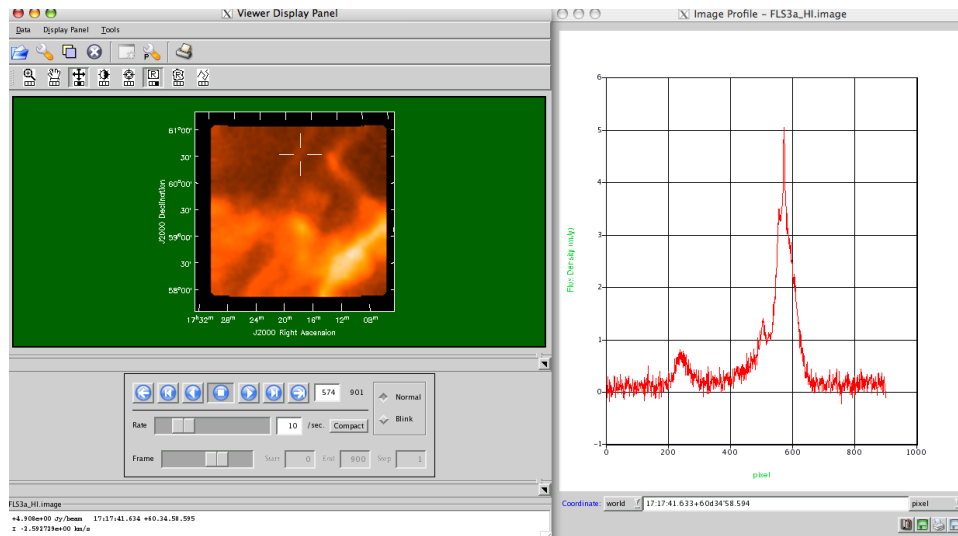
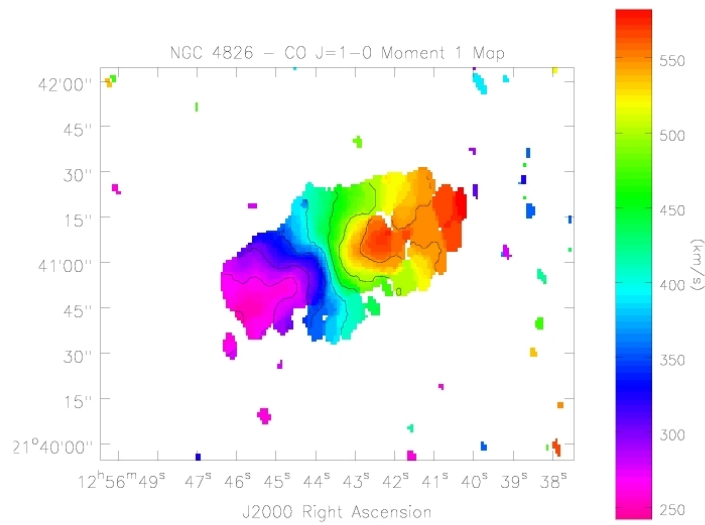


CASA Synthesis & Single Dish Reduction Cookbook



Version: September 5, 2007

Contents

1	Introduction	13
1.1	CASA Basics — Information for First-Time Users	15
1.1.1	Before Starting CASA	15
1.1.2	Starting CASA	16
1.1.3	Ending CASA	16
1.1.4	What happens if something goes wrong?	16
1.1.5	Python Basics for CASA	17
1.1.5.1	Variables	17
1.1.5.2	Lists and Ranges	18
1.1.5.3	Indexes	18
1.1.5.4	Indentation	19
1.1.5.5	System shell access	19
1.1.5.6	Executing Python scripts	19
1.1.6	Getting Help in CASA	20
1.1.6.1	TAB key	20
1.1.6.2	<code>help <taskname></code>	20
1.1.6.3	<code>help</code> and <code>PAGER</code>	21
1.1.6.4	<code>help par.<parameter></code>	21
1.1.6.5	Python <code>help</code>	22
1.2	Tasks and Tools in CASA	23
1.2.1	Further Details About Tasks	24
1.2.2	Setting Parameters and Invoking Tasks	26
1.2.2.1	The <code>default</code> Command	28
1.2.2.2	The <code>go</code> Command	28
1.2.2.3	The <code>inp</code> Command	29
1.2.2.4	The <code>restore</code> Command	32
1.2.2.5	The <code>saveinputs</code> Command	32
1.2.2.6	The <code>.last</code> file	34
1.3	Getting the most out of CASA	34
1.3.1	Your command line history and the <code>logger</code>	35
1.3.2	Where are my data in CASA?	36
1.3.3	What's in my data?	37
1.3.4	Data Selection in CASA	38
1.4	From Loading Data to Images	38

1.4.1	Loading Data into CASA	39
1.4.1.1	VLA: Filling data from VLA archive format	40
1.4.1.2	Filling data from UVFITS format	40
1.4.1.3	Loading FITS images	40
1.4.1.4	Concatenation of multiple MS	41
1.4.2	Data Examination, Editing, and Flagging	41
1.4.2.1	Interactive X-Y Plotting and Flagging	41
1.4.2.2	Flag the Data Non-interactively	41
1.4.2.3	Viewing and Flagging the MS	42
1.4.3	Calibration	42
1.4.3.1	Setting the flux density scale	43
1.4.3.2	Gain Calibration	43
1.4.3.3	Bandpass Calibration	43
1.4.3.4	Examining Calibration Solutions	43
1.4.3.5	Bootstrapping Flux Calibration	43
1.4.3.6	Calibration Accumulation	44
1.4.3.7	Correcting the Data	44
1.4.3.8	Splitting the Data	44
1.4.4	Synthesis Imaging	44
1.4.4.1	Making a “dirty” image	45
1.4.4.2	Cleaning a single-field image	45
1.4.4.3	Cleaning a mosaic	45
1.4.4.4	Feathering in a Single-Dish image	45
1.4.5	Self Calibration	46
1.4.6	Data and Image Analysis	46
1.4.6.1	What’s in an image?	46
1.4.6.2	Moments of an Image Cube	46
1.4.6.3	Regridding an Image	47
1.4.6.4	Displaying Images	47
1.4.7	Getting data and images out of CASA	47
2	Visibility Data Import, Export, and Selection	48
2.1	CASA Measurement Sets	48
2.1.1	Under the Hood: Structure of the Measurement Set	49
2.2	Data Import and Export	51
2.2.1	UVFITS Import and Export	52
2.2.2	VLA: Filling data from archive format (<code>importvla</code>)	53
2.2.3	ALMA: Filling ALMA Science Data Model (ASDM) observations	55
2.3	Summarizing your MS (<code>listobs</code>)	56
2.4	Concatenating multiple datasets (<code>concat</code>)	59
2.5	Data Selection	60
2.5.1	General selection syntax	61
2.5.1.1	String Matching	61
2.5.2	The <code>field</code> Parameter	62
2.5.3	The <code>spw</code> Parameter	63

2.5.3.1	Channel selection in the <code>spw</code> parameter	64
2.5.4	The <code>selectdata</code> Parameters	65
2.5.4.1	The <code>antenna</code> Parameter	65
2.5.4.2	The <code>scan</code> Parameter	66
2.5.4.3	The <code>timerange</code> Parameter	66
2.5.4.4	The <code>uvrange</code> Parameter	67
2.5.4.5	The <code>msselect</code> Parameter	68
3	Data Examination and Editing	69
3.1	Plotting and Flagging Visibility Data in CASA	69
3.2	Managing flag versions with <code>flagmanager</code>	69
3.3	Flagging auto-correlations with <code>flagautocorr</code>	71
3.4	X-Y Plotting and Editing of the Data	71
3.4.1	Plot control	73
3.4.2	<code>plotoptions</code>	74
3.4.2.1	<code>plotsymbol</code>	74
3.4.2.2	Iteration	75
3.4.2.3	Subplots	76
3.4.3	Interactive Flagging in <code>plotxy</code>	76
3.4.4	Exiting <code>plotxy</code>	77
3.4.5	Example session using <code>plotxy</code>	77
3.5	Non-Interactive Flagging using <code>flagdata</code>	80
3.5.1	Flag Antenna/Channels	81
3.5.1.1	Clipping in <code>flagdata</code>	81
3.6	Interactive flagging using the <code>viewer</code>	82
3.7	Browse the Data	82
4	Synthesis Calibration	91
4.1	Calibration Tasks	91
4.2	Calibration models for absolute flux density (<code>setjy</code>)	92
4.2.1	Dealing with resolved calibrators	94
4.3	Complex Gain Calibration (<code>gaincal</code>)	95
4.3.1	“A priori” gain curve calibration	97
4.3.2	“A priori” atmospheric opacity correction	98
4.3.2.1	Determining opacity corrections for VLA data	98
4.3.3	Other <i>a priori</i> Calibrations and Corrections	98
4.3.4	Polarization-dependent Gain (G)	99
4.3.5	Polarization-independent Gain (T)	100
4.3.6	GSPLINE solutions	100
4.4	Flux density scale calibration	101
4.4.1	Resolved flux density calibrators	103
4.5	Spectral Bandpass Calibration (<code>bandpass</code>)	104
4.5.1	B solutions	105
4.5.2	BPOLY solutions	106
4.6	Instrumental Polarization Calibration (D)	107

4.7	Manipulating Calibration Tables	108
4.7.1	Calibration Smoothing (<code>smoothcal</code>)	108
4.7.2	Calibration Interpolation and Accumulation (<code>accum</code>)	109
4.7.2.1	Interpolation using (<code>accum</code>)	109
4.7.2.2	Incremental Calibration using (<code>accum</code>)	110
4.8	Plotting Calibration Solutions (<code>plotcal</code>)	113
4.9	Listing calibration solutions with (<code>listcal</code>)	115
4.10	Application of Calibration (<code>applycal</code>)	117
4.10.1	Examine calibrated source data	118
4.11	Resetting the Calibration using (<code>clearcal</code>)	119
4.12	Optional: Split out Calibrated uv data (<code>split</code>)	119
4.13	Advanced Calibration and UV-Plane Analysis	119
4.13.1	UV-Plane Continuum Subtraction (<code>uvcontsub</code>)	119
4.13.2	Baseline-based Calibration (<code>blcal</code>)	121
4.13.3	Fringe Fitting (<code>fringeal</code>)	122
4.13.4	UV-Plane Model Fitting (<code>uvmodelfit</code>)	123
4.14	Example of Calibration	125
5	Synthesis Imaging	136
5.1	Imaging Tasks Overview	136
5.2	Common Imaging Task Parameters	137
5.2.1	The <code>cell</code> Parameter	137
5.2.2	The <code>field</code> Parameter	138
5.2.3	The <code>imagename</code> Parameter	138
5.2.4	The <code>imsize</code> Parameter	138
5.2.5	The <code>mode</code> Parameter	138
5.2.6	The <code>restfreq</code> Parameter	140
5.2.7	The <code>spw</code> Parameter	140
5.2.8	The <code>stokes</code> Parameter	140
5.2.9	The <code>uvfilter</code> Parameter	141
5.2.10	The <code>weighting</code> Parameter	141
5.2.10.1	'natural' weighting	142
5.2.10.2	'uniform' weighting	142
5.2.10.3	'superuniform' weighting	142
5.2.10.4	'radial' weighting	142
5.2.10.5	'briggs' weighting	143
5.2.11	The <code>vis</code> Parameter	143
5.3	Making a Dirty Image and PSF (<code>invert</code>)	144
5.4	Deconvolution using CLEAN (<code>clean</code>)	144
5.4.1	Specific <code>clean</code> Parameters	147
5.4.1.1	The <code>alg</code> Parameter	147
5.4.1.2	The <code>cleanbox</code> Parameter	148
5.4.1.3	The <code>gain</code> Parameter	149
5.4.1.4	The <code>mask</code> Parameter	149
5.4.1.5	The <code>niter</code> Parameter	149

5.4.1.6	The <code>threshold</code> Parameter	149
5.4.2	Interactive Cleaning	149
5.5	Mosaic Deconvolution using CLEAN (<code>mosaic</code>)	150
5.5.1	The <code>cyclefactor</code> Parameter	151
5.5.2	The <code>cyclespeedup</code> Parameter	152
5.5.3	The <code>ftmachine</code> Parameter	152
5.5.4	The <code>minpb</code> Parameter	152
5.5.5	The <code>modelimage</code> Parameter	152
5.5.6	The <code>mosweight</code> Parameter	153
5.5.7	The <code>phasecenterid</code> Parameter	153
5.5.8	The <code>scaletype</code> Parameter	153
5.5.9	The <code>sdimage</code> Parameter	153
5.6	Combined Single Dish and Interferometric Imaging (<code>feather</code>)	153
5.7	Making Deconvolution Masks (<code>makemask</code>)	154
5.8	Transforming an Image Model (<code>ft</code>)	156
5.9	Image-plane deconvolution (<code>deconvolve</code>)	156
5.10	Self-Calibration	157
5.11	Example of Imaging	157
6	Displaying Images	161
6.1	Starting the <code>viewer</code>	161
6.1.1	Starting the <code>casaviewer</code> outside of <code>casapy</code>	161
6.2	The <code>viewer</code> GUI	162
6.3	Viewing a raster map	165
6.4	Viewing a contour map	165
6.5	Viewing a MeasurementSet with visibility data	165
6.6	Adjusting Display Parameters	165
6.7	Adjusting Canvas Parameters/Multi-panel displays	166
6.8	Overlay contours on a raster map	166
7	Image Analysis	175
7.1	Summary of an Image and Headers	175
7.2	Computing the Moments of an Image Cube (<code>immoments</code>)	177
7.3	Regridding an Image (<code>regridimage</code>)	178
7.4	Image display in the <code>viewer</code>	178
7.4.1	Image statistics	178
7.5	Image Import/Export to FITS	179
8	Single Dish Data Processing	180
8.1	Guidelines for Use of ASAP and SDtasks in CASA	180
8.1.1	Environment Variables	180
8.1.2	Assignment	181
8.1.3	Lists	181
8.1.4	Dictionaries	182
8.1.5	Line Formatting	182

8.2	Single Dish Analysis Tasks	183
8.2.1	SDtask Summaries	184
8.2.2	A Single Dish Analysis Use Case With SDTasks	195
8.3	Using The ASAP Toolkit Within CASA	208
8.3.1	Environment Variables	210
8.3.2	Import	211
8.3.3	Scantable Manipulation	213
8.3.3.1	Data Selection	213
8.3.3.2	State Information	213
8.3.3.3	Masks	215
8.3.3.4	Scantable Management	215
8.3.3.5	Scantable Mathematics	215
8.3.3.6	Scantable Save and Export	216
8.3.4	Calibration	216
8.3.4.1	Tsys scaling	216
8.3.4.2	Flux and Temperature Unit Conversion	217
8.3.4.3	Gain-Elevation and Atmospheric Optical Depth Corrections	217
8.3.4.4	Calibration of GBT data	217
8.3.5	Averaging	218
8.3.6	Spectral Smoothing	220
8.3.7	Baseline Fitting	220
8.3.8	Line Fitting	221
8.3.9	Plotting	222
8.3.10	Single Dish Spectral Analysis Use Case With ASAP Toolkit	222
8.4	Single Dish Imaging	224
8.4.1	Single Dish Imaging Use Case With ASAP Toolkit	225
8.5	Known Issues, Problems, Deficiencies and Features	226
A	Appendix: Obtaining and Installing CASA	233
A.1	Installation Script	233
A.2	Startup	233
B	Appendix: Python and CASA	234
B.1	Automatic parentheses	234
B.2	Indentation	235
B.3	Lists and Ranges	235
B.4	System shell access	236
B.5	Logging	238
B.6	History and Searching	239
B.7	Macros	241
B.8	On-line editing	241
B.9	Executing Python scripts	242
B.10	How do I exit from CASA?	242

C	Appendix: Annotated Example Scripts	243
C.1	NGC 5921 — VLA red-shifted HI emission	243
C.1.1	NGC 5921 data summary	255
C.2	Jupiter — VLA continuum polarization	257
D	Appendix: CASA Dictionaries	282
D.1	AIPS – CASA dictionary	282
D.2	MIRIAD – CASA dictionary	282
D.3	CLIC – CASA dictionary	282

List of Tables

2.1	Common MS Columns	51
2.2	Commonly accessed MAIN Table columns	52
4.1	Recognized Flux Density Calibrators.	93
D.1	MIRIAD – CASA dictionary	283
D.2	CLIC–CASA dictionary	284

List of Figures

1.1	Issue/Defect Tracking system. Left: http://bugs.aoc.nrao.edu page showing login entry fields. Right: Screen after selecting the "Create New Issue" tab along the top.	17
1.2	Screen shot of the default CASA inputs for task <code>clean</code> .	30
1.3	The <code>clean</code> inputs where some parameters have been set to non-default values (blue). Note that some of the boldface ones have opened up new dependent sub-parameters (indented green).	31
1.4	The <code>clean</code> inputs where some parameters have been set to invalid values. These are drawn in red to draw attention to the problem.	32
1.5	CASA Logger GUI window	35
1.6	CASA Logger - Search example: Specify a string in the entry box to have all instances of the found string highlighted.	36
1.7	CASA Logger - Filter facility: The log output can be sorted by Priority, Time, Origin. One can also filter for a string found in the Message.	37
1.8	CASA Logger - Insert facility: The log output can be augmented by adding notes or comments during the reduction. The file should then be saved to disk to retain these changes.	38
1.9	Flow chart of the data processing operations that a general user will carry out in an end-to-end CASA reduction session.	39
2.1	The contents of a Measurement Set. These tables compose a Measurement Set named <code>ngc5921.ms</code> on disk. This display is obtained by using the File:Open menu in <code>browsetable</code> .	50
3.1	The <code>plotxy</code> plotter. The bottom set of buttons on the lower left are: 1,2,3) Home, Back, and Forward . Click to navigate between previously defined views (akin to web navigation). 4) Pan . Click and drag to pan to a new position. 5) Zoom . Click to define a rectangular region for zooming. 6) Subplot Configuration . Click to configure the parameters of the subplot and spaces for the figures. 7) Save . Click to launch a file save dialog box. The upper set of buttons in the lower left are: 1) Mark Region . Press this to begin marking regions (rather than zooming or panning). 2,3,4) Flag, Unflag, Locate . Click on these to flag, unflag, or list the data within the marked regions. 5) Next . Click to move to the next in a series of iterated plots. Finally, the cursor readout is on the bottom right.	83
3.2	<code>plotxy</code> iteration plot: The first set of plots from the example in § 3.4.2.2. Each time you press the Next button, you get the next series of plots.	84

3.3	Multi-panel display of visibility versus channel (top), antenna array configuration (bottom left) and the resulting uv coverage (bottom right). The commands to make these three panels respectively are: 1) <code>plotxy('n5921.ms', xaxis='channel', datacolumn='corrected', field='0', subplot=211, plotcolor='', plotsymbol='go')</code> , 2) <code>plotxy(xaxis='x', subplot=223, plotsymbol='r.')</code> , 3) <code>plotxy(xaxis='u', yaxis='v', subplot=224, plotsymbol='b.')</code>	85
3.4	Plot of amplitude versus uv distance, before (left) and after (right) flagging two marked regions. The call was: <code>plotxy(vis='n5921.ms', xaxis='uvdist', plotsymbol='b.', subplot=111, datacolumn='data', field='1445*')</code>	86
3.5	flagdata : Example showing before and after displays using a selection of one antenna and a range of channels. Note that each invocation of the flagdata task represents a cumulative selection, i.e., running <code>antenna='0'</code> will flag all data with antenna 0, while <code>antenna='0', spw='0:10 15'</code> will flag only those channels on antenna 0.	87
3.6	flagdata : Flagging example using the clip facility.	87
3.7	browsetable : The browser displays the main table within a frame. Hit the expand button to fill the browser frame (this has been done for this figure). You can scroll through the data (x=columns of the MAIN table, and y=the rows) or select a specific page or row as desired.	88
3.8	browsetable : You can use the Menu option View to look at other tables within an MS. If you select on View:Table Keywords you get the image displayed. You can then select on a table to view its contents.	89
3.9	browsetable : View the SOURCE table of the MS.	90
4.1	plotcal : Display of the amplitude solutions for short solution interval table (0.5 seconds: top) and the smoothed table using a smoothtime of 1000 seconds.	109
4.2	plotcal : Display of the amplitude solutions for NGC 5921; original (left), interpolated solutions-20s sampling (right).	110
4.3	plotcal : Display of the amplitude and phase gain solutions (for all data).	114
4.4	plotcal : Display of the amplitude and phase bandpass solutions (for all data).	115
4.5	plotcal : Display of a 3x2 grid of bandpass solutions, iterating over antenna identifier index.	116
4.6	Use of plotxy to display corrected data (red points) and uv model fit data (blue circles).	126
5.1	Screenshot of the interactive clean window during deconvolution of the VLA 6m Jupiter dataset. We have already cleaned 100 iterations in the region previously marked, and are ready to extend the mask to pick up the newly revealed emission. Note the boxes at the top right where the npercycle , niter , and threshold can be changed.	160
6.1	Viewer Display Panel with no data loaded. Each section of the GUI is explained below	167
6.2	casaviewer : Illustration of a raster image in the Viewer Display Panel(left) and the Load Data panel (right).	168
6.3	casaviewer : Illustration of a raster image in the Viewer Display Panel(left) and the Load Data panel (right).	169
6.4	casaviewer : Display of visibility data. The default axes are time vs. baseline.	170

6.5	casaviewer: Data display options. In the left panel, the Display axes, Hidden axes, and Basic Settings options are shown; in the right panel, the Position tracking and Axis labels options are shown.	171
6.6	casaviewer: Data display options. In this final, third panel , the Axis label properties are shown.	172
6.7	casaviewer: A multi-panel display set up through the Viewer Canvas Manager. . . .	173
6.8	casaviewer: Display contour overlay on top of a raster image.	174
8.1	Wiring diagram for the SDtask <code>sdca1</code> . The stages of processing within the task are shown, along with the parameters that control them.	229
8.2	Multi-panel display of the scantable. There are two plots per scan indicating the <code>_psr</code> (reference position data) and the <code>_ps</code> (source data).	230
8.3	Two panel plot of the calibrated spectra. The GBT data has a separate scan for the SOURCE and REFERENCE positions so scans 20,21,22 and 23 result in these two spectra.	230
8.4	Calibrated spectrum with a line at zero (using histograms).	231
8.5	FLS3a HI emission. The display illustrates the visualization of the data cube (left) and the profile display of the cube at the cursor location (right); the Tools menu of the Viewer Display Panel has a Spectral Profile button which brings up this display. By default, it grabs the left-mouse button. Pressing down the button and moving in the display will show the profile variations.	232

Chapter 1

Introduction

This document describes how to calibrate and image interferometric and single-dish radio astronomical data using the CASA (Common Astronomy Software Application) package. CASA is a suite of astronomical data reduction tools and tasks that can be run via the IPython interface to Python. CASA is being developed in order to fulfill the data post-processing requirements of the ALMA and EVLA projects, but also provides basic and advanced capabilities useful for the analysis of data from other radio, millimeter, and submillimeter telescopes.

Currently, CASA is in an **Alpha Release** stage. This means that only a small subset of the eventual functionality is available. Furthermore, the package is under intense development, and some features might change in future releases. This should be taken into account as users begin to learn the package. We will do our best to point out commands, tasks, and parameters that are likely to change underfoot. Unfortunately, bugs and crashes also come along with the Alpha territory. We will do our best to stamp these out as soon as we find them, but sometimes known bugs will persist until we can find the right time to fix them (like in a task that we know we want to make a big change to next month). See the release notes for the current version for more details. In this cookbook, we will try to point out known pitfalls and workarounds in the **Alpha Alert** boxes, or in ALPHA ALERTs in the text.

This cookbook is a task-based walkthrough of interferometric data reduction and analysis. In CASA, **tasks** represent the more streamlined operations that a typical user would carry out. The idea for having tasks is that they are simple to use, provide a more familiar interface, and are easy to learn for most astronomers who are familiar with radio interferometric data reduction (and hopefully for novice users as well). In CASA, the **tools** provide the full capability of the package, and are the atomic functions that form the basis of data reduction. These tools augment the tasks, or fill in gaps left by tasks that are under development but not yet available. See

Alpha Alert!

Boxes like this will bring to your attention some of the features (or lack thereof) in the current Alpha release version of CASA.

Inside the Toolkit:

Throughout this Cookbook, we will occasionally intersperse boxed-off pointers to parts of the toolkit that power users might want to explore.

the **CASA Toolkit Guide** for more details on the tools. Note that in most cases, the tasks are Python interface scripts to the tools, but with specific, limited access to them and a standardized interface for parameter setting. The tasks and tools can be used together to carry out more advanced data reduction operations.

For the moment, the audience is assumed to have some basic grasp of the fundamentals of synthesis imaging, so details of how a radio interferometer or telescope works and why the data needs to undergo calibration in order to make synthesis images are left to other documentation — a good place to start might be *Synthesis Imaging in Radio Astronomy II* (1999, ASP Conference Series Vol. 180, eds. Taylor, Carilli & Perley).

This cookbook is broken down by the main phases of data analysis:

- data import and export (Chapter 2),
- examination and flagging of data (Chapter 3),
- interferometric calibration (Chapter 4),
- interferometric imaging (Chapter 5),
- image display (Chapter 6), and
- image analysis (Chapter 7).

There is also a special chapter on Single Dish data analysis (Chapter 8).

The appendices provide more details on what's happening under the hood of CASA, as well as supplementary material on tasks, scripts, and relating CASA to other packages. These appendices include:

- obtaining and installing CASA (Appendix A),
- more details about Python and CASA (Appendix B),
- annotated scripts for typical data reduction cases (Appendix C), and
- CASA dictionaries to AIPS, MIRIAD, and CLIC (Appendix D).

The CASA User Documentation includes:

- **CASA Synthesis & Single Dish Reduction Cookbook** — this document, a task-based data analysis walk-through and instructions;
- **CASA in-line help** — accessed using `help` in the `casapy` interface;
- The **CASA Toolkit Guide** — useful when the tasks do not have everything you want and you need more power and functionality. Also contains more detailed descriptions of the philosophy of data analysis;

- The **CASA User Reference Manual** — details on a specific task or tool does and how to use it. **ALPHA ALERT:** Currently the Reference Manual describes only tools, not tasks.

The CASA home page can be found at:

- <http://casa.nrao.edu>

From there you can find documentation and assistance for the use of the package, including the User Documentation.

1.1 CASA Basics — Information for First-Time Users

This section assumes that CASA has been installed on your LINUX or OSX system. See Appendix A for instructions on how to obtain and install CASA.

1.1.1 Before Starting CASA

To define environment variables and the `casapy` alias, you will need to run one of the `casainit` shell scripts. The location of the startup scripts for CASA will depend upon where you installed CASA on your system. Sometimes, you will have multiple versions (for example, various released versions).

For example, NRAO-AOC testers would do the following on an AOC RHE4 machine:

```
In bash:
> . /home/casa/casainit.sh
or for csh:
> source /home/casa/casainit.csh
```

depending on what shell you are running (Bourne or [t]csh).

Before starting up `casapy`, you should set or reset any *environment variables* needed, as CASA will adopt these on startup. For example, the `PAGER` environment variable determines how help is displayed in the CASA terminal window (see § 1.1.6.3). The choices are (in `bash`):

```
PAGER=less
PAGER=more
PAGER=cat
```

or in `csh` or `tcsh`:

```
setenv PAGER less
setenv PAGER more
setenv PAGER cat
```

The actions of these are as if you were using the equivalent Unix shell command to view the help material. See § 1.1.6.3 for more information on these choices. We recommend using the `cat` option for most users, as this works smoothly both interactively and in scripts.

Note: there is currently no way within CASA to change these environment variables.

1.1.2 Starting CASA

After having run the appropriate `casainit` script, CASA is started by typing

```
casapy
```

on the command line. After startup information, you should get an IPython

```
CASA <1>:
```

command prompt in the xterm window where you started CASA. CASA will take approximately 10 seconds to initialize at startup in a new working directory; subsequent startups are faster. CASA is active when you get a

```
CASA <1>
```

prompt in the command line interface.

1.1.3 Ending CASA

You can exit CASA by typing:

```
CTRL-D, %exit, or quit.
```

If you don't want to see the question "Do you really want to exit [y]/n?", then just type

```
Exit
```

and CASA will stop right then and there.

1.1.4 What happens if something goes wrong?

First, always check that your inputs are correct; use the `help <taskname>` or `help par.<parameter_name>` to review the inputs/output.

You can submit a question/bug/enhancement via the site:

```
http://bugs.aoc.nrao.edu
```

Login (or register yourself if you don't have a login/password); click the 'Create New Issue' along the top tabs to file a question/bug/enhancement.

If something has gone wrong and you want to stop what is executing, then typing 'Control-C' will usually cleanly abort the application.

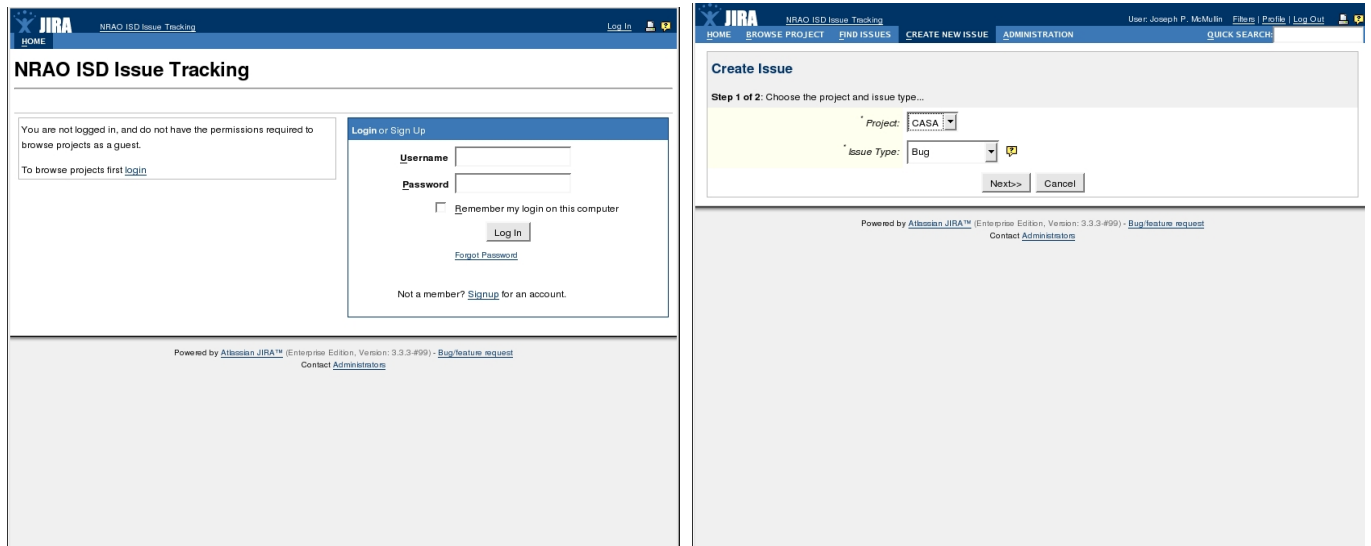


Figure 1.1: Issue/Defect Tracking system. **Left:** <http://bugs.aoc.nrao.edu> page showing login entry fields. **Right:** Screen after selecting the "Create New Issue" tab along the top.

1.1.5 Python Basics for CASA

Within CASA, you use Python to interact with the system. This does not mean an extensive Python course is necessary - basic interaction with the system (assigning parameters, running tasks) is straightforward. At the same time, the full potential of Python is at the more experienced user's disposal. Some further details about Python, IPython, and the interaction between Python and CASA can be found in Appendix B.

The following are some examples of helpful hints and tricks on making Python work for you in CASA.

1.1.5.1 Variables

Python variables are set using the `<parameter> = <value>` syntax. Python assigns the type dynamically as you set the value, and thus you can easily give it a non-sensical value, e.g.

```
vis = 'ngc5921.ms'
vis = 1
```

The CASA parameter system will check types when you run a task or tool, or more helpfully when you set inputs using `inp` (see below). CASA will check and protect the assignments of the global parameters in its namespace.

Note that Python variable names are case-sensitive:

```

CASA <109>: Foo = 'bar'
CASA <110>: foo = 'Bar'
CASA <111>: foo
  Out[111]: 'Bar'
CASA <112>: Foo
  Out[112]: 'bar'

```

so be careful.

Also note that mis-spelling a variable assignment will not be noticed (as long as it is a valid Python variable name) by the interface. For example, if you wish to set `correlation='RR'` but instead type `corellation='RR'` you will find `correlation` unset and a new `corellation` variable set. Command completion (see § 1.1.6.1) should help you avoid this.

1.1.5.2 Lists and Ranges

Sometimes, you need to give a task a list of indices. If these are consecutive, you can use the Python `range` function to generate this list:

```

CASA <1>: iflist=range(4,8)
CASA <2>: print iflist
[4, 5, 6, 7]
CASA <3>: iflist=range(4)
CASA <4>: print iflist
[0, 1, 2, 3]

```

See Appendix B.3 for more information.

1.1.5.3 Indexes

As in C, Python indices are 0-based. For example, the first element in a list `antlist` would be `antlist[0]`:

```

CASA <113>: antlist=range(5)
CASA <114>: antlist
  Out[114]: [0, 1, 2, 3, 4]
CASA <115>: antlist[0]
  Out[115]: 0
CASA <116>: antlist[4]
  Out[116]: 4

```

CASA also uses 0-based indexing internally for elements in the Measurement Set (MS – the basic construct that contains visibility and/or single dish data; see Chapter 2). Thus, we will often talk about Field or Antenna “ID”s which will start at 0. For example, the first field in an MS would have `FIELD_ID==0` in the `MSselect` syntax, and can be addressed as indexed as `field='0'` in most tasks, as well as by name `field='0137+331'` (assuming that's the name of the first field). You will see these indices in the MS summary from the task `listobs`.

1.1.5.4 Indentation

Python pays attention to the indentation of lines, as it uses indentation to determine the level of nesting in loops. Be careful when cutting and pasting: if you get the wrong indentation, then unpredictable things can happen (usually it just gives an error).

See Appendix B.2 for more information.

1.1.5.5 System shell access

Any input line beginning with a `'!` character is passed verbatim (minus the `'!`, of course) to the underlying operating system (*the sole exception to this is the `'cd` command which must be executed without the `'!`*). Also, several common commands (`ls`, `pwd`, `cd`, `less`) may be executed with or without the `'!`.

Example:

```
CASA <5>: !rm -r mydata.ms
```

Note that if you want to access a Unix environment variable, you will need to prefix with a double `$$` instead of a single `$` — for example, to print the value of the `$PAGER` variable, you would use

```
CASA <6>: !echo $$PAGER
```

See Appendix B.4 for more information.

1.1.5.6 Executing Python scripts

You can execute Python scripts (ASCII text files containing Python or `casapy` commands) using the `execfile` command. For example, to execute the script contained in the file `myscript.py` (in the current directory), you would type

```
CASA <7>: execfile('myscript.py')
```

or

```
CASA <8>: execfile 'myscript.py'
```

which will invoke the IPython auto-parenthesis feature.

NOTE: in some cases, you can use the IPython `run` command instead, e.g.

```
CASA <9>: run myscript.py
```

In this case, you do not need the quotes around the filename. This is most useful for re-initializing the task parameters, e.g.

```
CASA <10>: run clean.last
```

(see § 1.2.2.6).

See Appendix B.9 for more information.

1.1.6 Getting Help in CASA

1.1.6.1 TAB key

At **any** time, hitting the <TAB> key will complete any available commands or variable names and show you a list of the possible completions if there's no unambiguous result. It will also complete filenames in the current directory if no CASA or Python names match.

For example, it can be used to list the available functionality using minimum match; once you have typed enough characters to make the command unique, <TAB> will complete it.

```
CASA <15>: cle<TAB>
clean                clean_description      clearcal_check_params
clearplot            clearstat
clean_check_params   clear                    clearcal_defaults
clearplot_defaults   clearstat_defaults
clean_defaults       clearcal                  clearcal_description
clearplot_description clearstat_description
```

1.1.6.2 help <taskname>

Basic information on an application, including the parameters used and their defaults, can be obtained by typing `help task` (`pdoc task` and `task?` are equivalent commands with some additional programming information returned). `help task` provides a one line description of the task and then lists all parameters, a brief description of the parameter, the parameter default, an example setting the parameter and any options if there are limited allowed values for the parameter.

```
CASA <45>: help uvcontsub
-----> help(uvcontsub)
Help on function uvcontsub in module uvcontsub:

uvcontsub(vis=None, field=None, spw=None, channels=None, solint=None,
fitorder=None, fitmode=None, async=None)
    Continuum fitting and subtraction in the uv plane:

    Keyword arguments:
    vis -- Name of input visibility file (MS)
           default: <unset>; example: vis='ngc5921.ms'
    field -- Field name(s); this will use a minimum match on the strings
            default: field = '' means use all sources
            field = 1 # will get field_id=1 (if you give it an
```

```

        integer, it will retrieve the source with that index.
    field = '1328+307' specifies source '1328+307'
        minimum match can be used, egs field = '13*' will
        retrieve '1328+307' if unique or exists.
        source names with imbedded blanks cannot be included.
    spw -- Spectral window index identifier
        default=0; example: spw=1
    channels -- Range of channels to fit
        default::; example: channels=range(4,7)+range(50,60)
    solint -- Averaging time (seconds)
        default: 0.0 (scan-based); example: solint=10
    fitorder -- Polynomial order for the fit
        default: 0; example: fitorder=1
    fitmode -- Use of the continuum fit model
        default: 'subtract'; example: fitmode='replace'
    <Options:
        'subtract'-store continuum model and subtract from data,
        'replace'-replace vis with continuum model,
        'model'-only store continuum model>

```

1.1.6.3 help and PAGER

Your PAGER environment variable (§ 1.1.1) determines how help is displayed in the terminal window where you start CASA. If you set your bash environment variable PAGER=less (setenv PAGER less in csh) then typing help <taskname> will show you the help but the text will vanish and return you to the command line when you are done viewing it. Setting PAGER=more (setenv PAGER more) will scroll the help onto your command window and then return you to your prompt (but leaving it on display). Setting PAGER=cat (setenv PAGER cat) will give you the more equivalent without some extra formatting baggage and is the recommended choice.

If you have set PAGER=more or PAGER=less, the help display will be fine, but the display of 'taskname?' will often have confusing formatting content at the beginning (lots of ESC surrounding the text). This can be remedied by exiting casapy and doing an 'unset PAGER' (unsetenv PAGER in [t]csh) at the Unix command line.

You can see the current value of the PAGER environment variable with CASA by typing:

```
!echo $$PAGER
```

(note the double \$\$). This will show what command paging is pointed to.

1.1.6.4 help par.<parameter>

Typing help par.<parameter> provides a brief description of a given parameter <parameter>.

```
CASA <46>: help par.robust
```

Help on function robust in module parameter_dictionary:

```
robust()
  Brigg's robustness parameter.
```

```
Options: -2.0 (close to uniform) to 2.0 (close to natural)
```

1.1.6.5 Python help

Typing `help` at the `casapy` prompt with no arguments will bring up the native Python help facility, and give you the `help>` prompt for further information; hitting `<RETURN>` at the `help` prompt returns you to the `CASA` prompt. You can also get the short help for a `CASA` method by typing `'help tool.method'` or `'help task'`.

```
CASA <2>: help
-----> help()
```

Welcome to Python 2.5! This is the online help utility.

If this is your first time using Python, you should definitely check out the tutorial on the Internet at <http://www.python.org/doc/tut/>.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

```
help> keywords
```

Here is a list of the Python keywords. Enter any keyword to get more help.

<code>and</code>	<code>else</code>	<code>import</code>	<code>raise</code>
<code>assert</code>	<code>except</code>	<code>in</code>	<code>return</code>
<code>break</code>	<code>exec</code>	<code>is</code>	<code>try</code>
<code>class</code>	<code>finally</code>	<code>lambda</code>	<code>while</code>
<code>continue</code>	<code>for</code>	<code>not</code>	<code>yield</code>
<code>def</code>	<code>from</code>	<code>or</code>	
<code>del</code>	<code>global</code>	<code>pass</code>	
<code>elif</code>	<code>if</code>	<code>print</code>	

```
help>
```

```
# hit <RETURN> to return to CASA prompt
```

You are now leaving help and returning to the Python interpreter. If you want to ask for help on a particular object directly from the interpreter, you can type "help(object)". Executing "help('string')" has the same effect as typing a particular string at the help> prompt.

```
CASA <3>: help gaincal
-----> help(gaincal)
Help on function gaincal in module gaincal:
```

```
gaincal(vis=None, caltable=None, field=None, spw=None, selectdata=None,
timerange=None, uvrange=None, antenna=None, scan=None, msselect=None,
solint=None, preavg=None, refant=None, minsnr=None, solnorm=None,
gaintype=None, calmode=None, append=None, splinetime=None, npointaver=None,
phasewrap=None, gaincurve=None, opacity=None, tau=None, gaintable=None,
bptable=None, pointtable=None, async=None)
```

Determine temporal gains from calibrator observations:

The complex gains for each antenna/spwid are determined from the data column (raw data) and the model column for the specified calibrator sources. A solution interval or a spline fit can be obtained. Previous calibrations can be applied on the fly.

Keyword arguments:

```
vis -- Name of input visibility file
      default: none; example: vis='ngc5921.ms'
caltable -- Name of output gain calibration table
           default: none; example: caltable='ngc5921.gcal'
```

```
--- Data Selection (see help par.selectdata for more detailed information)
```

```
field -- Select field using field id(s) or field name(s).
        [run listobs to obtain the list id's or names]
        default: ''=all fields
```

etc. etc.

For a full list of keywords associated with the various tasks, see the **CASA User Reference Manual**. Further help in working within the Python shell is given in Appendix B. **ALPHA ALERT:** The User Reference Manual currently covers only tools, not tasks.

1.2 Tasks and Tools in CASA

Originally, CASA consisted of a collection of tools, combined in the so-called toolkit. Since the majority of prospective users is far more familiar with the concept of tasks, an effort is underway

to replace most - if not all - toolkit functionality by tasks.

While running CASA, you will have access to and be interacting with tasks, either indirectly by providing parameters to a task, or directly by running a task. Each task has a well defined purpose, and a number of associated parameters, the values of which are to be supplied by the user. Technically speaking, tasks are built on top of tools - when you are running a task, you are running tools in the toolkit, though this should be transparent.

As more tasks are being written, and the functionality of each task is enhanced, there will be less and less reason to run tools in the toolkit. We are working toward a system in which direct access to the underlying toolkit is unnecessary for all standard data processing.

1.2.1 Further Details About Tasks

As mentioned in the introduction, tasks in CASA are python interfaces to the more basic toolkit. Tasks are executed to perform a single job, such as loading, plotting, flagging, calibrating, and imaging the data.

Basic information on tasks, including the parameters used and their defaults, can be obtained by typing `help <taskname>` or `<taskname>?` at the CASA prompt, where `<taskname>` is the name of a given task. As described above in § 1.1.6.2, `help <taskname>` provides a description of the task and then lists all parameters, a brief description of the parameter, the parameter default, an example setting the parameter and any options if there are limited allowed values for the parameter.

To see what tasks are available in CASA, use `tasklist`.

```
CASA <4>: tasklist()
Available tasks:
```

Import/Export	Information	Editing	Display/Plot
-----	-----	-----	-----
importvla	listcal	flagautocorr	clearplot
importasdm	listhistory	flagdata	plotants
importfits	listobs	flagmanager	plotcal
importuvfits	imhead	plotxy	plotxy
exportfits			viewer
exportuvfits			
Calibration	Imaging	Modelling	Utility
-----	-----	-----	-----
accum	clean	setjy	help task
applycal	feather	uvcontsub	help par.parameter
bandpass	ft	uvmodelfit	taskhelp
blcal	invert		tasklist
gaincal (G)	makemask		browsetable
fluxscale	mosaic		clearplot
fringecal (K)			clearstat
clearcal			concat
listcal			filecatalog


```

pointcal                               startup
smoothcal                             split
Image Analysis Simulation              restore
-----
(imcontsub)      almasimmos
imhead
immoments
regridimage

```

Typing `taskhelp` provides a one line description of all available tasks.

```

CASA <5>: taskhelp()
Available tasks:

```

```

accum          : Accumulate calibration solutions into a cumulative table
almasimmos    : ALMA mosaic simulation task (prototype)
applycal      : Apply calculated calibration solutions
bandpass      : Calculate a bandpass calibration solution
blcal         : Calculate a baseline-based calibration solution
browsetable   : Browse a visibility data set or calibration table
clean         : Calculate a deconvolved image with selected clean algorithm
clearcal      : Re-initialize visibility data set calibration data
clearplot     : Clear matplotlib plotter and all layers
clearstat     : Clear all read/write locks on tables
concat        : Concatenate two visibility data sets
deconvolve    : Image based deconvolver
exportfits    : Convert a CASA image to a FITS image
exportuvfits  : Export MS to UVFITS file
feather       : Feather together an interferometer and a single dish image in the Fourier plane
filecatalog   : File Catalog GUI
flagautocorr  : Flag autocorrelations (typically in a filled VLA data set)
flagdata      : Flag data based on time, baseline, antenna, clip, etc
flagmanager   : Enable list, save, restore and delete of flag versions
fluxscale     : Bootstrap the flux density scale from standard calibrators
fringecal     : Calculate a baseline-based fringe-fitting solution (phase, delay, delay-rate)
ft            : Fourier transform the specified model (or component list)
gaincal       : Calculate gain calibration solutions
imhead        : List/set image header properties
immoments     : Compute moments from an image (see URM for mathematical details)
importasdm    : Convert an ALMA Science Data Model directory to a CASA visibility data set (MS)
importfits    : Convert a FITS image to a CASA image
importuvfits  : Convert a UVFITS file to a CASA visibility data set (MS)
importvla     : Convert VLA archive file(s) to a CASA visibility data set (MS)
invert        : Calculate a dirty image and dirty beam
listcal       : List calibration solutions to terminal
listhistory   : List the processing history of a data set
listobs       : List the observations in a data set
makemask      : Calculate mask from image or visibility data set
mosaic        : Calculate a multi-field deconvolved image with selected clean algorithm

```

```

plotants      : Plot the antenna distribution in local reference frame
plotcal       : Plot calibration solutions
plotxy        : Plot points for selected X and Y axes
pointcal      : Calculate pointing error calibration
regridimage   : Grid image to same shape and coordinates as template
restore       : Restore all parameters to defaults
setjy         : Compute the model visibility for a specified source flux density
smoothcal     : Produce a smoothed calibration table
split         : Create a new data set (MS) from a subset of an existing data set (MS)
uvcontsub     : Continuum fitting and subtraction in the uv plane
uvmodelfit    : Fit a single component source model to the uv data
viewer        : View an image or visibility data set

```

Typing `startup` will provide the startup page displayed when entering CASA.

Details on individual tasks are also presented in the **CASA User Reference Manual**. **ALPHA ALERT:** The User Reference Manual currently covers only tools, not tasks.

1.2.2 Setting Parameters and Invoking Tasks

Tasks require input parameters (sometimes called keywords). A task, like a tool, is a function under Python and may be written in Python, C, or C++ (the CASA toolkit is made up of C++ functions). Tasks and tools can be executed in several ways.

First, one may call tasks and tools by name with parameters set on the same line. Parameters may be set either as explicit `<parameter>=<value>` arguments, or as a series of comma delimited `<value>`s in the correct order for that task or tool. Note that missing parameters will retain the values previously given to those parameters – use `default <taskname>` to avoid this behavior. For example, the following are equivalent:

```

# Specify parameter names for each keyword input:
plotxy(vis='ngc5921.ms',xaxis='channel',yaxis='amp',datacolumn='data')
# when specifying the parameter name, order doesn't matter, e.g.:
plotxy(xaxis='channel',vis='ngc5921.ms',datacolumn='data',yaxis='amp')
# use parameter order for invoking tasks
plotxy('ngc5921.ms','channel','amp','data')

```

Second, one can set parameters for tasks (but currently not for tools) by performing the assignment within the CASA shell and then inspecting them using the `inp` command:

```

CASA <30>: default(plotxy)
CASA <31>: vis = 'ngc5921.ms'
CASA <32>: xaxis = 'channel'
CASA <33>: yaxis = 'amp'

```

Inside the Toolkit:

In the current version of CASA, you cannot use the task parameter setting features, such as the `inp`, `default`, or `go` commands, for the tools.

```

CASA <34>: datacolumn = 'data'
CASA <35>: inp(plotxy)
vis                = 'ngc5921.ms'      # Name of input visibility
xaxis              = 'channel'         # azimuth,elevation,hourangle,baseline,channel,time,u,v,w,uvdist
yaxis              = 'amp'             # azimuth,elevation,hourangle,baseline,amp,pha,u,v,w,uvdist
datacolumn         = 'data'           # data (raw), corrected, model
field              = ''                # Select data based on field name or index
spw                = ''                # Select data based on spectral window
selectdata         = False             # Select a subset of the data - opens selection params
average            = ''                # Select averaging mode: time or channel
subplot            = 111                # Panel number on display screen (yxn)
overplot           = False             # Overplot values on current plot (if possible)
showflags          = False             # Show flagged data
iteration           = ''                # Plot separate panels by field, antenna, baseline, scan, feed
plotsymbol         = '.'              # pylab plot symbol
plotcolor          = 'darkcyan'        # pylab plot color
markersize        = 5.0               # Size of plotted marks
linewidth          = 1.0               # Width of plotted lines
connect            = 'none'            # Specifies which points are connected with lines
plotrange          = [-1, -1, -1, -1] # The range of data to be plotted, can be time values
skipnpoints        = 1                 # Plot every nth point
multicolor         = False             # Plot polarizations and channels in different colors
replacetopplot    = False             # Replace the last plot or not when overplotting
removeoldpanels    = True              # Turn on/of automatic clearing of panels
title              = ''                # Plot title (above plot)
xlabel             = ''                # Label for x-axis
ylabel             = ''                # Label for y-axis
fontsize           = 10.0              # Font size for labels
windowsize         = 1.0               # Window size

```

See § 1.2.2.3 below for more details on the use of the `inputs` command.

All task parameters have **global** scope within CASA: the parameter values are common to all tasks and also at the CASA command line. This allows the convenience of not changing parameters that are shared between tasks but does require care when chaining together sequences of task invocations (to ensure proper values are provided).

Alpha Alert!

The `restore` command has been disabled for now. It will return in a later patch.

If you want to reset the input keywords for a single task, use the `default` command (§ 1.2.2.1). For example, to set the defaults for the `clean` task, type:

```
CASA <12>: default('clean')
```

To inspect a single parameter value just type it at the command line:

```

CASA <16>: alg          # type 'alg' to see the what the algorithm keyword is set to
Out[16]: 'clark'      # CASA tells you it is set to use the Clark algorithm

```

CASA parameters are just Python variables.

Parameters for a given task can be saved by using the `saveinputs` command (see § 1.2.2.5) and restored using the `execfile '<filename>'` command. Note that if the task is successfully executed, then a `<taskname>.last` file is created in the working directory containing the parameter values (see § 1.2.2.6).

We now describe the individual CASA task parameter interface commands and features in more detail.

1.2.2.1 The default Command

Each task has a special set of default parameters defined for its parameters. You can use the `default` command to reset the parameters for a specified task (or the current task as defined by the `taskname` variable) to their default.

For example, suppose we have been running CASA on a particular dataset, e.g.

```
CASA <40>: inp clean
-----> inp('clean')
vis                = 'ngc5921.ms'      # Name of input visibility file
imagename          = 'ngc5921'         # Pre-name of output images
mode               = 'mfs'             # Type of selection (mfs, channel, velocity, frequency)
alg                = 'csclean'         # Algorithm to use (hogbom, clark, csclean, multiscale)
niter              = 1000              # Number of iterations
...
```

and now we wish to switch to a different one. We can reset the parameter values using `default`:

```
CASA <41>: default
-----> default()

CASA <42>: inp
-----> inp()
vis                = ''               # Name of input visibility file
imagename          = ''               # Pre-name of output images
mode               = 'mfs'           # Type of selection (mfs, channel, velocity, frequency)
alg                = 'clark'         # Algorithm to use (hogbom, clark, csclean, multiscale)
niter              = 500             # Number of iterations
...
```

It is good practice to use `default` before running a task if you are unsure what state the CASA global variables are in.

1.2.2.2 The go Command

You can execute a task using the `go` command, either explicitly

```
CASA <44>: go listobs
-----> go(listobs)
Executing: listobs()
...
```

or implicitly if `taskname` is defined (e.g. by previous use of `default` or `inp`)

```
CASA <45>: taskname = 'clean'
CASA <46>: go
-----> go()
Executing: clean()
...
```

You can also execute a task simply by typing the `taskname`.

```
CASA <46>: clean
-----> clean()
Executing: clean()
...
```

1.2.2.3 The `inp` Command

You can set the values for the parameters for tasks (but currently not for tools) by performing the assignment within the CASA shell and then inspecting them using the `inp` command. This command can be invoked in any of three ways: via function call `inp('<taskname>')` or `inp(<taskname>)`, without parentheses `inp '<taskname>'` or `inp <taskname>`, or using the current `taskname` variable setting with `inp`. For example,

```
CASA <1>: inp('clean')
...
CASA <2>: inp 'clean'
-----> inp('clean')
...
CASA <3>: inp(clean)
...
CASA <4>: inp clean
-----> inp(clean)
...
CASA <5>: taskname = 'clean'
CASA <6>: inp
-----> inp()
```

all do the same thing.

When you invoke the task inputs via `inp`, you see a list of the parameters, their current values, and a short description of what that parameters does. For example, starting from the default values,

```

CASA <18>: inp('clean')
vis                =      ''          # Name of input visibility file
imagenam          =      ''          # Pre-name of output images
mode              =      'mfs'        # Type of selection (mfs, channel, velocity, frequency)
alg               =      'clark'      # Algorithm to use (hogbom, clark, csclean, multiscale)
niter             =      500          # Number of iterations
gain              =      0.1          # Loop gain for cleaning
threshold         =      0.0          # Flux level to stop cleaning (mJy)
mask              =      ['']        # Name of mask image used in cleaning
cleanbox          =      []           # clean box regions or file name or 'interactive'
imsize            =      [256, 256]   # Image size in pixels (nx,ny)
cell              =      ['1.0arcsec', '1.0arcsec'] # Cell size in arcseconds (x,y)
stokes            =      'I'          # Stokes parameter to image (I,IV,IQU,IQUV)
field             =      ''          # Field name
spw               =      ''          # Spectral window identifier
weighting         =      'natural'    # Weighting to apply to visibilities
                                     # (natural, uniform, briggs, radial, superuniform)
uvfilter          =      False        # Apply additional filtering/uv tapering of the visibilities
selecttime        =      '1960/01/01/00:00:00~2020/12/31/23:59:59' # range of time to select from
restfreq          =      ''          # restfrequency to use in image
async             =      False        # if True run in the background, prompt is freed

```

Figure 1.2 shows how this will look to you on your terminal. Note that some parameters are in boldface with a gray background. This means that some values for this parameter will cause it to *expand*, revealing new *sub-parameters* to be set.

```

CASA <7>: default clean
> default(clean)

CASA <8>: inp
> inp()
vis                =      ''          # Name of input visibility file
imagenam          =      ''          # Pre-name of output images
mode              =      'mfs'        # Type of selection (mfs, channel, velocity, frequency)
alg               =      'clark'      # Algorithm to use (hogbom, clark, csclean, multiscale)
niter             =      500          # Number of iterations
gain              =      0.1          # Loop gain for cleaning
threshold         =      0.0          # Flux level to stop cleaning (mJy)
mask              =      ['']        # Name of mask image used in cleaning
cleanbox          =      []           # clean box regions or file name
imsize            =      [256, 256]   # Image size in pixels (nx,ny)
cell              =      ''          # Cell size in arcseconds (x,y)
stokes            =      'I'          # Stokes parameter to image (I,IV,IQU,IQUV)
fieldid           =      0           # Field index identifier
field             =      ''          # Field name
spwid             =      0           # Spectral window identifier
weighting        =      'natural'    # Weighting to apply to visibilities
                                     # (natural, uniform, briggs, radial, superuniform)
uvfilter         =      False        # Apply additional filtering/uv tapering of the visibilities
timerange         =      '1960/01/01/00:00:00~2020/12/31/23:59:59' # range of time to select from
ata               =      ''          # restfrequency to use in image
restfreq          =      ''          # restfrequency to use in image

CASA <9>:

```

Figure 1.2: Screen shot of the default CASA inputs for task `clean`.

CASA uses color and font to indicate different properties of parameters and their values:

Parameter and Values in CASA inp

	Text Font	Text Color	Highlight	Indentation	Meaning
Parameters:					
	plain	black	none	none	standard parameter
	bold	black	grey	none	expandable parameter
	plain	green	none	yes	sub-parameter
Values:					
	plain	black	none	none	default value
	plain	blue	none	none	non-default value
	plain	red	none	none	invalid value

Figure 1.3 shows what happens when you set some of the `clean` parameters to non-default values. Some have opened up sub-parameters, which can now be seen and set. Figure 1.4 shows what happens when you set a parameter, in this case `vis` and `mode`, to an invalid value. Its value now appears in red. Reasons for invalidation include incorrect type, an invalid menu choice, or a filename that does not exist. For example, since `vis` expects a filename, it will be invalidated (red) if it is set to a non-string value, or a string that is not the name of a file that can be found. The `mode='happy'` is invalid because its not a supported choice ('mfs', 'channel', 'velocity', or 'frequency').

```

CASA <5>: run clean.last
CASA <6>: inp
         inp()
vis      = 'ngc5921_src.split.ms'      # Name of input visibility file
imagename = 'ngc5921_task'             # Pre-name of output images
mode     = 'channel'                  # Type of selection (mfs, channel, velocity, frequency)
  nchan  = 46                          # Number of channels to select
  start  = 0                            # Start channel
  step   = 1                            # Increment between channels/velocity
  width  = 1                            # Channel width (value > 1 indicates channel averaging)
alg      = 'hogbow'                   # Algorithm to use (hogbow, clark, csclean, multiscale)
niter    = 6000                        # Number of iterations
gain     = 0.1                         # Loop gain for cleaning
threshold = 8.0                        # Flux level to stop cleaning (mJy)
mask     = ''                          # Name of mask image used in cleaning
cleanbox = []                          # clean box regions or file name
imsize   = [256, 256]                 # Image size in pixels (nx,ny)
cell     = [15.0, 15.0]               # Cell size in arcseconds (x,y)
stokes   = 'I'                       # Stokes parameter to image (I,IV,IQU,IQUV)
fieldid  = 0                          # Field index identifier
field    = ''                          # Field name
spwid    = 0                          # Spectral window identifier
weighting = 'briggs'                  # Weighting to apply to visibilities
  rmode  = 'none'                      # (natural, uniform, briggs, radial, superuniform)
  robust = 0.5                         # Robustness mode (for Briggs weighting)
  noise  = '0.0Jy'                    # Briggs robustness parameter
  npixels = 0                          # noise parameter for briggs weighting when rmode='abs'
                                         # number of pixels to determine uv-cell size 0-> field of view
uvfilter = False                      # Apply additional filtering/uv tapering of the visibilities
timerange = '1960/01/01/00:00:00'2020/12/31/23:59:59' # range of time to select from data
ata       = ''                        # restfrequency to use in image
restfreq  = ''
CASA <7>:

```

Figure 1.3: The `clean` inputs where some parameters have been set to non-default values (blue). Note that some of the boldface ones have opened up new dependent sub-parameters (indented green).

```

CASA <7>: vis=2
CASA <8>: inp
> inp()
vis           =          2          # Name of input visibility file
imagename     =          ''         # Pre-name of output images
mode          =      'happy'       # Type of selection (mfs, channel, velocity, frequency)
alg           =      'clark'       # Algorithm to use (hogbow, clark, csclean, multiscale)
niter         =          500       # Number of iterations
gain          =          0.1       # Loop gain for cleaning
threshold     =          0.0       # Flux level to stop cleaning (mJy)
mask          =          ['']      # Name of mask image used in cleaning
clearbox      =          []        # clean box regions or file name
imsize       = [256, 256]         # Image size in pixels (nx,ny)
cell          = ['1.0arcsec', '1.0arcsec'] # Cell size in arcseconds (x,y)
stokes        =          'I'       # Stokes parameter to image (I,IV,IQU,IQV)
Fieldid       =          0         # Field index identifier
Field         =          ''        # Field name
spwid         =          0         # Spectral window identifier
weighting     =      'natural'     # Weighting to apply to visibilities
                                     # (natural, uniform, briggs, radial, superuniform)
uvfilter      =          False     # Apply additional filtering/uv tapering of the visibilities
timerange     = '1960/01/01/00:00:00~2020/12/31/23:59:59' # range of time to select from data
restfreq      =          ''        # restfrequency to use in image
CASA <9>:

```

Figure 1.4: The `clean` inputs where some parameters have been set to invalid values. These are drawn in red to draw attention to the problem.

1.2.2.4 The restore Command

If you want to reset all input keywords for all tasks to the *global default values*, use the `restore` command:

```
CASA <10>: restore
```

Note that the global default values for many parameters are different than the task-specific default values. This is because some parameters have different default values in the different tasks they appear in! Using the `default <taskname>` command is much safer.

Alpha Alert!

In the current version of CASA, the `restore` command has been disabled, as it is still difficult to keep the list of CASA globals stored in different places. When we sort out our parameter handling mechanisms, we will probably bring back `restore`.

1.2.2.5 The saveinputs Command

The `saveinputs` command will save the current values of a given task parameters to Python (plain ascii) file. It can take up to two arguments. The first is the usual `taskname` parameter. The second is the name for the output Python file. If there is no second argument, a file with name `<taskname>.saved` will be created (or overwritten if extant). If invoked with no arguments, it will use the current values of the `taskname` variable.

For example, starting from default values

```
CASA <1>: default listobs
-----> default(listobs)
```



```
CASA <2>: inp
-----> inp()
vis           =      ''           # Name of input visibility file (MS)
verbose       =      False        # Extended summary list of data set in logger
```

Now set and run again

```
CASA <3>: vis='ngc5921.ms'
```

```
CASA <4>: inp
-----> inp()
vis           = 'ngc5921.ms'      # Name of input visibility file (MS)
verbose       =      False        # Extended summary list of data set in logger
```

Now save them, using the default name 'listobs.saved', and then look at the file:

```
CASA <5>: saveinputs
-----> saveinputs()

CASA <6>: !more 'listobs.saved'    # view the listobs.saved file on disk.
IPython system call: more 'listobs.saved'
taskname      = "listobs"
vis           = "ngc5921.ms"
verbose       = False
#listobs(vis="ngc5921.ms",verbose=False)
```

To read these back in, use the Python `execfile` command. For example,

```
CASA <7>: vis='someotherfile.ms'
```

```
CASA <8>: inp
-----> inp()
vis           = 'someotherfile.ms' # Name of input visibility file (MS)
verbose       =      False        # Extended summary list of data set in logger
```

```
CASA <9>: execfile 'listobs.saved'
-----> execfile('listobs.saved')
```

```
CASA <10>: inp
-----> inp()
vis           = 'ngc5921.ms'      # Name of input visibility file (MS)
verbose       =      False        # Extended summary list of data set in logger
```

and we are back.

You can also save to a custom named file:

```
CASA <11>: verbose = True
```

```
CASA <12>: saveinputs 'listobs','ngc5921_listobs.par'
-----> saveinputs('listobs','ngc5921_listobs.par')
```

```
CASA <13>: !more 'ngc5921_listobs.par'
IPython system call: more 'ngc5921_listobs.par'
taskname      = "listobs"
vis           = "ngc5921.ms"
verbose       = True
#listobs(vis="ngc5921.ms",verbose=False)
```

1.2.2.6 The .last file

Whenever you successfully execute a CASA task, a Python script file called `<taskname>.last` will be written (or over-written) into the current working directory. For example, if you ran the `listobs` task as detailed above, then

```
CASA <14>: vis = 'ngc5921.ms'
```

```
CASA <15>: verbose = True
```

```
CASA <16>: listobs()
```

```
CASA <17>: !more 'listobs.last'
IPython system call: more listobs.last
taskname      = "listobs"
vis           = "ngc5921.ms"
verbose       = True
#listobs(vis="ngc5921.ms",verbose=False)
```

You can restore the parameter values from the save file using

```
CASA <18>: execfile('listobs.last')
```

or

```
CASA <19>: run listobs.last
```

Note that the `.last` file is generally not created until the task actually finished (successfully), so it is often best to manually create a save file beforehand using the `saveinputs` command if you are running a critical task that you strongly desire to have the inputs saved for.

1.3 Getting the most out of CASA

There are some other general things you should know about using CASA in order to make things go smoothly during your data reduction.

1.3.1 Your command line history and the logger

Your command line history is automatically maintained and stored in the local directory as `ipython.log`. This file can be edited and re-executed as appropriate using the `execfile '<filename>'` feature.

The output from CASA commands is sent to the file `casapy.log`, also in your local directory.

The output contained in `casapy.log` is also displayed in a separate window using the *logger*. Generally, the logger window will be brought up when `casapy` is started. If you do not want the logger GUI to appear, then start `casapy` using the `--nolog` option,

```
casapy --nolog
```

which will run CASA in the terminal window.

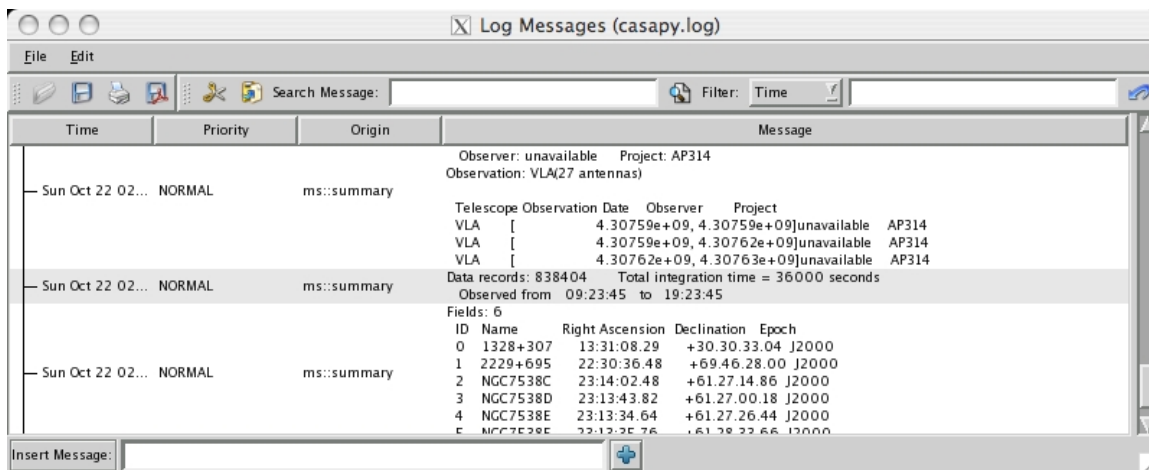


Figure 1.5: CASA Logger GUI window

The CASA logger is shown in Figures 1.5–1.8. The logger has a range of features, which include:

- **Search** — search messages by entering text in the Search window and clicking the search icon. The search currently just matches the exact text you type anywhere in the message. See Figure 1.6 for an example.
- **Filter** — a filter to sort by message priority, time, task/tool of origin, and message contents. Enter text in the Filter window and click the filter icon to the right of the window. Use the pull-down at the left of the Filter window to choose what to filter. The matching is for the exact text currently (no regular expressions). See Figure 1.7 for an example.
- **View** — show and hide columns (Time, Priority, Origin, Message) by checking boxes under the **View** menu pull-down. You can also change the font here.
- **Insert Message** — insert additional comments as “notes” in the log. Enter the text into the “Insert Message” box at the bottom of the logger, and click on the Add (+) button, or choose

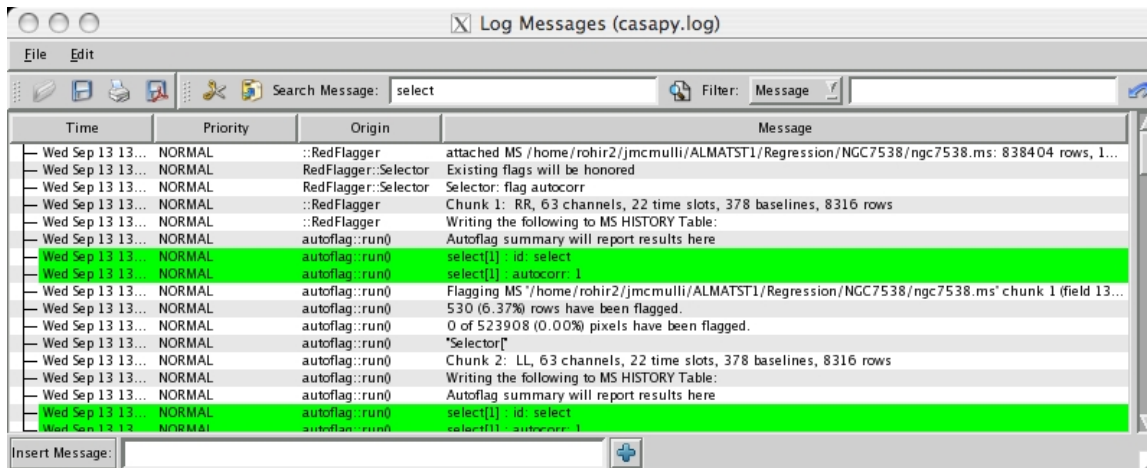


Figure 1.6: CASA Logger - Search example: Specify a string in the entry box to have all instances of the found string highlighted.

to enter a longer message. The entered message will appear with a priority of “NOTE” with the Origin as your username. See Figure 1.8 for an example.

- **Copy** — left-click on a row, or click-drag a range of rows, or click at the start and shift click at the end to select. Use the Copy button or **Edit** menu Copy to put the selected rows into the clipboard. You can then (usually) paste this where you wish. ALPHA ALERT: this does not work routinely in the current version. You are best off going to the `casapy.log` file if you want to grab text.
- **Open** — ALPHA ALERT: there is an Open function in the **File** menu, and an Open button, but these are “grayed-out” in the Alpha. Sorry!

Other operations are also possible from the menu or buttons. Mouse “flyover” will reveal the operation of buttons, for example.

1.3.2 Where are my data in CASA?

Interferometric data are filled into a so-called Measurement Set (or MS). In its logical structure, the MS looks like a generalized description of data from any interferometric or single dish telescope. Physically, the MS consists of several tables in a directory on disk.

Tables in CASA are actually directories containing files that are the sub-tables. For example, when you create a MS called `AM675.ms`, then the name of the directory where all the tables are stored will be called `AM675.ms/`. See Chapter 2 for more information on Measurement Set and Data Handling in CASA.

The data that you originally get from a telescope can be put in any directory that is convenient to you. Once you “fill” the data into a measurement set that can be accessed by CASA, it is generally

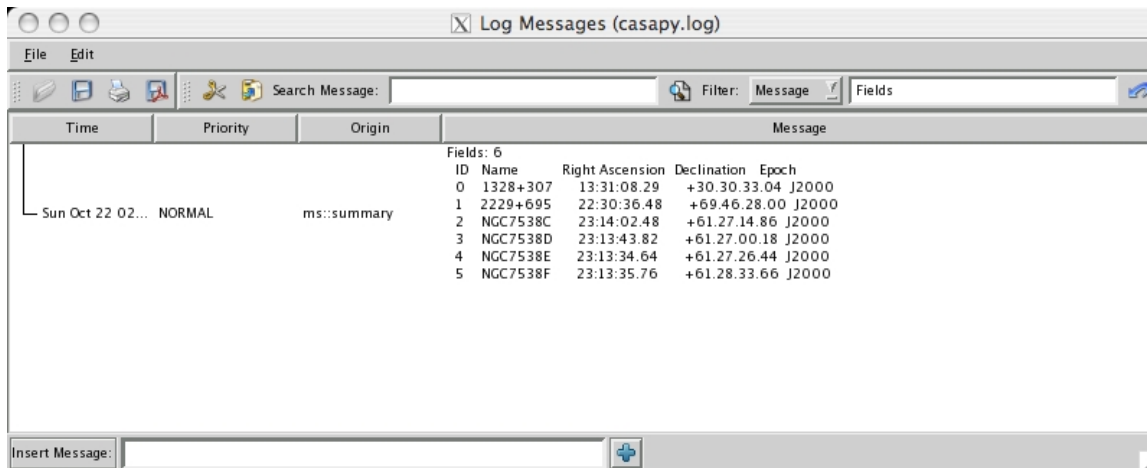


Figure 1.7: CASA Logger - Filter facility: The log output can be sorted by Priority, Time, Origin. One can also filter for a string found in the Message.

best to keep that MS in the same directory where you started CASA so you can get access to it easily (rather than constantly having to specify a full path name).

When you generate calibration solutions or images (again these are in table format), these will also be written to disk. It is a good idea to keep them in the directory in which you started CASA. Note that when you delete a measurement set, calibration table, or image, you must delete the top level directory, and all underlying directories and files, using the file delete method of the operating system you started CASA from. For example, when running CASA on a Linux system, in order to delete the measurement set named `AM675.ms` type:

```
CASA <5>: !rm -r AM675.ms
```

from within CASA. The `!` tells CASA that a system command follows (see § 1.1.5.5), and the `-r` makes sure that all subdirectories are deleted recursively.

It is convenient to prefix all MS, calibration tables, and output files produced in a run with a common string. For example, one might prefix all files from VLA project AM675 with `AM675`, e.g. `AM675.ms`, `AM675.cal`, `AM675.clean`. Then,

```
CASA <6>: !rm -r AM675*
```

will clean up all of these.

1.3.3 What's in my data?

The actual data is in a large MAIN table that is organized in such a way that you can access different parts of the data easily. This table contains a number of “rows”, which are effectively a single timestamp for a single spectral window (like an IF from the VLA) and a single baseline (for an interferometer).

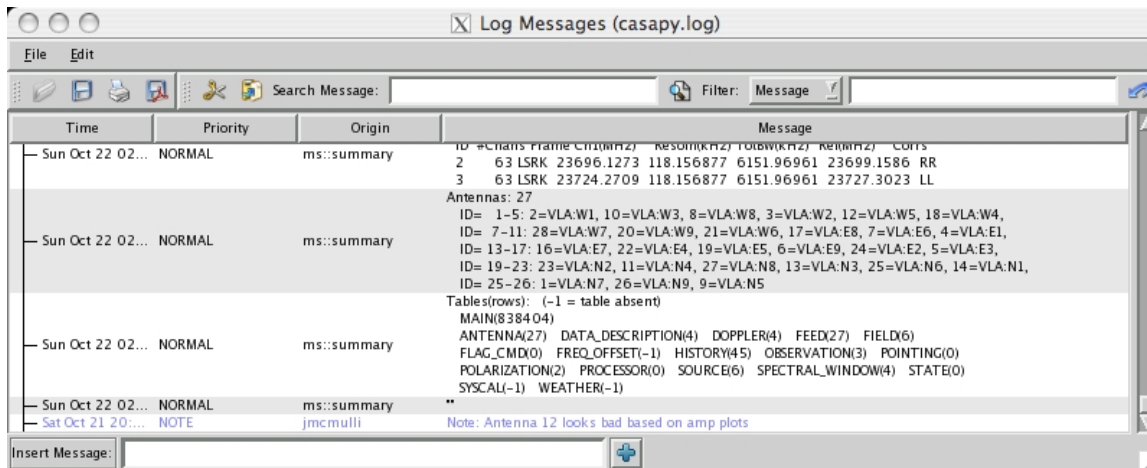


Figure 1.8: CASA Logger - Insert facility: The log output can be augmented by adding notes or comments during the reduction. The file should then be saved to disk to retain these changes.

There are a number of “columns” in the MS, the most important of which for our purposes is the DATA column — this contains the original visibility data from when the MS was created or filled. There are other helpful “scratch” columns which hold useful versions of the data or weights for further processing: the CORRECTED_DATA column, which is used to hold calibrated data; the MODEL_DATA column, which holds the Fourier inversion of a particular model image; and the IMAGING_WEIGHT column which can hold the weights to be used in imaging. The creation and use of the scratch columns is generally done behind the scenes, but you should be aware that they are there (and when they are used). We will occasionally refer to the rows and columns in the MS.

More on the contents of the MS can be found in § 2.1.

1.3.4 Data Selection in CASA

We have tried to make the CASA task interface as uniform as possible. If a given parameter appears in multiple tasks, it should, as far as is possible, mean the same thing and be used in the same way in each. There are groups of parameters that appear in a number of tasks to do the same thing, such as for data selection.

The parameters `field`, `spw`, and `selectdata` (which if `True` expands to a number of sub-parameters) are commonly used in tasks to select data on which to work. These common data selection parameters are described in § 2.5.

1.4 From Loading Data to Images

The subsections below provide a brief overview of the steps you will need to load data into CASA and obtain a final, calibrated image. Each subject is covered in more detail in Chapters 2 through

7.

An end-to-end workflow diagram for CASA data reduction for interferometry data is shown in Figure 1.9. This might help you chart your course through the package. In the following subsections, we will chart a rough course through this process, with the later chapters filling in the individual boxes.

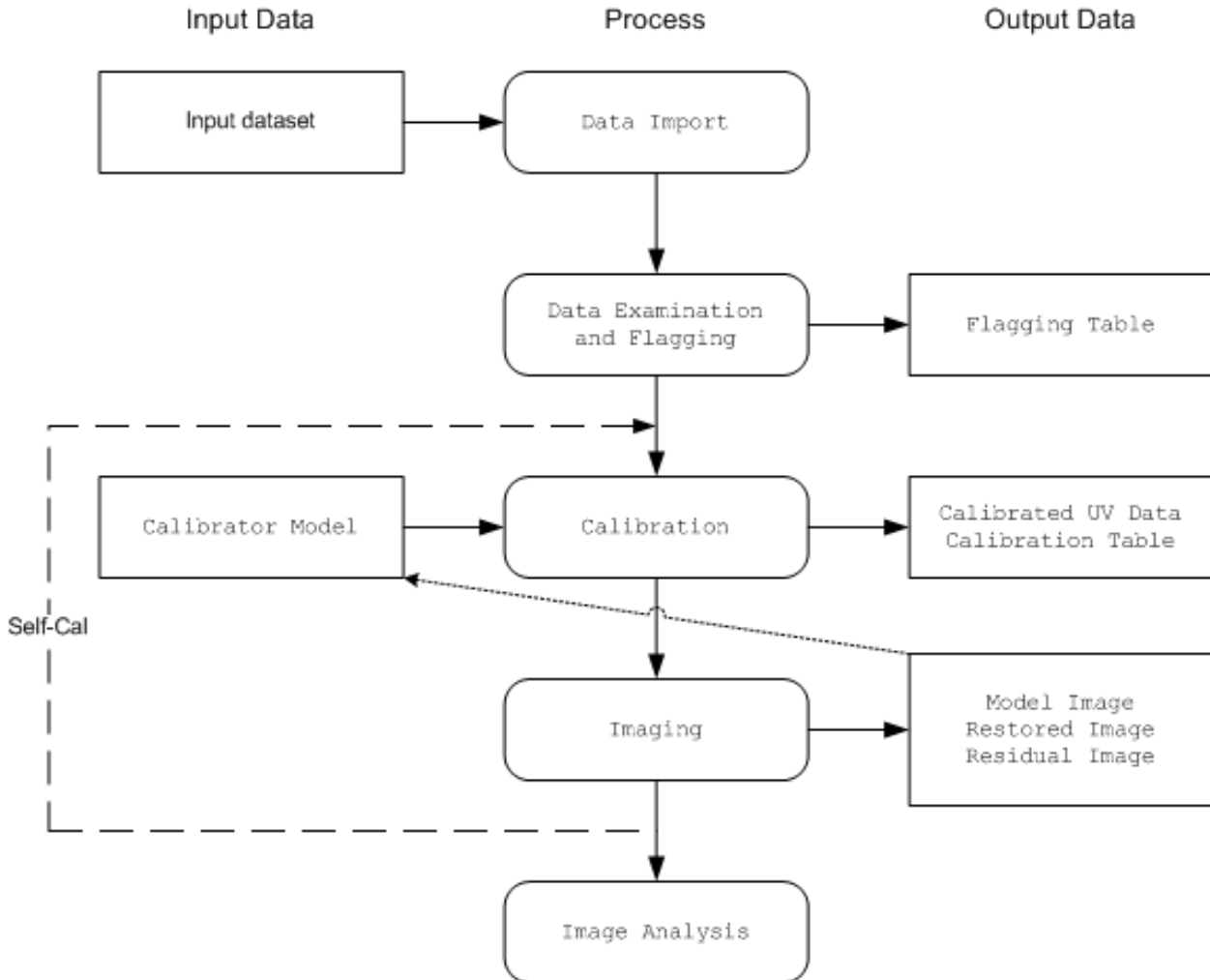


Figure 1.9: Flow chart of the data processing operations that a general user will carry out in an end-to-end CASA reduction session.

Note that single-dish data reduction (for example with the ALMA single-dish system) follows a similar course. This is detailed in Chapter 8.

1.4.1 Loading Data into CASA

The key data and image import tasks are:

- `importuvfits` — import visibility data in UVFITS format (§ 2.2.1)
- `importvla` — import data from VLA that is in *export* format (§ 2.2.2)
- `importasdm` — import data in ALMA ASDM format (§ 2.2.3)
- `importfits` — import a FITS image into a CASA *image* format table (§ 7.5)

These are used to bring in your interferometer data, to be stored as a CASA Measurement set (MS), and any previously made images or models (to be stored as CASA image tables).

The data import tasks will create a MS with a path and name specified by the `vis` parameter. See § 1.3.2 for more information on MS in CASA. The measurement set is the internal data format used by CASA, and conversion from any other native format is necessary for most of the data reduction tasks.

Once data is imported, there are other operations you can use to manipulate the datasets:

- `concat` — concatenate a second MS into a given MS (§ 2.4)

Data import, export, concatenation, and selection detailed in Chapter 2.

1.4.1.1 VLA: Filling data from VLA archive format

VLA data in “archive” format are read into CASA from disk using the `importvla` task (see § 2.2.2). This filler supports the new naming conventions of EVLA antennas when incorporated into the old VLA system.

Note that future data from the EVLA in ASDM format will use a different filler. This will be made available in a later release.

1.4.1.2 Filling data from UVFITS format

For UVFITS format, use the `importuvfits` task. A subset of popular flavors of UVFITS (in particular UVFITS as written by AIPS) is supported by the CASA filler. See § 2.2.1 for details.

1.4.1.3 Loading FITS images

For FITS format images, such as those to be used as calibration models, use the `importfits` task. Most, though not all, types of FITS images written by astronomical software packages can be read in.

See § 7.5 for more information.

1.4.1.4 Concatenation of multiple MS

ONce you have loaded data into measurement sets on disk, you can use the `concat` task to combine them. Currently, `concat` will add a second MS to an existing MS (not producing a new one). This would be run multiple times if you had more than two sets to combine.

See § 2.4 for details.

1.4.2 Data Examination, Editing, and Flagging

The main data examination and flagging tasks are:

- `listobs` — summarize the contents of a MS (§ 2.3)
- `flagmanager` — save and manage versions of the flagging entries in the measurement set (§ 3.2)
- `flagautocorr` — non-interactive flagging of auto-correlations (§ 3.3)
- `plotxy` — interactive X-Y plotting and flagging of visibility data (§ 3.4)
- `flagdata` — non-interactive flagging (and unflagging) of specified data (§ 3.5)
- `viewer` — the CASA viewer can display (as a raster image) MS data, with some editing capabilities (§ 3.6)

These tasks allow you to list, plot, and/or flag data in a CASA MS.

There will eventually be tasks for “automatic” flagging to data based upon statistical criteria. Stay tuned.

Examination and editing of synthesis data is described in Chapter 3.

1.4.2.1 Interactive X-Y Plotting and Flagging

The principal tool for making X-Y plots of visibility data is `plotxy` (see § 3.4). Amplitudes and phases (among other things) can be plotted against several x-axis options.

Interactive flagging (i.e., “see it – flag it”) is possible on the `plotxy` X-Y displays of the data (§ 3.4.3). Since flags are inserted into the measurement set, it is useful to backup (or make a copy) of the current flags before further flagging is done, using `flagmanager` (§ 3.2). Copies of the flag table can also be restored to the MS in this way.

1.4.2.2 Flag the Data Non-interactively

The `flagdata` task (§ 3.5) will flag the visibility data set based on the specified data selections. The `listobs` task (§ 2.3) may be run (e.g. with `verbose=True`) to provide some of the information needed to specify the flagging scope.

1.4.2.3 Viewing and Flagging the MS

The CASA `viewer` can be used to display the data in the MS as a (grayscale or color) raster image. The MS can also be edited. Use of the `viewer` on an MS is detailed in § 3.6.

1.4.3 Calibration

The major calibration tasks are:

- `setjy` — Computes the model visibility for a specified source flux density (§ 4.2)
- `bandpass` — Solves for frequency-dependent (bandpass) complex gains (§ 4.5)
- `gaincal` — Solves for time-dependent complex gains (§ 4.3)
- `fluxscale` — Bootstraps the flux density scale from standard calibrators (§ 4.4)
- `accum` — Accumulates incremental calibration solutions into a cumulative calibration table (§ 4.7.2)
- `smoothcal` — Smooths calibration solutions derived from one or more sources (§ 4.7.1)
- `applycal` — Applies calculated calibration solutions (§ 4.10)
- `clearcal` — Re-initializes calibration data for a given visibility data set (§ 4.11)
- `listcal` — Lists calibration solutions (§ 4.9)
- `plotcal` — Plots (and optionally flags) calibration solutions (§ 4.8)
- `uvcontsub` — carry out uv-plane continuum subtraction for spectral-line data (§ 4.13.1)
- `split` — write out a new (calibrated) MS for specified sources (§ 4.12)

During the course of calibration, the user will specify a set of calibrations to pre-apply before solving for a particular type of effect, for example gain or bandpass or polarization. The solutions are stored in a calibration table (subdirectory) which is specified by the user, *not* by the task: care must be taking in naming the table for future use. The user then has the option, as the calibration process proceeds, to accumulate the current state of calibration in a new cumulative table. Finally, the calibration can be applied to the dataset.

Synthesis data calibration is described in detail in Chapter 4.

1.4.3.1 Setting the flux density scale

The `setjy` task places the Fourier transform of a standard calibration source model in the `MODEL_DATA` column of the measurement set. This can then be used in later calibration tasks. Currently, `setjy` knows the flux density as a function of frequency for several standard VLA flux calibrators, and the value of the flux density can be manually inserted for any other source. If the source is not well-modeled as a point source, then a model image of that source structure can be used (with the total flux density scaled by the values given or calculated above for the flux density). Models are provided for the standard VLA calibrators.

See § 4.2 for more details.

1.4.3.2 Gain Calibration

The `gaincal` task determines solutions for the time-based complex antenna gains, for each spectral window, from the specified calibration sources. A solution interval may be specified. For the VLA, antenna gain curves may be pre-applied before solving for the gains.

A spline fit for the solution (solution type `GSPLINE`) may be carried out instead of the default time-slot based solutions.

See § 4.3 for more on gain calibration.

1.4.3.3 Bandpass Calibration

The `bandpass` task calculates a bandpass calibration solution: that is, it solves for gain variations in frequency as well as in time. Since the bandpass (relative gain as a function of frequency) generally varies much more slowly than the changes in overall (mean) gain solved for by `gaincal`, one generally uses a long time scale when solving for the bandpass.

Bandpass calibration is discussed in detail in § 4.5.

1.4.3.4 Examining Calibration Solutions

The `plotcal` task (§ 4.8) will plot the solutions in a calibration table as a function of time (and channel for `bandpass` calibration). The `plotcal` interface and plotting surface is similar to that in `plotxy`. Eventually, `plotcal` will allow you to flag and unflag calibration solutions in the same way that data can be edited in `plotxy`.

The `listcal` task (§ 4.9) will print out the calibration solutions in a specified table.

1.4.3.5 Bootstrapping Flux Calibration

The `fluxscale` task bootstraps the flux density scale from “primary” standard calibrators to the “secondary” calibration sources. Note that the flux density scale must have been previously

established on the “primary” calibrator(s), typically using `setjy`, and of course a calibration table containing valid solutions for all calibrators must be available.

See § 4.4 for more.

1.4.3.6 Calibration Accumulation

The `accum` task applies an incremental solution table to a previous calibration table, and writes out a cumulative solution table. Different interpolation schemes may be selected.

A description of this process is given in § 4.7.2.

1.4.3.7 Correcting the Data

The final step in the calibration process, `applycal` may be used to apply several calibration tables (e.g., `gaincal`, `bandpass`, `pointing`). The corrections are applied to the `DATA` column of the visibility, writing the `CORRECTED_DATA` column which can then be plotted (e.g. in `plotxy`), `split` out as the `DATA` column of a new MS, or imaged (e.g. using `clean`). Any existing corrected data are overwritten.

See § 4.10 for details.

1.4.3.8 Splitting the Data

After a suitable calibration is achieved, it may be desirable to create one or more new measurement sets containing the data for selected sources. This can be done using the `split` task (§ 4.12).

Further imaging and calibration (e.g. self-calibration) can be carried out on these split MSs.

1.4.4 Synthesis Imaging

The key synthesis imaging tasks are:

- `invert` — Creates a dirty image and dirty beam (point spread function) (§ 5.3)
- `clean` — Calculates a deconvolved image based on the visibility data, using one of several clean algorithms (§ 5.4)
- `mosaic` — Calculates a multi-field deconvolved image based on visibility data, using one of several deconvolution algorithms (§ 5.5)
- `feather` — Combines a single dish and synthesis image in the Fourier plane (§ 5.6)

Most of these tasks are used to take calibrated interferometer data, with the possible addition of a single-dish image, and reconstruct a model image of the sky.

There are several other utility imaging tasks of interest:

- `makemask` — Makes a mask image from a `cleanbox` `blc`, `trc` region (§ 5.7)
- `ft` — Fourier transforms the specified model (or component list) and insert the source model into the `MODEL_DATA` column of the MS (§ 5.8)
- `deconvolve` — Deconvolve an input image from a provided PSF, using one of several image-plane deconvolution algorithms (§ 5.9)

These are not discussed in this walk-through — see the individual Cookbook entries for details.

See Chapter 5 for more on synthesis imaging.

1.4.4.1 Making a “dirty” image

Often, the first step in imaging is to make a simple gridded Fourier inversion of the calibrated data to make a “dirty” image. This can then be examined to look for the presence of noticeable emission above the noise, and to assess the quality of the calibration by searching for artifacts in the image.

The `invert` task is provided for this purpose. See § 5.3 for details.

1.4.4.2 Cleaning a single-field image

The CLEAN algorithm is the most popular and widely-studied method for reconstructing a model image based on interferometer data. It effectively iteratively removes at each step a fraction of the flux in the brightest pixel in a defined region of the current “dirty” image as a point source, placing this in the model image. The `clean` task implements the CLEAN algorithm for single-field data. The user can choose from a number of options for the particular flavor of CLEAN to use.

See § 5.4 for an in-depth discussion of the `clean` task.

1.4.4.3 Cleaning a mosaic

The `mosaic` task generalizes the `clean` task to allow CLEAN deconvolution for a mosaic of observed fields.

See § 5.5 for more on mosaic CLEANing.

1.4.4.4 Feathering in a Single-Dish image

If you have a single-dish image of the large-scale emission in the field, this can be “feathered” in to the image obtained from the interferometer data. This is carried out using the `feather` tasks as the weighted sum in the uv-plane of the gridded transforms of these two images. While not as accurate as a true joint reconstruction of an image from the synthesis and single-dish data together, it is sufficient for most purposes.

See § 5.6 for details on the use of the `feather` task.

1.4.5 Self Calibration

Once a calibrated dataset is obtained, and a first deconvolved model image is computed, a “self-calibration” loop can be performed. Effectively, the model (not restored) image is passed back to another calibration process (on the target data). This refines the calibration of the target source, which up to this point has had (usually) only external calibration applied. This process follows the regular calibration procedure outlined above.

Any number of self-calibration loops can be performed. As long as the images are improving, it is usually prudent to continue the self-calibration iterations.

This process is described in § 5.10.

1.4.6 Data and Image Analysis

The key data and image analysis tasks are:

- **imhead** — summarize and manipulate the “header” information in a CASA image (§ 7.1)
- **immoments** — compute the moments of an image cube (§ 7.2)
- **regridimage** — regrid an image onto the coordinate system of another image (§ 7.3)
- **viewer** — there are useful region statistics and image cube slice and profile capabilities in the viewer (§ 7.4)

1.4.6.1 What’s in an image?

The **imhead** task will print out a summary of whats in an image (the “header” entries). This task can also be used to change the header values.

See § 7.1 for more.

1.4.6.2 Moments of an Image Cube

The **immoments** task will compute a “moments” image of an input image cube. A number of options are available, from the traditional true moments (zero, first, second) and variations thereof, to other images such as median, minimum, or maximum along the moment axis.

See § 7.2 for details.

1.4.6.3 Regridding an Image

It is occasionally necessary to regrid an image onto a new coordinate system. The `regridimage` task can be used to regrid an input image onto the coordinate system of an existing template image, creating a new output image.

See § 7.3 for a description of this task.

1.4.6.4 Displaying Images

To display an image use the `viewer` task. The viewer will display images in raster, contour, or vector form. Blinking and movies are available for spectral-line image cubes. To start the viewer, type:

```
viewer
```

Executing the `viewer` task will bring up two windows: a viewer screen showing the data or image, and a file catalog list. Click on an image or ms from the file catalog list, choose the proper display, and the image should pop up on the screen. Clicking on the wrench tool (second from left on upper left) will obtain the data display options. Most functions are self-documenting.

The viewer can be run outside of casapy by typing `casaviewer`.

See § 7.4 for more on viewing images.

1.4.7 Getting data and images out of CASA

The key data and image export tasks are:

- `exportuvfits` — export a CASA MS in UVFITS format (§ 2.2.1)
- `exportfits` — export a CASA image table as FITS (§ 7.5)

These tasks can be used to export a CASA MS or image to UVFITS or FITS respectively. See the individual sections referred to above for more on each.

Chapter 2

Visibility Data Import, Export, and Selection

To use CASA to process your data, you first will need to get it into a form that is understood by the package. These are “measurement sets” for synthesis (and single dish) data, and “image” tables for images.

There are a number of tasks used to fill telescope-specific data, to import/export standard formats, to list data contents, and to concatenate multiple datasets. These are:

- `importuvfits` — import visibility data in UVFITS format (§ 2.2.1)
- `importvla` — import data from VLA that is in *export* format (§ 2.2.2)
- `importasdm` — import data in ALMA ASDM format (§ 2.2.3)
- `importfits` — import a FITS image into a CASA *image* format table (§ 7.5)
- `exportuvfits` — export a CASA MS in UVFITS format (§ 2.2.1)
- `exportfits` — export a CASA image table as FITS (§ 7.5)
- `listobs` — summarize the contents of a MS (§ 2.3)
- `concat` — concatenate a second MS into a given MS (§ 2.4)

In CASA, there is a standard syntax for selection of data that is employed by multiple tasks. This is described in this chapter (§ 2.5).

2.1 CASA Measurement Sets

Data is handled in CASA via the `table` system. In particular, visibility data are stored in a CASA table known as a Measurement Set (MS). Details of the physical and logical MS structure are given

below, but for our purposes here an MS is just a construct that contains the data. An MS can also store single dish data (essentially a set of auto-correlations of a 1-element interferometer), though there are also data formats more suitable for single-dish spectra (see § 8).

Note that images are handled through special `image` tables, although standard FITS I/O is also supported. Images and `image` data are described in a separate chapter.

Unless your data was previously processed by CASA or software based upon its predecessor `aips++`, you will need to import it into CASA as an MS. Supported formats include some “standard” flavors of UVFITS, the VLA “Export” archive format, and most recently, the ALMA Science Data Model (ASDM) format. These are described below in § 2.2.

Inside the Toolkit:
Measurement sets are handled in the `ms` tool. Import and export methods include `ms.fromfits` and `ms.tofits`.

Once in Measurement Set form, your data can be accessed through various tools and tasks with a common interface. The most important of these is the *data selection interface* (§ 2.5) which allows you to specify the subset of the data on which the tasks and tools will operate.

2.1.1 Under the Hood: Structure of the Measurement Set

It is not necessary that a casual CASA user know the specific details on how the data in the MS is stored and the contents of all the sub-tables. However, we will occasionally refer to specific “columns” of the MS when describing the actions of various tasks, and thus we provide the following synopsis to familiarize the user with the necessary nomenclature.

All CASA data files, including Measurement Sets, are written into the current working directory by default, with each CASA 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 CASA directly, or via full path names.

An MS consists of a `MAIN` table containing the visibility data and associated sub-tables containing auxiliary or secondary information. The various MS tables and sub-tables can be seen by listing the contents of the MS directory itself (e.g. using Unix `ls`), or via the `browsetable` task (§ 3.7). See Fig 2.1 for an example of the contents of a MS directory. Or, from the `casapy` prompt,

```
CASA <1>: ls ngc5921.ms
IPython system call: ls -F ngc5921.ms
ANTENNA          POLARIZATION    table.f1        table.f3_TSM1  table.f8
DATA_DESCRIPTION PROCESSOR        table.f10       table.f4        table.f8_TSM1
FEED             SORTED_TABLE    table.f10_TSM1  table.f5        table.f9
FIELD           SOURCE          table.f11       table.f5_TSM1  table.f9_TSM1
FLAG_CMD        SPECTRAL_WINDOW table.f11_TSM1  table.f6        table.info
HISTORY         STATE          table.f2        table.f6_TSM0  table.lock
OBSERVATION     table.dat      table.f2_TSM1  table.f7
POINTING        table.f0       table.f3        table.f7_TSM1
```

Note that the `MAIN` table information is contained in `table.dat` file. Each of the sub-table sub-directories contain their own `table.dat` and other files, e.g.

```
CASA <2>: ls ngc5921.ms/SOURCE
IPython system call: ls -F ngc5921.ms/SOURCE
table.dat  table.f0  table.f0i  table.info  table.lock
```

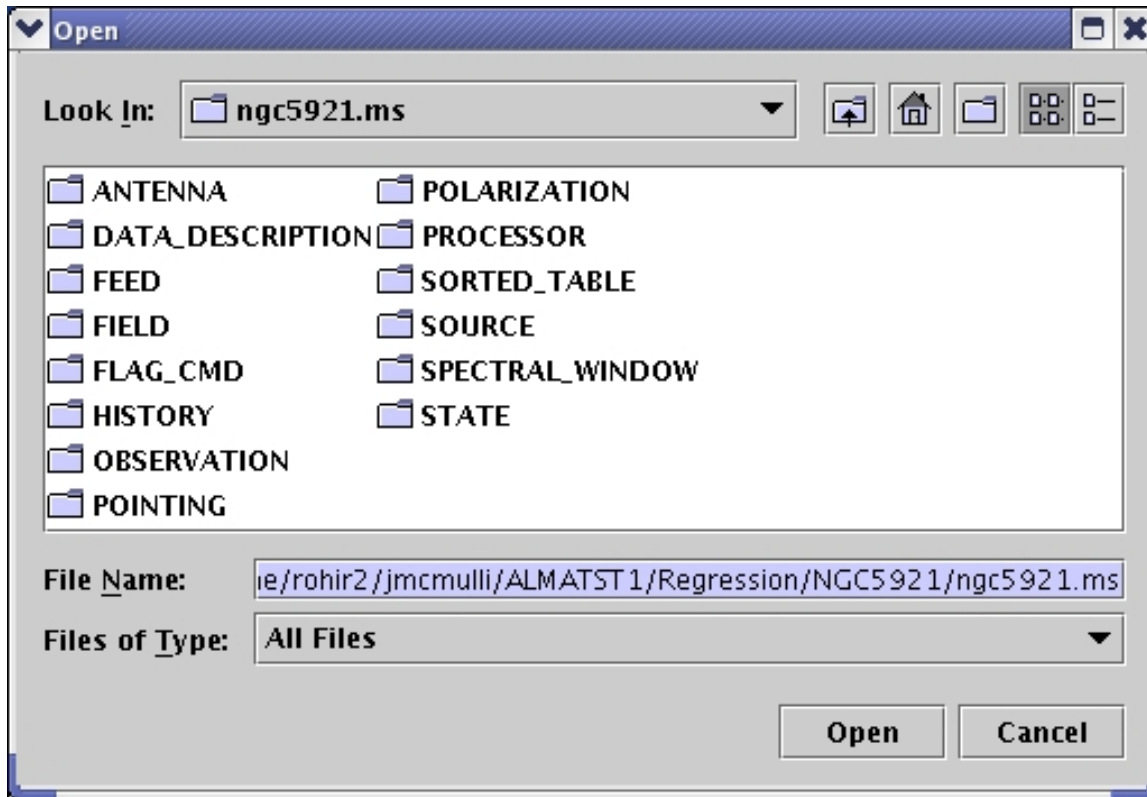


Figure 2.1: The contents of a Measurement Set. These tables compose a Measurement Set named `ngc5921.ms` on disk. This display is obtained by using the **File:Open** menu in `browsetable`.

Each “row” in a table contains entries for a number of specified “columns”. For example, in the `MAIN` table of the MS, the original visibility data is contained in the `DATA` column — each “cell” contains a matrix of observed complex visibilities for that row 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 correlator for an array.

Table 2.1 lists the non-data columns of the `MAIN` table that are most important during a typical data reduction session. Table 2.2 lists the key data columns of the `MAIN` table of an interferometer MS. The MS produced by fillers for specific instruments may insert special columns, such as `ALMA_PHASE_CORR`, `ALMA_NO_PHAS_CORR` and `ALMA_PHAS_CORR_FLAG_ROW` for ALMA data filled using the `importasdm` filler (§ 2.2.3).

Note that when you examine table entries for IDs such as `FIELD_ID` or `DATA_DESC_ID`, you will see 0-based numbers.

Table 2.1: Common columns in the MAIN table of the MS.

Parameter	Contents
ANTENNA1	First antenna in baseline
ANTENNA2	Second antenna in baseline
FIELD_ID	Field (source no.) identification
DATA_DESC_ID	Spectral window number, polarization identifier pair (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

The MS can contain a number of “scratch” columns, which are used to hold hold useful versions of other columns such as the data or weights for further processing. The most common scratch columns are:

- `CORRECTED_DATA` — used to hold calibrated data for imaging or display;
- `MODEL_DATA` — holds the Fourier inversion of a particular model image for calibration or imaging;
- `IMAGING_WEIGHT` —holds the gridding weights to be used in imaging.

The creation and use of the scratch columns is generally done behind the scenes, but you should be aware that they are there (and when they are used).

The most recent specification for the MS is **Aips++ MeasurementSet definition version 2.0** (<http://casa.nrao.edu/Memos/229.html>).

2.2 Data Import and Export

There are a number of tasks available to bring data in various forms into CASA as a Measurement Set:

- UVFITS format can be imported into and exported from CASA (`importuvfits` and `exportuvfits`)
- VLA Archive format data can be imported into CASA (`importvla`)
- ALMA and EVLA Science Data Model format data can be imported into CASA (`importasdm`)

Table 2.2: Commonly accessed MAIN Table data-related columns. Note that the columns ALMA_PHASE_CORR, ALMA_NO_PHAS_CORR and ALMA_PHAS_CORR_FLAG_ROW are specific to ALMA data filled using the `importasdm` filler.

Column	Format	Comments
DATA	Complex(N_c , N_f)	Complex visibility matrix =ALMA_PHASE_CORR by default
FLAG	Bool(N_c , N_f)	cumulative data flags
WEIGHT	Float(N_c)	Weight for a row
WEIGHT_SPECTRUM	Float(N_c , N_f)	Weight for whole data matrix
ALMA_PHASE_CORR	Complex(N_c , N_f)	On-line phase corrected complex visibility matrix (<i>Not in VLA data</i>)
ALMA_NO_PHAS_CORR	Bool(N_c , N_f)	Complex visibility matrix that has not been phase corrected (<i>Not in VLA data</i>)
ALMA_PHAS_CORR_FLAG_ROW	Bool(N_c , N_f)	Flag to use phase-corrected data or not, Default=F (<i>not in VLA data</i>)
MODEL_DATA	Complex(N_c , N_f)	Scratch: created by calibrator or imager tools
CORRECTED_DATA	Complex(N_c , N_f)	Scratch: created by calibrator or imager tools
IMAGING_WEIGHT	Float(N_c)	Scratch: created by calibrator or imager tools

2.2.1 UVFITS Import and Export

To import UVFITS format data into CASA, use the `importuvfits` task:

```
CASA <1>: inp(importuvfits)
fitsfile      =      '' # Name of input UVFITS file
vis           =      '' # Name of output visibility file (MS)
async        =      False # if True run in the background, prompt is freed
```

This is straightforward, since all it does is read in a UVFITS file and convert it as best it can into a MS.

The `exportuvfits` task will take a MS and write it out in UVFITS format.

```

CASA <2>: inp(exportuvfits)
vis                =      '' # Name of input visibility file
fitsfile          =      '' # Name of output UVFITS file)
datacolumn       = 'corrected' # which data to write (data, corrected, model)
fieldid          =      -1 # Field index identifier)
field            =      '' # Field name list
spwid           =      -1 # Spectral window identifier
nchan           =      -1 # Number of channels to select
start           =      0 # Start channel
width           =      1 # Channel averaging width (value>1 indicates averaging)
writesyscal     =      False # Write GC and TY tables
multisource     =      True # Write in multi-source format
combinespw      =      True # Combine spectral windows (True for AIPS)
writestation    =      False # Write station name instead of antenna name
async          =      False # if True run in the background, prompt is freed

```

ALPHA ALERT: This will be upgraded to use the common data selection parameters (§2.5).

2.2.2 VLA: Filling data from archive format (importvla)

VLA data in archive format (i.e., as downloaded from the VLA data archive) are read into CASA from disk using the `importvla` task. The inputs are:

```

archivefiles      =      '' # Name of input VLA archive file(s)
vis              =      '' # Name of output visibility file
bandname         =      '' # VLA frequency band name: '' => obtain all bands in archive fi
frequencytol     = 150000.0 # Tolerance in frequency shift to define
                    a unique spectral window (Hz).
project          =      '' # Project name: (') => all projects in file
starttime       = '1970/1/31/00:00:00' # start time to search for data
stoptime        = '2199/1/31/23:59:59' # end time to search for data
autocorr        =      False # import autocorrelations to ms, if set to True
antnamescheme   =      'new' # 'old' or 'new'; if 'new', antenna names are
                    'VA04' for VLA antenna 4, 'EA13' for EVLA antenna 13
async           =      False # if True run in the background, prompt is freed

```

The parameters are:

Import VLA archive file(s) to a measurement set:

Imports an arbitrary number of VLA archive-format data sets into a casa measurement set. If more than one band is present, they will be put in the same measurement set but in a separate spectral window.

Keyword arguments:

```

archivefiles -- Name of input VLA archive file(s)
              default: none. Must be supplied
              example: archivefiles=['AP314_A950519.xp1', 'AP314_A950519.xp2']

```

```

vis -- Name of output visibility file
      default: none. Must be supplied.
      example: vis='NGC7538.ms'
      Will not over-write existing ms of same name.
bandname -- VLA Frequency band
      default: '' = all bands
      example: bandname='K'
      Options: '4'=48-96 MHz, 'P'=298-345 MHz, 'L'=1.15-1.75 GHz,
      'C'=4.2-5.1 GHz, 'X'=6.8-9.6 GHz, 'U'=13.5-16.3 GHz,
      'K'=20.8-25.8 GHz, 'Q'=38-51 GHz
frequencytol -- Tolerance in frequency shift in making spectral windows
      default: 150000 (Hz). For Doppler shifted data, <10000 Hz may
      may produce too many unnecessary spectral windows.
      example: frequencytol = 1500000.0 (units = Hz)
project -- Project name to import from archive files:
      default: '' => all projects in file
      example: project='AL519'
      project = 'al519' will work, but
      project = 'AL0519' will not.
starttime -- Time after which data will be considered for importing
      default: '1970/1/31/00:00:00'
stoptime -- Time before which data will be considered for importing
      default: '2199/1/31/23:59:59'
autocorr -- import autocorrelations to ms
      default = False (no autocorrelations)
antnamescheme -- 'old' or 'new' antenna names.
      default = 'new' gives antnenna names
      'VA04' for VLA telescope 4 or
      'EA13' for EVLA telescope 13.
      'old' gives names '4' and '13'
asynch -- Run asynchronously
      default = false; do not run asynchronously

```

Note that autocorrelations are filled into the data set if `autocorr=True`. Generally for the VLA, autocorrelation data is not useful, and furthermore the imaging routine will try to image the autocorrelation data (it assumes it is single dish data) which will swamp any real signal. Thus, if you do fill the autocorrelations, you will have to flag them before imaging.

The *importvla* task allows selection on frequency band. Suppose that you have 1.3 cm line observations in K-band and you have copied the archive data files `AP314_A95019.xp*` to your working directory and started `casapy`. Then,

```

importvla(
    archivefiles=['AP314_A950519.xp1', 'AP314_A950519.xp2', 'AP314_A950519.xp3'],
    vis='ngc7538.ms',
    bandname='K',
    frequencytol=10e6)

```

If the data is located in a different directory on disk, then use the full path name to specify each archive file, e.g.:

```
archivefiles=['/home/rohir2/jmcmulli/ALMATST1/Data/N7538/AP314_A950519.xp1',\
             '/home/rohir2/jmcmulli/ALMATST1/Data/N7538/AP314_A950519.xp2',\
             '/home/rohir2/jmcmulli/ALMATST1/Data/N7538/AP314_A950519.xp3']
```

The `antnamescheme` parameter controls whether `importvla` will try to use a naming scheme where EVLA antennas are prefixed with EA (e.g. 'EA16') and old VLA antennas have names prefixed with VA (e.g. 'VA11'). Our method to detect whether an antenna is EVLA is not yet perfected, and thus unless you require this feature, simply use `antnamescheme='old'`.

2.2.3 ALMA: Filling ALMA Science Data Model (ASDM) observations

The `importasdm` task will fill an ASDM into a CASA visibility data set (MS).

ALPHA ALERT: Note that ASDM data are not available at this time. Soon they will be obtained at the ALMA Test Facility (ATF); right now, some simulated data exist. Thus, this filler is in a development stage.

Currently there are no options for filling the data (you get the whole data set!). For example:

```
CASA <1>: importasdm '/home/basho3/jmcmulli/ASDM/ExecBlock3'
-----> importasdm('/home/basho3/jmcmulli/ASDM/ExecBlock3')
```

```
Parameter: asdm is: /home/basho3/jmcmulli/ASDM/ExecBlock3 and has type <type 'str'>.
Taking the dataset /home/basho3/jmcmulli/ASDM/ExecBlock3 as input.
Time spent parsing the XML medata :1.16 s.
The measurement set will be filled with complex data
About to create a new measurement set '/home/basho3/jmcmulli/ASDM/ExecBlock3.ms'
The dataset has 4 antennas...successfully copied them into the measurement set.
The dataset has 33 spectral windows...successfully copied them into the measurement set.
The dataset has 4 polarizations...successfully copied them into the measurement set.
The dataset has 41 data descriptions...successfully copied them into the measurement set.
The dataset has 125 feeds...successfully copied them into the measurement set.
The dataset has 2 fields...successfully copied them into the measurement set.
The dataset has 0 flags...
The dataset has 0 historys...
The dataset has 1 execBlock(s)...successfully copied them into the measurement set.
The dataset has 12 pointings...successfully copied them into the measurement set.
The dataset has 3 processors...successfully copied them into the measurement set.
The dataset has 72 sources...successfully copied them into the measurement set.
The dataset has 3 states...
The dataset has 132 calDevices...
The dataset has 72 mains...
Processing row # 0 in MainTable
Entree ds getDataCols
About to clear
About to getData
About to new VMSData
Exit from getDataCols
ASDM Main table row #0 transformed into 40 MS Main table rows
```

```

Processing row # 1 in MainTable
Entree ds getDataCols
About to clear
About to getData
About to new VMSData
Exit from getDataCols
ASDM Main table row #1 transformed into 40 MS Main table rows
...
ASDM Main table row #71 transformed into 40 MS Main table rows

...successfully copied them into the measurement set.
About to flush and close the measurement set.
Overall time spent in ASDM methods to read/process the ASDM Main table : cpu = 5.31 s.
Overall time spent in AIPS methods to fill the MS Main table : cpu = 1.3

```

2.3 Summarizing your MS (listobs)

Once you import your data into a CASA Measurement Set, you can get a summary of the MS contents with the `listobs` task.

The inputs are:

```

vis           =      ''           # Name of input visibility file (MS)
verbose       =      True         # Extended summary list of data set in logger

```

The summary will be written to the logger and to the `casapy.log` file. For example, using `verbose=False`:

```
listobs('n5921.ms',False)
```

results in the logger messages:

```

Thu Jul 5 17:20:55 2007    NORMAL ms::summary:

      MeasurementSet Name:  /home/scamper/CASA/N5921/n5921.ms      MS Version 2

      Observer: TEST      Project:
      Observation: VLA(28 antennas)

Thu Jul 5 17:20:55 2007    NORMAL ms::summary:
Data records: 22653      Total integration time = 5280 seconds
      Observed from 09:19:00 to 10:47:00

Thu Jul 5 17:20:55 2007    NORMAL ms::summary:
Fields: 3
ID   Name                Right Ascension  Declination     Epoch
0   1331+30500002_013:31:08.29  +30.30.32.96  J2000
1   1445+09900002_014:45:16.47  +09.58.36.07  J2000

```



```
2 N5921_2 15:22:00.00 +05.04.00.00 J2000
```

```
Thu Jul 5 17:20:55 2007 NORMAL ms::summary:
Spectral Windows: (1 unique spectral windows and 1 unique polarization setups)
  SpwID #Chans Frame Ch1(MHz) Resoln(kHz) TotBW(kHz) Ref(MHz) Corrs
  0      63 LSRK 1412.68608 24.4140625 1550.19688 1413.44902 RR LL
```

```
Thu Jul 5 17:20:55 2007 NORMAL ms::summary:
Antennas: 27
  ID= 0-3: '1'='VLA:N7', '2'='VLA:W1', '3'='VLA:W2', '4'='VLA:E1',
  ID= 4-7: '5'='VLA:E3', '6'='VLA:E9', '7'='VLA:E6', '8'='VLA:W8',
  ID= 8-11: '9'='VLA:N5', '10'='VLA:W3', '11'='VLA:N4', '12'='VLA:W5',
  ID= 12-15: '13'='VLA:N3', '14'='VLA:N1', '15'='VLA:N2', '16'='VLA:E7',
  ID= 16-19: '17'='VLA:E8', '18'='VLA:W4', '19'='VLA:E5', '20'='VLA:W9',
  ID= 20-24: '21'='VLA:W6', '22'='VLA:E4', '24'='VLA:E2', '25'='VLA:N6',
  ID= 25-26: '26'='VLA:N9', '27'='VLA:N8'
```

```
Thu Jul 5 17:20:55 2007 NORMAL ms::summary:
```

```
Tables(rows): (-1 = table absent)
  MAIN(22653)
  ANTENNA(28) DATA_DESCRIPTION(1) DOPPLER(-1) FEED(28) FIELD(3)
  FLAG_CMD(0) FREQ_OFFSET(-1) HISTORY(310) OBSERVATION(1) POINTING(168)
  POLARIZATION(1) PROCESSOR(0) SOURCE(3) SPECTRAL_WINDOW(1) STATE(0)
  SYSCAL(-1) WEATHER(-1)
```

```
Thu Jul 5 17:20:55 2007 NORMAL ms::summary ""
```

```
Thu Jul 5 17:20:55 2007 NORMAL ms::close:
Readonly measurement set: just detaching from file.
```

If you choose the (default) `verbose=True` option, there will be more information. For example,

```
listobs('n5921.ms',True)
```

will result in the logger messages:

```
Thu Jul 5 17:23:55 2007 NORMAL ms::summary:
      MeasurementSet Name: /home/scamper/CASA/N5921/n5921.ms MS Version 2
Observer: TEST      Project:
Observation: VLA
Thu Jul 5 17:23:55 2007 NORMAL ms::summary:
Data records: 22653      Total integration time = 5280 seconds
Observed from 09:19:00 to 10:47:00
Thu Jul 5 17:23:55 2007 NORMAL ms::summary:
```

```

ObservationID = 0      ArrayID = 0
Date      Timerange      Scan  FldId  FieldName      SpwIds
13-Apr-1995/09:19:00.0 - 09:24:30.0    1     0 1331+30500002_0 [0]
           09:27:30.0 - 09:29:30.0    2     1 1445+09900002_0 [0]
           09:33:00.0 - 09:48:00.0    3     2 N5921_2         [0]
           09:50:30.0 - 09:51:00.0    4     1 1445+09900002_0 [0]
           10:22:00.0 - 10:23:00.0    5     1 1445+09900002_0 [0]
           10:26:00.0 - 10:43:00.0    6     2 N5921_2         [0]
           10:45:30.0 - 10:47:00.0    7     1 1445+09900002_0 [0]

```

Thu Jul 5 17:23:55 2007 NORMAL ms::summary:

Fields: 3

ID	Name	Right Ascension	Declination	Epoch
0	1331+30500002_0	13:31:08.29	+30.30.32.96	J2000
1	1445+09900002_0	14:45:16.47	+09.58.36.07	J2000
2	N5921_2	15:22:00.00	+05.04.00.00	J2000

Thu Jul 5 17:23:55 2007 NORMAL ms::summary:

Spectral Windows: (1 unique spectral windows and 1 unique polarization setups)

SpwID	#Chans	Frame	Ch1(MHz)	Resoln(kHz)	TotBW(kHz)	Ref(MHz)	Corrs
0	63	LSRK	1412.68608	24.4140625	1550.19688	1413.44902	RR LL

Thu Jul 5 17:23:55 2007 NORMAL ms::summary:

Feeds: 28: printing first row only

Antenna	Spectral Window	# Receptors	Polarizations
1	-1	2	[R, L]

Thu Jul 5 17:23:55 2007 NORMAL ms::summary:

Antennas: 27:

ID	Name	Station	Diam.	Long.	Lat.
0	1	VLA:N7	25.0 m	-107.37.07.2	+33.54.12.9
1	2	VLA:W1	25.0 m	-107.37.05.9	+33.54.00.5
2	3	VLA:W2	25.0 m	-107.37.07.4	+33.54.00.9
3	4	VLA:E1	25.0 m	-107.37.05.7	+33.53.59.2
4	5	VLA:E3	25.0 m	-107.37.02.8	+33.54.00.5
5	6	VLA:E9	25.0 m	-107.36.45.1	+33.53.53.6
6	7	VLA:E6	25.0 m	-107.36.55.6	+33.53.57.7
7	8	VLA:W8	25.0 m	-107.37.21.6	+33.53.53.0
8	9	VLA:N5	25.0 m	-107.37.06.7	+33.54.08.0
9	10	VLA:W3	25.0 m	-107.37.08.9	+33.54.00.1
10	11	VLA:N4	25.0 m	-107.37.06.5	+33.54.06.1
11	12	VLA:W5	25.0 m	-107.37.13.0	+33.53.57.8
12	13	VLA:N3	25.0 m	-107.37.06.3	+33.54.04.8
13	14	VLA:N1	25.0 m	-107.37.06.0	+33.54.01.8
14	15	VLA:N2	25.0 m	-107.37.06.2	+33.54.03.5
15	16	VLA:E7	25.0 m	-107.36.52.4	+33.53.56.5
16	17	VLA:E8	25.0 m	-107.36.48.9	+33.53.55.1
17	18	VLA:W4	25.0 m	-107.37.10.8	+33.53.59.1
18	19	VLA:E5	25.0 m	-107.36.58.4	+33.53.58.8
19	20	VLA:W9	25.0 m	-107.37.25.1	+33.53.51.0

20	21	VLA:W6	25.0 m	-107.37.15.6	+33.53.56.4
21	22	VLA:E4	25.0 m	-107.37.00.8	+33.53.59.7
23	24	VLA:E2	25.0 m	-107.37.04.4	+33.54.01.1
24	25	VLA:N6	25.0 m	-107.37.06.9	+33.54.10.3
25	26	VLA:N9	25.0 m	-107.37.07.8	+33.54.19.0
26	27	VLA:N8	25.0 m	-107.37.07.5	+33.54.15.8
27	28	VLA:W7	25.0 m	-107.37.18.4	+33.53.54.8

Thu Jul 5 17:23:55 2007 NORMAL ms::summary:

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	310 rows
OBSERVATION	1 row
POINTING	168 rows
POLARIZATION	1 row
PROCESSOR	<empty>
SOURCE	3 rows
SPECTRAL_WINDOW	1 row
STATE	<empty>
SYSCAL	<absent>
WEATHER	<absent>

Thu Jul 5 17:23:55 2007 NORMAL ms::summary ""

Thu Jul 5 17:23:55 2007 NORMAL ms::close:
Readonly measurement set: just detaching from file.

The most useful extra information that `verbose=True` gives is the list of the scans in the dataset.

2.4 Concatenating multiple datasets (concat)

Once you have your data in the form of CASA Measurement Sets, you can go ahead and process your data using the editing, calibration, and imaging tasks. In some cases, you will most efficiently operate on single MS for a particular session (such as calibration). Other tasks will (eventually) take multiple MS as input. For others, it is easiest to combine your multiple data files into one.

If you need to combine multiple datasets, you can use the `concat` task. The default inputs are:

```
# concat :: Concatenate two visibility data sets:
```

```

vis      =      '' # Name of input visibility file
concatvis =      '' # Name of visibility file to append to input
freqtol  =      '' # Frequency shift tolerance for combining same spectral window
dirtol   =      '' # Pointing direction tolerance for combining the same field
async    =      False # if True run in the background, prompt is freed

```

This currently will add the second MS (given by `concatvis`) into an existing MS (given by `vis`). The parameters `freqtol` and `dirtol` control how close together in frequency and angle on the sky spectral windows or field locations need to be before calling them the same.

2.5 Data Selection

Once in MS form, subsets of the data can be operated on using the tasks and tools. In CASA, there are three common data selection parameters used in the various tasks: `field`, `spw`, and `selectdata`. In addition, the `selectdata` parameter, if set to `True`, will open up a number of other sub-parameters for selection. The selection operation is unified across all the tasks. The available `selectdata` parameters may not be the same in all tasks. But if present, the same parameters mean the same thing and behave in the same manner when used in any task.

Alpha Alert!

Data selection is being changed over to this new unified system. In various tasks, you may find relics of the old way, such as `fieldid` or `spwid`.

For example:

```

field      =      '' # field names or index of calibrators ''=>all
spw        =      '' # spectral window:channels: ''=>all
selectdata =      False # Other data selection parameters

```

versus

```

field      =      '' # field names or index of calibrators ''=>all
spw        =      '' # spectral window:channels: ''=>all
selectdata =      True # Other data selection parameters
  timerange =      '' # time range: ''=>all
  uvrange   =      '' # uv range''=all
  antenna   =      '' # antenna/baselines: ''=>all
  scan      =      '' # scan numbers: Not yet implemented
  msselect  =      '' # Optional data selection (Specialized. but see help)

```

The following are the general syntax rules and descriptions of the individual selection parameters of particular interest for the tasks:

2.5.1 General selection syntax

Most of the selections are effected through the use of selection strings. This sub-section describes the general rules used in constructing and parsing these strings. Note that some selections are done through the use of numbers or lists. There are also parameter-specific rules that are described under each parameter.

All lists of basic selection specification-units are comma separated lists and can be of any length. White-spaces before and after the commas (e.g. '3C286, 3C48, 3C84') are ignored, while white-space within sub-strings is treated as part of the sub-string (e.g. '3C286, VIRGO A, 3C84').

All integers can be of any length (in terms of characters) composed of the characters 0–9. Floating point numbers can be in the standard format (DIGIT.DIGIT, DIGIT., or .DIGIT) or in the mantissa-exponent format (e.g. 1.4e9). Places where only integers make sense (e.g. IDs), if a floating point number is given, only the integer part is used (it is truncated).

Range of numbers (integers or real numbers) can be given in the format 'N0~N1'. For integer ranges, it is expanded into a list of integers starting from N0 (inclusive) to N1 (inclusive). For real numbers, it is used to select all values present for the appropriate parameter in the Measurement Set between N0 and N1 (including the boundaries). Note that the '~' character is used rather than the more obvious '-' in order to accomodate hyphens in strings and minus signs in numbers.

Wherever appropriate, units can be specified. The units are used to convert the values given to the units used in the Measurement Set. For ranges, the unit is specified only once (at the end) and applies to both the range boundaries.

2.5.1.1 String Matching

String matching can be done in three ways. Any component of a comma separated list that cannot be parsed as a number, a number range, or a physical quantity is treated as a regular expression or a literal string. If the string does not contain the characters '*', '{', '}' or '?', it is treated as a literal string and used for exact matching. If any of the above mentioned characters are part of the string, they are used as a regular expression. As a result, for most cases, the user does not need to supply any special delimiters for literal strings and/or regular expressions. For example:

```
field = '3'      # match field ID 3 and not select field named "3C286".

field = '3*'    # used as a pattern and matched against field names. If
                # names like "3C84", "3C286", "3020+2207" are found,
                # all will match. Field ID 3 will not be selected
                # (unless of course one of the above mentioned field
                # names also correspond to field ID 3!).

field = '30*'   # will match only with "3020+2207" in above set.
```

However if it is required that the string be matched exclusively as a regular expression, it can be supplied within a pair of '/' as delimiters (e.g. '/.+BAND.+/'). A string enclosed within double quotes ('"') is used exclusively for pattern matching (patterns are a simplified form of regular

expressions - used in most UNIX commands for string matching). Patterns are internally converted to equivalent regular expressions before matching. See the Unix command "info regex", or visit <http://www.regular-expressions.info>, for details of regular expressions and patterns.

Strings can include any character except the following:

```
',' ';' '"' '/' NEWLINE
```

(since these are part of the selection syntax). Strings that do not contain any of the characters used to construct regular expressions or patterns are used for exact matches. Although it is highly discouraged to have name in the MS containing the above mentioned reserved characters, if one *does* choose to include the reserved characters as parts of names etc., those names can only be matched against quoted strings (since regular expression and patterns are a super-set of literal strings – i.e., a literal string is also a valid regular expression).

This leaves '""', '*"', '{', '}' or '?' as the list of printable character that cannot be part of a name (i.e., a name containing this character can never be matched in a MSSelection expression). These will be treated as pattern-matching even inside double double quotes ('" "')). There is currently no escape mechanism (e.g. via a backslash).

Some examples of strings, regular expressions, and patterns:

- The string 'LBAND' will be used as a literal string for exact match. It will match only the exact string LBAND.
- The wildcarded string '*BAND*' will be used as a string pattern for matching. This will match any string which has the sub-string BAND in it.
- The string '"*BAND*"' will also be used as a string pattern, matching any string which has the sub-string BAND in it.
- The string '/.+BAND.+/' will be used as a regular expression. This will also match any string which has the sub-string BAND in it. (the .+ regex operator has the same meaning as the * wildcard operator of patterns).

2.5.2 The field Parameter

The `field` parameter is a string that specifies which field names or ids will be processed in the task or tool. The field selection expression consists of comma separated list of field specifications inside the string.

Field specifications can be literal field names, regular expressions or patterns (see § 2.5.1.1). Those fields for which the entry in the NAME column of the FIELD MS sub-table match the literal field name/regular expression/pattern are selected. If a field name/regular expression/pattern fails to match any field name, the given name/regular expression/pattern are matched against the field code. If still no field is selected, an exception is thrown.

Field specifications can also be give by their integer IDs. IDs can be a single or a range of IDs. Field ID selection can also be done as a boolean expression. For a field specification of the form '>ID', all field IDs greater than ID are selected. Similarly for '<ID' all field IDs less than the ID are selected.

For example, if the MS has the following observations:

MS summary:

```
=====
```

FIELDID	SPWID	NChan	Pol	NRows	Source Name
0	0	127	RR	10260	0530+135
1	0	127	RR	779139	05582+16320
2	0	127	RR	296190	05309+13319
3	0	127	RR	58266	0319+415
4	0	127	RR	32994	1331+305
5	1	1	RR,RL,LL,RR	23166	KTIP

one might select

```
field = '0~2,KTIP'      # FIELDID 0,1,2 and field name KTIP
field = '0530+135'     # field 0530+135
field = '05*'          # fields 0530+135,05582+16320,05309+13319
```

2.5.3 The spw Parameter

The `spw` parameter is a string that indicates the specific spectral windows and the channels within them to be used in subsequent processing. Spectral window selection ('`SPWSEL`') can be given as a spectral window integer ID, a list of integer IDs, a spectral window name specified as a literal string (for exact match) or a regular expression or pattern. A range of frequencies are used to select all spectral windows which are within the given range. Frequencies can be specified with an optional unit — the default unit being Hz.

The `spw` can also be selected via comparison for integer IDs. For example, '>ID' will select all spectral windows with ID greater than the specified value, while '<ID' will select those with ID lesser than the specified value.

Spectral window selection using strings follows the standard rules:

```
spw = '1'              # SPWID 1
spw = '1,3,5'          # SPWID 1,3,5
spw = '0~3'            # SPWID 0,1,2,3
spw = '0~3,5'          # SPWID 0,1,2,3 and 5
spw = '<4,5'           # SPWID 0,1,2,3 and 5
spw = '*'              # All spectral windows
```

In some cases, the spectral windows may allow specification by name. For example,

```
spw = '3mmUSB, 3mmLSB'      # choose by names (if available)
```

might be meaningful for the dataset in question.

Note that the order in which multiple `spw` are given may be important for other parameters. For example, the `mode = 'channel'` in `clean` uses the first `spw` as the origin for the channelization of the resulting image cube.

2.5.3.1 Channel selection in the `spw` parameter

Channel selection can be included in the `spw` string in the form `'SPWSEL:CHANSEL'` where `CHANSEL` is the channel selector. In the end, the spectral selection within a given spectral window comes down to the selection of specific channels. We provide a number of shorthand selection options for this. These `CHANSEL` options include:

Alpha Alert!

Not all options are available yet, such as percentages or velocities. Stay tuned!

- *Channel ranges*: `'START~STOP'`
- *Frequency ranges*: `'FSTART~FSTOP'`
- *Velocity ranges*: `'VSTART~VSTOP'` (**not yet available**)
- *Bandwidth percentages*: `'PSTART~PSTOP'` or `'PWIDTH'` (**not yet available**)
- *Channel striding/stepping*: `'START~STOP^STEP'` or `'START^STEP'` or `'^STEP'`

The most common selection is via specifying channel, frequency or velocity ranges `'START~STOP'`:

```
spw = '2:16~40'           # spw 2, channels 16-40, inclusive
spw = '2:5134~5138MHz'   # spw 2, 5134-5138MHz section only
spw = '2:51~76km/s'      # spw 2, 51-76km/s section only
```

All ranges are inclusive, with the channel given by, or containing the frequency or velocity given by, `START` and `STOP` plus all channels between included in the selection.

You can also specify multiple spectral window or channel ranges, e.g.

```
spw = '2:16, 3:32~34'     # spw 2, channel 16 plus spw 3 channels 32-34
spw = '2:1~3;57~63'      # spw 2, channels 1-3 and 57-63
spw = '1~3:10~20'        # spw 1-3, channels 10-20
spw = '*:4~56'           # all spw, channels 4-56
```

Note the use of the wildcard in the last example.

A step can be also be included using `'^STEP'` as a postfix:


```

spw = '0:10~100^2'      # chans 10,12,14,...,100 of spw 0
spw = ':^4'            # chans 0,4,8,... of all spw
spw = ':100~150GHz^10GHz' # closest chans to 100,110,...,150GHz

```

A step in frequency or velocity will pick the channel in which that frequency or velocity falls, or the nearest channel.

2.5.4 The selectdata Parameters

The `selectdata` parameter, if set to `True`, will expand the inputs to include a number of sub-parameters, given below and in the individual task descriptions (if different). If `selectdata = False`, then the sub-parameters are treated as blank for selection by the task. The default for `selectdata` is `False`.

The common `selectdata` expanded sub-parameters are:

2.5.4.1 The antenna Parameter

The `antenna` selection string is a semi-colon (';') separated list of baseline specifications. A baseline specification is of the form:

- `'ANT1'` — select all baselines including the antenna(s) specified by the selector `ANT1`,
- `'ANT1&'` — select only baselines between the antennas specified by the selector `ANT1`,
- `'ANT1&ANT2'` — select only baselines between the antennas specified by selector `ANT1` and antennas specified by selector `ANT2`. Thus `'ANT1&'` is an abbreviation for `'ANT1&ANT1'`.

The selectors `ANT1` and `ANT2` are comma-separated lists of antenna integer-IDs or literal antenna names, patterns, or regular expressions. The `ANT` strings are parsed and converted to a list of antenna integer-IDs or IDs of antennas whose name match the given names/pattern/regular expression. Baselines corresponding to all combinations of the elements in lists on either side of ampersand are selected.

Integer IDs can be specified as single values or a range of integers. When items of the list are parsed as literal strings or regular expressions or patterns (see § 2.5.1 for more details on strings). All antenna names that match the given string (exact match)/regular expression/pattern are selected.

The comma is used only as a separator for the list of antenna specifications. The list of baselines specifications is a semi-colon separated list, e.g.

```
antenna = '1~3 & 4~6 ; 10&11'
```

will select baselines between antennas 1,2,3 and 4,5,6 (`'1&4'`, `'1&5'`, ..., `'3&6'`) plus baseline `'10&11'`.

The wildcard operator (`'*'`) will be the most often used pattern. To make it easy to use, the wildcard (and only this operator) can be used without enclosing it in quotes. For example, the selection

```
antenna = 'VA*'
```

will match all antenna names which have 'VA' as the first 3 characters in the name (irrespective of what follows after these characters).

Antenna numbers as names: Needless to say, naming antennas such that the names can also be parsed as a valid token of the syntax is a bad idea. Nevertheless, antenna names that contain any of the reserved characters and/or can be parsed as integers or integer ranges can still be used by enclosing the antenna names in double quotes (' "ANT" '). E.g. the string

```
antenna = '10~15,21,VA22'
```

will expand into an antenna ID list 10,11,12,13,14,15,21,22 (assuming the index of the antenna named 'VA22' is 22). If the antenna with ID index 50 is named '21', the string

```
antenna = '10~15,"21",VA22'
```

will expand into an antenna ID list of 10,11,12,13,14,15,50,22.

Read elsewhere (e.g. `info regex` under Unix) for details of regular expression and patterns.

2.5.4.2 The scan Parameter

The `scan` parameter selects the scan ID numbers of the data. There is currently no naming convention for scans. The scan ID is filled into the MS depending on how the data was obtained, so use this with care.

Examples:

```
scan = '3'           # scan number 3.
scan = '1~8'        # scan numbers 1 through 8, inclusive
scan = '1,2,4,6'    # scans 1,2,4,6
scan = '<9'          # scans <9 (1-8)
```

NOTE: ALMA and VLA/EVLA number scans starting with 1 and not 0. You can see what the numbering is in your MS using the `listobs` task with `verbose=True` (see § 2.3).

2.5.4.3 The timerange Parameter

The time strings in the following (T0, T1 and dT) can be specified as YYYY/MM/DD/HH:MM:SS.FF. The time fields (i.e., YYYY, MM, DD, HH, MM, SS and FF), starting from left to right, may be omitted and they will be replaced by context sensitive defaults as explained below.

Some examples:

1. `timerange='T0~T1'`: Select all time stamps from T0 to T1.

Fields missing in T0 are replaced by the fields in the time stamp of the first valid row in the MS. Fields missing in T1 are replaced by the corresponding fields of T0 (after its defaults are set).

2. `timerange='T0'`: Select all time stamps that are within an integration time of T0.

Integration time is determined from the first valid row (more rigorously, an average integration time should be computed). Default settings for the missing fields of T0 are as in (1).

3. `timerange='T0+dT'`: Select all time stamps starting from T0 and ending with time stamp T0+dT.

Defaults of T0 are set as usual. Defaults for dT are set from the time corresponding to MJD=0. Thus, dT is a specification of length of time from the assumed nominal "start of time".

4. `timerange='>T0'`: Select all times greater than T0.

5. `timerange='<T1'`: Select all times less than T1.

Default settings for T0 and T1 are as above.

For example, a typical `timerange` selection might be

```
timerange = '25/22:40:0 ~ 26/03:30:0'
```

where the YY/MM/ part of the selection has been defaulted to the start of the MS. An ultra-conservative selection might be:

```
timerange = '1960/01/01/00:00:00~2020/12/31/23:59:59'
```

which would choose all possible data!

2.5.4.4 The `uvrange` Parameter

Rows in the MS can also be selected based on the uv-distance or physical baseline length that the visibilities in each row correspond to. This `uvrange` can be specified in various formats.

The basic building block of uv-distance specification is a valid number with optional units in the format N[UNIT] (the unit in square brackets is optional). We refer to this basic building block as UVDIST. The default unit is meter. Units of length (such as 'm' and 'km') select physical baseline distances (independent of wavelength). The other allowed units are in wavelengths (such as 'l', 'kl' and 'Ml' for lambda, kilo-lambda and mega-lambda respectively) and are true uv-plane radii

$$r_{uv} = \sqrt{u^2 + v^2}. \quad (2.1)$$

If only a single UVDIST is specified, all rows, the uv-distance of which exactly matches the given UVDIST, are selected.

UVDIST can be specified as a range in the format 'N0~N1[UNIT]' (where N0 and N1 are valid numbers). All rows corresponding to uv-distance between N0 and N1 (inclusive) when converted the specified units are selected.

UVDIST can also be selected via comparison operators. When specified in the format '>UVDIST', all visibilities with uv-distances greater than the given UVDIST are selected. Likewise, when specified in the format '<UVDIST', all rows with uv-distances less than the given UVDIST are selected.

Any number of above mentioned uv-distance specifications can be given as a comma-separated list.

Examples:

```
uvrange = '100~200km'           # an annulus in physical baseline length
uvrange = '24~35M1, 40~45M1'   # two annuli in units of mega-wavelengths
uvrange = '< 45k1'             # less than 45 kilolambda
uvrange = '> 01'               # greater than zero length (no auto-corrs)
uvrange = '100km'             # baselines of length 100km
uvrange = '100k1'             # visibilities with uv-radius 100 kilolambda
```

2.5.4.5 The mselect Parameter

More complicated selections within the MS structure are possible using the Table Query Language (TaQL). This is accessed through the `mselect` parameter.

Note that the TaQL syntax does not follow the rules given in § 2.5.1 for our other selection strings. TaQL is explained in more detail in **Aips++ NOTE 199 — Table Query Language** (<http://aips2.nrao.edu/docs/notes/199/199.html>). This will eventually become a CASA document. The specific columns of the MS are given in the most recent MS specification document: **Aips++ NOTE 229 — MeasurementSet definition version 2.0** (<http://aips2.nrao.edu/docs/notes/229/229.html>). This documentation will eventually be updated to the CASA document system.

Most selection can be carried out using the other selection parameters. However, these are merely shortcuts to the underlying TaQL selection. For example, field and spectral window selection can be done using `mselect` rather than through `field` or `spw`:

```
mselect='FIELD_ID == 0'           # Field id 0 only
mselect='FIELD_ID <= 1'         # Field id 0 and 1
mselect='FIELD_ID IN [1,2]'     # Field id 1 and 2
mselect='FIELD_ID==0 && DATA_DESC_ID==3' # Field id 0 in spw id 3 only
```

Chapter 3

Data Examination and Editing

3.1 Plotting and Flagging Visibility Data in CASA

The tasks available for plotting and flagging of data are:

- `flagmanager` — manage versions of data flags
- `flagautocorr` — non-interactive flagging of auto-correlations
- `plotxy` — create X-Y plots of data in MS, flag data
- `flagdata` — non-interactive flagging of data
- `viewer` — use viewer to look at and flag MS
- `browsetable` — browse data in any CASA table (including a MS)

The following sections describe the use of these tasks.

Information on other related operations can be found in:

- `listobs` — list what's in a MS (§ 2.3)
- `selectdata` — general data selection syntax (§ 2.5)

3.2 Managing flag versions with `flagmanager`

The `flagmanager` task will allow you to manage different versions of flags in your data. These are stored inside a CASA `flagversions` table, under the name of the MS `<msname>.flagversions`. For example, for the MS `jupiter6cm.usecase.ms`, there will need to be `jupiter6cm.usecase.ms.flagversions` on disk. This is created when flagging is first done, such as with `plotxy`.

The inputs for `flagmanager` are:

```
vis          =      ''          # Name of input visibility file (MS)
mode         =      'list'      # Flag management operation (list,save,restore,delete)
```

The mode='list' option will list the available flagversions from the <msname>.flagversions file. For example:

```
CASA <103>: vis = 'jupiter6cm.usecase.ms'
CASA <104>: mode = 'list'
CASA <105>: flagmanager()
See logger for flag versions for this file
```

```
Tue Jun 26 20:52:55 2007 NORMAL : Table : /home/sandrock2/smyers/jupiter6cm.usecase.ms
Tue Jun 26 20:52:55 2007 NORMAL : main : working copy in main table
Tue Jun 26 20:52:55 2007 NORMAL : xyflags : Plotxy flags
```

The mode parameter expands the options. For example, if you wish to save the current flagging state of vis=<msname>,

```
mode          =      'save'      # Flag management operation (list,save,restore,delete)
versionname   =      ''         # Name of flag version (no spaces)
comment       =      ''         # Short description of flag version
merge        =      'replace'    # Merge option (replace, and, or)
```

with the output version name specified by versionname. For example, the above xyflags version was written using:

```
default('flagmanager')
vis = 'jupiter6cm.usecase.ms'
mode = 'save'
versionname = 'xyflags'
comment = 'Plotxy flags'
flagmanager()
```

and you can see that there is now a sub-table in the flagversions directory

```
CASA <106>: ls jupiter6cm.usecase.ms.flagversions/
IPython system call: ls -F jupiter6cm.usecase.ms.flagversions/
flags.xyflags/ FLAG_VERSION_LIST
```

It is recommended that you use this facility regularly to save versions during flagging.

You can restore a previously saved set of flags using the mode='restore' option:

```
mode          =      'restore'   # Flag management operation (list,save,restore,delete)
versionname   =      ''         # Name of flag version (no spaces)
merge        =      'replace'    # Merge option (replace, and, or)
```

The `merge` sub-parameter will control the action. For `merge='replace'`, the flags in `versionname` will replace those in the MAIN table of the MS. For `merge='and'`, only data that is flagged in BOTH the current MAIN table and in `versionname` will be flagged. For `merge='or'`, data flagged in EITHER the MAIN or in `versionname` will be flagged.

The `mode='delete'` option can be used to remove `versionname` from the flagversions:

```
mode                = 'delete'          # Flag management operation (list,save,restore,delete)
  versionname       = ''                # Name of flag version (no spaces)
```

3.3 Flagging auto-correlations with `flagautocorr`

The `flagautocorr` task can be used if all you want to do is to flag the auto-correlations out of the MS. Nominally, this can be done upon filling from the VLA for example, but you may be working from a dataset that still has them.

This task has a single input, the MS file name:

```
vis                 = ''                # Name of input visibility file (MS)
```

To use it, just set and go:

```
CASA <90>: vis = 'jupiter6cm.usecase.ms'
CASA <91>: flagautocorr()
```

Note that the auto-correlations can also be flagged using `flagdata` (§ 3.5) but the `flagautocorr` task is a handy shortcut for this common operation.

3.4 X-Y Plotting and Editing of the Data

The principal way to get X-Y plots of visibility data is using the `plotxy` task. This task also provides editing capability. CASA uses the `matplotlib` plotting library to display its plots. You can find information on `matplotlib` at <http://matplotlib.sourceforge.net/>.

The `plotxy` plotter is shown in figure 3.1.

To bring up this plotter use the `plotxy` task. The inputs are:

```
# plotxy :: Plot points for selected X and Y axes:

vis                 = ''                # Name of input visibility
axis                = 'time'          # azimuth,elevation,hourangle,baseline,channel,time,u,v,w,uvdist,x
```

Inside the Toolkit:

Access to `matplotlib` is also provided through the `p1` tool. See below for a description of the `p1` tool functions.

```

yaxis          =      'amp'      # azimuth,elevation,hourangle,baseline,amp,pha,u,v,w,uvdist
datacolumn    =      'data'     # data (raw), corrected, model, residual (corrected - model)
field         =      ''        # field names or index of calibrators:
spw           =      ''        # spectral window:channels: ''=>all, spw='1:5~57'
selectdata    =      False     # Other data selection parameters
average       =      ''        # Select averaging mode: time or channel
subplot       =      111       # Panel number on display screen (yxn)
overplot      =      False     # Overplot values on current plot (if possible)
showflags     =      False     # Show flagged data
iteration      =      ''        # Separate panels by: field, antenna, baseline, scan, feed
plotsymbol    =      '.'       # pylab plot symbol
plotcolor     =      'darkcyan' # pylab plot color
markersize   =      5.0       # Size of plotted marks
linewidth     =      1.0       # Width of plotted lines
connect       =      'none'    # Specifies which points are connected with lines
plotrange     =      [-1, -1, -1, -1] # The range of data to be plotted, can be time values
skipnpoints   =      1        # Plot every nth point
multicolor    =      'none'    # Plot polarizations and channels in different colors
replacetopplot =      False   # Replace the last plot or not when overplotting
removeoldpanels =      True    # Turn on/of automatic clearing of panels
title         =      ''        # Plot title (above plot)
xlabel        =      ''        # Label for x-axis
ylabel        =      ''        # Label for y-axis
fontsize      =      10.0     # Font size for labels
windowsize    =      1.0      # Window size

```

Setting `selectdata=True` opens up several sub-parameters:

```

selectdata    =      True      # Select a subset of the data - opens selection params
  antenna     =      ''        # Select data based on antenna/baseline
  timerange   =      ''        # Select data based on time
  correlation =      ''        # Correlation(s) to plot (RR,LL,RR LL,XX,YY,XX YY)
  scan        =      ''        # Select data based on scan number
  feed        =      ''        # Select data based on feed number - NOT IMPLEMENTED
  array       =      ''        # Select data based on the array
  uvrange     =      ''        # Select data based on uv range

```

The parameter `average` is used to request averaging either by time or by frequency. Option `average='time'` leads to the option of setting the sub-parameter `averagenpoints`, which sets the number of *rows* to average when plotting. This is somewhat like time averaging. Option `average='channel'` affects the interpretation of the `spw` parameter, with ⁵ meaning "average 5 channels" rather than "show every 5th channel." **ALPHA ALERT:** the averaging options are in the process of being refined and extended.

For example:

```

plotxy(vis='n75.ms',          # channel plot for the n75.ms data set
        xaxis='channel',      # plot channels on x-axis
        yaxis='amp',         # plot amplitude on y-axis
        plotsymbol=',',     # use red, slightly large dots

```



```

datacolumn='corrected',           # plot corrected data
selectdata=True,                 # open data selection
field='1328',                    # Plot only source 1328+307 (minimum match)
spw='2',                         # Plot channels in spectral window 2.
correlation='RR LL',            # Plot RR and LL correlations
multicolor=True)                # Allow plotxy to plot different colors.

```

3.4.1 Plot control

You can use the various buttons on the `plotxy` GUI to control its operation – in particular, to determine flagging and unflagging behaviors.

There is a standard row of buttons at the bottom. These include (left to right):

- **Home** — The “house” button (1st on left) returns to the original zoom level.
- **Step** — The left and right arrow buttons (2nd and 3rd from left) step through the zoom settings you’ve visited.
- **Pan** — The “four-arrow button” (4th from left) lets you pan in zoomed plot.
- **Zoom** — The most useful is the “magnifying glass” (5th from the left) which lets you draw a box and zoom in on the plot.
- **Panels** — The “window-thingy” button (second from right) brings up a menu to adjust the panel placement in the plot.
- **Save** — The “disk” button (last on right) saves a `.png` copy of the plot to a generically named file on disk.

In a row above these, there are a set of other buttons (left to right):

- **Mark Region** — If depressed lets you draw rectangles to mark points in the panels. This is done by left-clicking and dragging the mouse. You can Mark multiple boxes before doing something. Clicking the button again will un-depress it and forget the regions. `ESC` will remove the last region marked.
- **Flag** — Click this to Flag the points in a marked region.
- **Unflag** — Click this to Unflag any flagged point that would be in that region (even if invisible).
- **Locate** — Print out some information to the logger on points in the marked regions.
- **Next** — Step to the next plot in an iteration.
- **Quit** — Exit `plotcal`, clear the window and detach from the MS.

These buttons are shared with the `plotcal` tool.

3.4.2 plotoptions

The `plotoptions` parameters work in concert with the native `matplotlib` functionality to enable flexible representations of data displays. In particular, the parameters `overplot`, `subplot`, and `plotsymbol` allow visual comparisons of data results for analysis during the reduction process.

3.4.2.1 plotsymbol

The `plotsymbol` parameter defines both the line or symbol for the data being drawn as well as the color; from the `matplotlib` online documentation (e.g., type `pl.plot?` for help):

The following line styles are supported:

```
-      : solid line
--     : dashed line
- .    : dash-dot line
:      : dotted line
.      : points
,      : pixels
o      : circle symbols
^      : triangle up symbols
v      : triangle down symbols
<      : triangle left symbols
>      : triangle right symbols
s      : square symbols
+      : plus symbols
x      : cross symbols
D      : diamond symbols
d      : thin diamond symbols
1      : tripod down symbols
2      : tripod up symbols
3      : tripod left symbols
4      : tripod right symbols
h      : hexagon symbols
H      : rotated hexagon symbols
p      : pentagon symbols
|      : vertical line symbols
_      : horizontal line symbols
steps : use gnuplot style 'steps' # kwarg only
```

The following color abbreviations are supported

```
b : blue
g : green
r : red
c : cyan
```

Inside the Toolkit:

For even more functionality, you can access the `pl` tool directly using `Py-lab` functions that allow one to annotate, alter, or add to any plot displayed in the `matplotlib` plotter (e.g. `plotxy`). See the **CASA Toolkit Guide**.

```

m : magenta
y : yellow
k : black
w : white

```

In addition, you can specify colors in many weird and wonderful ways, including full names 'green', hex strings '#008000', RGB or RGBA tuples (0,1,0,1) or grayscale intensities as a string '0.8'.

Line styles and colors are combined in a single format string, as in 'bo' for blue circles.

3.4.2.2 Iteration

There are currently four iteration options available: 'antenna', 'time', 'baseline', and 'field_id'. If one of these options is chosen, the data will be split into separate plot displays for each value of the iteration axis (e.g., for the VLA, the 'antenna1' option will get you 27 displays, one for each antenna).

An example iteration session:

```

CASA <1>: default plotxy
CASA <1>: plotxy('n5921.ms', 'channel', datacolumn='corrected', subplot=311, iteration='antenna1')

Number of rows in the selected Measurement Set : 22653
Thu Jul 5 20:49:44 2007      WARN :
Iterating on ANTENNA1 instead of ANTENNA
plotsymbols: .
connect: none
Attaching the main selected MS
Thu Jul 5 20:49:44 2007      WARN :

May need to read 2854278 values from disk.
Preparing the plotter..
Reading data...
Time spent reading X and Y data from disk : 0.09 sec.
Time to extract Flags : 0.02
Number of new points being plotted : 195651
Number of new points not being plotted : 8343
Total Plotting time : 1.013 sec.
Preparing the plotter..
Reading data...
Time spent reading X and Y data from disk : 0.08 sec.
Time to extract Flags : 0.01
Number of new points being plotted : 188290
Number of new points not being plotted : 8144
Total Plotting time : 1.507 sec.
Preparing the plotter..
Reading data...
Time spent reading X and Y data from disk : 0.08 sec.

```

```

Time to extract Flags : 0.01
Number of new points being plotted : 181111
Number of new points not being plotted : 7763
Total Plotting time : 1.934 sec.

```

etc.

3.4.2.3 Subplots

The `plotxy` argument `subplot` takes three numbers. The first is the number of y panels (stacking vertically), the second is the number of xpanels (stacking horizontally) and the third is the number of the panel you want to draw into. For example, `subplot=212` would draw into the lower of two panels stacked vertically in the figure.

An example use of subplot capability is shown in Fig 3.3. These were drawn with the commands (for the top, bottom left, and bottom right panels respectively):

```

plotxy('n5921.ms','channel',          # plot channels for the n5921.ms data set
      field='0',                       # plot only first field
      datacolumn='corrected',         # plot corrected data
      plotcolor='',                   # over-ride default plot color
      plotsymbol='go',                 # use green circles
      subplot=211)                     # plot to the top of two panels
plotxy('n5921.ms','x',                 # plot antennas for n5921.ms data set
      field='0',                       # plot only first field
      datacolumn='corrected',         # plot corrected data
      subplot=223,                     # plot to 3rd panel (lower left) in 2x2 grid
      plotcolor='',                   # over-ride default plot color
      plotsymbol='r.')                 # red dots
plotxy('n5921.ms','u','v',            # plot uv-coverage for n5921.ms data set
      field='0',                       # plot only first field
      datacolumn='corrected',         # plot corrected data
      subplot=224,                     # plot to the lower right in a 2x2 grid
      plotcolor='',                   # over-ride default plot color
      plotsymbol='b,')                 # blue, somewhat larger dots
# NOTE: You can change the gridding and panel
# size by manipulating the ny x nx grid.

```

3.4.3 Interactive Flagging in plotxy

Interactive flagging, on the principle of “see it — flag it”, is possible on the X-Y display of the data plotted by `plotxy`. The user can use the cursor to mark one or more regions, and then flag, unflag, or list the data that falls in these zones of the display.

Hint!

In the plotting environments such as `plotxy`, the ESC key can be used to remove the last region box drawn.

There is a row of buttons below the plot in the window. You can punch the **Mark Region** button (which will appear to depress), then mark a region by left-clicking and dragging the mouse (each click and drag will mark an additional region). You can get rid of all your regions by clicking again on the **Mark Region** button (which will appear to un-depress), or you can use the **ESC** key to remove the last box you drew. Once regions are marked, you can then click on one of the other buttons to take action:

1. **Flag** — flag the points in the region(s),
2. **Unflag** — unflag flagged points in the region(s),
3. **Locate** — spew out a list of the points in the region(s) to the logger (Warning: this could be a long list!).

Whenever you click on a button, that action occurs without forcing a diskwrite (unlike previous versions). If you quit `plotxy` and re-enter, you will see your previous edits.

A table with the name `<msname>.flagversions` (where `vis=<msname>`) will be created in the same directory if it does not exist already.

It is recommended that you save important flagging stages using the `flagmanager` task (§ 3.2).

3.4.4 Exiting plotxy

You can use the **Quit** button to clear the plot from the window and detach from the MS. You can also dismiss the window by killing it with the **X** on the frame, which will also detach the MS.

You can also just leave it alone. The plotter pretty much keeps running in the background even when it looks like it's done! You can keep doing stuff in the plotter window, which is where the `overplot` parameter comes in. Note that the `plotcal` task (§ 4.8) will use the same window, and can also overplot on the same panel.

If you leave `plotxy` running, beware of (for instance) deleting or writing over the MS without stopping. It may work from a memory version of the MS or crash.

3.4.5 Example session using plotxy

The following is an example of interactive plotting and flagging using `plotxy` on the Jupiter 6cm continuum VLA dataset. This is extracted from the script `jupiter6cm.usecase.py` available in the script area.

This assumes that the MS `jupiter6cm.usecase.ms` is on disk with `flagautocorr` already run.

ALPHA ALERT: Exact syntax may be slightly different in your version as the Alpha Patches progress.

```
default('plotxy')

vis = 'jupiter6cm.usecase.ms'

# The fields we are interested in: 1331+305,JUPITER,0137+331
selectdata = True

# First we do the primary calibrator
field = '1331+305'

# Plot only the RR and LL for now
correlation = 'RR LL'

# Plot amplitude vs. uvdist
xaxis = 'uvdist'
yaxis = 'amp'
multicolor = 'both'

# The easiest thing is to iterate over antennas
iteration = 'antenna'

plotxy()

# You'll see lots of low points as you step through RR LL RL LR
# A basic clip at 0.75 for RR LL and 0.055 for RL LR will work
# If you want to do this interactively, set
iteration = ''

plotxy()

# You can also use flagdata to do this non-interactively
# (see below)

# Now look at the cross-polar products
correlation = 'RL LR'

plotxy()

#-----
# Now do calibrator 0137+331
field = '0137+331'
correlation = 'RR LL'
xaxis = 'uvdist'
spw = ''
iteration = ''
antenna = ''

plotxy()

# You'll see a bunch of bad data along the bottom near zero amp
# Draw a box around some of it and use Locate
```

```

# Looks like much of it is Antenna 9 (ID=8) in spw=1

xaxis = 'time'
spw = '1'
correlation = ''

# Note that the strings like antenna='9' first try to match the
# NAME which we see in listobs was the number '9' for ID=8.
# So be careful here (why naming antennas as numbers is bad).
antenna = '9'

plotxy()

# YES! the last 4 scans are bad. Box 'em and flag.

# Go back and clean up
xaxis = 'uvdist'
spw = ''
antenna = ''
correlation = 'RR LL'

plotxy()

# Box up the bad low points (basically a clip below 0.52) and flag

# Note that RL,LR are too weak to clip on.

#-----
# Finally, do JUPITER
field = 'JUPITER'
correlation = ''
iteration = ''
xaxis = 'time'

plotxy()

# Here you will see that the final scan at 22:00:00 UT is bad
# Draw a box around it and flag it!

# Now look at whats left
correlation = 'RR LL'
xaxis = 'uvdist'
spw = '1'
antenna = ''
iteration = 'antenna'

plotxy()

# As you step through, you will see that Antenna 9 (ID=8) is often
# bad in this spw. If you box and do Locate (or remember from
# 0137+331) its probably a bad time.

```

```
# The easiset way to kill it:

antenna = '9'
iteration = ''
xaxis = 'time'
correlation = ''

plotxy()

# Draw a box around all points in the last bad scans and flag 'em!

# Now clean up the rest
xaxis = 'uvdist'
correlation = 'RR LL'
antenna = ''
spw = ''

# You will be drawing many tiny boxes, so remember you can
# use the ESC key to get rid of the most recent box if you
# make a mistake.

plotxy()

# Note that the end result is we've flagged lots of points
# in RR and LL. We will rely upon imager to ignore the
# RL LR for points with RR LL flagged!
```

3.5 Non-Interactive Flagging using flagdata

Task `flagdata` will flag the visibility data set based on the specified data selections, most of the information coming from a run of the `listobs` task (with/without `verbose=True`). Currently you can select based on any combination of:

- antennas (`antenna`)
- baselines (`antenna`)
- spectral windows and channels (`spw`)
- correlation types (`correlation`)
- field ids or names (`field`)
- uv-ranges (`uvrange`)
- times (`timerange`) or scan numbers (`scan`)

- antenna arrays (`array`)

and choose to flag, unflag, clip (`setclip` and sub-parameters), and remove the first part of each scan (`setquack`) and/or the autocorrelations (`autocorr`).

The inputs to `flagdata` are:

```
vis           =      ''      # Name of input visibility file
antenna       =      ''      # Select data based on antenna/baseline
spw           =      ''      # Select data based on spectral-window/frequency/channel
correlation   =      ''      # Select data based on correlation
field         =      ''      # Select data based on field name or index
uvrange       =      ''      # Select data based on uv range
timerange     =      ''      # Select data based on time
scan          =      ''      # Select data based on scan number
feed          =      ''      # Select data based on feed number - NOT ENABLED
array         =      ''      # Select data based on the array
mode          = 'manualflag' # manualflag
    autocorr   =      False   # Flag autocorrelations
    unflag     =      False   # Unflag the data specified
    setclip    = [' ', [], True] # Setup Clipping [clipexpr, [min,max], outside(T/F)]
    setquack   =      []      # Setup VLA Quack [quack interval, quack length] - NOT TESTED
```

ALPHA ALERT: the modes `'autoflag'`, `'summary'`, `'query'`, and `'extend'` are not currently supported.

3.5.1 Flag Antenna/Channels

The following commands give the results shown in Figure 3.5:

```
default plotxy
plotxy('ngc5921.ms', 'channel', iteration='antenna1', subplot=311)
default flagdata
flagdata(vis='ngc5921.ms', antenna='0', spw='0:10~15')
default plotxy
plotxy('ngc5921.ms', 'channel', iteration='antenna1', subplot=311)
```

3.5.1.1 Clipping in `flagdata`

The following commands give the results shown in Figure 3.6:

```
default plotxy
plotxy('ngc5921.ms', 'uvdist')
default flagdata
flagdata(vis='ngc5921.ms', setclip=['LL', [0.0, 1.6], True])
plotxy('ngc5921.ms', 'uvdist')
```

3.6 Interactive flagging using the viewer

Coming soon.

3.7 Browse the Data

The `browsetable` task is available for viewing data directly (and handles all CASA tables, including MeasurementSets, calibration tables, and images).

The default inputs are:

```
# browsetable :: Browse a table (MS, calibration table, image)
tablename          =          ''          # Name of input table
```

Currently, its single input is the `tablename`, so an example would be:

```
CASA <2>: browsetable('ngc5921.ms')
```

For an MS such as this, it will come up with a browser of the `MAIN` table (see Fig 3.7). If you want to look at sub-tables, use the **View:Table Keywords** to bring up a panel with the sub-tables listed (Fig 3.8), then choose (left-click) a table and **View:Details** to bring it up (Fig 3.9). You can left-click on a cell in a table to view the contents.

ALPHA ALERT: The `browsetable` task brings up the CASA Java Tablebrowser, which is a separate program. You may encounter some crashes, particularly if you use the **File:Open** menu to browse sub-tables or MS.

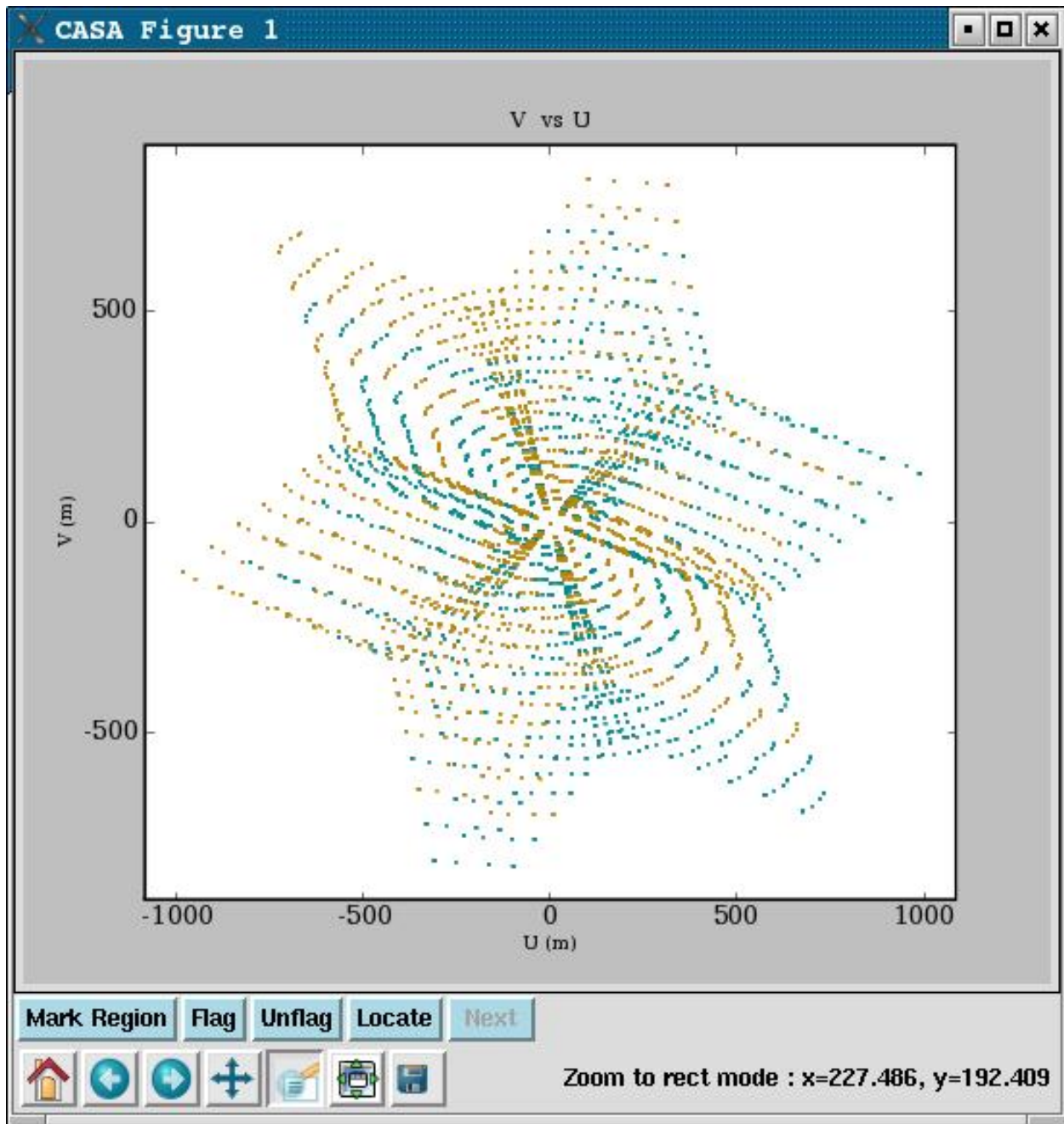


Figure 3.1: The plotxy plotter. The **bottom set of buttons** on the lower left are: 1,2,3) **Home, Back, and Forward**. Click to navigate between previously defined views (akin to web navigation). 4) **Pan**. Click and drag to pan to a new position. 5) **Zoom**. Click to define a rectangular region for zooming. 6) **Subplot Configuration**. Click to configure the parameters of the subplot and spaces for the figures. 7) **Save**. Click to launch a file save dialog box. The **upper set of buttons in the lower left** are: 1) **Mark Region**. Press this to begin marking regions (rather than zooming or panning). 2,3,4) **Flag, Unflag, Locate**. Click on these to flag, unflag, or list the data within the marked regions. 5) **Next**. Click to move to the next in a series of iterated plots. Finally, the **cursor readout** is on the bottom right.

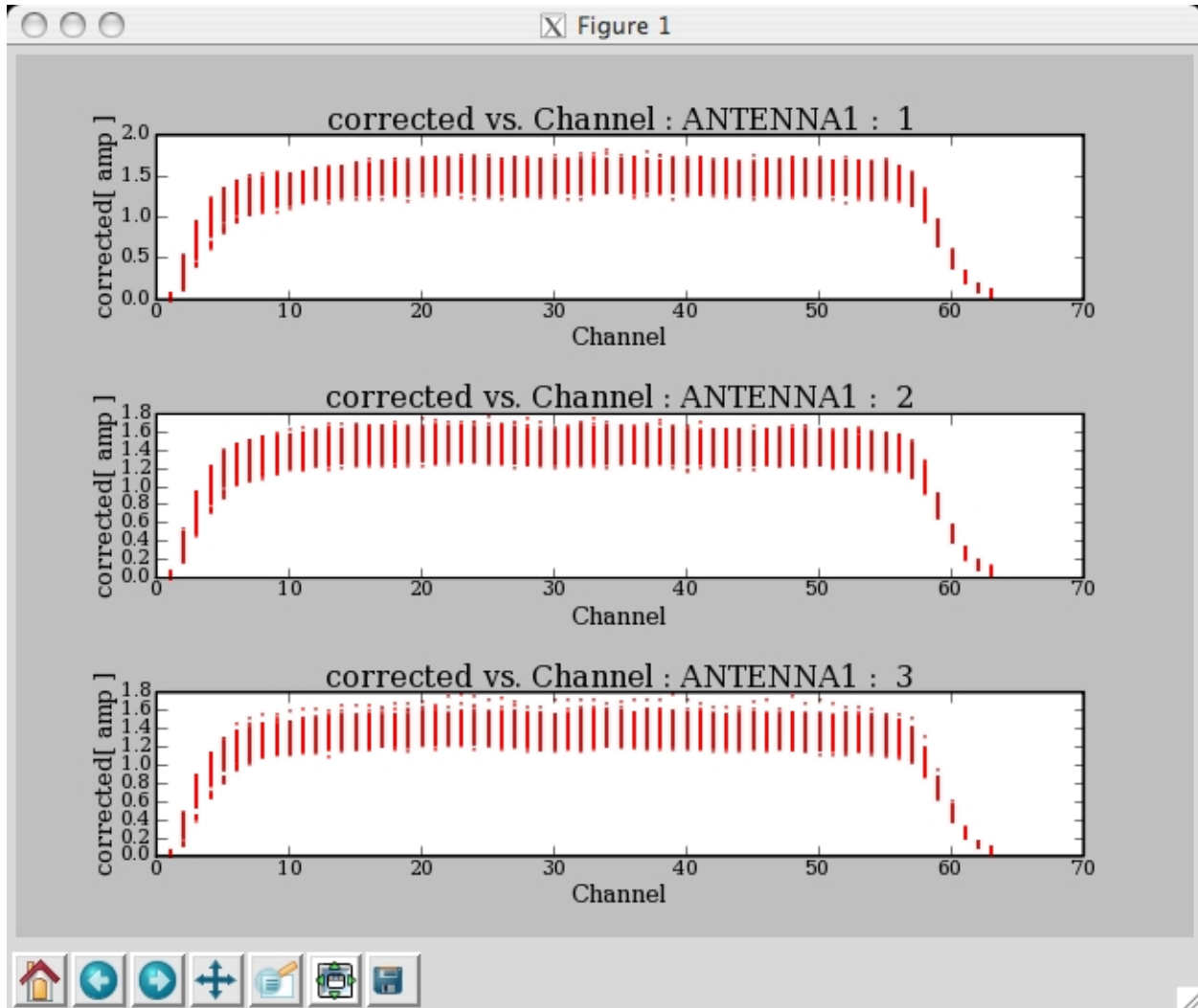


Figure 3.2: plotxy iteration plot: The first set of plots from the example in § 3.4.2.2. Each time you press the **Next** button, you get the next series of plots.

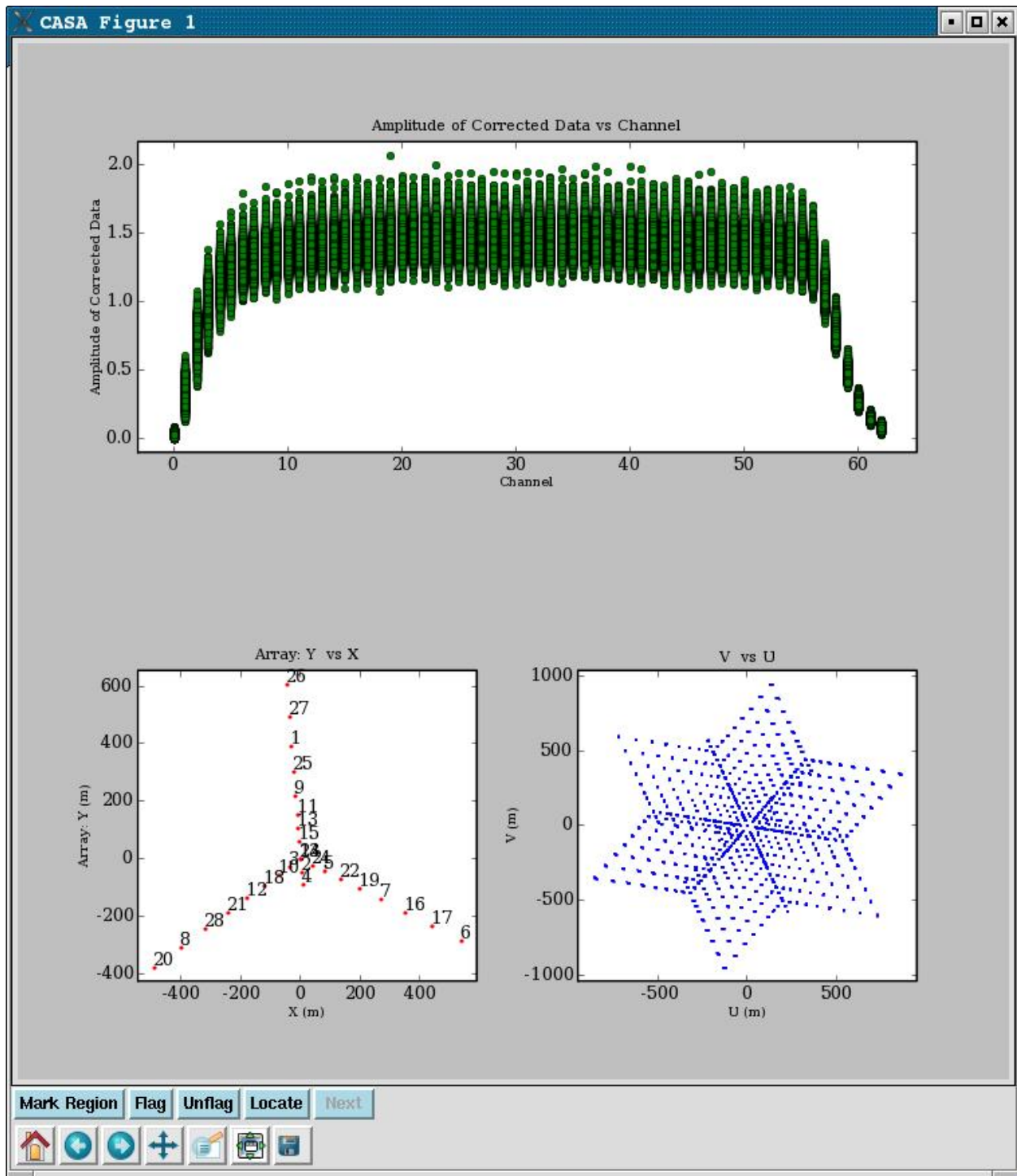


Figure 3.3: Multi-panel display of visibility versus channel (**top**), antenna array configuration (**bottom left**) and the resulting uv coverage (**bottom right**). The commands to make these three panels respectively are: 1) `plotxy('n5921.ms', xaxis='channel', datacolumn='corrected', field='0', subplot=211, plotcolor='', plotsymbol='go')`, 2) `plotxy(xaxis='x', subplot=223, plotsymbol='r')`, 3) `plotxy(xaxis='u', yaxis='v', subplot=224, plotsymbol='b')`.

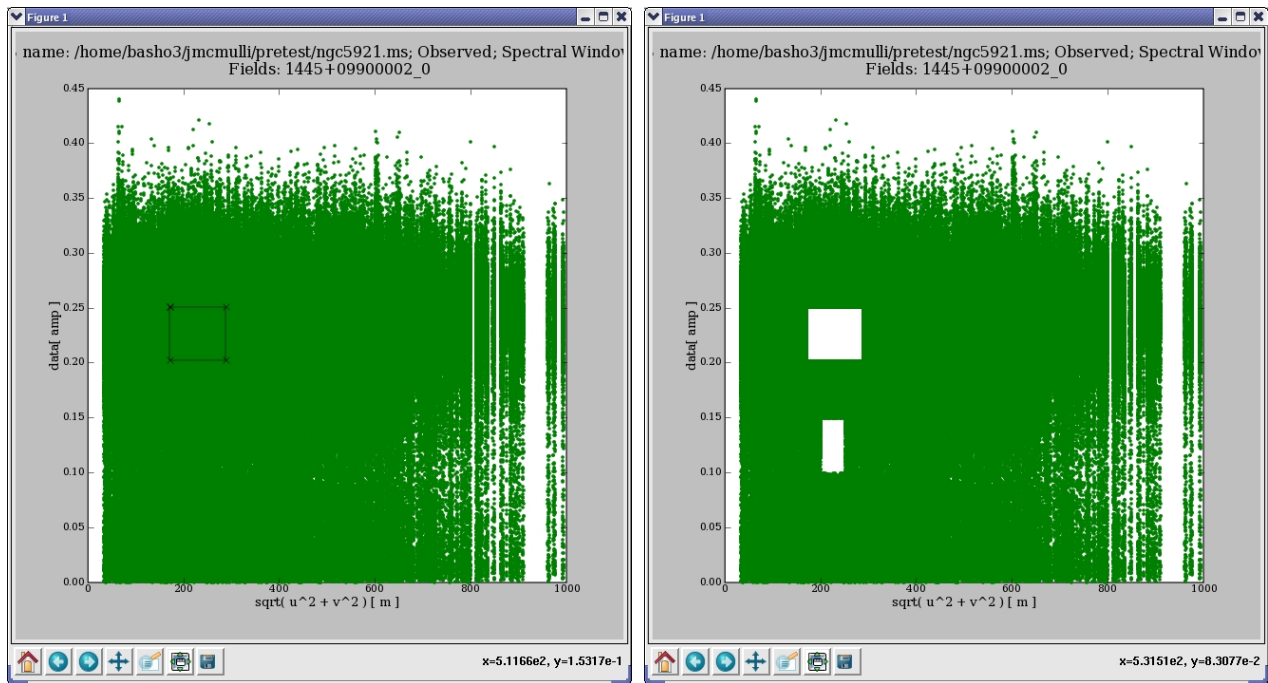


Figure 3.4: Plot of amplitude versus uv distance, before (left) and after (right) flagging two marked regions. The call was: `plotxy(vis='n5921.ms',xaxis='uvdist', plotsymbol='b,', subplot=111, datacolumn='data', field='1445*')`.

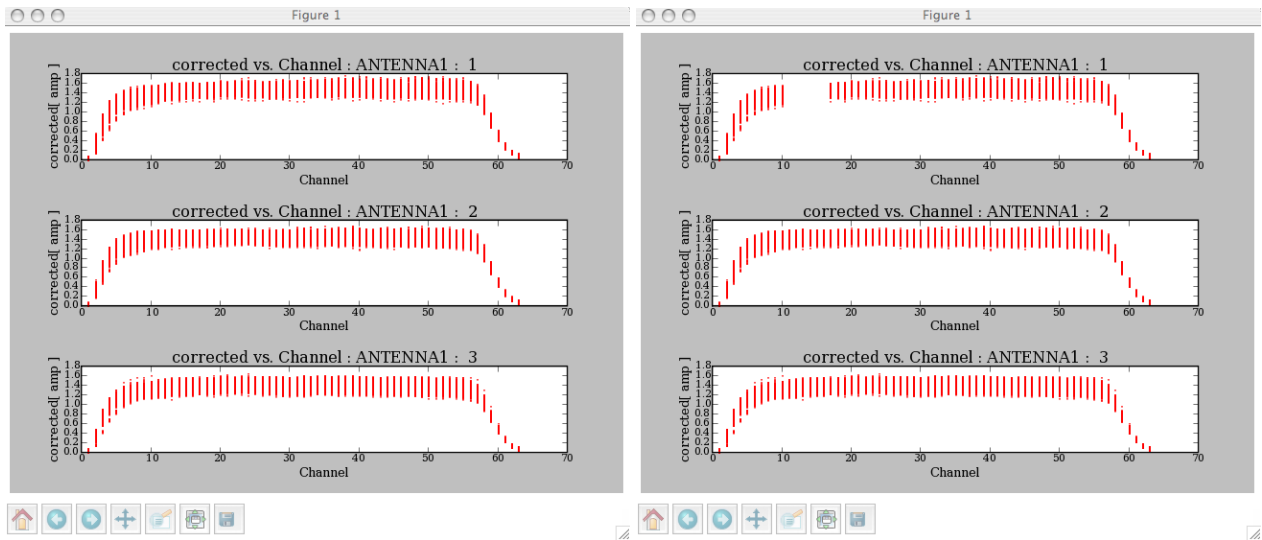


Figure 3.5: `flagdata`: Example showing before and after displays using a selection of one antenna and a range of channels. Note that each invocation of the `flagdata` task represents a cumulative selection, i.e., running `antenna='0'` will flag all data with antenna 0, while `antenna='0', spw='0:10 15'` will flag only those channels on antenna 0.

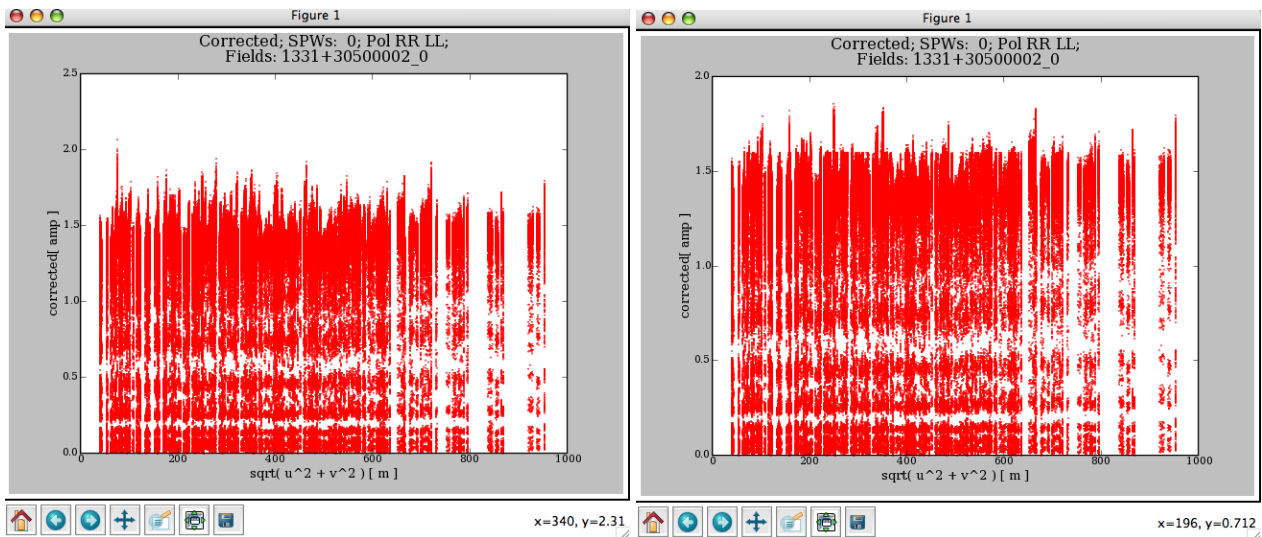


Figure 3.6: `flagdata`: Flagging example using the clip facility.

	UVW	FLAG	FLAG_CATEGORY	WEIGHT	SIGMA	ANTENNA1	ANTENNA2	ARRAY
1	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	1	0
2	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	27	27	0
3	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	7	7	0
4	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	2	2	0
5	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	11	11	0
6	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	17	17	0
7	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	9	9	0
8	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	19	19	0
9	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	20	20	0
10	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	18	18	0
11	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	3	3	0
12	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	15	15	0
13	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	21	21	0
14	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	6	6	0
15	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	23	23	0
16	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	5	5	0
17	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	4	4	0
18	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	10	10	0
19	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	14	14	0
20	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	26	26	0
21	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	12	12	0
22	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	24	24	0
23	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	13	13	0
24	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	25	25	0
25	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	0	0	0
26	0, 0]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	8	8	0
27	9.058, 250.84, -139.587]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	27	0
28	4.356, 324.988, -175.103]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	7	0
29	2322, -3.75874, -17.7603]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	2	0
30	4.913, 124.532, -79.1585]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	11	0
31	3904, 72.8845, -54.3907]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	17	0
32	3905, 29.9394, -33.9058]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	9	0
33	4.819, 406.108, -213.945]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	19	0
34	9.15, 183.892, -107.539]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	20	0
35	20.742, 0.179511, 82.5222]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	18	0

Figure 3.7: browsetable: The browser displays the main table within a frame. Hit the expand button to fill the browser frame (this has been done for this figure). You can scroll through the data (x=columns of the MAIN table, and y=the rows) or select a specific page or row as desired.

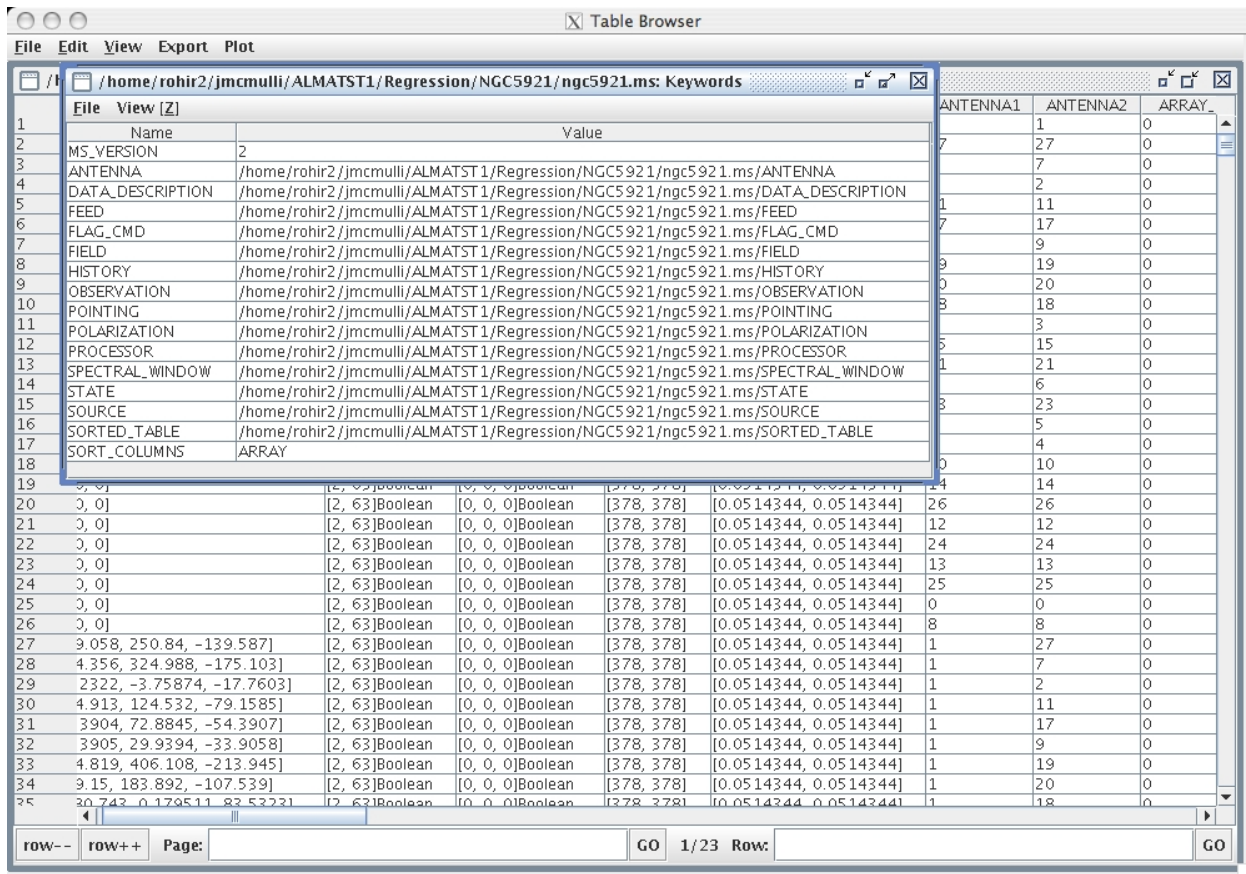


Figure 3.8: browstable: You can use the Menu option **View** to look at other tables within an MS. If you select on **View:Table Keywords** you get the image displayed. You can then select on a table to view its contents.

The screenshot shows a window titled "Table Browser" with a menu bar containing "File", "Edit", "View", "Export", and "Plot". The window displays a table with the following data:

	DIRECTION	PROPER_MOTION	CALIBRATION_GROUP	CODE	INTERVAL	NAME	NUM_LINES	SOURCE_ID
1	[-2.74393, 0.532485]	[0, 0]	-1		1.79769e+308	1331+20500002_0	1	0
2	[-2.42045, 0.174126]	[0, 0]	-1		1.79769e+308	1445+09900002_0	1	1
3	[-2.2602, 0.08843]	[0, 0]	-1		1.79769e+308	N5921_2	1	2
33	4.819, 406.108, -213.945]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	19	0
34	9.15, 183.892, -107.539]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	20	0
35	20.742, 0.179511, 82.53221]	[2, 63]Boolean	[0, 0, 0]Boolean	[378, 378]	[0.0514344, 0.0514344]	1	18	0

Navigation controls at the bottom include "row--", "row++", "Page:", "GO", "1/1 Row:", and "GO". A second set of controls shows "1/23 Row:".

Figure 3.9: browsetable: View the SOURCE table of the MS.

Chapter 4

Synthesis Calibration

This chapter explains how to calibrate interferometer data within the CASA task system. Calibration is the process of determining the complex correction factors that must be applied to each visibility in order to make them as close as possible to what an idealized interferometer would measure,

such that when the data is imaged an accurate picture of the sky is obtained. This is not an arbitrary process, and there is a methodology that has been developed to carry out synthesis calibration and an algebra to describe the various corruptions that data might be subject to: the Hamaker-Bregman-Sault Measurement Equation (ME).¹ The user need not worry about the details of this mathematics as the CASA software does that for you! Anyway, its just matrix algebra, and your familiar scalar methods of calibration (such as in AIPS) are encompassed in this more general approach! However, the curious can find a detailed description of the ME and calibration in the **CASA Toolkit Guide**.

Inside the Toolkit:

The workhorse for synthesis calibration is the `cb` tool.

4.1 Calibration Tasks

The standard set of `calibration` tasks are:

- `accum` — Accumulate incremental calibration solutions into a cumulative cal table (§ 4.7.2),
- `applycal` — Apply calculated calibration solutions (§ 4.10),
- `bandpass` — B calibration solving; supports pre-apply of other calibrations (§ 4.5),
- `clearcal` — Re-initialize visibility data set calibration data (§ 4.11),
- `fluxscale` — Bootstrap the flux density scale from standard calibration sources (§ 4.4),
- `gaincal` — G calibration solving; supports pre-apply of other calibrations (§ 4.3),

¹Hamaker, J.P., Bregman, J.D. & Sault, R.J. (1996), *Astronomy and Astrophysics Supplement*, v.117, p.137-147

- `listcal` — list calibration solutions (§ 4.9),
- `plotcal` — Plot calibration solutions (§ 4.8),
- `setjy` — Compute the model visibility for a specified source flux density (§ 4.2),
- `smoothcal`— Smooth calibration solutions derived from one or more sources (§ 4.7.1)
- `split`— Write out new MS containing calibrated data from a subset of the original MS (§ section:cal.split).

There are also more advanced and experimental calibration tasks available in this release:

- `blcal` — baseline-based G (or B) calibration; supports pre-apply of other calibrations (§ 4.13.2),
- `fringecal` — *Experimental*: baseline-based fringe-fitting calibration solving; supports pre-apply of other calibrations (§ 4.13.3),
- `uvcontsub`— uv-plane continuum fitting and subtraction (§ 4.13.1),
- `uvmodelfit`— Fit a component source model to the uv data (§ 4.13.4).

The following sections outline the use of these tasks in standard calibration processes.

Information on other useful tasks and parameter setting can be found in:

- `listobs` — list whats in a MS (§ 2.3),
- `plotxy` — X-Y plotting and editing (§ 3.4),
- `flagdata` — non-interactive data flagging (§ 3.5),
- data selection — general data selection syntax (§ 2.5).

4.2 Calibration models for absolute flux density (`setjy`)

When solving for visibility-plane calibration, CASA calibration applications compare the observed DATA column with the MODEL_DATA column. The first time that an imaging or calibration task is executed 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 `setjy` task is then used to set the proper flux density for flux calibrators. For sources that are recognized flux calibrators (listed in Table 4.1), `setjy` will calculate the 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).

The inputs for `setjy` are:

Table 4.1:

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

```
# setjy :: Place flux density of sources in the measurement set:
```

```
vis           =      ''      # Name of input visibility file
field         =      ''      # Field name list or field ids list
spw           =      ''      # Spectral window identifier (list)
modimage      =      ''      # Model image name
fluxdensity   =      -1      # Specified flux density [I,Q,U,V]
standard      = 'Perley-Taylor 99' # Flux density standard
```

By default the `setjy` task will cycle through all fields and spectral windows, setting the flux density either to 1 Jy (unpolarized), or if the source is recognized as one of the calibrators in the above table, to the flux density (assumed unpolarized) appropriate to the observing frequency. For example, to run `setjy` on a measurement set called `data.ms`:

```
setjy(vis='data.ms') # This will set all fields and spectral windows
```

To limit this operation to certain fields and spectral windows, use the `field` and/or `spw` parameters, which take the usual data selection strings (§ 2.5). For example, to set the flux density of the first field (all spectral windows)

```
setjy(vis='data.ms',field='0')
```

or to set the flux density of the second field in spectral window 17

```
setjy(vis='data.ms',field='1',spw='17')
```

The full-polarization flux density (I,Q,U,V) may also be explicitly provided:

```
setjy(vis='data.ms',
      field='1',spw='16', # Run setjy on field id 1, spw id 17
      fluxdensity=[3.5,0.2,0.13,0.0]) # and set I,Q,U,V explicitly
```

4.2.1 Dealing with resolved calibrators

If the flux density calibrator is resolved at the observing frequency, the point source model generated by `setjy` will not be appropriate. If available, a model image of the resolved source at the observing frequency may be used to generate the appropriate visibilities using the `modimage` parameter (or in older versions explicitly with the `ft` task).

Model images for some flux density calibrators are provided with CASA:

- RPM RHE4: located in `/usr/lib/casapy/data/nrao/VLA/CalModels`
- MAC OSX .dmg: located in `/opt/casa/data/nrao/VLA/CalModels`
- NRAO-AOC stable: `/home/casa/data/nrao/VLA/CalModels`
- NRAO-AOC daily: `/home/ballista/casa/daily/data/nrao/VLA/CalModels`

The models available are:

```
3C138_C.im/ 3C138_Q.im/ 3C147_K.im/ 3C286_C.im/ 3C286_Q.im/ 3C48_C.im/ 3C48_Q.im/
3C138_K.im/ 3C138_U.im/ 3C147_Q.im/ 3C286_K.im/ 3C286_U.im/ 3C48_K.im/ 3C48_U.im/
3C138_L.im/ 3C138_X.im/ 3C147_U.im/ 3C286_L.im/ 3C286_X.im/ 3C48_L.im/ 3C48_X.im/
```

These are all un-reconvolved images of AIPS CC lists, properly scaled to the Perley-Taylor 1999 flux density for the frequencies at which they were observed.

It is important that the model image *not* be one convolved with a finite beam; it must have units of Jy/pixel (not Jy/beam). Also, the frequency range of the image must cover the frequencies in the dataset. Finally, the amplitude scale in the image must be correct (beware of variation due to non-zero spectral index).

Copy the model image to the working directory; the following illustrates its use.

```
# Import the data
importvla(archivefiles='AS776_A031015.xp2', vis='ngc7538_XBAND.ms',
          freqtol=10000000.0, bandname='X')

# Flag the ACs
flagautocorr('ngc7538_XBAND.ms')

# METHOD 1: Use point source model for 3C48, plus uvrage in solve

# Use point source model for 3C48
setjy(field='0');

# Limit 3C48 (fieldid=0) solutions to uvrage = 0-40 klambda
gaincal(vis='ngc7538_XBAND.ms', caltable='cal.G', field='0',
        solint=60.0, refant=10, uvrage=[0,40],
        append=False, gaincurve=False, opacity=False)
```

```

# Append phase-calibrator's solutions (no uvrage) to the same table
gaincal(vis='ngc7538_XBAND.ms', caltable='cal.G', field='2',
        solint=60.0, refant=10, uvrage=[0], append=True,
        gaincurve=False, opacity=False)

# Fluxscale
fluxscale(vis='ngc7538_XBAND.ms', caltable='cal.G', reference=['0137+331'],
          transfer=['2230+697'], fluxtable='cal.Gflx', append=False)

# METHOD 2: use a resolved model copied from the data repository
#   for 3C48, and no uvrage
# (NB: detailed freq-dep flux scaling TBD)

setjy(field='0', modimage='3C48_X.im')

# Solutions on both calibrators with no uvrage
gaincal(vis='ngc7538_XBAND.ms', caltable='cal.G2', field='0,2',
        solint=60.0, refant=10, uvrage=[0],
        append=False, gaincurve=False, opacity=False)

# Fluxscale
fluxscale(vis='ngc7538_XBAND.ms', caltable='cal.G2', reference=['0137+331'],
          transfer=['2230+697'], fluxtable='cal.G2flx', append=False)

# Both methods give 2230 flux densities ~0.7 Jy, in good agreement with
#   AIPS

```

4.3 Complex Gain Calibration (gaincal)

The fundamental calibration to be done on your interferometer data is to calibrate the antenna-based gains as a function of time in the various frequency channels and polarization products. Some of these calibrations are known beforehand (“a priori”) and others must be determined from observations of calibrators, or from observations of the target itself (“self-calibration”).

The `gaincal` task has the following default inputs:

```

# gaincal :: Determine temporal gains from calibrator observations:

vis           =      '' # Name of input visibility file
caltable      =      '' # Name of output calibration table
field         =      '' # Select data based on field name or index
spw          =      '' # Select spectral window and channels (''=all)
selectdata    =      False # Activate data selection details
gaintype     =      'G' # Type of solution (G, T, or GSPLINE)
solint       =      0.0 # Solution interval (sec); 0 = scan
refant       =      '' # Reference antenna name or ID number

```

```

minsnr          =      0.0 # Reject solutions below this SNR
solnorm         =      False # Normalize solution amplitudes (G,T) post-solve.
append         =      False # Append to (existing) table. False==>overwrite
calmode        =      'ap' # Type of solution (a,p,ap)
append         =      False # Append solutions to (existing) table
gaintable      =      '' # Previous gain calibration solutions to apply
bptable        =      '' # Previous bandpass calibration solutions to apply
pointtable     =      '' # Previous pointing calibration solutions to apply
gaincurve      =      True # Apply VLA antenna gain curve correction
opacity        =      False # Apply an opacity correction (True/False)
preavg         =      -1.0 # Pre-averaging interval (sec)
async          =      False # if True run in the background, prompt is freed

```

Data selection is done through the standard field, spw and `selectdata` expandable sub-parameters (see § 2.5). Several other parameters are also expandable:

```

gaintype       = 'GSPLINE' # Type of solution (G, T, or GSPLINE)
  splintime    = 10800.0 # Spline timescale (sec), default=3 hours
  npointaver   = 10 # Points to average for phase wrap
  phasewrap    = 180 # Wrap phase when greater than this

```

and

```

opacity        =      True # Apply an opacity correction (True/False)
  tau          =      0.001 # Opacity value to apply (if opacity=True)

```

which can be used to pre-apply atmospheric opacity corrections (see below).

There are controls for applying previous calibration

```

gaincurve      =      True # Apply VLA antenna gain curve correction
opacity        =      False # Apply an opacity correction (True/False)
gaintable      =      '' # Previous gain calibration solutions to apply
bptable        =      '' # Previous bandpass calibration solutions to apply
pointtable     =      '' # Previous pointing calibration solutions to apply

```

ALPHA ALERT: These will likely eventually required to be pre-applied to a single input cal table using `accum`.

Data selection is done through a series of standard parameters. These are general among the various calibration tasks. These are described in § 2.5. For example, a typical simple selection might be:

```

field = '1331+305'
spw = '0:2~56'

```

to select channels 2–56 in spectral window 0 for the calibrator 1331+305. See § 2.5.3.1 for a description of the combined spectral window / channel selection syntax.

Setting `selectdata = True` expands the selection possibilities further:


```

selectdata      =      True      # Activate data selection details
  timerange     =      ''       # Select data based on time
  uvrange       =      ''       # Select data based on uv range
  antenna       =      ''       # Select data based on antenna/baseline
  scan          =      ''       # Select data based on scan number
  msselect      =      ''       # Optional data selection (see help)

```

In the current scheme, the `msselect` parameter is key to choosing which sources, fields, and antennas are included in the calibration. From the `help gaincal` in-line help:

```

msselect -- Optional data selection (field,spw,time,etc)
  default:'' means select all; example:msselect='FIELD_ID==0',
  msselect='FIELD_ID IN [0,1,2]' means select fields 0,1 and 2
  msselect='FIELD_ID <= 1' means select fields 0, 1
  msselect='FIELD_ID==0 && ANTENNA1 IN [0] && ANTENNA2 IN [2:26]'
    means select field 0 and antennas 0 to 26, except antenna 1.
  Other msselect fields are: 'DATA_DESC_ID', 'SPECTRAL_WINDOW_ID',
  'POLARIZATION_ID', 'SCAN_NUMBER', 'TIME', 'UVW'

```

4.3.1 “A priori” gain curve calibration

Gain curve calibration involves compensating for the effects of elevation on the amplitude of the received signals at each antenna. Antennas are not absolutely rigid, and so their effective collecting area and net surface accuracy vary with elevation as gravity deforms the surface. This calibration is especially important at higher frequencies where the deformations represent a greater fraction of the observing wavelength. By design, this effect is usually minimized (i.e., gain maximized) for elevations between 45 and 60 degrees, with the gain decreasing at higher and lower elevations. Gain curves are most often described as 2nd- or 3rd-order polynomials in zenith angle.

At this writing, gain curve calibration has been implemented in CASA for the VLA, with gain curve polynomial coefficients available directly from the CASA data repository. To make gain curve corrections for VLA data, set the `gaincurve` parameter to `True` for any of the calibration tasks, e.g.,:

```
gaincal('data.ms', 'cal.GO', gaincurve=True, solint=0., refant=11)
```

Note `gaincurve=True` is the default so this parameter can be omitted for VLA data:

```
gaincal('data.ms', 'cal.GO', solint=0., refant=11)
```

is equivalent. **NOTE: Set `gaincurve=False` if you are not using VLA data.**

The gain curve will be calculated per timestamp. Upon execution of a calibration task (e.g., `gaincal`, `bandpass`, `correct`, etc), the gain curve data appropriate to the observing frequencies will be automatically retrieved from the data repository and applied.

4.3.2 “A priori” atmospheric opacity correction

The troposphere is not completely transparent. At high radio frequencies (>15 GHz), water vapor and molecular oxygen begin to have a substantial effect on radio observations. According to the physics of radiative transmission, the effect is threefold. First, radio waves from astronomical sources are absorbed (and therefore attenuated) before reaching the antenna. Second, since a good absorber is also a good emitter, significant noise-like power will be added to the overall system noise. Finally, the optical path length through the troposphere introduces a time-dependent phase error. In all cases, the effects become worse at lower elevations due to the increased air mass through which the antenna is looking. In CASA, the opacity correction described here compensates only for the first of these effects, tropospheric attenuation, using a plane-parallel approximation for the troposphere to estimate the elevation dependence.

Opacity corrections are a component of calibration type 'T'. To make opacity corrections in CASA, an estimate of the zenith opacity is required (see observatory-specific chapters for how to measure zenith opacity). With a value for zenith opacity in hand (0.1 nepers, say), use the following parameters:

```
gaincal('data.ms', 'cal.G0', solint=0., refant=11, opacity=True, tau=0.1)
```

The calibration task in this example will apply an elevation-dependent opacity correction (scaled to 0.1 nepers at the zenith for all antennas for this example) calculated at each scan (`solint=0`). Set `solint=-1` instead to get a solution every timestamp.

Generalizations to antenna- and time-dependent opacities, including derivation (from weather information) and solving (directly from the visibility data) capabilities, will be made available in the future.

4.3.2.1 Determining opacity corrections for VLA data

For VLA data, zenith opacity can be measured at the frequency and during the time observations are made using a VLA tipping scan in the observe file. Historical tipping data are available at:

<http://www.vla.nrao.edu/astro/calib/tipper>

Choose a year, and click **Go** to get a list of all tipping scans that have been made for that year.

If a tipping scan was made for your observation, then select the appropriate file. Go to the bottom of the page and click on the button that says **Press here to continue..** The results of the tipping scan will be displayed. Go to the section called 'Overall Fit Summary' to find the fit quality and the fitted zenith opacity in percent. If the zenith opacity is reported as 6%, then the actual zenith opacity value is `tau=0.060` for `gaincal`.

4.3.3 Other *a priori* Calibrations and Corrections

Other *a priori* calibrations will be added to the calibrator tool in the near future. These will include antenna-position (phase) corrections, system temperature normalization (amplitude) corrections,

tropospheric phase corrections derived from Water Vapor Radiometry (WVR) measurements, instrumental line-length corrections, etc. Where appropriate, solving capabilities for these effects will also be added.

4.3.4 Polarization-dependent Gain (G)

Systematic time-dependent complex gain errors are almost always the dominant calibration effect, and a solution for them is almost always necessary before proceeding with any other calibration. Traditionally, this calibration type has been a catch-all for a variety of similar effects, including: the relative amplitude and phase gain for each antenna, phase and amplitude drifts in the electronics of each antenna, amplitude response as a function of elevation (gain curve), and tropospheric amplitude and phase effects. In CASA, it is possible to handle many of these effects separately, as available information and circumstances warrant, but it is still possible to solve for the net effect using calibration type G.

Generally speaking, type G can represent any per-spectral window multiplicative polarization- and time-dependent complex gain effect downstream of the polarizers. (Polarization *independent* effects *upstream* of the polarizers may also be treated with G.) Multi-channel data (per spectral window) will be averaged in frequency before solving (use calibration type B to solve for frequency-dependent effects within each spectral window).

To solve for G on, say, fields 1 & 2, on a 90s timescale, and apply, e.g., gain curve corrections:

```
gaincal('data.ms',
        caltable='cal.G',          # Write solutions to disk file 'cal.G'
        field='0,1',              # Restrict field selection
        solint=90,                # Solve for phase and amp on a 90s timescale
        refant=3)                 #
                                  # Note: gaincurve=True by default

plotcal('cal.G','amp')           # Inspect solutions
```

These G solution will be referenced to antenna 4. Choose a well-behaved antenna that is located near the center of the array for the reference antenna. For non-polarization datasets, reference antennas need not be specified although you can if you want. If no reference antenna is specified, an effective phase reference that is an average over the data will be calculated and used. For data that requires polarization calibration, you must choose a reference antenna that has a constant phase difference between the right and left polarizations (e.g. no phase jumps or drifts). If no reference antenna (or a poor one) is specified, the phase reference may have jumps in the R-L phase, and the resulting polarization angle response will vary during the observation, thus corrupting the polarization imaging.

To apply this solution to the calibrators and the target source (field 2, say):

```
correct('data.ms',
        field='0,1,2',           # Restrict field selection (cals + src)
        opacity=False,          # Don't apply opacity correction
```

```

    gaintable='cal.G')      # Apply G solutions and correct data
                           # (written to the CORRECTED_DATA column)
                           # Note: calwt=True by default
plotxy('data.ms',xaxis='channel',datacolumn='data',subplot=211)
plotxy('data.ms',xaxis='channel',datacolumn='corrected',subplot=212)

```

4.3.5 Polarization-independent Gain (T)

At high frequencies, it is often the case that the most rapid time-dependent gain errors are introduced by the troposphere, and are polarization-independent. It is therefore unnecessary to solve for separate time-dependent solutions for both polarizations, as is the case for G. Calibration type T is available to calibrate such tropospheric effects, differing from G only in that a single common solution for both polarizations is determined. In cases where only one polarization is observed, type T is adequate to describe the time-dependent complex multiplicative gain calibration.

In the following example, we assume we have a 'G' solution obtained on a longish timescale (longer than a few minutes, say), and we want a residual T solution to track the polarization-independent variations on a very short timescale:

```

gaincal('data.ms',          # Visibility dataset
        caltable='cal.T',   # Specify output table name
        gaintype='T',      # Solve for T
        field='0,1',       # Restrict data selection to calibrators
        solint=3.,         # Obtain solutions on a 3s timescale
        gaintable='cal120.G') # Pre-apply prior G solution

```

For dual-polarization observations, it will always be necessary to obtain a G solution to account for differences and drifts between the polarizations (which traverse different electronics), but solutions for rapidly varying polarization-independent effects such as those introduced by the troposphere will be optimized by using T. Note that T can be used in this way for self-calibration purposes, too.

4.3.6 GSPLINE solutions

At high radio frequencies, where tropospheric phase fluctuates rapidly, it is often the case that there is insufficient SNR to obtain a robust G or T solution on timescales short enough to track the variation. In this case it is desirable to solve for a best-fit functional form for each antenna using the GSPLINE solver. The GSPLINE solver fits time-series of cubic B-splines to the phase and/or amplitude of the calibrator visibilities. Unlike ordinary G, a single common GSPLINE solution will be determined from data for all selected spectral windows and fields specified in `msselect`, and the resulting solution will be applicable to any field or spectral window in the same Measurement Set. An important consequence of this is that all fields used to obtain a GSPLINE amplitude solution must have models with accurate relative flux densities. (Use of incorrect relative flux densities will introduce spurious variations in the GSPLINE amplitude solution.)

The GSPLINE solver requires a number of unique additional parameters, compared to ordinary G and T solving.

```

gaintype      = 'GSPLINE' # Type of solution (G, T, or GSPLINE)
  splinetime   = 10800.0 # Spline timescale (sec), default=3 hours
  npointaver   = 10 # Points to average for phase wrap
  phasewrap    = 180 # Wrap phase when greater than this

```

The duration of each spline segment is controlled by `splinetime`. The actual `splinetime` will be adjusted such that an integral number of equal-length spline segments will fit within the overall range of data.

Phase splines require that cycle ambiguities be resolved prior to the fit; this operation is controlled by `npointaver` and `phasewrap`. The `npointaver` parameter controls how many contiguous points in the time-series are used to predict the cycle ambiguity of the next point in the time-series, and `phasewrap` sets the threshold phase jump (in degrees) that would indicate a cycle slip. Large values of `npointaver` improve the SNR of the cycle estimate, but tend to frustrate ambiguity detection if the phase rates are large. The `phasewrap` parameter may be adjusted to influence when cycles are detected. Generally speaking, large values ($> 180^\circ$) are useful when SNR is high and phase rates are low. Smaller values for `phasewrap` can force cycle slip detection when low SNR conspires to obscure the jump, but the algorithm becomes significantly less robust. More robust algorithms for phase-tracking are under development (including fringe-fitting).

To solve for GSPLINE phase and amplitudes, with splines of duration 600s:

```

gaincal('data.ms',
  caltable='cal.spline.ap',
  gaintype='GSPLINE' # Solve for GSPLINE
  calmode='ap' # Solve for amp & phase
  field='0,1', # Restrict data selection to calibrators
  splinetime=600.) # Set spline timescale to 10min

```

The GSPLINE solutions can not yet be plotted using `plotcal`.

4.4 Flux density scale calibration

The 'G' or 'T' solutions obtained from calibrators for which the flux density was unknown and assumed to be 1 Jy are correct in a time- and antenna- relative sense, but are mis-scaled by a factor equal to the inverse of the square root of the true flux density. This scaling can be corrected by enforcing the constraint that mean gain amplitudes determined from calibrators of unknown flux density should be the same as determined from those with known flux densities. The `fluxscale` task exists for this purpose.

The inputs for `fluxscale` are:

```

vis           =      '' # Name of input visibility file
caltable     =      '' # Name of input calibration table
fluxtable    =      '' # Name of output, flux-scaled calibration table
reference     =      '' # Reference field name (transfer flux scale from)
transfer     =      '' # Transfer field name(s)
append       =      False # Append solutions?
refspwmap    =      [-1] # Scale across spectral window boundaries

```

Before running `fluxscale`, one will have first `setjy` for the `reference` sources and run a `gaincal` on both `reference` and `transfer` fields. After running `fluxscale` the output `fluxtable` `caltable` will have been scaled such that the correct scaling will be applied to the `transfer` sources.

Given a 'G' table, 'cal.G', containing solutions for a flux density calibrator (3C286, say) and for one or more random calibrator sources with unknown flux densities (0234+285 and 0323+022, say) use the `fluxscale` task as follows:

```

fluxscale(vis='data.ms',
          caltable='cal.G',           # Select input table
          fluxtable='cal.Gflx',      # Write scaled solutions to cal.Gflx
          reference='3C286',         # 3C286 = flux calibrator
          transfer='0234+258, 0323+022') # Select calibrators to scale

```

The output table, 'cal.Gflx', contains solutions that are properly scaled for all calibrators.

Note that the assertion that the gain solutions are independent of the calibrator includes the assumption that the gain amplitudes are strictly not systematically time dependent. While synthesis antennas are designed as much as possible to achieve this goal, in practice, a number of effects conspire to frustrate it. When relevant, it is advisable to pre-apply gain curve and opacity corrections when solving for the 'G' solutions that will be fluxscaled. When the G solutions are essentially constant for each calibrator separately, the `fluxscale` operation is likely to be robust.

The `fluxscale` task can be executed on either 'G' or 'T' solutions, but it should only be used on one of these types if solutions exist for both. GSPLINE solutions do not yet support `fluxscale`.

If the reference and transfer fields were observed in different spectral windows, the `refspwmap` parameter may be used to achieve the scaling calculation across spectral window boundaries. The `refspwmap` parameter takes a vector of indices indicating the spectral window mapping for the reference fields, such that `refspwmap[i]=j` means that reference field amplitudes from spectral window `j` will be used for spectral window `i`.

```

fluxscale(vis='data.ms',
          caltable='cal.G',           # Select input table
          fluxtable='cal.Gflx',      # Write scaled solutions to cal.Gflx
          reference='3C286',         # 3C286 = flux calibrator
          transfer='0234+258,0323+022' # Select calibrators to scale
          refspwmap=[0,0,0])         # Use spwid 0 scaling for spwids 1 & 2

```

will use `spw=0` to scale the others, while in

```
fluxscale(vis='data.ms',
          caltable='cal.G',           # Select input table
          fluxtable='cal.Gflx',      # Write scaled solutions to cal.Gflx
          reference='3C286',         # 3C286 = flux calibrator,
          transfer='0234+285, 0323+022', # select calibrators to scale,
          refspwmap=[0,0,1,1])      # select spwids for scaling,
```

the reference amplitudes from spectral window 0 will be used for spectral windows 0 and 1 and reference amplitudes from spectral window 2 will be used for spectral windows 2 and 3.

4.4.1 Resolved flux density calibrators

If the flux density calibrator is resolved, the assumption that it is a point source will cause solutions on outlying antennas to be biased in amplitude. In turn, the flux-density scaling step will be biased on these antennas as well. In general, it is best to use model for the calibrator, but if such a model is not available, it is important to limit the solution on the flux density calibrator to only the subset of antennas that have baselines short enough that the point-source assumption is valid. This can be done by using antenna and uvrange selection when solving for the flux density calibrator. For example, if antennas 1 through 8 are the antennas among which the baselines are short enough that the point-source assumption is valid, and we want to be sure to limit the solutions to the use of baselines shorter than 15000 wavelengths, then we can assemble properly scaled solutions for the other calibrator as follows (note: specifying both an antenna and a uvrange constraint prevents inclusion of antennas with only a small number of baselines within the specified uvrange from being included in the solution; such antennas will have poorly constrained solutions):

As an example, we first solve for gain solutions for the flux density calibrator (3C286 observed in field 1) using a subset of antennas

```
gaincal(vis='data.ms',
        caltable='cal.G',           # write solutions to cal.G
        field='0'                  # Select the flux density calibrator
        selectdata=True,          # Expand other selectors
        antenna='0~7',            # antennas 0-7,
        uvrange='0~15kl',        # limit uvrange to 0-15klambda
        solint=90)                # on 90s timescales, write solutions
                                  # to table called cal.G
```

Now solve for other calibrator (0234+285 in field 2) using all antennas (implicitly) and append these solutions to the same table

```
gaincal(vis='data.ms',
        caltable='cal.G',           # write solutions to cal.G
        field='1',
        solint=90,
        append=T)                  # Set up to write to the same table
```

Finally, run fluxscale to adjust scaling

```
fluxscale(vis='data.ms',
          caltable='cal.G',      # Input table with unscaled cal solutions
          fluxtable='cal.Gflx', # Write scaled solutions to cal.Gflx
          reference='3C286',    # Use 3c286 as ref with limited uvrage
          transfer='0234+285') # Transfer scaling to 0234+285
```

The fluxscale calculation will be performed using only the antennas common to both fields, but the result will be applied to all antennas on the transfer field.

4.5 Spectral Bandpass Calibration (bandpass)

For channelized data, it is often desirable to solve for the gain variations in frequency as well as in time. Variation in frequency arises as a result of non-uniform filter passbands or other dispersive effects in signal transmission. It is usually the case that these frequency-dependent effects vary on timescales much longer than the time-dependent effects handled by types 'G' and 'T'. Thus, it makes sense to solve for them as a separate term: 'B', using the `bandpass` task.

The inputs to `bandpass` are:

```
CASA <42>: inp('bandpass')
vis          =      '' # Name of input visibility file (MS)
caltable     =      '' # Name of output bandpass calibration table
field        =      '' # Select data based on field name or index
spw          =      '' # Select data based on spectral window
mode         =      'none' # Frequency data selection (none,channel,velocity)
selectdata   =      True # Activate data selection details
  timerange  =      '' # Select data based on time
  uvrange    =      '' # Select data based on uv range
  antenna    =      '' # Select data based on antenna/baseline
  scan       =      '' # Select data based on scan number
  msselect   =      '' # Optional data selection (see help)

solint       =      864000.0 # Solution interval (sec)
refant       =      '' # Reference antenna name (not ID number)
solnorm      =      False # Normalize bandpass amplitudes and phases
bandtype     =      'B' # Type of bandpass solution (B or BPOLY)
append       =      False # Append solutions to (existing) table
visnorm      =      False # Normalize data prior to BPOLY solution
gaincurve    =      False # Apply VLA antenna gain curve correction
opacity      =      False # Apply an opacity correction (True/False)
gaintable    =      '' # Previous gain calibration solutions to apply
gainselect   =      '' # Select subset of calibration solutions from gaintable
bptable      =      '' # Previous bandpass calibration solutions to apply
pointtable   =      '' # Previous pointing calibration solutions to apply
async        =      False # if True run in the background, prompt is freed
```

There are two non-selection expandable parameters:


```

bandtype      = 'BPOLY' # Type of bandpass solution (B or BPOLY)
  degamp      = 3 # Polynomial degree for BPOLY amplitude solution
  degphase    = 3 # Polynomial degree for BPOLY phase solution
  maskcenter  = 0 # Number of channels in BPOLY to avoid in center of band
  maskedge    = 0 # Percent of channels in BPOLY to avoid at each band edge

```

and

```

opacity       = True # Apply an opacity correction (True/False)
  tau         = 0.001 # Opacity value to apply (if opacity=True)

```

As in `gaincal` (§ 4.3), there are the standard selection parameters (§ 2.5),

```

field         = '' # Select data based on field name or index
spw           = '' # Select data based on spectral window
selectdata    = True # Activate data selection details
  timerange   = '' # Select data based on time
  uvrange     = '' # Select data based on uv range
  antenna     = '' # Select data based on antenna/baseline
  scan        = '' # Select data based on scan number
  msselect    = '' # Optional data selection (see help)

```

There are also controls for applying previous calibration

```

gaincurve     = True # Apply VLA antenna gain curve correction
opacity       = False # Apply an opacity correction (True/False)
gaintable     = '' # Previous gain calibration solutions to apply
gainselect    = '' # Select subset of calibration solutions from gaintable
bptable       = '' # Previous bandpass calibration solutions to apply
pointtable    = '' # Previous pointing calibration solutions to apply

```

Note that `opacity` will expand to input `tau` if set to `True`. **ALPHA ALERT:** These will likely eventually required to be pre-applied to a single input cal table using `accum`.

4.5.1 B solutions

Calibration type 'B' differs from 'G' only in that it is determined for each channel in each spectral window. It is possible to solve for it as a function of time, but it is most efficient to keep the 'B' solving timescale as long as possible, and use 'G' or 'T' for rapid timescale variations.

'B' solutions are limited by the SNR available per channel, which may be quite small. It is therefore important that the data be coherent over the time-range of the 'B' solutions. As a result, 'B' solutions are almost always preceded by a 'G' or 'T' solve. In turn, if the 'B' solution improves the frequency domain coherence significantly, a 'G' or 'T' solution following it will be better than the original.

To solve for 'B' (on a very long timescale, i.e., constant for an observation shorter than 1 day), using a prior G solution:

```
bandpass(vis='data.ms',          # input data set
         caltable='cal.B',      #
         spw='0:2~56',         # Use channels 2-56 (avoid end channels)
         field='0',            # Select bandpass calibrator (field 0)
         gaintable='cal.G',    # Pre-apply gain solutions derived previously
         solint=86400.,        # Setup a long timescale (assumes bandpass
         refant='14')          # is constant over this length).
```

Note that the solution has only been obtained for the subset of 55 channels starting with channel 2. Explicit channel selection like this is only necessary if it is desired that some channels be avoided (e.g., end channels that may not be well-behaved). The default is to obtain a solution for every channel.

4.5.2 BPOLY solutions

For some observations, it may be the case that the SNR per channel is insufficient to obtain a usable per-channel 'B' solution. In this case it is desirable to solve instead for a best-fit functional form for each antenna using the BPOLY solver. The 'BPOLY' solver fits polynomials to the amplitude and phase of the calibrator visibilities as a function of frequency. Unlike ordinary 'B', a single common BPOLY solution will be determined for all spectral windows specified (or implicit) in the `setdata` execution. As such, it is usually most meaningful to select individual spectral windows for BPOLY solves, unless groups of adjacent spectral windows are known *a priori* to share a single continuous bandpass response over their combined frequency range (e.g., PdBI data). Currently, BPOLY solutions cannot be solved for in a time-dependent manner.

The 'BPOLY' solver requires a number of unique parameters:

```
bandtype      = 'BPOLY' # Type of bandpass solution (B or BPOLY)
  degamp      = 3 # Polynomial degree for BPOLY amplitude solution
  degphase    = 3 # Polynomial degree for BPOLY phase solution
  maskcenter  = 0 # Number of channels in BPOLY to avoid in center of band
  maskedge   = 0 # Percent of channels in BPOLY to avoid at each band edge
```

The `degamp` and `degphase` parameters indicate the polynomial degree desired for the amplitude and phase solutions. The `maskcenter` parameter is used to indicate the number of channels in the center of the band to avoid passing to the solution (e.g., to avoid Gibbs ringing in central channels for PdBI data).

For example, to solve for a BPOLY (5th order in amplitude, 7th order in phase), using data from field 2, with G corrections pre-applied:

```
bandpass(vis='data.ms',          # input data set
         caltable='cal.BPOLY',   #
         spw='0:2~56',         # Use channels 3-57 (avoid end channels)
         field='0',            # Select bandpass calibrator (field 0)
         bandtype='BPOLY',     # Select bandpass polynomials
         degamp=5,             # 5th order amp
```

```

    degphase=7,          # 7th order phase
    gaintable='cal.G',  # Pre-apply gain solutions derived previously
    refant='14')        #

```

Note that all available spectral windows will be used to obtain a single solution spanning them all. If separate solutions for each spectral window are desired, solve for each separately, e.g., if there are 3 spectral windows (0,1,2):

```

bandpass(vis='data.ms',
         caltable='cal.BPOLY.0',
         spw='0:2~56',
         field='0',
         bandtype='BPOLY',
         degamp=5,
         degphase=7,
         gaintable='cal.G',
         refant='14')

```

```

bandpass(vis='data.ms',
         caltable='cal.BPOLY.1',
         spw='1:2~56',
         bandtype='BPOLY',
         degamp=5,
         degphase=7,
         gaintable='cal.G',
         refant='14')

```

```

bandpass(vis='data.ms',
         caltable='cal.BPOLY.2',
         spw='2:2~56',
         field='0',
         bandtype='BPOLY',
         degamp=5,
         degphase=7,
         gaintable='cal.G',
         refant='14')

```

Each solution is stored in a separate table. As a result, subsequent calibration operations must be undertaken for each spectral window separately.

The BPOLY solutions can not yet be plotted using `plotcal`.

4.6 Instrumental Polarization Calibration (D)

The `polcal` task has not yet been created. You can use the toolkit to do polarization calibration if necessary, or wait for us to catch up.

4.7 Manipulating Calibration Tables

Calibration tables can be manipulated in various ways, such as by interpolating between times (and sources), smoothing of solutions, and accumulating various separate calibrations into a single table.

4.7.1 Calibration Smoothing (smoothcal)

ALPHA ALERT: This task may be called `smooth` in older patches.

The `smoothcal` task will smooth calibration solutions (most usefully G or T) over a longer time interval to reduce noise and outliers. The inputs are:

```
vis           =      ''      # Name of input visibility file
tablein      =      ''      # Input calibration table
caltable     =      ''      # Output calibration table
field        =      ''      # Field name list
smoothtype   =      'median' # Smoothing filter to use
smoothtime   =      60.0     # Smoothing time (sec)
async        =      False    # if True run in the background, prompt is freed
```

The smoothing will use the `smoothtime` and `smoothtype` parameters to determine the new data points which will replace the previous points on the same time sampling grid as for the `tablein` solutions. The currently supported `smoothtype` options:

1. `'mean'` — use the mean of the points within the window defined by `smoothtime` (a “boxcar” average),
2. `'median'` — use the median of the points within the window defined by `smoothtime` (most useful when many points lie in the interval).

Note that `smoothtime` defines the width of the time window that is used for the smoothing.

An example using the `smoothcal` task to smooth an existing table:

```
gaincal(vis='ngc5921.ms',
        caltable='ngc5921_05s.gcal',
        spw='0:2~56',
        field='0,1',
        solint=0.5) #
                  # Note: we use the defaults for other parameters

plotcal('ngc5921_05s.gcal','amp') # Plot the amplitudes

smoothcal(vis='ngc5921.ms',
          tablein='ngc5921_05s.gcal', # input calibration table
          caltable='ngc5921_sm.gcal', # output calibration table (smoothed)
          smoothtime=1000.)          # use 1000 seconds for the smoothing time and
                                     # the default smoothtype='median'

plotcal('ngc5921_sm.gcal','amp') # Plot the smoothed amplitudes
```

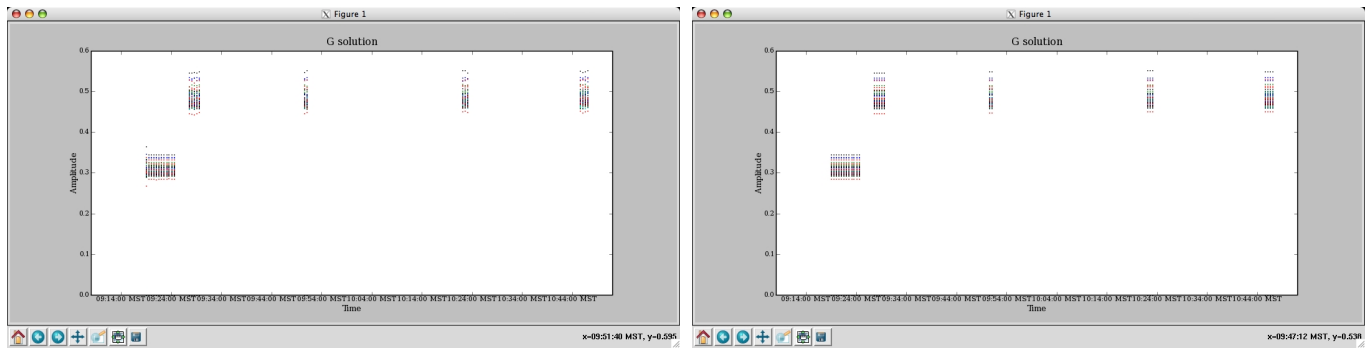


Figure 4.1: plotcal: Display of the amplitude solutions for short solution interval table (0.5 seconds: top) and the smoothed table using a smoothtime of 1000 seconds.

4.7.2 Calibration Interpolation and Accumulation (`accum`)

The `accum` task is used to interpolate calibration solutions onto a different time grid, or to *accumulate* incremental calibrations into a *cumulative* calibration table.

Its inputs are:

```

vis           =      '' # Name of input visibility file (MS)
tablein      =      '' # Input (cumulative) calibration table
incrtable    =      '' # Input incremental calibration table
caltable     =      '' # Output (cumulative) calibration table
field        =      '' # List of field names to process from tablein
calfield     =      '' # List of field names to use from incremental table
interp       =      'linear' # Interpolation mode to use on incremental
accumtime    =      -1.0 # Cumulative table timescale when creating
spwmap       =      [-1] # Spectral window combinations to apply

```

We now describe the two uses of `accum`.

4.7.2.1 Interpolation using (`accum`)

Calibration solutions (most notably G or T) must be interpolated onto the timestamps of the science target observations. Currently, the time-dependent interpolation options available for specification in the `interp` parameter are:

1. `'nearest'` — apply the calibration factor nearest in time to each datum,
2. `'linear'` — apply to each datum an amplitude and phase (separately) linearly interpolated or extrapolated from the two nearest (in time) calibration solutions,

3. `'aipslin'` — emulates the on-the-fly calibration interpolation in classic AIPS, with amplitude interpolated linearly (as in `'linear'`), and phase interpolated from linear interpolation of the real and imaginary parts.

For most purposes, the `'linear'` option should suffice.

The `'linear'` and `'aipslin'` options differ in how they treat phase interpolation. Using the `linear` option, it is assumed that there is no phase cycle ambiguity to consider, i.e., the direction of the smaller phase difference (necessarily always < 180 degrees) between the two solutions is considered the correct direction for interpolation. The `aipslin` option avoids the complication of determining the minimum phase ambiguity, but the result is decidedly non-linear in phase for interpolations over more than a few 10s of degrees. As the phase difference between interpolating solutions approaches 180 degrees, `aipslin` tends toward `nearest` for the phase interpolation.

An example using `accum` to interpolate an existing table onto a new time grid.

```
accum(vis='ngc5921.ms',          #
      incrtable='ngc5921.gcal',  # this is the table that will be interpolated
      caltable='ngc5921_20s.gcal', # this will be the interpolated output table
      interp='linear',          # linear interpolation
      accumtime=20.0)          # timescale for sampling the output table
```

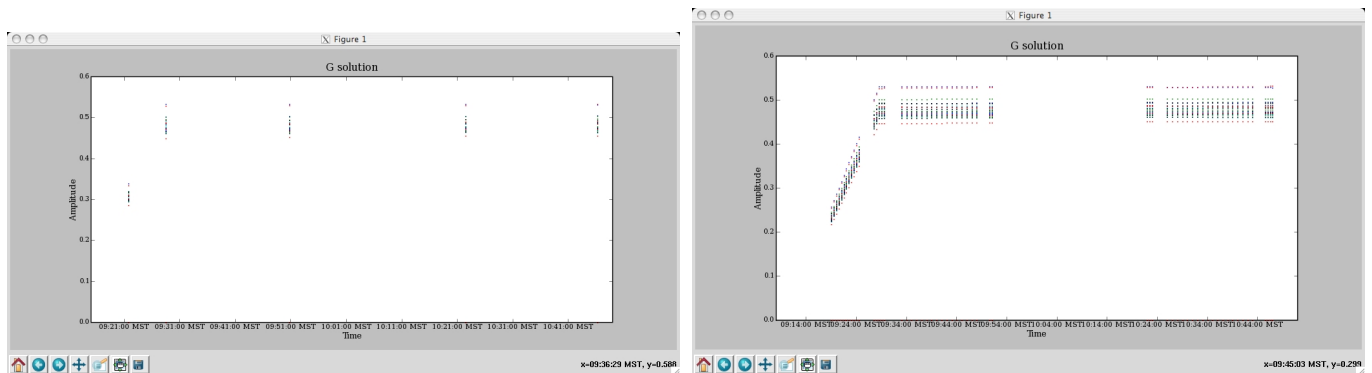


Figure 4.2: `plotcal`: Display of the amplitude solutions for NGC 5921; original (left), interpolated solutions-20s sampling (right).

4.7.2.2 Incremental Calibration using (`accum`)

It is occasionally desirable to solve for and apply calibration incrementally. This is the case when a calibration table of a certain type already exists (from a previous solve), an incremental solution *of the same type and relative to the first* is required, and it is not possible to recover the cumulative solution by a single solve.

Much of the time, it is, in fact, possible to recover the cumulative solution. This is because the equation describing the solution for the incremental solution (using the original solution), and that describing the solution for their product are fundamentally the same equation—the cumulative solution, if unique, must always be the same no matter what initial solution is. One circumstance where an incremental solution is necessary is the case of *phase-only* self-calibration relative to a full amplitude and phase calibration already obtained (from a different field).

For example, a phase-only “G” self-calibration on a target source may be desired to tweak the full amplitude and phase “G” calibration already obtained from a calibrator. The initial calibration (from the calibrator) contains amplitude information, and so must be carried forward, yet the phase-only solution itself cannot (by definition) recover this information, as a full amplitude and phase self-calibration would. In this case, the initial solution must be applied while solving for the phase-only solution, then the two solutions combined to form a cumulative calibration embodying the net effect of both. In terms of the Measurement Equation, the net calibration is the product of the initial and incremental solutions.

The analog of accumulate in classic AIPS is the use of CLCAL to combine a series of (incremental) SN calibration tables to form successive (cumulative) CL calibration tables.

Cumulative calibration tables also provide a means of generating carefully interpolated calibration, on variable user-defined timescales, that can be examined prior to application to the data with `correct`. The solutions for different fields and/or spectral windows can be interpolated in different ways, with all solutions stored in the same table.

The only difference between incremental and cumulative calibration tables is that incremental tables are generated directly from the calibration solving tasks (`gaincal`, `bandpass`, etc), and cumulative tables are generated from other cumulative and incremental tables via `accum`. In all other respects (internal format, application to data with `correct`, plotting with `plotcal`, etc.), they are the same, and therefore interchangeable. Thus, accumulate and cumulative calibration tables need only be used when circumstances require it.

The `accum` task represents a generalization on the classic AIPS CLCAL model of cumulative calibration in that its application is not limited to accumulation of “G” solutions (SN/CL tables in classic AIPS are the analog of “G” (and, implicitly, “T”) in `aips++`). In principle, any basic calibration type can be accumulated (onto itself), as long as the result of the accumulation (matrix product) is of the same type. This is true of all the basic types, except “D”. Accumulation is currently supported for “B”, “G”, and “T”, and, in future, “F” (ionospheric Faraday rotation), “J” (generic full-polarization calibration), fringe-fitting, and perhaps others. Accumulation of certain specialized types (e.g., “GSPLINE”, “TOPAC”, etc.) onto the basic types will be supported in the near future. The treatment of various calibration from ancillary data (e.g., system temperatures, weather data, WVR, etc.), as they become available, will also make use of accumulate to achieve the net calibration.

Note that accumulation only makes sense if treatment of a uniquely incremental solution is required (as described above), or if a careful interpolation or sampling of a solution is desired. In all other cases, re-solving for the type in question will suffice to form the net calibration of that type. For example, the product of an existing “G” solution and an amplitude and phase “G” self-cal (solved with the existing solution applied), is equivalent to full amplitude and phase “G” selfcal (with no

prior solution applied), as long as the timescale of this solution is at least as short as that of the existing solution.

The `tablein` parameter is used to specify the existing cumulative calibration table to which an incremental table is to be applied. Initially, no such table exists, and `accumulate` will generate one from scratch (on-the-fly), using the timescale (in seconds) specified by the parameter `accumtime`. These nominal solutions will be unit-amplitude, zero-phase (i.e., unit matrix) calibration, ready to be adjusted by accumulation according to the settings of other parameters. When `accumtime` is negative (the default), the table name specified in `tablein` must exist and will be used.

The `incrtable` parameter is used to specify the incremental table that should be applied to `tablein`. The calibration type of `incrtable` sets the type assumed in the operation, so `tablein` (if specified) must be of the same type. If it is not, `accum` will exit with an error message. (Certain combinations of types and subtypes will be supported by `accum` in the future.)

The `caltable` parameter is used to specify the name of the output table to write. If un-specified (or “”), then `tablein` will be overwritten. Use this feature with care, since an error here will require building up the cumulative table from the most recent distinct version (if any).

The `field` parameter specifies those field names in `tablein` to which the incremental solution should be applied. The solutions for other fields will be passed to `caltable` unaltered. If the cumulative table was created from scratch in this run of `accumulate`, then the solutions for these other fields will be unit-amplitude, zero-phase, as described above.

The `calfield` parameter is used to specify the fields to select from `incrtable` to use when applying to `tablein`. Together, use of `field` and `calfield` permit completely flexible combinations of calibration accumulation with respect to fields. Multiple runs of `accum` can be used to generate a single table with many combinations. In future, a “self” mode will be enabled that will simplify the accumulation of field-specific solutions.

The `interp` parameter is used to specify the interpolation type to use on the incremental solutions. The currently available interpolation types are “nearest”, “linear”, and “aipslin”.

Here is an example:

```
# obtain G solutions from calibrator
gaincal(vis='ap366.sim',
        caltable='cal.G0',
        field='9,11',
        solint=300);

# obtain proper flux density scale
fluxscale (vis='ap366.sim',
          caltable='cal.G0',
          fluxtable='cal.G1',
          reference='1328+307')

# generate cumulative table for target source on 20s timescale
accum(vis='ap366.sim',
      tablein='',
```



```

    incrtable='cal.G1',
    caltable='cal.cG0',
    field='0957+561',      # calibrate target...
    calfield='0917+624',  # ...with calibrator
    interp='linear',
    accumtime=20)

# apply this calibration to target
correct(vis='ap366.sim',
        gaintable='cal.cG0')

# (image target with imager tool)

# phase-selfcal target on (faster) 60s timescale
gaincal(vis='ap366.sim',
        caltable='cal.G2',
        field='10',
        calmode='p',      # phase-only
        solint=60,
        gaintable='cal.cG0');

# accumulate new solution onto existing one
accum(vis='ap366.sim',
      tablein='cal.cG0',      # existing cumulative (input)
      incrtable='cal.G2',    # new incremental (input)
      caltable='cal.cG1',    # new cumulative (output)
      field='0957+561',     # calibrate target...
      calfield='0957+561',  # ...with its own solutions
      interp='linear')

# apply new cumulative solution to data
correct(vis='ap366.sim',
        gaintable='cal.cG1')

# (another round of imaging, etc.)

```

4.8 Plotting Calibration Solutions (plotcal)

The `plotcal` task is available for examining solutions of all of the basic solvable types (G, T, B, D, M, MF, K). The inputs are:

```

tablein      =      '' # Name of input calibration table
yaxis        =      'amp' # Value to plot along yaxis (amp, phase, delay, delayrate)
poln         =      '' # Polarization to plot (RL,R,L,XY,X,Y,R/L,R-L)
field        =      '' # Field selection
baseline     =      '' # Antenna selection
spw          =      '' # Spectral Window selection
subplot      =      111 # Panel number on display screen (yxn)
multiplot    =      False # Plot data on separate plots

```

```

plotsymbol      =      '.' #  pylab plot symbol
overplot       =      False #  Overplot data display on existing display

```

The controls for the `plotcal` window are the same as for `plotxy` (see § 3.4.1).

The plot types available are:

- 'amp' — amplitude,
- 'phase' — phase,
- 'delay' — the phase delay,
- 'delayrate' — the phase delay rate.

while `poln` determines what polarization or combination of polarization is being plotted. The `poln='RL'` plots both R and L polarizations on the same plot, while the `poln='R-L'` and `poln='R+L'` options do the same thing and amplitude ratios or phase differences between R and L. The respective XY options do equivalent things. The `field`, `spw`, and `antenna` selection parameters are available to obtain plots of subsets of solutions. (**ALPHA ALERT:** `antenna` may be named `baseline` in older versions.)

For example, to plot amplitude or phase as a function of time for G solutions:

```

plotcal('cal.G', 'amp')
plotcal('cal.G', 'phase')

```

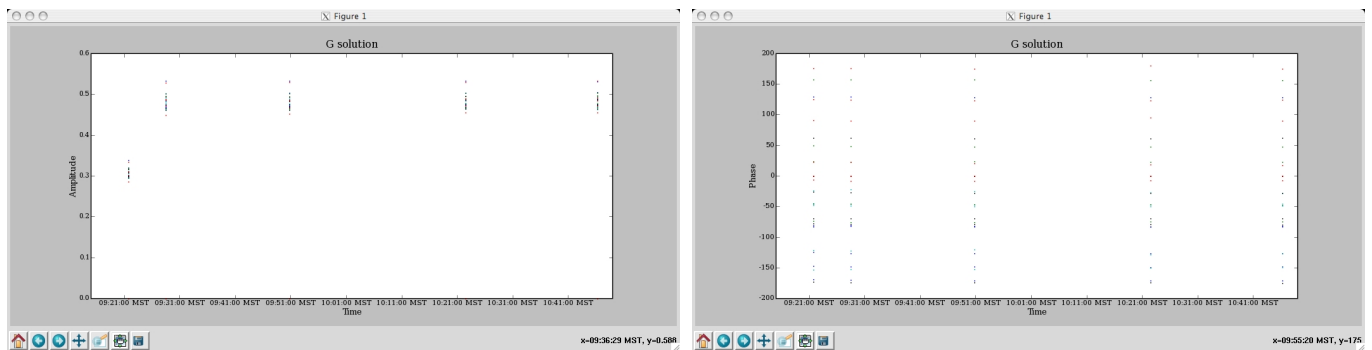


Figure 4.3: `plotcal`: Display of the amplitude and phase gain solutions (for all data).

Similarly, to plot amplitude or phase as a function of channel for B solutions:

```

plotcal('cal.B', 'amp')
plotcal('cal.B', 'phase')

```

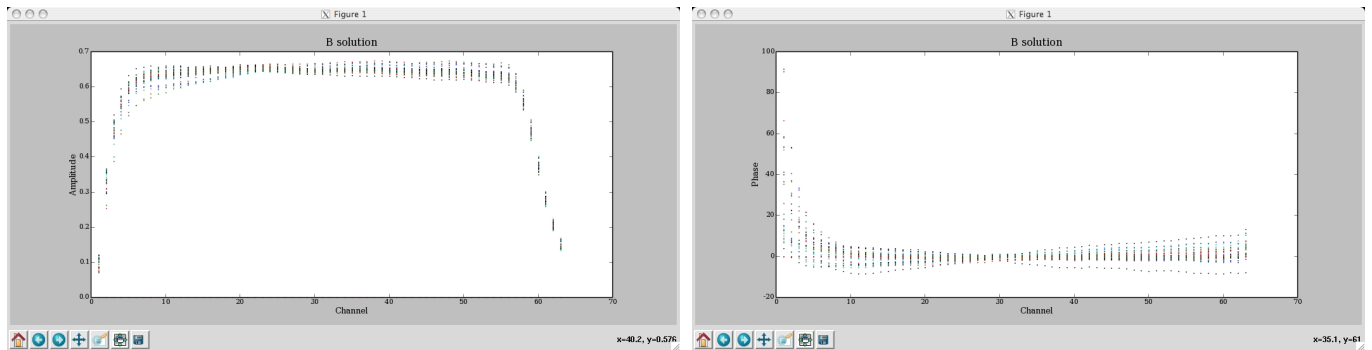


Figure 4.4: `plotcal`: Display of the amplitude and phase bandpass solutions (for all data).

The previous examples will show the solutions for all antennas and spectral windows on a single plot. If per-antenna solution plots are desired, use `multiplot=True`, and specify the number of plots to appear on each page using `subplot`. The format is `subplot=yxn` where `yxn` is an integer with digit `y` representing the number of plots in the y-axis, digit `x` the number of panels along the x-axis, and digit `n` giving the location of the plot in the panel array (where `n = 1, \dots, xy`, in order upper left to right, then down). If `multiplot=True` then set `n` to 1.

For example to show 5 plots per page (arranged vertically) of G phase solutions on each page, arranged vertically:

```
plotcal('cal.B', 'amp',           # show antennas bandpasses
        subplot=511,             # 5 vertical panels per page
        multiplot=True)         # draw separate plots
```

Use the **Next** button on the `plotcal` window to advance to the next set of plots. **ALPHA ALERT:** iteration in older versions may be done through a terminal query.

Note that if there is more than one timestamp in a B table, the user will be queried to interactively advance the plot to each timestamp, or if `multiplot=True`, the antennas plots will be cycled through for each timestamp in turn.

To show 6 plots per page of B amplitudes on a 3x2 grid:

```
plotcal('cal.B', 'amp', subplot=231, multiplot=True)
```

See Figure 4.5 for this example.

4.9 Listing calibration solutions with (`listcal`)

The `listcal` task will list the solutions in a specified calibration table.

The inputs are:


```

'9'      : 0.989  -12.4   0.992  -13.5
'10'     : 1.000F -4.2F  1.000F -3.2F
'11'     : 0.896   -0.0   0.890   -0.0
'12'     : 0.996  -10.6   0.996   -4.2
'13'     : 1.009   -8.4   1.011   -6.1
'14'     : 0.993  -17.6   0.994  -16.1
'15'     : 1.002   -0.8   1.002   -1.1
'16'     : 1.010   -9.9   1.012   -8.6
'17'     : 1.014   -8.0   1.017   -7.1
'18'     : 0.998   -3.0   1.005   -1.0
'19'     : 0.997  -39.1   0.994  -38.9
'20'     : 0.984   -5.7   0.986    3.0
'21'     : 1.000F -4.2F  1.000F -3.2F
'22'     : 1.003  -11.8   1.004  -10.4
'23'     : 1.007  -13.8   1.009  -11.7
'24'     : 1.000F -4.2F  1.000F -3.2F
'25'     : 1.000F -4.2F  1.000F -3.2F
'26'     : 0.992    3.7   1.000   -0.2
'27'     : 0.994   -5.6   0.991   -4.3
'28'     : 0.993  -10.7   0.997   -3.8

```

ALPHA ALERT: It is likely that the format of this listing will change to better present it to the user.

4.10 Application of Calibration (applycal)

ALPHA ALERT: in older versions this task may be named `correct`.

After all relevant calibration types have been determined, they must be applied to the target source(s) before splitting off to a new MS or before imaging. This is currently done by explicitly taking the data in the `DATA` column in the `MAIN` table of the MS, applying the relevant calibration tables, and creating the `CORRECTED_DATA` scratch column. The original `DATA` column is untouched.

The `applycal` task does this. The inputs are:

```

vis          =      '' # Name of input visibility file
field       =      '' # Select data based on field name or index
spw        =      '' # Select data based on spectral window
selectdata  =      False # Activate data selection details
gaincurve   =      False # Apply VLA antenna gain curve correction
opacity     =      False # Apply an opacity correction (True/False)
gaintable   =      '' # Gain calibration solutions to apply
gainselect  =      '' # Select subset of calibration solutions from gaintable
bptable     =      '' # Bandpass calibration solutions to apply
blbased     =      False # Apply baseline-based solutions (from blcal)
pointtable  =      '' # Pointing or other additional calibration solutions to apply
calwt       =      True # Apply calibration also to the WEIGHTS

```

```
spwmap          =      [-1] # Spectral window map of solutions
async          =      False # if True run in the background, prompt is freed
```

As in other tasks, setting `selectdata=True` will open up the other selection sub-parameters (see § 2.5).

For example, to apply G and B solutions to source fields 2,3,4:

```
applycal(vis='ngc5921.ms',      # Visibility set to correct
         field='2,3,4',        # restrict correction to specified fields
         gaintable='cal.Gflx',  # gain solutions to apply (time dependent)
         bptable='cal.B')      # bandpass solutions to apply (freq dependent)
```

Different detailed combinations of calibration application can be achieved by running this sequence more than once, and including specific field and/or spectral window selections as appropriate. For example, to limit the G solutions applied to field 3 to those obtained from field 1 (otherwise, same as above):

```
applycal(vis='ngc5921.ms',
         field='2,4',
         gaintable='cal.Gflx',
         bptable='cal.B')
applycal(vis='ngc5921.ms',
         field='3',
         gaintable='cal.Gflx',
         gainselect='FIELD_ID==1',
         bptable='cal.B')
```

It is important to remember the relative nature of each calibration term. A term solved for in the presence of others is, in effect, residual to the others, and so must be used in combination with them (or new versions of them) in subsequent processing. At the same time, it is important to avoid isolating the same calibration effects in more than one term, e.g., by solving for both G and T separately (without applying the other), and then using them together. It is always a good idea to examine the corrected data after calibration (using `plotxy` to compare the raw ('data') and corrected ('corrected') visibilities).

4.10.1 Examine calibrated source data

Once the source data is calibrated using `applycal`, you should examine the *uv* data and flag anything that looks bad. If you find source data that has not been flanked by calibration scans, delete it (it will not be calibrated).

For example:

```
plotxy('data.ms', 'uvdist', 'amp')          # Display data for flagging
```

will show the `CORRECTED_DATA` column by default.

See Chapter on Data Editing for descriptions of how to display/edit the data in `plotxy` and in the `viewer`.

4.11 Resetting the Calibration using (clearcal)

The `applycal` task will set the `CORRECTED_DATA` column. The `clearcal` task will reset it.

There is only a single input to `clearcal`:

```
# clearcal :: Re-initializes calibration for an ms
vis          =          ''          # Name of input visibility file
```

4.12 Optional: Split out Calibrated uv data (split)

The `split` task will apply calibration and output a new sub-MS containing a specified list of sources (usually a single source). The inputs are:

```
# split :: Create a visibility subset from an existing visibility set:
vis          =          ''          # Name of input visibility file
outputvis    =          ''          # Name of output visibility file
field        =          ''          # Field name list
spw          =          ''          # Spectral window identifier
antenna      =          ''          # Antenna selection
timebin      =          '-1s'      # time averaging of data
timerange    =          ''          # time range for subset of data
datacolumn   =          'corrected' # which column to split (data, corrected, model)
async        =          False      # if True run in the background, prompt is freed
```

It is usual to run with the default `datacolumn='corrected'` as previous operations (e.g. `applycal`) will have placed the calibrated data in the `CORRECTED_DATA` column of the MS.

For example, to split out 46 channels (5-50) from `spw 1`:

```
split(vis='ngc5921.ms',          # input MS
      outputvis='ngc5921_src.split.ms', # Output just the source data
      field='2',                # Select the third source (0-based)
      spw='0:5~50',            # Select 46 channels from the first spectral window
      datacolumn='CORRECTED_DATA') # Take the calibrated data column
```

ALPHA ALERT: The ability to average channels in `split` is on the way.

4.13 Advanced Calibration and UV-Plane Analysis

4.13.1 UV-Plane Continuum Subtraction (uvcontsub)

ALPHA ALERT: this was `contsub` in older patches.

At this point, consider whether you are likely to need continuum subtraction. If there is significant continuum emission present in what is intended as a spectral line observation, continuum subtraction may be desirable. You can estimate and subtract continuum emission in the *uv*-plane prior to imaging or wait and subtract an estimate of it in the image-plane. Note that neither method is ideal, and the choice depends primarily upon the distribution and strength of the continuum emission. Subtraction in the *uv*-plane is desirable if continuum emission dominates the source, since deconvolution of the line emission will be more robust if not subject to errors in deconvolution of the brighter continuum. There is also a performance benefit since the continuum is probably the same in each channel of the observation, and it is desirable to avoid duplication of effort. However, the main drawback of subtraction in the *uv*-plane is that it is only strictly correct for the phase center, since without the Fourier transform, the visibilities only describe the phase center. Thus, *uv*-plane continuum subtraction will be increasingly poor for emission distributed further from the phase center. If the continuum emission is relatively weak, it is usually adequate to subtract it in the image plane; this is described in the Image Analysis section of this cookbook. Here, we describe how to do continuum subtraction in the *uv*-plane.

The *uv*-plane continuum subtraction is performed by the `uvcontsub` task. First, determine which channels in your data cube do not have line emission, perhaps by forming a preliminary image as described in the next chapter. This image will also help you decide whether or not you need to come back and do *uv*-plane continuum subtraction at all.

The inputs are:

```
vis           =      '' # Name of input visibility file (MS)
field        =      '' # Field name(s)-min matches; use spaces to separate fields
spw          =      '' # Spectral window identifier (0-based)
channels     =      [] # Range of channels to fit
solint       =      0.0 # Averaging time (sec)
fitorder     =      0 # Polynomial order for the fit
fitmode      =      'subtract' # Use of continuum fit (subtract,replace,model)
splitdata    =      False # Split out continuum, continuum-subtracted data
async        =      False # if True run in the background, prompt is freed
```

ALPHA ALERT: The `spw` parameter can currently only be used to specify the Spectral Window, not channelization. For now, we provide the `channel` parameter (see the example below).

For each baseline, and over the timescale specified in `solint`, `uvcontsub` will provide a simple linear fit to the real and imaginary parts of the (continuum-only) channels specified in `channels`, and subtract this model from all channels. Choose the timescale to be shorter than the timescale for changes in the visibility function of the continuum, but be careful not to make it so short that the SNR of the estimated continuum suffers substantially. For example:

```
uvcontsub(vis='data.ms', # Select visibility data set
          field='2',      # Choose the source data (field 3),
          spw='0',        # spectral window 0,
          channels=range(4,7)+range(50,60), # line free channels 4-6 & 50-59,
          solint=0.0,     # estimate/subtract per scan
          fitmode='subtract'); # & subtract cont. from line data.
```



```
# Line-only data will be written into
# the CORRECTED_DATA column.
```

Running `uvcontsub` with `fitmode='subtract'` will replace the `CORRECTED_DATA` column in the MS with continuum-subtracted line data and the `MODEL_DATA` column with the continuum model. You can use `fitmode='replace'` to replace the `CORRECTED_DATA` column with the continuum model; however, it is probably better to use `fitmode='subtract'` and then use `split` to select the `MODEL_DATA` and form a dataset appropriate for forming an image of the estimated continuum. Note that a continuum image formed from this model will only be strictly correct near the phase center, for the reasons described above.

The `splitdata` parameter can be used to have `uvcontsub` write out split MS for both the continuum-subtracted data and the continuum. It will leave the input MS in the state as if `fitmode='subtract'` was used. Note that the entire channel range of the MS will be written out, so do `split` manually if you want to restrict the output channel range. If `splitdata=True`, then `uvcontsub` will make two output MS with names `<input msname>.contsub` and `<input msname>.cont`. **ALPHA ALERT:** be sure to run with `fitmode='subtract'` if setting `splitdata=True`.

Note that it is currently the case that `uvcontsub` will overwrite the `CORRECTED_DATA` column. Therefore, it is desirable to first `split` the relevant corrected data into a new Measurement Set. If you run `uvcontsub` on the original dataset, you will have to re-apply the calibration as described in the previous chapter.

So, the recommended procedure is as follows:

- Finish calibration as described in the previous chapter.
- Use `split` to form a separate dataset.
- Use the `invert` or `clean` task on the `split` result to form an exploratory image that is useful for determining the line-free channels.
- Use `uvcontsub` with `mode='subtract'` to subtract the continuum from the `CORRECTED_DATA` in the MS, and write the continuum model in the `MODEL_DATA` column. Set `splitdata=True` to have it automatically split out continuum-subtracted and continuum datasets, else do this manually.
- Image the line-only emission with the `clean` task.
- If an image of the estimated continuum is desired, and you did not use `splitdata=True`, then run `split` again (on the `uvcontsub`'d dataset), and select the `MODEL_DATA`; then run `clean` to image it.

4.13.2 Baseline-based Calibration (`blcal`)

You can use the `blcal` task to solve for baseline-dependent (non-closing) errors. **WARNING:** this is in general a very dangerous thing to do, since baseline-dependent errors once introduced are

difficult to remove. You must be sure you have an excellent model for the source (better than the magnitude of the baseline-dependent errors).

The inputs are:

```
# blcal :: Calculate a baseline-based calibration solution (gain or bandpass)

vis           =      ''      # Name of input visibility file (MS)
caltable      =      ''      # Name of output bandpass calibration table
field         =      ''      # Select data based on field name or index
spw           =      ''      # Select data based on spectral window
selectdata    =      True    # Activate data selection details
    timerange  =      ''      # Select data based on time
    uvrange    =      ''      # Select data based on uv range
    antenna    =      ''      # Select data based on antenna/baseline
    scan       =      ''      # Select data based on scan number
    msselect   =      ''      # Optional data selection

freqdep       =      False   # Solve for frequency dependent solutions
gaincurve     =      False   # Apply VLA antenna gain curve correction
opacity       =      0.0     # Opacity correction to apply (nepers)
gaintable     =      ''      # Gain calibration solutions to apply
gainselect    =      ''      # Select subset of calibration solutions from gaintable
solint        =      0.0     # Solution interval (sec)
```

The `freqdep` parameter controls whether `blcal` solves for “gain” (`freqdep=True`) or “bandpass” (`freqdep=False`) style calibration.

Other parameters are the same as in other calibration tasks.

4.13.3 Fringe Fitting (`fringecal`)

WARNING: This is an experimental calibration task, and has not been extensively tested!

The `fringecal` task provides the capability for solving for *baseline-based* phase, phase-delay, and delay-rate terms in the gains (G-type). This is not full antenna-based “fringe-fitting” as is commonly used in VLBI. The main use is to calibrate ALMA or EVLA commissioning data where the delays may be improperly set, and to test “fringe” solutions as a way for dealing with non-dispersive atmospheric terms.

The inputs are:

```
# fringecal :: BL-based fringe-fitting solution:

vis           =      ''      # Name of input visibility file (MS)
caltable      =      ''      # Name of output bandpass calibration table
field         =      ''      # Select data based on field name or index
spw           =      ''      # Select data based on spectral window
selectdata    =      True    # Activate data selection details
```

```

timerange      =      ''      # Select data based on time
uvrange        =      ''      # Select data based on uv range
antenna        =      ''      # Select data based on antenna/baseline
scan           =      ''      # Select data based on scan number
msselect       =      ''      # Optional data selection (see help)

gaincurve      =      True     # Apply VLA antenna gain curve correction
opacity        =      0.0     # Opacity correction to apply (nepers)
gaintable      =      ''      # Gain calibration solutions to apply
gainselect     =      ''      #
solint         =      0.0     # Solution interval (sec)
refant         =      ''      # Reference antenna
async          =      False    # if True run in the background, prompt is freed

```

The action of the parameters in `fringecal` is the same as in the other calibration “solver” tasks (such as `gaincal` and `bandpass`).

4.13.4 UV-Plane Model Fitting (`uvmodelfit`)

It is often desirable to fit simple analytic source component models directly to visibility data. Such fitting has its origins in early interferometry, especially VLBI, where arrays consisted of only a few antennas and the calibration and deconvolution problems were poorly constrained. These methods overcame the calibration uncertainties by fitting the models to calibration-independent closure quantities and the deconvolution problem by drastically limiting the number of free parameters required to describe the visibilities. Today, even with larger and better calibrated arrays, it is still desirable to use visibility model fitting in order to extract geometric properties such as the positions and sizes of discrete components in radio sources. Fits for physically meaningful component shapes such as disks, rings, and optically thin spheres, though idealized, enable connecting source geometry directly to the physics of the emission regions.

Visibility model fitting is controlled entirely by the `uvmodelfit` task, which allows fits for a single component point or Gaussian. The user specifies the number of non-linear solution iterations (`niter`), the component type (`comptype`), an initial guess for the component parameters (`sourcepar`), and optionally, a vector of Booleans selecting which component parameters should be allowed to vary (`fixpar`), and a filename in which to store a CASA componentlist for use in other applications (`file`). The function returns a vector containing the resulting parameter list. This vector can be edited at the command line, and specified as input (`sourcepar`) for another round of fitting.

The `sourcepar` parameter is currently the only way to specify the starting parameters for the fit. For points, there are three parameters: I (total flux density), and relative direction (RA, Dec) offsets (in arcsec) from the observation’s phase center. For Gaussians, there are three additional parameters: the Gaussian’s semi-major axis width (arcsec), the aspect ratio, and position angle (degrees). It should be understood that the quality of the result is very sensitive to the starting parameters provided by the user. If this first guess is not sufficiently close to the global χ^2 minimum, the algorithm will happily converge to an incorrect local minimum. In fact, the χ^2 surface, as a function of the component’s relative direction parameters, has a shape very much like the

inverse of the absolute value of the dirty image of the field. Any peak in this image (positive or negative) corresponds to a local χ^2 minimum that could conceivably capture the fit. It is the user's responsibility to ensure that the correct minimum does the capturing.

Currently, `uvmodelfit` relies on the likelihood that the source is very near the phase center (within a beamwidth) and/or the user's savvy in specifying the starting parameters. This fairly serious constraint will soon be relieved somewhat by enabling a rudimentary form of uv-plane weighting to increase the likelihood that the starting guess is on a slope in the correct χ^2 valley.

Improvements in the works for visibility model fitting include:

- User-specifiable uv-plane weighting
- Additional component shapes, including elliptical disks, rings, and optically thin spheroids.
- Optional calibration pre-application
- Multiple components. The handling of more than one component depends mostly on efficient means of managing the list itself (not easy in command line options), which are currently under development.
- Combined component and calibration fitting.

Example (See Figure 4.6):

```
#
# Note: It's best to channel average the data if many channels
# before running a modelfit
#
uvmodelfit('1445_avg.ms',      # use averaged data
           niter=5,           # Do 5 iterations
           comptype='P',      # P=Point source, G=Gaussian, D=Disk
           sourcepar=[2.0,.1,.1], # Source parameters for a point source
                               # [flux, long offset, lat offset]
           spw='0',          #
           file='gcal.cl')   # Output component list file
                               # Initial guess is that it's close to the phase center
                               # and has a flux of 2.0 (a priori we know it's 2.47)

# Output looks like:
CASA <25>:
uvmodelfit('1445_avg.ms/', niter=5, comptype='P',
           sourcepar=[2.0,.1,.1], file='gcal.cl', spw='0')

Tue Dec 12 23:02:05 2006      WARN Calibrator::setdata:
Selection is empty: reverting to sorted MeasurementSet
There are 19656 - 3 = 19653 degrees of freedom.
iter=0:  reduced chi2=0.0413952:  I=2,  dir=[0.1, 0.1] arcsec
iter=1:  reduced chi2=0.0011285:  I=2.48495,  dir=[-0.0265485, -0.0189735] arcsec
iter=2:  reduced chi2=0.00112653:  I=2.48547,  dir=[-0.00196871, 0.00409329] arcsec
iter=3:  reduced chi2=0.00112653:  I=2.48547,  dir=[-0.00195744, 0.00411176] arcsec
```

```

iter=4:   reduced chi2=0.00112653:  I=2.48547,  dir=[-0.00195744, 0.00411178] arcsec
iter=5:   reduced chi2=0.00112653:  I=2.48547,  dir=[-0.00195744, 0.00411178] arcsec

```

If data weights are arbitrarily scaled, the following formal errors will be underestimated by at least a factor $\sqrt{\text{reduced chi2}}$. If the fit is systematically poor, the errors are much worse.

```

I = 2.48547 +/- 0.0172627
x = -0.00195744 +/- 0.159619 arcsec
y = 0.00411178 +/- 0.170973 arcsec

```

Writing componentlist to file: /Users/jmcmulli/ALMA/TST5/Regression/Scripts/gcal.cl

```

# Looks reasonable - got the right flux around the phase center
# chi2 went down: expect chi2 = 2*number of visibilities/number of degrees of freedom
# degrees of freedom = 3 for point source (flux and long,lat offsets)
# Now use the component list to generate model data

```

```

ft('1445_avg.ms',
  complist='gcal.cl')      # Fourier transform the component list -
                          # this writes it into the MODEL_DATA column
                          # of the MS

plotxy('data.ms',
  xaxis='uvdist',         # Plot data versus uv distance
  field='1',             # Select 1445+0990
  datacolumn='corrected') # Plot corrected data

plotxy('data.ms',
  xaxis='uvdist',         #
  field='1',             #
  overplot=True,         # Specify overplot
  plotsymbol='bo')      # Specify blue circles for model data

```

4.14 Example of Calibration

The following is an example calibration using the NGC5921 VLA observations as the demonstration. This uses the CASA tasks as of Alpha Patch 1. This data is available with the CASA release and so you can try this yourself.

The full NGC5921 example script can be found in Appendix C.1.

ALPHA ALERT: Note that the syntax has been changing recently and this may get out of date quickly!

```

#####
#
# Use Case Script for NGC 5921
#
# Converted by STM 2007-05-26
#

```

model vs. uv distance MS name: /home/basho3/jmcmulli/pretest/ngc5921.ms;
 Model; Spectral Windows: 0; Polarization: RR LL;
 Fields: 1445+09900002_0

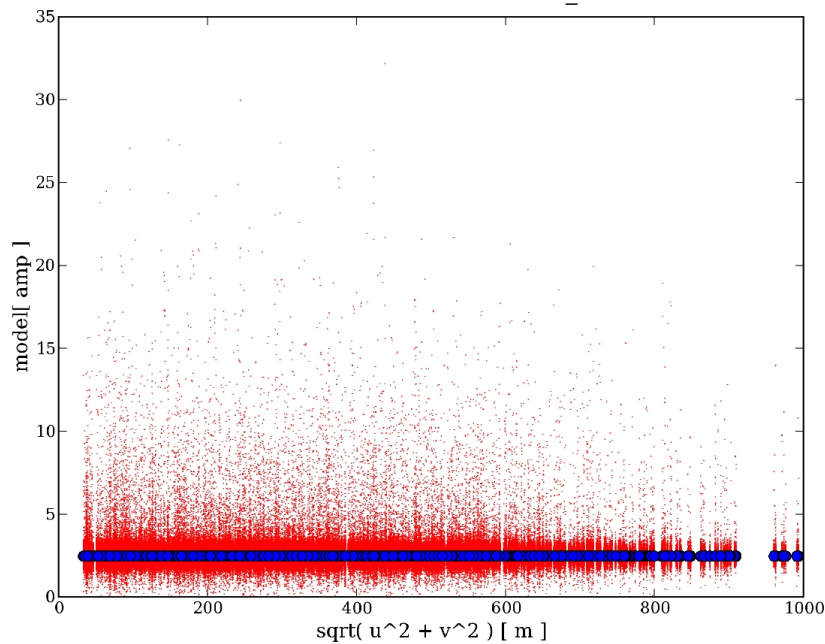


Figure 4.6: Use of plotxy to display corrected data (red points) and uv model fit data (blue circles).

```
# Updated      STM 2007-06-15 (Alpha Patch 1)           #
# Last Updated STM 2007-09-05 (Alpha Patch 2+)         #
#                                                       #
#####

import time
import os

#
# Set up some useful variables
#
# Get to path to the CASA home and strip off the name
pathname=os.environ.get('AIPSPATH').split()[0]

# This is where the NGC5921 UVFITS data will be
fitsdata=pathname+'/data/demo/NGC5921.fits'

# The prefix to use for all output files
prefix='ngc5921.usecase'

# Clean up old files
os.system('rm -rf '+prefix+'*')
```

```

#
#=====
#
# Import the data from FITS to MS
#
print '--Import--'

# Safest to start from task defaults
default('importuvfits')

# Set up the MS filename and save as new global variable
msfile = prefix + '.ms'

# Use task importuvfits
fitsfile = fitsdata
vis = msfile
importuvfits()

#
# Note that there will be a ngc5921.usecase.ms.flagversions
# there containing the initial flags as backup for the main ms
# flags.
#
#=====
#
# List a summary of the MS
#
print '--Listobs--'

# Don't default this one and make use of the previous setting of
# vis. Remember, the variables are GLOBAL!

# You may wish to see more detailed information, like the scans.
# In this case use the verbose = True option
verbose = True

listobs()

# You should get in your logger window and in the casapy.log file
# something like:
#
# MeasurementSet Name: /home/sandrock2/smyers/Testing2/Sep07/ngc5921.usecase.ms
# MS Version 2
#
# Observer: TEST      Project:
# Observation: VLA
#
# Data records: 22653      Total integration time = 5280 seconds
#   Observed from 09:19:00 to 10:47:00
#
#   ObservationID = 0      ArrayID = 0

```

```

# Date          Timerange          Scan  FldId  FieldName      SpwIds
# 13-Apr-1995/09:19:00.0 - 09:24:30.0    1     0 1331+30500002_0 [0]
#                   09:27:30.0 - 09:29:30.0    2     1 1445+09900002_0 [0]
#                   09:33:00.0 - 09:48:00.0    3     2 N5921_2         [0]
#                   09:50:30.0 - 09:51:00.0    4     1 1445+09900002_0 [0]
#                   10:22:00.0 - 10:23:00.0    5     1 1445+09900002_0 [0]
#                   10:26:00.0 - 10:43:00.0    6     2 N5921_2         [0]
#                   10:45:30.0 - 10:47:00.0    7     1 1445+09900002_0 [0]
#
# Fields: 3
#  ID  Code Name          Right Ascension  Declination  Epoch
#  0   C  1331+30500002_013:31:08.29    +30.30.32.96  J2000
#  1   A  1445+09900002_014:45:16.47    +09.58.36.07  J2000
#  2           N5921_2          15:22:00.00    +05.04.00.00  J2000
#
# Spectral Windows: (1 unique spectral windows and 1 unique polarization setups)
# SpwID #Chans Frame Ch1(MHz)  Resoln(kHz) TotBW(kHz) Ref(MHz) Corrs
#  0           63 LSRK  1412.68608  24.4140625  1550.19688  1413.44902  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.
#  0   1    VLA:N7  25.0 m -107.37.07.2 +33.54.12.9
#  1   2    VLA:W1  25.0 m -107.37.05.9 +33.54.00.5
#  2   3    VLA:W2  25.0 m -107.37.07.4 +33.54.00.9
#  3   4    VLA:E1  25.0 m -107.37.05.7 +33.53.59.2
#  4   5    VLA:E3  25.0 m -107.37.02.8 +33.54.00.5
#  5   6    VLA:E9  25.0 m -107.36.45.1 +33.53.53.6
#  6   7    VLA:E6  25.0 m -107.36.55.6 +33.53.57.7
#  7   8    VLA:W8  25.0 m -107.37.21.6 +33.53.53.0
#  8   9    VLA:N5  25.0 m -107.37.06.7 +33.54.08.0
#  9  10    VLA:W3  25.0 m -107.37.08.9 +33.54.00.1
# 10  11    VLA:N4  25.0 m -107.37.06.5 +33.54.06.1
# 11  12    VLA:W5  25.0 m -107.37.13.0 +33.53.57.8
# 12  13    VLA:N3  25.0 m -107.37.06.3 +33.54.04.8
# 13  14    VLA:N1  25.0 m -107.37.06.0 +33.54.01.8
# 14  15    VLA:N2  25.0 m -107.37.06.2 +33.54.03.5
# 15  16    VLA:E7  25.0 m -107.36.52.4 +33.53.56.5
# 16  17    VLA:E8  25.0 m -107.36.48.9 +33.53.55.1
# 17  18    VLA:W4  25.0 m -107.37.10.8 +33.53.59.1
# 18  19    VLA:E5  25.0 m -107.36.58.4 +33.53.58.8
# 19  20    VLA:W9  25.0 m -107.37.25.1 +33.53.51.0
# 20  21    VLA:W6  25.0 m -107.37.15.6 +33.53.56.4
# 21  22    VLA:E4  25.0 m -107.37.00.8 +33.53.59.7
# 23  24    VLA:E2  25.0 m -107.37.04.4 +33.54.01.1
# 24  25    VLA:N6  25.0 m -107.37.06.9 +33.54.10.3
# 25  26    VLA:N9  25.0 m -107.37.07.8 +33.54.19.0
# 26  27    VLA:N8  25.0 m -107.37.07.5 +33.54.15.8

```



```

# 27 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 273 rows
# OBSERVATION 1 row
# POINTING 168 rows
# POLARIZATION 1 row
# PROCESSOR <empty>
# SOURCE 3 rows
# SPECTRAL_WINDOW 1 row
# STATE <empty>
# SYSCAL <absent>
# WEATHER <absent>
#
#
#=====
#
# Get rid of the autocorrelations from the MS
#
print '--Flagautocorr--'

# Don't default this one either, there is only one parameter (vis)

flagautocorr()

#
#=====
#
# Set the fluxes of the primary calibrator(s)
#
print '--Setjy--'
default('setjy')

vis = msfile

#
# 1331+305 = 3C286 is our primary calibrator
# Use the wildcard on the end of the source name
# since the field names in the MS have inherited the
# AIPS qualifiers
field = '1331+305*'

# This is 1.4GHz D-config and 1331+305 is sufficiently unresolved

```

```

# that we dont need a model image. For higher frequencies
# (particularly in A and B config) you would want to use one.
modimage = ''

# Setjy knows about this source so we dont need anything more

setjy()

#
# You should see something like this in the logger and casapy.log file:
#
# 1331+30500002_0 spwid= 0 [I=14.76, Q=0, U=0, V=0] Jy, (Perley-Taylor 99)
#
# So its using 14.76Jy as the flux of 1331+305 in the single Spectral Window
# in this MS.
#
#=====
#
# Bandpass calibration
#
print '--Bandpass--'
default('bandpass')

# We can first do the bandpass on the single 5min scan on 1331+305
# At 1.4GHz phase stability should be sufficient to do this without
# a first (rough) gain calibration. This will give us the relative
# antenna gain as a function of frequency.

vis = msfile

# set the name for the output bandpass caltable
btable = prefix + '.bcal'
caltable = btable

# No gain tables yet
gaintable = ''

# Use flux calibrator 1331+305 = 3C286 (FIELD_ID 0) as bandpass calibrator
field = '0'
# all channels
spw = ''
# No other selection
selectdata = False

# In this band we do not need a-priori corrections for
# antenna gain-elevation curve or atmospheric opacity
# (at 8GHz and above you would want these)
gaincurve = False
opacity = 0.0

# Choose bandpass solution type

```

```

# Pick standard time-binned B (rather than BPOLY)
bandtype = 'B'

# set solution interval arbitrarily long (get single bpass)
solint = 86400.0

# reference antenna Name 15 (15=VLA:N2) (Id 14)
refant = '15'

bandpass()

# You can use plotcal to examine the solutions
#default('plotcal')
#tablein = btable
#yaxis = 'amp'
#field = '0'
#multiplot = True
#plotcal()
#
#yaxis = 'phase'
#plotcal()
#
# Note the rolloff in the start and end channels. Looks like
# channels 6-56 (out of 0-62) are the best

#=====
#
# Gain calibration
#
print '--Gaincal--'
default('gaincal')

# Armed with the bandpass, we now solve for the
# time-dependent antenna gains

vis = msfile

# set the name for the output gain caltable
gtable = prefix + '.gcal'
caltable = gtable

# Use our previously determined bandpass
# Note this will automatically be applied to all sources
# not just the one used to determine the bandpass
bptable = btable

# Gain calibrators are 1331+305 and 1445+099 (FIELD_ID 0 and 1)
field = '0,1'

# We have only a single spectral window (SPW 0)
# Choose 51 channels 6-56 out of the 63

```

```

# to avoid end effects.
# Channel selection is done inside spw
spw = '0:6~56'

# No other selection
selectdata = False

# In this band we do not need a-priori corrections for
# antenna gain-elevation curve or atmospheric opacity
# (at 8GHz and above you would want these)
gaincurve = False
opacity = 0.0

# scan-based G solutions for both amplitude and phase
gaintype = 'G'
solint = 0.
calmode = 'ap'

# reference antenna 15 (15=VLA:N2)
refant = '15'

gaincal()

# You can use plotcal to examine the gain solutions
#default('plotcal')
#tablein = gtable
#yaxis = 'amp'
#field = '0,1'
#multiplot = True
#plotcal()
#
#yaxis = 'phase'
#plotcal()
#
# The amp and phase coherence looks good

#=====
#
# Bootstrap flux scale
#
print '--Fluxscale--'
default('fluxscale')

vis = msfile

# set the name for the output rescaled caltable
ftable = prefix + '.fluxscale'
fluxtable = ftable

# point to our first gain cal table
caltable = gtable

```

```

# we will be using 1331+305 (the source we did setjy on) as
# our flux standard reference - note its extended name as in
# the FIELD table summary above (it has a VLA seq number appended)
reference = '1331*'

# we want to transfer the flux to our other gain cal source 1445+099
transfer = '1445*'

fluxscale()

# In the logger you should see something like:
# Flux density for 1445+09900002_0 in SpW=0 is:
# 2.48576 +/- 0.00123122 (SNR = 2018.94, nAnt= 27)

# If you run plotcal() on the tablein = 'ngc5921.usecase.fluxscale'
# you will see now it has brought the amplitudes in line between
# the first scan on 1331+305 and the others on 1445+099

#=====
#
# Apply our calibration solutions to the data
# (This will put calibrated data into the CORRECTED_DATA column)
#
print '--ApplyCal--'
default('applycal')

vis = msfile

# We want to correct the calibrators using themselves
# and transfer from 1445+099 to itself and the target N5921

# Start with the fluxscale/gain and bandpass tables
bptable = btable
gaintable = ftable

# all channels
spw = ''
selectdata = False

# as before
gaincurve = False
opacity = 0.0

# select the fields for 1445+099 and N5921
field = '1,2'

# pick the 1445+099 out of the gain table for transfer
# (NOTE: this currently uses TaQL strings)
gainselect = 'FIELD_ID==1'

```

```

applycal()

# Now for completeness apply 1331+305 to itself

field = '0'
gainselect = 'FIELD_ID==0'

# The CORRECTED_DATA column now contains the calibrated visibilities

applycal()

#=====
#
# Split the gain calibrator data
#
print '--Split (cal data)--'
default('split')

vis = msfile

# We first want to write out the corrected data for the calibrator

# Make an output vis file
calsplitms = prefix + '.cal.split.ms'
outputvis = calsplitms

# Select the 1445+099 field, all chans
field = '1445*'
spw = ''

# pick off the CORRECTED_DATA column
datacolumn = 'corrected'

split()

#=====
#
# UV-plane continuum subtraction on the target
# (this will update the CORRECTED_DATA column)
#
print '--UV Continuum Subtract--'
default('uvcontsub')

vis = msfile

# Pick off N5921
field = 'N5921*'

# Use channels 4-6 and 50-59 for continuum
#spw = '0:4~6;50~59'
# ALPHA ALERT: still does not use standard notation

```

```
spw = '0'
channels = range(4,7)+range(50,60)

# Averaging time (none)
solint = 0.0

# Fit only a mean level
fitorder = 0

# Do the uv-plane subtraction
fitmode = 'subtract'

# Let it split out the data automatically for us
splitdata = True

uvcontsub()

# You will see it made two new MS:
# ngc5921.usecase.ms.cont
# ngc5921.usecase.ms.contsub

srcsplitms = msfile + '.contsub'

# Note that ngc5921.usecase.ms.contsub contains the uv-subtracted
# visibilities (in its DATA column), and ngc5921.usecase.ms.contsub
# the pseudo-continuum visibilities (as fit).

# The original ngc5921.usecase.ms now contains the uv-continuum
# subtracted vis in its CORRECTED_DATA column and the continuum
# in its MODEL_DATA column as per the fitmode='subtract'
```

Chapter 5

Synthesis Imaging

This chapter describes how to make and deconvolve images starting from calibrated interferometric data, possibly supplemented with single-dish data or an image made from single-dish data. This data must be available in CASA (see § 2 on importing data). See § 4 for information on calibrating synthesis data. In the following sections, the user will learn how to make various types of images from synthesis data, reconstruct images of the sky using the available deconvolution techniques, include single-dish information in the imaging process, and to prepare to use the results of imaging for improvement of the calibration process (“self-calibration”).

Inside the Toolkit:

The `im` tool handles synthesis imaging operations.

5.1 Imaging Tasks Overview

At this intermediate stage of alpha-development, the imaging capabilities in CASA are being converted from tools to tasks. Tasks that allow you to do most standard imaging have been created. The current imaging tasks are:

- `invert` — create a dirty image and point-spread function (PSF) (§ 5.3)
- `clean` — calculate a deconvolved image with a selected clean algorithm (§ 5.4)
- `mosaic` — calculate a multi-field deconvolved image with selected clean algorithm (§ 5.5)
- `feather` — combine a single dish and synthesis image in the Fourier plane (§ 5.6)
- `deconvolve` — image-plane only deconvolution based on the dirty image and beam, using one of several algorithms (§ 5.9)

There are also tasks that help you set up the imaging or interface imaging with calibration:

- `makemask` - create “cleanbox” deconvolution regions (§ 5.7)
- `ft` - Fourier transform the specified model (or component list) and insert the source model into the MODEL column of a visibility set (§ 5.8)

The full “tool kit” that allows expert-level imaging must still be used if you do not find enough functionality within the tasks above.

Information on other useful tasks and parameter setting can be found in:

- `listobs` — list whats in a MS (§ 2.3),
- `split`— Write out new MS containing calibrated data from a subset of the original MS (§ section:cal.split),
- data selection — general data selection syntax (§ 2.5).

5.2 Common Imaging Task Parameters

We now describe some parameters are are common to the imaging tasks. These should behave the same way in any imaging task that they are found in. These are in alphabetical order.

NOTE: In the current version of CASA, there are a subset of data selection parameters used in the imaging tasks: `field`, `spw`, `selecttime`. In a later patch, we will unify these across tasks into the standard data selection set (§ 2.5).

Inside the Toolkit:

The `im.setimage` method is used to set many of the common image parameters. The `im.advise` method gives helpful advice for setting up for imaging.

5.2.1 The `cell` Parameter

The `cell` parameter defines the pixel size in the x and y axes for the output image. If given as floats or integers, this is the cell size in arc seconds, e.g.

```
cell=[0.5,0.5]
```

make 0.5'' pixels. You can also give the cell size in *quantities*, e.g.

```
cell=['1arcmin', '1arcmin']
```

If a single value is given, then square pixels of that size are assumed.

5.2.2 The field Parameter

The `field` parameter selects the field indexes or names to be used in imaging. Unless you making a `mosaic`, this is usually a single index or name:

```
field = '0'           # First field (index 0)
field = '1331+305'   # 3c286
field = '*'          # all fields in dataset
```

The syntax for `field` selection is given in § 2.5.2.

5.2.3 The imagename Parameter

The value of the `imagename` parameter is used as the root name of the output image. Depending on the particular task and the options chosen, one or more images with names built from that root will be created. For example, the `clean` task run with `imagename='ngc5921'` a series of output images with names `ngc5921.clean`, `ngc5921.residual`, and `ngc5921.model` will be created.

If an image with that name already exists, it will in general be overwritten. Beware using names of existing images however. If the `clean` is run using an `imagename` where `<imagename>.residual` and `<imagename>.model` already exist then `clean` will continue starting from these (effectively restarting from the end of the previous `clean`). Thus, if multiple runs of `clean` are run consecutively with the same `imagename`, then the cleaning is incremental (as in the `difmap` package).

5.2.4 The imsize Parameter

The image size in numbers of pixels on the x and y axes is set by `imsize`. For example,

```
imsize = [256, 256]
```

makes a square image 256 pixels on a side. If a single value is given, then a square image of that dimension is made. This need not be a power of two, but should not be a prime number.

5.2.5 The mode Parameter

The `mode` parameter defines how the frequency channels in the synthesis MS are mapped onto the image. The allowed values are: `mfs`, `channel`, `velocity`, `frequency`. The `mode` parameter is expandable, with some options uncovering a number of sub-parameters, depending upon its value.

The default `mode='mfs'` emulates multi-frequency synthesis in that each visibility-channel datum k with baseline vector \mathbf{B}_k at wavelength λ_k is gridded into the uv-plane at $\mathbf{u}_k = \mathbf{B}_k/\lambda_k$. The result is a single image plane, regardless of how many channels are in the input dataset. This image plane is at the frequency given by the midpoint between the highest and lowest frequency channels in the

input `spw(s)`. Currently, there is no way to choose the center frequency of the output image plane independently.

If `mode='channel'` is chosen, then an image cube will be created. This is an expandable parameter, with dependent parameters:

```
mode      = 'channel'  # Type of selection (mfs, channel, velocity, frequency)
nchan     =      -1    # Number of channels to select
start     =         0  # Start channel
step      =         1  # Increment between channels/velocity
width     =         1  # Channel width (value > 1 indicates channel averaging)
```

The channelization of the resulting image is determined by the channelization in the first MS of `vis` of the first `spw` specified (the “reference `spw`”). The resulting image cube will have `nchan` channels spaced evenly in frequency. The first output channel will be located at the frequency of channel `start` in the reference `spw`. The output channel spacing is given by every `step` in the reference `spw` of the MS. Channels in `spw` beyond the first are mapped into the nearest output image channel within half a channel (if any). Image channels that lie outside the MS frequency range or have no data mapped to them will be blank in the output image, but will be in the cube. If `width > 1`, then input MS channels with centers within a frequency range given by $(width + 1)/2$ times the reference `spw` spacing will be gridded together (as in `mode = 'mfs'` above) into the channels of the output image cube.

For `mode='frequency'`, an output image cube is created with `nchan` channels spaced evenly in frequency.

```
mode      = 'frequency' # Type of selection (mfs, channel, velocity, frequency)
nchan     =      -1    # Number of channels to select
start     =         0  # Frequency of first image channel: e.g '1.4GHz'
step      =         1  # image channel width in frequency units: e.g '1.0kHz'
```

The frequency of the first output channel is given by `start` and spacing by `step`. The sign of `step` determines whether the output channels ascend or descend in frequency. Output channels have a width also given by `step`. Data from the input MS with centers that lie within one-half an input channel overlap of the frequency range of $\pm step/2$ centered on the output channels are gridded together.

If `mode='velocity'` is chosen, then an output image cube with `nchan` channels will be created, with channels spaced evenly in velocity. Parameters are:

```
mode      = 'velocity'  # Type of selection (mfs, channel, velocity, frequency)
nchan     =      -1    # Number of channels to select
start     =         0  # Velocity of first image channel: e.g '0.0km/s'
step      =         1  # image channel width in velocity units: e.g '-1.0km/s'
```

The velocity of the first output channel is given by `start` and spacing by `step`. Note that the velocity frame is given by the rest frequency in the MS header, which can be overridden by the `restfreq` parameter. Averaging is as in `mode='frequency'`.

5.2.6 The `restfreq` Parameter

The value of the `restfreq` parameter, if set, will over-ride the rest frequency in the header of the first input MS to define the velocity frame of the output image.

5.2.7 The `spw` Parameter

The `spw` parameter selects the spectral windows that will be used to form the image, and possibly a subset of channels within these windows.

The syntax for `spw` selection is given in § 2.5.3.

The `spw` parameter is a string or an integer or a list of integers, e.g.

```
spw = '1'
spw = 1
spw = '0,1,2,3'
```

Note that the order in which multiple `spws` are given is important for `mode = 'channel'`, as this defines the origin for the channelization of the resulting image.

5.2.8 The `stokes` Parameter

The `stokes` parameter specifies the Stokes parameters for the resulting images. Note that forming Stokes Q and U images requires the presence of cross-hand polarizations (e.g. RL and LR for circularly polarized systems such as the VLA) in the data. Stokes V requires both parallel hands (RR and LL) for circularly polarized systems or the cross-hands (XY and YX) for linearly polarized systems such as ALMA and ATCA.

This parameter is specified as a string of up to four letters (IQUV). For example,

```
stokes = 'I'           # Intensity only
stokes = 'IQU'        # Intensity and linear polarization
stokes = 'IV'         # Intensity and circular polarization
stokes = 'IQUV'       # All Stokes imaging
```

are common choices. () The output image will have planes (along the “polarization axis”) corresponding to the chosen Stokes parameters.

If the `stokes` parameter is being input to deconvolution tasks such as `clean`, then the chosen Stokes images will be deconvolved jointly rather than sequentially as in AIPS. This is strictly true for `alg='clark'`, and `'cs'`, but cleaning is sequential for `'hogbom'` `clean`.

ALPHA ALERT: The `stokes = 'QU'` for linear polarization only is not currently an option. There is also no option to make single polarization product (e.g. separate RR and LL, or XX and YY) images from data with dual polarizations available. You currently would have to make `stokes='I'` images from data with a single polarization product (e.g. RR or LL) split out.

5.2.9 The `uvfilter` Parameter

This controls the radial weighting of visibilities in the uv-plane (see § 5.2.10 below) through the multiplication of the visibilities by the Fourier transform of an elliptical Gaussian. This is itself a Gaussian, and thus the visibilities are “tapered” with weights decreasing as a function of uv-radius.

The `uvfilter` parameter expands the menu upon setting `uvfilter=True` to reveal the following sub-parameters:

```
uvfilter          =      True  # Apply additional filtering/uv tapering of the visibilities
  uvfilterbmaj    =      1.0  # Major axis of filter (arcseconds)
  uvfilterbmin    =      1.0  # Minor axis of filter (arcseconds)
  uvfilterbpa     =      0.0  # Position angle of filter (degrees)
```

The sub-parameters specify the size and orientation of this Gaussian in the image plane (in arcseconds). Note that since this filter effectively *multiplies* the intrinsic visibility weights, the resulting image will not have a PSF given by the size of the filter, but a PSF given by its intrinsic size convolved by the filter. Thus you should end up with a synthesized beam of size equal to the quadratic sum of the original beam and the filter.

5.2.10 The weighting Parameter

In order to image your data, we must have a map from the visibilities to the image. Part of that map, which is effectively a convolution, is the weights by which each visibility is multiplied before gridding. The first factor in the weighting is the “noise” in that visibility, represented by the data weights in the MS (which is calibrated along with the visibility data). The weighting function can also depend upon the uv locus of that visibility (e.g. a “taper” to change resolution). This is actually controlled by the `uvfilter` parameter (see § 5.2.9). The weighting matrix also includes the convolutional kernel that distributes that visibility onto the uv-plane during gridding before Fourier transforming to make the image of the sky. This depends upon the density of visibilities in the uv-plane (e.g. “natural”, “uniform”, “robust” weighting).

The user has control over all of these.

ALPHA ALERT: You can find a weighting description in the online User Reference Manual at:

<http://casa.nrao.edu/docs/casaref/imager.weight.html>

The `weighting` parameter expands the menu to include various sub-parameters depending upon the mode chosen:

Inside the Toolkit:

The `im.weight` method has more weighting options than available in the imaging tasks. See the **User Reference Manual** for more information on imaging weights.

5.2.10.1 'natural' weighting

For `weighting='natural'`, visibilities are weighted only by the data weights, which are calculated during filling and calibration and should be equal to the inverse noise variance on that visibility. Imaging weight w_i of sample i is given by

$$w_i = \omega_i = \frac{1}{\sigma_k^2} \quad (5.1)$$

where the data weight ω_i is determined from σ_i is the rms noise on visibility i . When data is gridded into the same uv-cell for imaging, the weights are summed, and thus a higher uv density results in higher imaging weights. No sub-parameters are linked to this mode choice. It is the default imaging weight mode, and it should produce “optimum” image with the lowest noise (highest signal-to-noise ratio). Note that this generally produces images with the poorest angular resolution, since the density of visibilities falls radially in the uv-plane

5.2.10.2 'uniform' weighting

For `weighting = 'uniform'`, the data weights are calculated as in 'natural' weighting. The data is then gridded to a number of cells in the uv-plane, and after all data is gridded the uv-cells are re-weighted to have “uniform” imaging weights. This pumps up the influence on the image of data with low weights (they are multiplied up to be the same as for the highest weighted data), which sharpens resolution and reduces the sidelobe level in the field-of-view, but increases the rms image noise. No sub-parameters are linked to this mode choice.

5.2.10.3 'superuniform' weighting

The `weighting = 'superuniform'` mode is similar to the 'uniform' weighting mode but there is now an additional `npixels` sub-parameter that specifies a change to the number of cells on a side (with respect to uniform weighting) to define a uv-plane patch for the weighting renormalization. If `npixels=0` you get uniform weighting.

5.2.10.4 'radial' weighting

The `weighting = 'radial'` mode is a seldom-used option that increases the weight by the radius in the uv-plane, ie.

$$w_i = \omega_i \cdot \sqrt{u_i^2 + v_i^2}. \quad (5.2)$$

Technically, I would call that an inverse uv-taper since it depends on uv-coordinates and not on the data per-se. Its effect is to reduce the rms sidelobes for an east-west synthesis array. This option has limited utility.

5.2.10.5 'briggs' weighting

The `weighting = 'briggs'` mode is an implementation of the flexible weighting scheme developed by Dan Briggs in his PhD thesis. See:

<http://www.aoc.nrao.edu/dissertations/dbriggs/>

This choice brings up four sub-parameters:

```
weighting      = 'briggs' # Weighting to apply to visibilities
                # (natural, uniform, briggs, radial, superuniform)
      rmode     = 'none'  # Robustness mode (for Briggs weighting)
      robust    = 0.0     # Briggs robustness parameter
      noise     = '0.0Jy' # noise parameter for briggs weighting when rmode='abs'
      npixels   = 0       # number of pixels to determine uv-cell size 0=> field of view
```

The key parameter is the `robust` parameter, which sets R in the Briggs equations. The scaling of R is such that $R = 0$ gives a good tradeoff between resolution and sensitivity. The `robust` R takes value between -2.0 (close to uniform weighting) to 2.0 (close to natural).

Briggs sub-parameter `rmode` controls how the `robust` parameter is used. If `rmode='none'`, Briggs weighting is turned off and `robust` is not used.

If `rmode='norm'`, a different Briggs weighting is used, with the sub-parameter `noise` factoring in also.

Superuniform weighting can be combined with Briggs weighting using the `npixels` sub-parameter. This works as in 'superuniform' weighting (§ 5.2.10.3).

See <http://casa.nrao.edu/docs/casaref/imager.weight.html> for a more detailed description of the Briggs weighting modes.

5.2.11 The vis Parameter

The value of the `vis` parameter is either the name of a single MS, or a list of strings containing the names of multiple MSs, that should be processed to produce the image. The MS referred to by the first name in the list (if more than one) is used to determine properties of the image such as channelization and rest frequency.

For example,

```
vis = 'ngc5921.ms'
```

set a single input MS, while

```
vis = ['ngc5921_day1.ms', 'ngc5921_day2.ms', 'ngc5921_day3.ms']
```

Alpha Alert!

Multi-MS handling is not percolated to the tasks yet, as we are still working on this. Use single MS only.

points to three separate measurement sets that will be gridded together to form the image. This means that you do not have to concatenate datasets, for example from different configurations, before imaging.

5.3 Making a Dirty Image and PSF (invert)

To create a “dirty” image of your calibrated *uv* data, and to make a point spread function (psf) associated with that data, use the `invert` task.

The default inputs to `invert` are:

```
# invert :: Calculate a dirty image and dirty beam:

vis          =      '' # Name of input visibility file (MS)
imagename    =      '' # Name of output image
mode         =      'mfs' # Type of selection (mfs, channel, velocity)
imsize      = [256, 256] # Image size i spatial pixels [x,y]; symmetric for single value
cell         = ['1arcsec', '1arcsec'] # Cell size in arcseconds [x,y];
stokes       =      'I' # Stokes parameter to image (I,IV,IQU,IQUV)
field        =      '0' # Field name
spw          =      '0' # Spectral window identifier
weighting    = 'natural' # Weighting to apply to visibilities
                # (natural, uniform, briggs, radial, superuniform)
restfreq     =      '' # restfrequency to use in image
async       =      False # if True run in the background, prompt is freed
```

The `invert` task uses many of the common imaging parameters. These are described above in § 5.2. The output of `invert` will be a set of images named using the `imagename` string as the root (see § 5.2.3).

5.4 Deconvolution using CLEAN (clean)

To create an image and then deconvolve it with the CLEAN algorithm, use the `clean` task. This task will work for single-field data. If you want to deconvolve multi-field data, use the `mosaic` task (§ 5.5) instead. The `clean` task uses many of the common imaging parameters. These are described above in § 5.2. There are also a number of parameters specific to `clean`. These are listed and described below.

The default inputs to `clean` are:

```
# clean :: Calculates a deconvolved image with a selected clean algorithm

vis          =      '' # Name of input visibility file
imagename    =      '' # Pre-name of output images
mode         =      'mfs' # Type of selection (mfs, channel, velocity, frequency)
```



```

alg          = 'clark' # Algorithm to use (hogbom, clark, csclean, multiscale)
niter       = 500 # Number of iterations
gain        = 0.1 # Loop gain for cleaning
threshold   = 0.0 # Flux level to stop cleaning (mJy)
mask        = '' # Name of mask image used in cleaning
cleanbox    = [] # clean box regions or file name or 'interactive'
imsize      = [256, 256] # Image size in pixels [nx,ny]
cell        = ['1.0arcsec', '1.0arcsec'] # Cell size in arcseconds [x,y]
stokes      = 'I' # Stokes parameter to image (I,IV,IQU,IQUV)
field       = '0' # Field name
spw         = '' # Spectral window identifier
weighting   = 'natural' # Weighting to apply to visibilities
              # (natural, uniform, briggs, radial, superuniform)
uvfilter    = False # Apply additional filtering/uv tapering of the visibilities
selecttime  = '1960/01/01/00:00:00~2020/12/31/23:59:59' # range of time to select from data
restfreq    = '' # restfrequency to use in image
async      = False # if True run in the background, prompt is freed

```

A typical setup for clean on the NGC5921 dataset, after setting parameter values, might look like:

```

CASA <102>: inp('clean')
vis          = 'ngc5921_src.split.ms' # Name of input visibility file
imagename    = 'ngc5921_im' # Pre-name of output images
mode         = 'channel' # Type of selection (mfs, channel, velocity, frequency)
  nchan      = 46 # Number of channels to select
  start      = 0 # Start channel
  step       = 1 # Increment between channels/velocity
  width      = 1 # Channel width (value > 1 indicates channel averaging)

alg          = 'csclean' # Algorithm to use (hogbom, clark, csclean, multiscale)
niter       = 6000 # Number of iterations
gain        = 0.1 # Loop gain for cleaning
threshold   = 8.0 # Flux level to stop cleaning (mJy)
mask        = '' # Name of mask image used in cleaning
cleanbox    = [] # clean box regions or file name
imsize      = [512, 512] # Image size in pixels [nx,ny]
cell        = [15.0, 15.0] # Cell size in arcseconds [x,y]
stokes      = 'I' # Stokes parameter to image (I,IV,IQU,IQUV)
field       = '*' # Field name
spw         = '' # Spectral window identifier
weighting   = 'briggs' # Weighting to apply to visibilities
              # (natural, uniform, briggs, radial, superuniform)
  rmode      = 'norm' # Robustness mode (for Briggs weighting)
  robust     = 0.5 # Briggs robustness parameter
  noise      = '0.0Jy' # noise parameter for briggs weighting when rmode='abs'
  npixels   = 0 # number of pixels to determine uv-cell size 0=> field of view

uvfilter    = False # Apply additional filtering/uv tapering of the visibilities
selecttime  = '' # range of time to select from data
restfreq    = '' # restfrequency to use in image
async      = False # if True run in the background, prompt is freed

```

Note that you can also execute the same thing directly using functional form

```
CASA <103>: clean('ngc5921_src.split.ms', 'ngc5921_im', 'channel',
.....: niter=6000, gain=0.1, threshold=8.0, mask='',
.....: nchan=46, start=0, step=1, field='0', imsize=[256, 256],
.....: cell=[15., 15.], weighting='briggs', rmode='norm', robust=0.5)
```

An example of the `clean` task to create a continuum image from many channels is given below:

```
default('clean')                # Make sure the inputs are set to their defaults first!

clean(vis='source.split.ms',    # Use data in source.split.ms
      imagename='ggtau',        # Name output images 'ggtau.*' on disk
      alg='clark',              # Use the Clark CLEAN algorithm
      niter=1000, gain=0.1,     # Iterate 1000 times using gain of 0.1
      mode='mfs',               # make a multi-frequency synthesis map (combine channels)
      nchan=1, start=3, width=58, # Make 1 channel, starting with 5, using 58
      imsize=[200, 200])        # Set image size = 200x200 pixels
      cell=[0.1, 0.1],          # Using 0.1 arcsec pixels
      spw='0,1,2',              # Combine channels from 3 spectral windows
      field='0',                # Use the first field in this split dataset
      stokes='I',               # Image stokes I polarization
      weighting='briggs',       # Use Briggs robust weighting with robustness
      rmode='norm',             # parameter of 0.5
      robust=0.5)
```

This example will clean the entire inner quarter of the primary beam. However, if you want to limit the region over which you allow the algorithm to find clean components then you can make a deconvolution region (or mask). To create a deconvolution mask, use the `makemask` task and input that mask as a keyword into the task above.

Or you can set up a simple `cleanbox` region. To do this, make a first cut at the image and clean the inner quarter. Then use the `viewer` to look at the image and get an idea of where the emission is located. You can use the `viewer adjustment` panel to view the image in pixel coordinates and read out the pixel locations of your cursor.

Then, you can use those pixel read-outs you just go to define a clean box region where you specify the bottom-left-corner (blc) x & y and top-right-corner x& y locations. For example, say you have a continuum source near the center of your image between `blcx`, `blcy`, `trcx`, `trcy` = 80, 80, 120, 120. Then to clean the same image above with this region:

```
default('clean')                # Make sure the inputs are set to their defaults first!

clean(vis='source.split.ms',    # Use data in source.split.ms
      imagename='ggtau',        # Name output images 'ggtau.*' on disk
      alg='clark',              # Use the Clark CLEAN algorithm
```

Inside the Toolkit:

The `im.clean` method is used for CLEANing data. There are a number of methods used to set up the clean, including `im.setoptions`.

```

niter=1000, gain=0.1,      # Iterate 1000 times using gain of 0.1
mode='mfs',              # make a multi-frequency synthesis map (combine channels)
nchan=1, start=3, width=58, # Make 1 channel, starting with 5, using 58
imsize=[200,200])       # Set image size = 200x200 pixels
cell=[0.1,0.1],         # Using 0.1 arcsec pixels
spw='0,1,2',            # Combine channels from 3 spectral windows
field='0',               # Use the first field in this split dataset
stokes='I',             # Image stokes I polarization
weighting='briggs',     # Use Briggs robust weighting with robustness
rmode='norm',           # parameter of 0.5
robust=0.5,
cleanbox=[80,80,120,120]) # Set the deconvolution region as a simple box in the center.

```

5.4.1 Specific clean Parameters

The following are the clean specific parameters and their allowed values:

5.4.1.1 The `alg` Parameter

The `alg` parameter chooses the CLEAN “algorithm” that will be used. The value types are strings. Allowed choices are: `'clark'`, `'hogbom'`, `'csclean'`, and `'multiscale'`. The default is `alg = 'clark'`. If `'multiscale'` is chosen, then the `scales` sub-parameter will be revealed.

The `hogbom` algorithm is the “Classic” image-plane CLEAN, where model pixels are found iteratively by searching for the peak. Each point is subtracted from the full residual image using the shifted and scaled point spread function. In general, this is not a good choice for most imaging problems (`clark` or `csclean` are preferred) as it does not calculate the residuals accurately.

In the `'clark'` algorithm, the cleaning is split into minor and major cycles. In the minor cycles only the brightest points are cleaned, using a subset of the point spread function. In the major cycle, the points thus found are subtracted correctly by using an FFT-based convolution. This algorithm is reasonably fast.

The `csclean` choice specifies the Cotton-Schwab algorithm. Cleaning is split into minor and major cycles. For each field, a Clark-style minor cycle is performed. In the major cycle, the points thus found are subtracted from the original visibilities. A fast variant does a convolution using a FFT. This will be faster for large numbers of visibilities. Double the image size from that used for the Clark clean and set a mask to clean only the inner quarter. This is probably the best choice for high-fidelity deconvolution of images without lots of large-scale structure.

The `multiscale` algorithm uses “Multi-scale CLEAN” to deconvolve using delta-functions and circular Gaussians as the basis functions for the model, instead of just delta-functions or pixels as in the other clean algorithms. This algorithm is still in the experimental stage, mostly because we are working on better algorithms for setting the scales

Inside the Toolkit:

The `im.setscales` method sets the multi-scale Gaussian widths. In addition to choosing a list of sizes in pixels, you can just pick a number of scales and get a geometric series of sizes.

for the Gaussians. The sizes of the Gaussians are set using the `scales` sub-parameter.

The `scale` sub-parameter specifies a list of scales for multiscale CLEAN. These are given in numbers of pixels, e.g.

```
scales = [0,3,10,30]      # Four scales including point sources
scales = [0]              # A delta-function, effectively a Hogbom clean
```

Presumably, these are the FWHM of the Gaussians.

We are working on defining a better algorithm for scale setting. In the toolkit, there is an `nscale` argument which sets scales

$$\theta_i = \theta_{bmin} 10^{(i-N/2)/2} \quad (5.3)$$

where $N = \text{nscales}$ and θ_{bmin} is the fitted FWHM of the minor axis of the CLEAN beam.

5.4.1.2 The `cleanbox` Parameter

If you set `cleanbox='interactive'`, then this will set the interactive mode (see below) where you will get a window in which you can define mask regions while you clean. This also opens up the `npercycle` sub-parameter.

You can give `cleanbox` a list giving the coordinates of a “box” region of the image to restrict the search for components. The default is to restrict `clean` to the inner quarter of the image.

If `cleanbox` is given a list, these are taken to be pixel coordinates for the `blc` and `trc` (bottom-left and top-right corners) of one or more rectangular boxes. For example,

```
cleanbox = [110,110,150,145, 180,70,190,80]
```

defines two boxes.

If `cleanbox` is given a string, then this should point to an ASCII file containing the BLC, TRC of the boxes with one box per line. Each line should contain five numbers

```
<fieldindex> <blc-x> <blc-y> <trc-x> <trc-y>
```

with whitespace separators. Currently the `<fieldindex>` is ignored.

NOTE: In future patches we will include options for the specification of circular and polygonal regions in the `cleanbox` file, as well as the use of world coordinates (not just pixel) and control of plane ranges for the boxes. For now, use the `mask` mechanism for more complicated CLEAN regions.

5.4.1.3 The gain Parameter

The `gain` parameter sets the fraction of the flux density in the residual image that is removed and placed into the clean model at each minor cycle iteration. The default value is `gain = 0.1` and is suitable for a wide-range of imaging problems. Setting it to a smaller gain per cycle, such as `gain = 0.05`, can sometimes help when cleaning images with lots of diffuse emission. Larger values, up to `gain=1`, are probably too aggressive and are not recommended.

5.4.1.4 The mask Parameter

The `mask` parameter takes a string pointing to the name of a mask image to be used for CLEAN to search for components. You can use the `makemask` task to construct this mask.

5.4.1.5 The niter Parameter

The `niter` parameter sets the maximum total number of minor-cycle CLEAN iterations to be performed during this run of `clean`. If restarting from a previous state, it will carry on from where it was. Note that the `threshold` parameter can cause the CLEAN to be terminated before the requested number of iterations is reached.

5.4.1.6 The threshold Parameter

The `threshold` parameter instructs `clean` to terminate when the maximum (absolute?) residual reaches this level or below. Note that it may not reach this residual level due to the value of the `niter` parameter which may cause it to terminate early.

5.4.2 Interactive Cleaning

If `cleanbox='interactive'` is set, then an interactive window will appear at various “cycle” stages while you clean, so you can set and change mask regions. These breakpoints are controlled by the `npercycle` sub-parameter which sets the number of iterations of clean before stopping.

```
cleanbox      = 'interactive'  # clean box regions or file name or 'interactive'
npercycle    =      100       # number of iteration before interactive masking prompt
```

ALPHA ALERT: this is currently the only way (`npercycle`) to control the breakpoints in interactive clean.

The window controls are fairly self-explanatory. It is basically a form of the `viewer`. An example is shown in Figure 5.1. You assign one of the drawing functions (rectangle or polygon, default is rectangle) to the right-mouse button (usually), then use it to mark out regions on the image. Zoom in if necessary (standard with the left-mouse button assignment). Double-click inside the marked region to add it to the mask. If you want to reduce the mask, change “Clean Regions” to **Erase**,

then mark and select as normal. When finished changing your mask, click the green “Masking” **Done** button. If you want to finish your clean with no more changes to the mask, hit the yellow “Masking” **No More** button. If you want to terminate the clean, click the red “Clean” **Stop** button.

For strangely shaped emission regions, you may find using the polygon region marking tool (the second from the right in the button assignment toolbar) the most useful.

For spectral cube images you can use the tapedeck to move through the channels.

See the example use-case script for the Jupiter dataset in Appendix C.2 for examples of using interactive clean.

5.5 Mosaic Deconvolution using CLEAN (mosaic)

To create an image from multiple fields (observations of a region taken with separate pointings) and perform a joint deconvolution on all fields at the same time then you will want to use the `mosaic` task. In other respects, this behaves as the `clean` task (§ 5.4) and shares many of the same inputs. It also uses the common imaging task parameters (§ 5.2).

The default inputs to `mosaic` are:

```
# mosaic :: Calculate a multi-field deconvolved image with selected clean algorithm:

vis           =      '' # Name of input visibility file (MS)
imagename     =      '' # Name of output images: restored=imagename.restored
                # (residual=imagename.residual, model=imagename.model)
mode         =      'mfs' # image spectral definition (mfs, channel, velocity, frequency)
alg          =      'clark' # Algorithm for deconvolution: clark, hogbom, multiscale, entropy
imsize       = [256, 256] # Image size in spatial pixels [x,y]; symmetric for single value
cell         = ['1arcsec', '1arcsec'] # Cell size in arcseconds [x,y];
phasecenter  =      '' # Field Identifier or direction of the mosaic phase center
stokes       =      'I' # Stokes parameter to image (I,IV,IQU,IQUV)
niter        =      500 # Number of iterations; set to zero for no CLEANing
gain         =      0.1 # Loop gain for CLEANing
threshold    =      0.0 # Flux level to stop CLEANing (mJy)
mask         =      [''] # Name(s) of mask image(s) used in CLEANing
cleanbox     =      [] # clean box regions or file name or 'interactive'
field        =      -1 # Field ids list to use in mosaic
spw          =      -1 # Spectral window identifier (0-based)
selecttime   =      '' # range of time to select from data (Not implemented)
restfreq     =      '' # restfrequency to use in image
sdimage      =      '' # Input Single Dish image to use as model
modelimage   =      '' # Output model image name (default=imagename.model)
weighting    =      'natural' # Weighting to apply to visibilities
                # (natural, uniform, briggs, radial, superuniform)
mosweight    =      False # Individually weight the fields of the mosaic
ftmachine    =      'mosaic' # Gridding option (ft, sd, both, mosaic)
cyclefactor  =      1.5 # Change threshold for major cycles (lower=more often)
```

```

cyclespeedup =      -1  # Double clean threshold if not reached in this many iterations
scaletype    =      'NONE' # Image plane flux scale type (NONE, SAULT)
minpb        =       0.1 # Minimum PB level to use
async        =       False # if True run in the background, prompt is freed

```

The `alg`, `mode`, `cleanbox`, and `weighting` parameters open up other sub-parameters. See the `clean` task (§ 5.4) for information on these.

An example of a simple mosaic call is shown below:

```

default('mosaic')           # Make sure the inputs are set to their defaults first!
mosaic(vis='split.n75.ms',  # Use data in split.n75.ms
        imagename='n75',    # Name output images 'n75.*' on disk
        alg='clark',        # Use Clark CLEAN algorithm
        mode='channel',     # Clean individual channel
        niter=10000, gain=0.1, # Allow up to 10000 iterations with a gain of 0.1
        threshold=1,       # Clean down to a threshold of 1 mJy/beam
        nchan=55, start=3, step=1, # Clean 55 channels, starting with 3
        field='0,1,2,3',   # Mosaic the 1st 4 fields in the dataset
        spw='0',           # Select first spectral window
        imsize=[400,400],  # Make an image that is 400x400 pixels
        cell=[1.,1.],      # using 1arcsec pixels
        weighting='briggs', # Use Briggs robust weighting with robustness
        rmode='norm',      # parameter of 0.5
        robust=0.5
        mask='n75.mask')   # You have a mask already made called n75.mask

```

We now describe the use of the `mosaic` specific parameters. See `clean` (§ 5.4) for a description of the parameters in common with that task.

5.5.1 The `cyclefactor` Parameter

The `cyclefactor` parameter allows the user to change the threshold at which the deconvolution cycle will stop and then `degrid` and subtract the model from the visibilities to form the residual. This is with respect to the breaks between minor and major cycles that the `clean` part would normally force. Larger values force a major cycle more often.

Inside the Toolkit:

The `im.setmfcontrol` method sets the parameters that control the cycles and primary beam used in mosaicing.

If your uv-coverage results in a poor PSF, then you should reconcile often (a `cyclefactor` of 4 or 5); For good PSFs, use `cyclefactor` in the range 1.5 to 2.0.

This parameter in effect controls the threshold used by CLEAN to test whether a major cycle break and reconciliation occurs:

$$\text{cycle threshold} = \text{cyclefactor} * \text{max sidelobe} * \text{max residual}$$

5.5.2 The `cyclespeedup` Parameter

The `cyclespeedup` parameter allows the user to let `mosaic` to raise the threshold at which a major cycle is forced if it is not converging to that threshold. To do this, set `cyclespeedup` to an integer number of iterations at which if the threshold is not reached, the threshold will be doubled. See `cyclefactor` above for more details. By default this is turned off (`cyclespeedup = -1`).

5.5.3 The `ftmachine` Parameter

The `ftmachine` parameter controls the gridding method and kernel to be used to make the image. A string value type is expected. Choices are: `'ft'`, `'sd'`, `'both'`, or `'mosaic'` (the default).

The `'ft'` option uses the standard gridding kernel (as used in `invert` or `clean`).

The `'sd'` option forces gridding as in single-dish data.

For combining single-dish and interferometer MS in the imaging, the `'both'` option will allow `mosaic` to choose the `'ft'` or `'sd'` machines as appropriate for the data.

The `'mosaic'` option (the default) uses the Fourier transform of the primary beam (the aperture cross-correlation function in the uv-plane) as the gridding kernel. This allows the data from the multiple fields to be gridded down to a single uv-plane, with a significant speed-up in performance in most (non-memory limited) cases. The effect of this extra convolution is an additional multiplication (apodization) by the primary beam in the image plane. This can be corrected for, but does result in an image with optimal signal to noise ratio across it.

Inside the Toolkit:

The `im.setoptions` method sets the parameters relevant to mosaic imaging, such as the `ftmachine`.

5.5.4 The `minpb` Parameter

The `minpb` parameter sets the level down to which the primary beam (or more correctly the voltage patterns in the array) can go and have a given pixel included in the image. This is important as it defines where the edge of the visible mosaic is. The default is 0.1 or equivalent to the 10% response level. If there is a lot of emission near the edge of the mosaic, then set this lower if you want to be able to clean it out.

5.5.5 The `modelimage` Parameter

The `modelimage` parameter specifies a name to use for the output model image, rather than defaulting from the `imagename` root. This is useful when a single-dish image is input using `sdimage`.

5.5.6 The `mosweight` Parameter

The `mosweight` parameter expects a boolean (`True/False`) to control whether the individual mosaic fields should receive independent weights (for optimum signal to noise ratio) or should be uniformly weighted (to make the signal to noise ratio as uniform as possible across the mosaic).

5.5.7 The `phasecenterid` Parameter

The `phasecenterid` parameter indicates which of the field IDs should be used to define the phase center of the mosaic image. The default action is to use the first one given in the `fieldid` list.

NOTE: This parameter will likely change when we finish updating to the unified selection system.

5.5.8 The `scaletype` Parameter

The `scaletype` parameter controls weighting of pixels in the image plane. The default action `'none'` does no scaling. If `scaletype='sault'` then the image will be re-weighted to have constant noise across it. In this case, the image will also be rescaled to have the correct flux scale across it. This option should be used with care, particularly if your data has very different exposure times (and hence intrinsic noise levels) between the mosaic fields.

Inside the Toolkit:

The `im.setmfcontrol` method gives more options for controlling the primary beam and noise across the image.

5.5.9 The `sdimage` Parameter

The `sdimage` parameter should be used to indicate an image to be used as an input model. The output model will contain this model plus clean components found during deconvolution. This is meant as a way to incorporate single-dish data in the form of an image.

Inclusion of the SD image here is superior to feathering it in later. See § 5.6 for more information on feathering.

5.6 Combined Single Dish and Interferometric Imaging (`feather`)

The term “feathering” is used in radio imaging to describe how to combine or “feather” two images together by forming a weighted sum of their Fourier transforms in the (gridded) uv -plane. Intermediate size scales are down-weighted to give interferometer resolution while preserving single-dish total flux density. For a detailed description of the feathering algorithm see the CASA Toolkit Guide.

The inputs for `feather` are:

```

imagename      =      '' # Name of output feathered image
highres        =      '' # Name of high resolution (synthesis) image
lowres         =      '' # Name of low resolution (single dish) image

```

Note that the only inputs are for images. Note that `feather` does not do any deconvolution but combines presumably deconvolved images after the fact.

Starting with a cleaned synthesis image and a low resolution image from a single dish telescope, the following examples shows how they can be feathered. Note that the single dish image must have a well-defined beam shape and the correct flux units so use task `imhead` first to set some image header properties that are needed first.

```

default('imhead')           # Make sure the inputs are set to their defaults first!
imhead(imagename='single_dish.im', # Select the single-dish image
        brightnessunit='Jy/beam',  # Set the brightness Unit in the header to be Jy/beam
        restoringbeam=['55arcsec', '55arcsec', '0deg'])
                                # Given a known beam shape for this map,
                                # set it as a 55 arcsec Gaussian beam.

default('feather')          # Make sure the inputs are set to their defaults first!
feather(imagename='feather.im',    # Create an image called feather.im
        highres='synth.im',        # The synthesis image is called synth.im
        lowres='single_dish.im')   # The SD image is called single_dish.im
                                    # All images reside in the directory in which
                                    # you started CASA.

```

5.7 Making Deconvolution Masks (makemask)

For most careful imaging, you will want to restrict the region over which you allow CLEAN components to be found. To do this, you can create a 'deconvolution region' or 'mask' image using the `makemask` task. This is useful if you have a complicated region over which you want to clean and it will take many clean boxes to specify.

The parameter inputs for `makemask` are:

```

# makemask :: Derive a mask image from a cleanbox and set of imaging parameters:

cleanbox      =      [] # Clean box file or regions
vis           =      '' # Name of input visibility file (if no input image)
imagename     =      '' # Name of output mask images
mode         =      'mfs' # Type of selection (mfs, channel, velocity)
imsize       = [256, 256] # Image size in spatial pixels [x,y]
cell         =      [1, 1] # Cell size in arcseconds
phasecenter  =      '' # Field identifier or direction of the phase center
stokes       =      'I' # Stokes parameter to image (I,IV,IQU,IQUV)
field        =      '0' # Field ids list to use in mosaic
spw          =      '0' # Spectral window identifier (0-based)

```

The majority of the parameters are the standard imaging parameters (§ 5.2). The `cleanbox` parameter (see § 5.4.1.2 in `clean` above) gives the region to be masked. The `imagename` parameter specifies the name for the output mask image.

You can use the `viewer` to figure out the `cleanbox` blc-trc x-y settings, make the mask image, and then bring it into the viewer as a contour image over your deconvolved image to compare exactly where your mask regions are relative to the actual emission. In this example, create a mask from many `cleanbox` regions specified in a file on disk (`cleanboxes.txt`) containing

```
1 80 80 120 120
2 20 40 24 38
3 70 42 75 66
```

where each line specifies the field index and the blc x-y and trc x-y positions of that `cleanbox`. For example, in `casapy`, you can do this easily:

```
CASA <29>: !cat > cleanboxes.txt
IPython system call: cat > cleanboxes.txt
1 80 80 120 120
2 20 40 24 38
3 70 42 75 66
<CNTRL-D>
CASA <30>: !cat cleanboxes.txt
IPython system call: cat cleanboxes.txt
1 80 80 120 120
2 20 40 24 38
3 70 42 75 66
```

Then, in `CASA`,

```
default('makemask')                # Make sure the inputs are set to their defaults first!

makemask(vis='source.ms',
         imagename='source.mask',
         cleanbox='cleanboxes.txt',
         mode='mfs',                 # make a multi-frequency synthesis map (combine channels)
         imsize=[200,200])          # Set image size = 200x200 pixels
         cell=[0.1,0.1],            # Using 0.1 arcsec pixels
         spw='0,1,2',               # Combine channels from 3 spectral windows
         field='0',                 # Use the first field in this split dataset
         stokes='I')                # Image stokes I polarization
```

This task will then create a mask image that has the 3 `cleanboxes` specified in the `cleanboxes.txt` file.

Note that you must specify a visibility dataset and create the image properties so the mask image will have the same dimensions as the image you want to actually clean.

Eventually we will add functionality to deal with the creation of non-rectangular regions and with multi-plane masks.

5.8 Transforming an Image Model (ft)

The `ft` task will Fourier transform an image and insert the resulting model into the `MODEL_DATA` column of a Measurement Set. You can also convert a CLEAN component list to a model and insert that into the `MODEL_DATA` column. The MS `MODEL_DATA` column is used, for example, to hold the model for calibration purposes in the tasks and toolkit. This is especially useful if you have a resolved calibrator and you want to start with a model of the source before you derive accurate gain solutions. This is also necessary for self-calibration (see § 5.10 below).

The inputs for `ft` are:

```
vis           =      '' # Name of input visibility file
fieldid      =      0  # Field index identifier
field        =      '' # Field name list
model        =      '' # Name of input model image
complist     =      '' # Name of component list
incremental  =      False # Add to the existing MODEL_DATA column?
```

An example of how to do this:

```
default('ft') # Make sure the inputs are set to their defaults first!

ft(vis='n75.ms', # Start with the visibility dataset n75.ms
   field='1328', # Select field name '1328+307' (minimum match)
   model='1328.model.image') # Name of the model image you have already
```

This task will Fourier transform the model image and insert the resulting model in the `MODEL_DATA` column of the rows of the MS corresponding to the source 1328+307.

Note that after `clean`, the transform of the final model is left in the `MODEL_DATA` column so you can go directly to a self-calibration step without explicitly using `ft`.

5.9 Image-plane deconvolution (deconvolve)

If you have only an image (obtained from some telescope) and an image of its point spread function, then you can attempt a simple image-plane deconvolution. Note that for interferometer data, full uv-plane deconvolution using `clean` or similar algorithm is superior!

The default inputs for `deconvolve` are:

```
# deconvolve :: Deconvolving a point spread function from an image
```

Inside the Toolkit:

The `im.ft` method does what the `ft` task does. Its main use is setting the `MODEL_DATA` column in the MS so that the `cb` tool can use it for subsequent calibration.

```

imagenam  =      '' # Name of image to deconvolve
model     =      '' # Name of output image to which deconvolved components are stored
psf       =      '' # Name of psf or gaussian parameters if psf is assumed gaussian
alg       =      'clark' # Deconvolution algorithm to use
niter     =      10 # number of iteration to use in deconvolution process
gain      =      0.1 # CLEAN gain parameter
threshold =      '0.0Jy' # level below which sources will not be deconvolved
mask      =      '' # Name of image that has mask to limit region of deconvolution
async    =      False # if True run in the background, prompt is freed

```

The algorithm (`alg`) options are: 'clark', 'hogbom', 'multiscale' or 'mem'. The 'multiscale' and 'mem' options will open the usual set of sub-parameters for these methods.

5.10 Self-Calibration

Once you have a model image or set of model components reconstructed from your data using one of the deconvolution techniques described above, you can use it to refine your calibration. This is called *self-calibration* as it uses the data to determine its own calibration (rather than observations of special calibration sources).

In principle, self-calibration is no different than the calibration process we described earlier (§ 4). In effect, you alternate between calibration and imaging cycles, refining the calibration and the model as you go. The trick is you have to be careful, as defects in early stages of the calibration can get into the model, and thus prevent the calibration from improving. In practice, it is best to not clean very deeply early on, so that the CLEAN model contains correct components only.

Hint:

The `clearcal` command can be used during the self-calibration if you need to clear the `CORRECTED_DATA` column and revert to the original `DATA`.

The key imaging task to allow self-calibration is currently `ft`, which fills the `MODEL_DATA` column with the Fourier transform of the model (see § 5.8 above). NOTE: in later patches we will change the tasks so that users need not worry what is contained in the MS scratch columns and how to fill them. CASA will handle that underneath for you!

For now, we refer the user back to the calibration chapter for a reminder on how to run the calibration tasks.

ALPHA ALERT: We will have more information here in later updates of this documentation.

5.11 Example of Imaging

The following is an example use of `clean` on the NGC5921 VLA data that we calibrated in the previous Chapter (§ 4.14). This assumes you have already run that script and have all of the defined variable in your session, as well as the split calibrated ms files on disk.

The full NGC5921 example script can be found in Appendix C.1.

ALPHA ALERT: Note that the syntax has been changing recently and this may get out of date quickly!

```
#####  
#  
# Done with calibration  
# Now clean an image cube of N5921  
#  
print '--Clean--'  
default('clean')  
  
# Pick up our split source data  
vis = srcsplitms  
  
# Make an image root file name  
imname = prefix + '.clean'  
imagenname = imname  
  
# Set up the output image cube  
mode = 'channel'  
nchan = 46  
start = 5  
step = 1  
  
# This is a single-source MS with one spw  
field = '0'  
spw = ''  
  
# Set the output image size and cell size (arcsec)  
imsize = [256,256]  
cell = [15.,15.]  
  
# Do a simple Hogbom clean, standard gain factor 0.1  
alg = 'hogbom'  
gain = 0.1  
  
# Fix maximum number of iterations  
niter = 6000  
  
# Also set flux residual threshold (in mJy)  
threshold=8.0  
  
# Set up the weighting  
# Use Briggs weighting (a moderate value, on the uniform side)  
weighting = 'briggs'  
rmode = 'norm'  
robust = 0.5  
  
# No clean mask or cleanbox for now
```

```
mask = ''
cleanbox = []

# But if you had a cleanbox saved in a file, e.g. "regionfile.txt"
# you could use it:
#cleanbox='regionfile.txt'
#
# and if you wanted to use interactive clean
#cleanbox='interactive'

clean()

# Should find stuff in the logger like:
#
# Fitted beam used in restoration: 51.5643 by 45.6021 (arcsec) at pa 14.5411 (deg)
#
# It will have made the images:
# -----
# ngc5921.usecase.clean.image
# ngc5921.usecase.clean.model
# ngc5921.usecase.clean.residual

clnimage = imname+'.image'

#=====
#
# Done with imaging
# Now view the image cube of N5921
#
print '--View image--'
viewer(clnimage,'image')

# Be sure to play through the cube using the tapedeck play button
# and watch the emission move with channel.

#=====
```

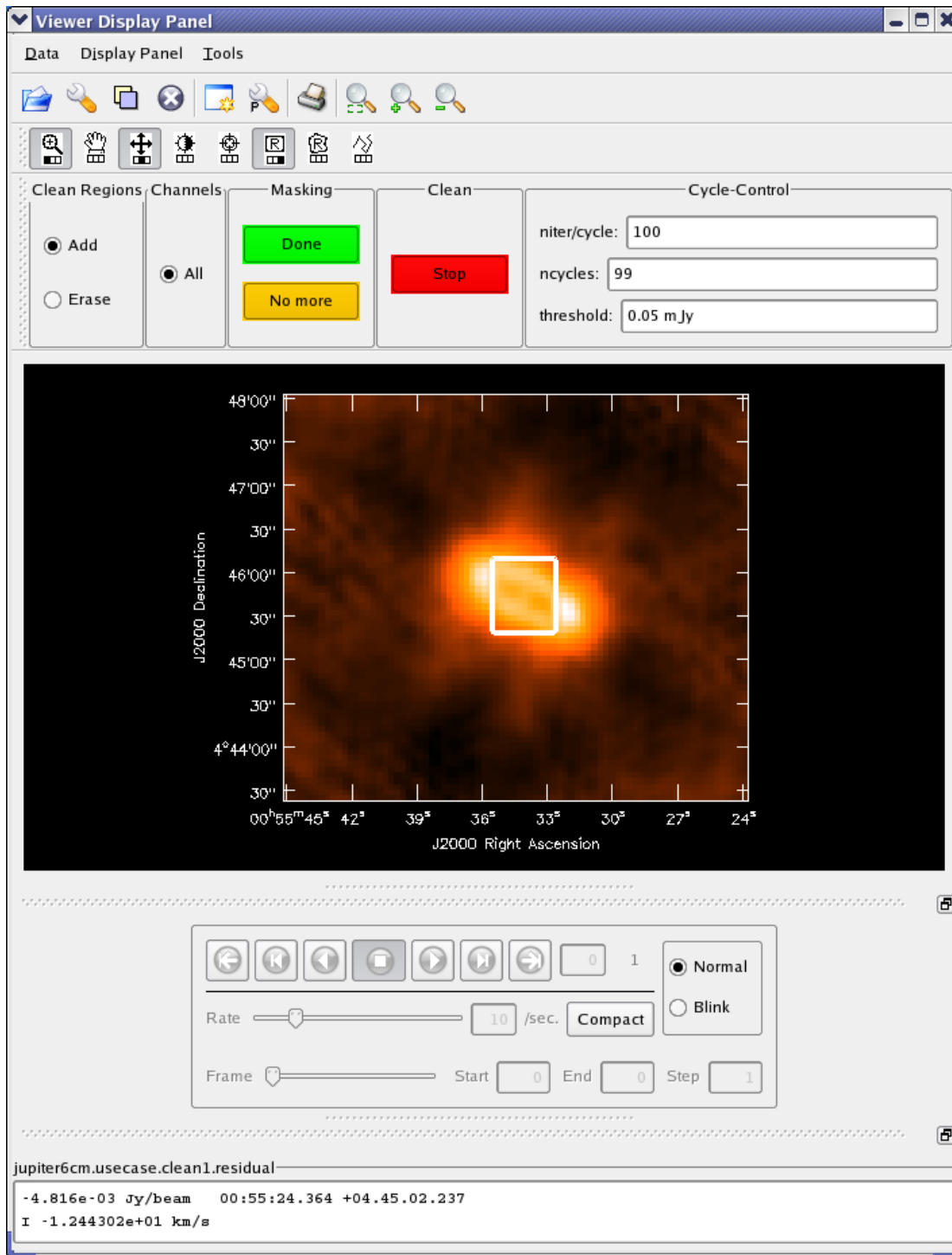


Figure 5.1: Screenshot of the interactive `clean` window during deconvolution of the VLA 6m Jupiter dataset. We have already cleaned 100 iterations in the region previously marked, and are ready to extend the mask to pick up the newly revealed emission. Note the boxes at the top right where the `npercycle`, `niter`, and `threshold` can be changed.

Chapter 6

Displaying Images

This chapter describes how to display data with the `casaviewer` either as a stand-alone or through the `viewer` task. You can display both images and MeasurementSets.

6.1 Starting the viewer

Within the `casapy` environment, there is a `viewer` task which can be used to call up an image. The inputs are:

```
# viewer :: View an image or visibility data set.

infile          =          '' # Name of file to visualize
filetype        =  'image' # Type of file (ms, image, or vector)
```

Examples of starting the viewer:

```
CASA <4>: viewer()

CASA <5>: viewer('ngc5921_task.image')

CASA <6>: viewer('ngc5921.ms', 'ms')
```

ALPHA ALERT: the `viewer` task cannot currently figure out whether a given file is an image or MS, so for now you need to specify `filetype='ms'` explicitly if you want to view an MS in raster mode.

6.1.1 Starting the `casaviewer` outside of `casapy`

The `casaviewer` is the name of the stand-alone application that is available with a CASA installation. From outside `casapy`, you can call this command from the command line in the following ways:

Start the `casaviewer` with no default image/MS loaded; it will pop up the **Load Data** frame and a blank, standard "Viewer Display Panel. Selecting a file on disk in the **Load Data** panel will provide options for how to display the data. Images can be displayed as: 1) Raster Image, 2) Contour Map, 3) Vector map or 4) Marker Map. MS's can only be displayed as raster.

```
> casaviewer &
```

Start the `casaviewer` with the selected image; the image will be displayed in the **Viewer Display Panel**. If the image is a cube (more than one plane for frequency or polarization) then it will be one the first plane of the cube.

```
> casaviewer image_filename &
```

Start the `casaviewer` with the selected MeasurementSet; note the additional parameter indicating that it is an ms; the default is 'image'.

```
> casaviewer ms_filename ms &
```

6.2 The viewer GUI

The main parts of the GUI are the menus:

- Data
 - Open - open an image from disk
 - Register - register selected image (menu expands to the right containing all loaded images)
 - Close - close selected image (menu expands to the right)
 - Adjust - open the adjust panel
 - Print - print the displayed image
 - Close Panel - close the Viewer Display Panel
 - Quit Viewer - currently disabled
- Display Panel
 - New Panel - create a new Viewer Display Panel
 - Panel Options - open the panel options frame
 - Print - print displayed image
 - Close Panel - close the Viewer Display Panel
- Tools

- Currently blank - will hold annotations and image analysis tools

Below this are icons for fast access to some of these menu items:

- folder - Data:Open shortcut – pulls up Load Data panel
- wrench - Data:Adjust shortcut – pulls up Data Display Options panel
- panels - Data:Register shortcut – pull up menu of loaded data
- delete - Data:Close shortcut – closes/unloads selected data
- panel - Display Panel:New Panel
- panel wrench - Display Panel:Panel Options – pulls up Viewer Canvas Manager
- print - Display Panel:Print – print data

Important Bug Note: Please use the icon buttons whenever possible instead of the menus. The Register and Close menus especially are known to lead to viewer crashes in some cases. You'll usually find that the first four icon buttons are all you need. Click on the display panel titlebar then hover over the buttons for brief reminders of their purpose.

Below this are the eight mouse control buttons. These allow/show the assignment of the mouse buttons for different operations. Clicking in one of these buttons will re-assign a mouse button to that operation.

- **Zooming (magnifying glass icon)** Zooming is accomplished by pressing down the selected mouse button at the start point, dragging the mouse away from that point, and releasing the selected mouse button when the zoom box encloses the desired zoom area. Once the button is released, the zoom rectangle can be moved by clicking inside it with the selected mouse button and dragging it around. To zoom in, simply double click with the selected button inside the rectangle. Double clicking outside the rectangle will result in a zoom out.
- **Panning (hand icon)** Panning is accomplished by pressing down on the selected mouse button at the point you wish to move, dragging the mouse to the position where you want the first point moved to, and releasing the selected mouse button. *Note: The arrow keys, Page Up, Page Down, Home and End keys, and scroll wheel (if any) can also be used to scroll through your data once you have zoomed in. For these to work, the mouse must be over the display panel drawing area, but no mouse tool need be active. Note: this is currently not enabled.*
- **Stretch-shift colormap fiddling**
- **Brightness-contrast colormap fiddling**

- **Positioning** This enables the user to place a crosshair marker on the image to indicate a position. Depending on the context, the positions may be used to flag MeasurementSet data (not yet enabled) or display image spectral profiles (also not currently enabled). Click on the position to place the crosshair; once placed you can drag it to move to another location. Double click is not needed for this control.
- **Rectangle and Polygon region drawing** A rectangle region is generated exactly the same way as the zoom rectangle, and is set by double clicking within the rectangle. Polygon regions can be constructed by progressively clicking the selected mouse button at the desired location of each vertex, and clicking in the same location twice to complete the polygon. Once constructed, it can be moved by dragging inside the polygon, and reshaped by dragging the various handles at the vertices.
- **Polyline drawing** A polyline can be constructed with this button selected. It is almost identical to the polygon region tool. Create points by clicking at the positions wanted and then double-click to finish the line.

Below this area is the actual display surface.

Below the display is the 'tape deck' which provides basic movement between image planes along a selected third dimension of an image cube. This set of buttons is only enabled when the first-registered image reports that it has more than one plane along the 'Z axis'. In the most common case, the animator controls the frequency channel being viewed. From left to right, the tape deck controls allow the user to:

- rewind to the start of the sequence (i.e., the first plane)
- step backwards by one plane
- play backwards, or repetitively step backwards
- stop any current play
- play forward, or repetitively step forward
- step forward by one plane
- fast forward to the end of the sequence

To the right of the tape deck is an editable text box indicating the current frame number and a sunken label showing the total number of frames. One can type a channel number into the current frame to jump to that channel. Below this is a slider for controlling the animation speed. To the right of this is the 'Full/Compact' toggle. In full mode, additional controls for blinking and for controlling the frame value and step are available; the default setting is for compact. In 'Blink' mode, when more than one raster image is registered in the Viewer Display Panel, the tapedeck will control which is being displayed at the moment. The images registered should cover the same portion of the sky, using the same coordinate projection.

6.3 Viewing a raster map

A raster map of an image shows pixel intensities in a two-dimensional cross-section of gridded data with colors selected from a finite set of (normally) smooth and continuous colors, i.e., a colormap.

Starting the `casaviewer` with an image as a raster map will look something like:

You will see the GUI which consists of two main windows, entitled "Viewer Display Panel" and "Load Data". In the "Load Data" panel, you will see all of the files in the current working directory along with their type (Image, MeasurementSet, etc). After selecting a file, you are presented with the available data types for these data. Clicking on the button **Raster Map** will create a display as above. The main parts of the "Viewer Display Panel" GUI are discussed in the following Section.

6.4 Viewing a contour map

Viewing a contour image is similar the process above. 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.

6.5 Viewing a MeasurementSet with visibility data

Visibility data can also be displayed and flagged directly from the viewer (*Note: flagging is not currently enabled*). For MeasurementSet files the only option for display is 'Raster' (similar to AIPS task TVFLG).

Note: There is also a bug in the current MS viewing which disables display of the data and flags; use the 'Adjust' panel 'Flagging Options' Menu to change the 'Show Flagged Regions' option to 'Masked to Background'. This will be the default for Patch 2.

6.6 Adjusting Display Parameters

The data display can be adjusted by the user as needed. The following illustrate the available options in the categories of:

- Display axes
- Hidden axes
- Basic Settings
- Position tracking
- Axis labels

- Axis label properties

This older web page gives details of individual display options. Although it has not yet been integrated into the reference manual for the newer CASA, it is accurate in most cases:

<http://aips2.nrao.edu/daily/docs/user/Display/node267.html>

6.7 Adjusting Canvas Parameters/Multi-panel displays

The display area or Canvas can also be manipulated through two sets of values:

- Margins - specify the spacing for the left, right, top, and bottom margins
- Number of panels - specify the number of panels in x and y and the spacing between those panels.

The following illustrates a multi-panel display along with the Viewer Canvas Manager settings which created it.

6.8 Overlay contours on a raster map

Contours of either a second data set or the same data set can be used for comparison or to enhance visualization of the data. The Adjust Panel will have multiple tabs which allow adjusting each data set individually (Note tabs along the top). To enable this simply open up the Load Data panel (Use the Data menu or click on the Folder icon), select the data set and select Contour.

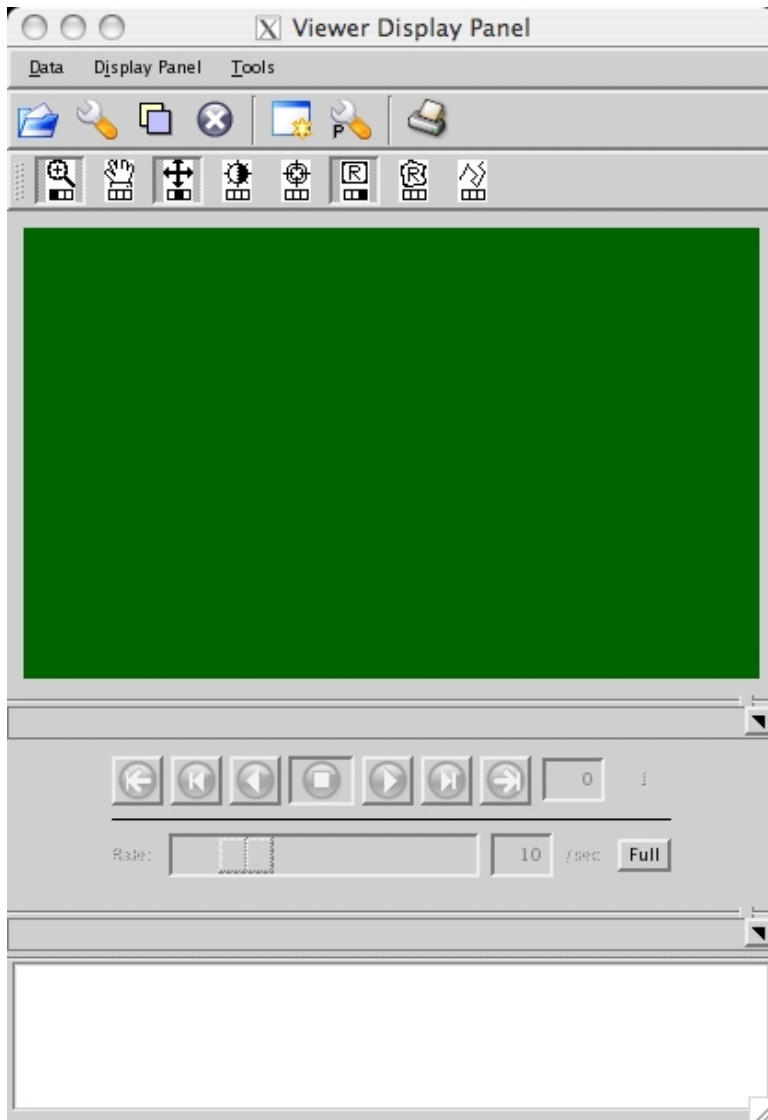


Figure 6.1: Viewer Display Panel with no data loaded. Each section of the GUI is explained below

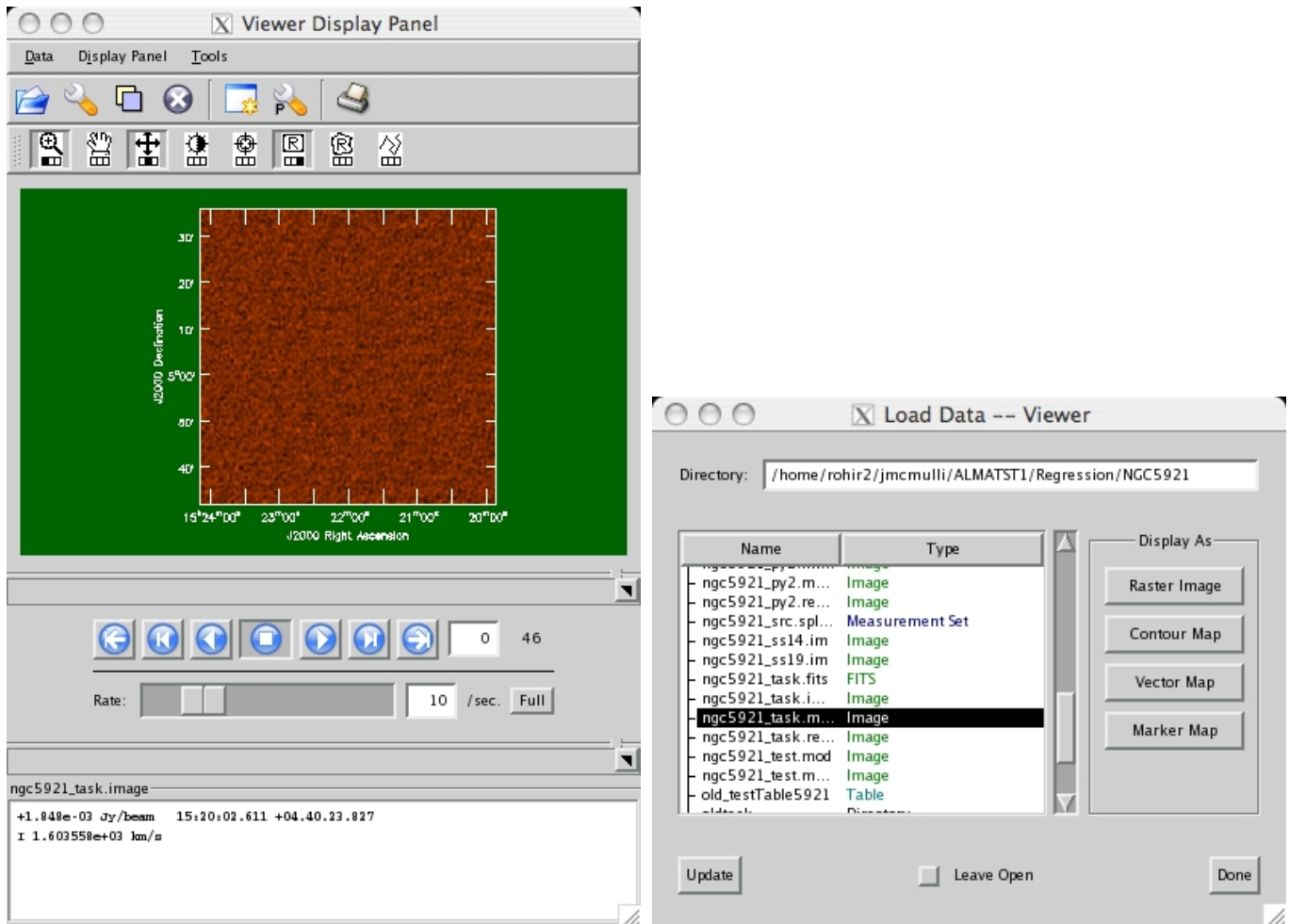


Figure 6.2: casaviewer: Illustration of a raster image in the Viewer Display Panel(left) and the Load Data panel (right).

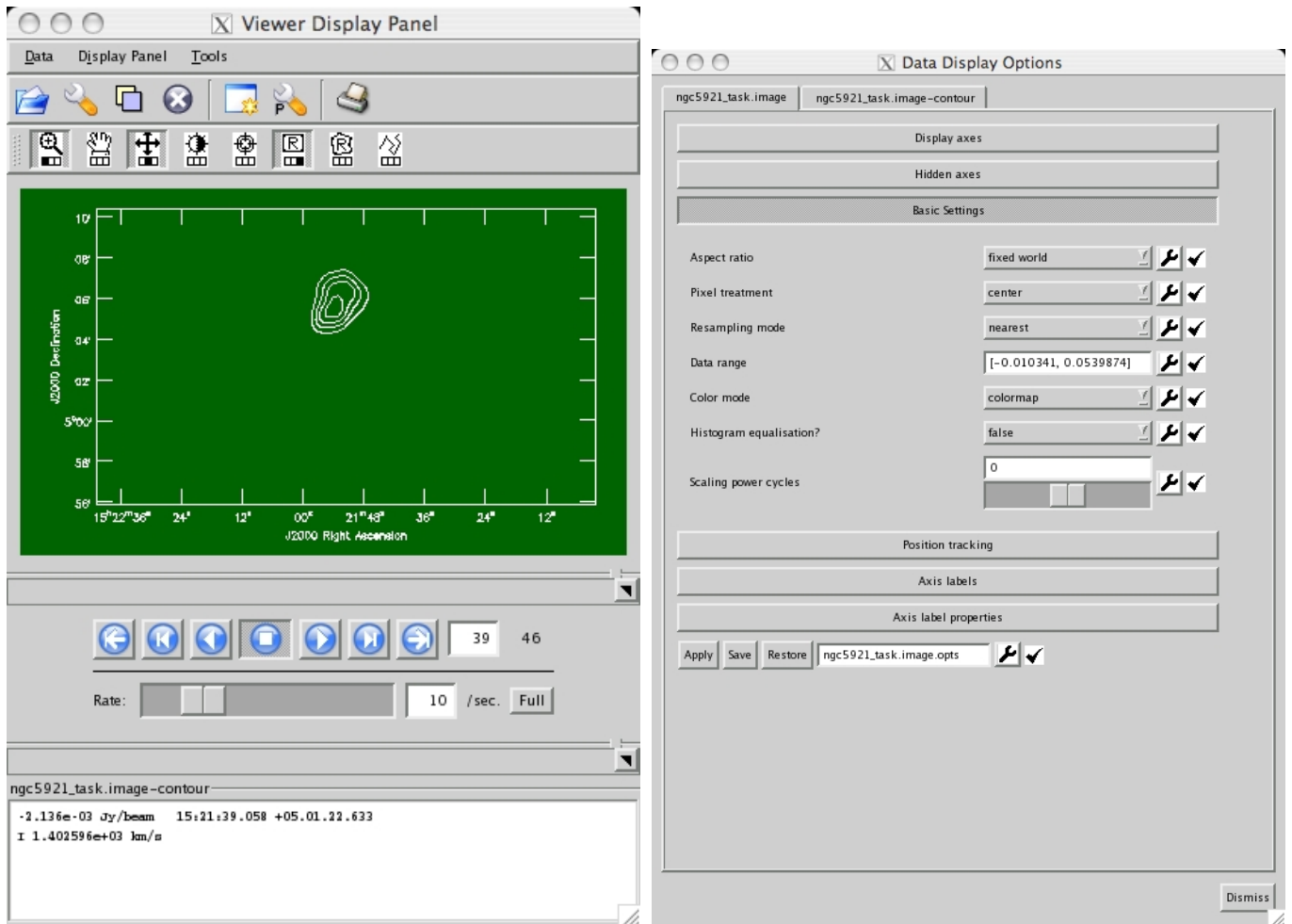


Figure 6.3: casaviewer: Illustration of a raster image in the Viewer Display Panel(left) and the Load Data panel (right).

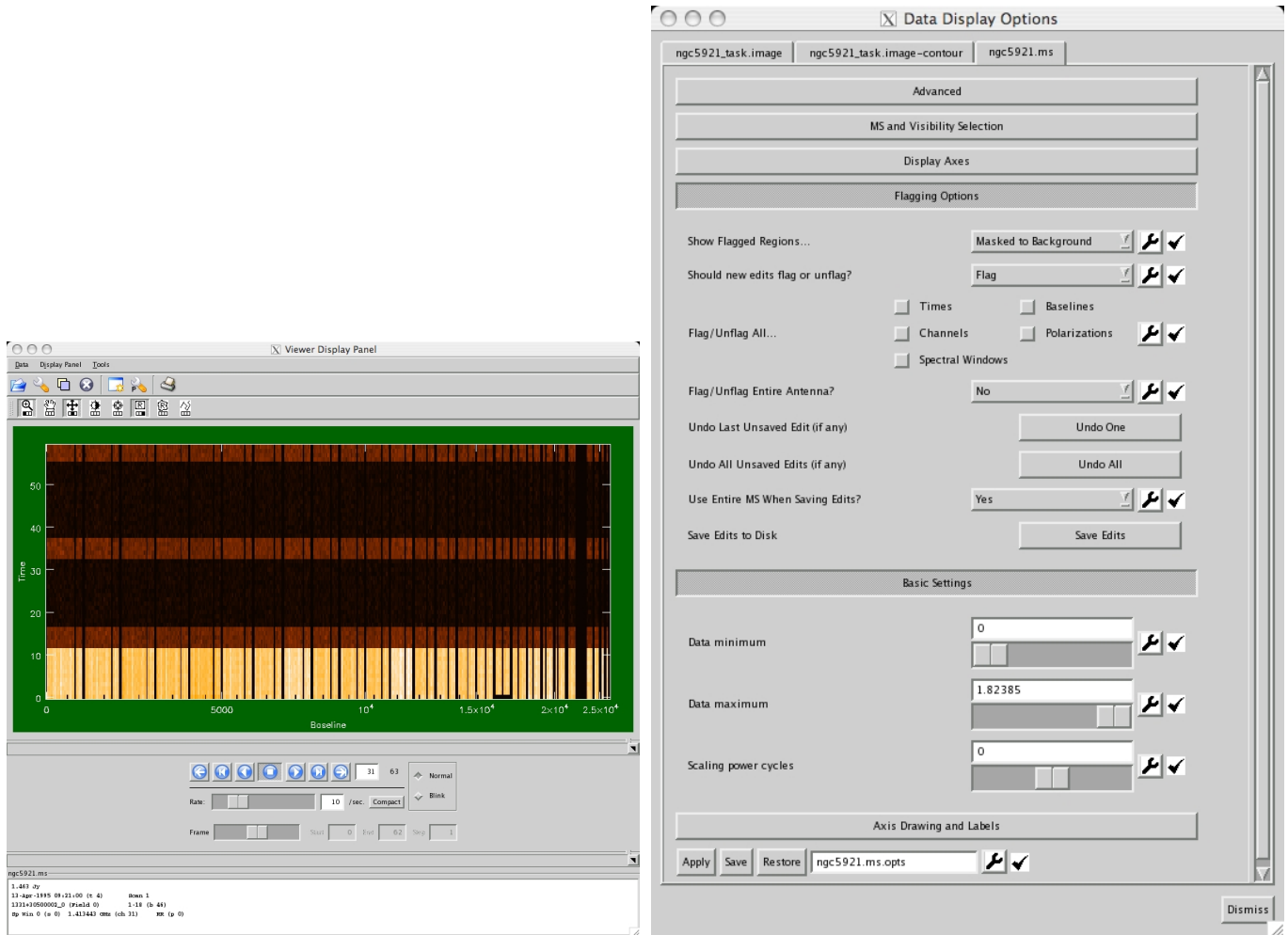


Figure 6.4: casaviewer: Display of visibility data. The default axes are time vs. baseline.

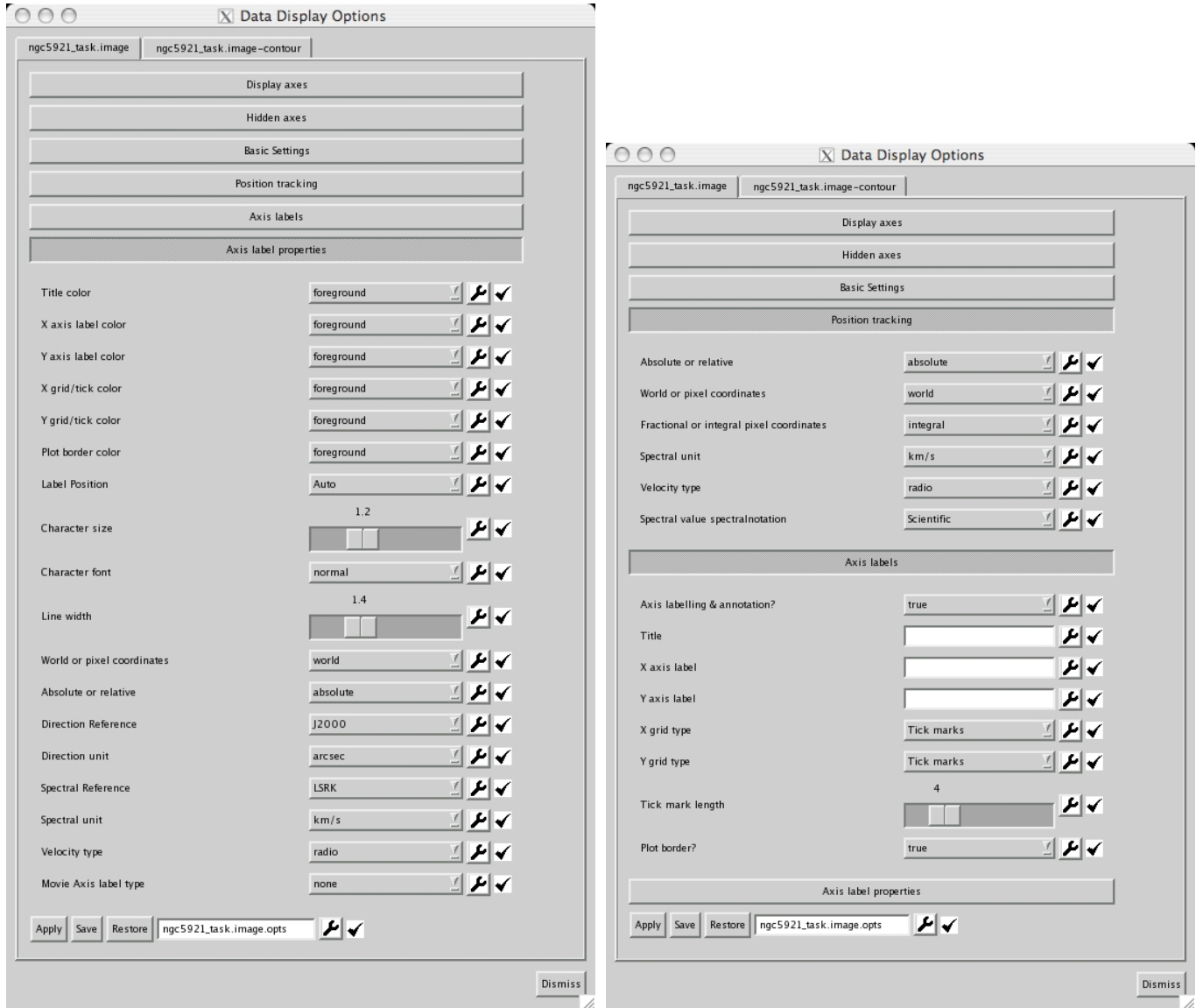


Figure 6.5: casaviewer: Data display options. In the left panel, the Display axes, Hidden axes, and Basic Settings options are shown; in the right panel, the Position tracking and Axis labels options are shown.

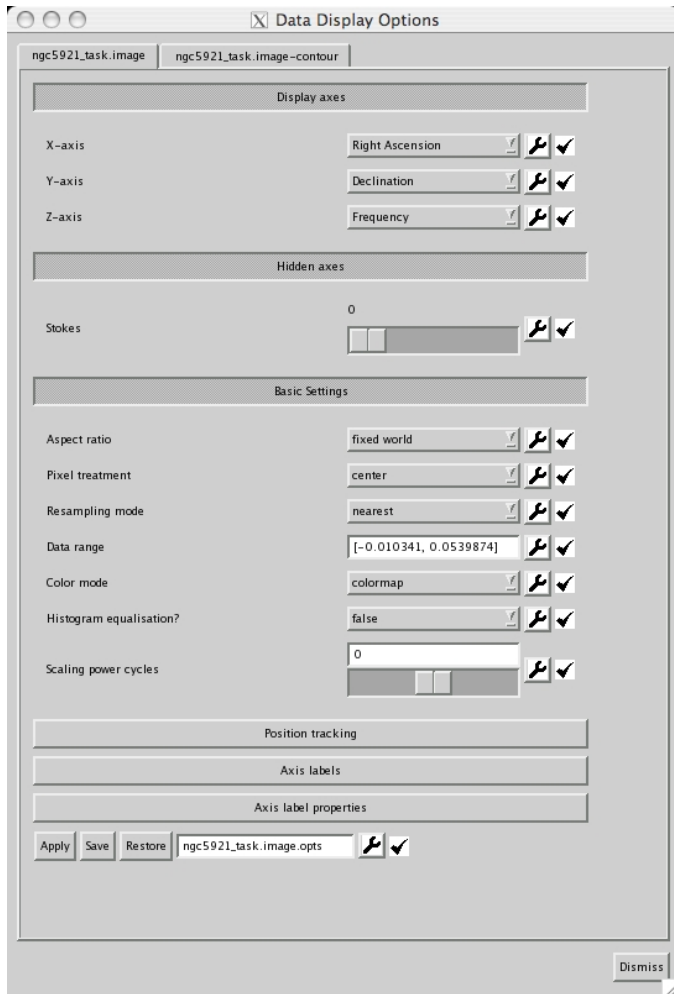


Figure 6.6: casaviewer: Data display options. In this final, third panel , the Axis label properties are shown.



Figure 6.7: casaviewer: A multi-panel display set up through the Viewer Canvas Manager.

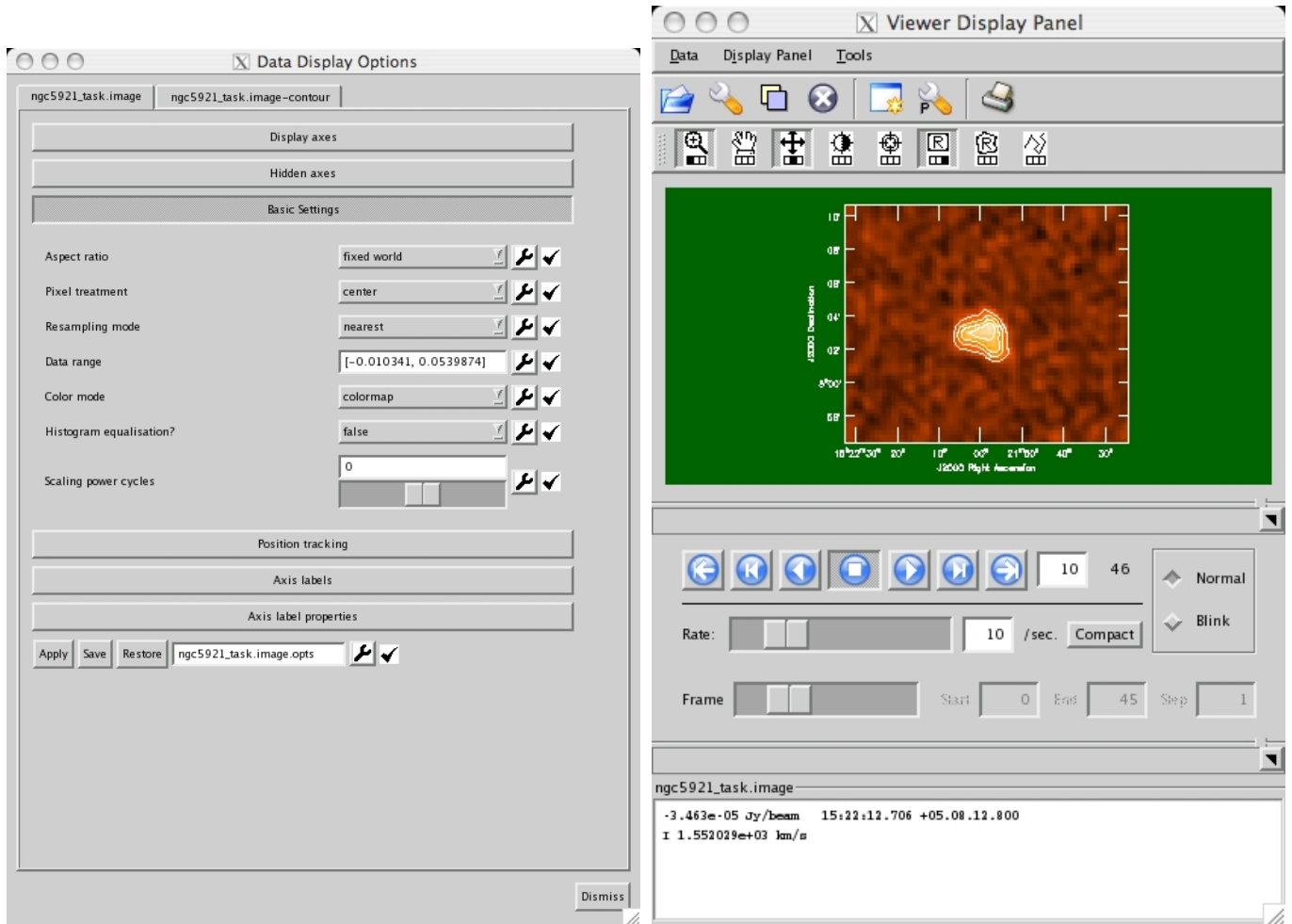


Figure 6.8: casaviewer: Display contour overlay on top of a raster image.

Chapter 7

Image Analysis

Once data has been calibrated (and imaged in the case of synthesis data), the resulting image or image cube must be displayed or analyzed in order to extract quantitative information, such as statistics or moment images. In addition, there need to be facilities for the coordinate conversion of images for direct comparison.

ALPHA ALERT: We have assembled a skeleton of image analysis tasks for this alpha release. Many more are still under development.

The image analysis tasks are:

- `imhead` — summarize and manipulate the “header” information in a CASA image (§ 7.1)
- `immoments` — compute the moments of an image cube (§ 7.2)
- `regridimage` — regrid an image onto the coordinate system of another image (§ 7.3)
- `viewer` — there are useful region statistics and image cube slice and profile capabilities in the viewer (§ 7.4)

Inside the Toolkit:

Image analysis is handled in the `ia` tool. Many options exist there, including moments and image math. See the **CASA Toolkit Guide** for more information.

7.1 Summary of an Image and Headers

To get a summary of the properties of your image, type:

```
imhead(imagename='image.im')
```

You will get 2 outputs: A nicely formatted summary in the log window, e.g.:

```

Sun Mar 4 22:20:09 2007   NORMAL image::summary:
Direction reference : J2000
Spectral reference   : TOPO (-> LSRK)
Velocity type      : RADIO
Pointing center    : 23:13:43.815918 +61.27.00.178403
Telescope         : VLA
Observer          : unavailable
Date observation   : 1995/05/19/09:23:45

```

Axis	Coord Type	Name	Proj	Shape	Tile	Coord value at pixel	Coord incr	Units
1	1	Direction Right Ascension	SIN	128	64	23:13:43.816	65.00	-4.000000e+00 arcsec
2	1	Direction Declination	SIN	128	32	+61.27.00.178	65.00	4.000000e+00 arcsec
3	2	Stokes Stokes		1	1	I		
4	3	Spectral Frequency		55	11	2.36918e+10	1.00	9.765625e+04 Hz

along with a detailed list of header info on the command line:

```

CASA <27>: summary
Out[27]:
{'header': {'axisnames': array(['Right Ascension', 'Declination', 'Stokes', 'Frequency'],
 dtype='|S16'),
 'axisunits': array(['rad', 'rad', '', 'Hz'],
 dtype='|S4'),
 'defaultmask': '',
 'hasmask': False,
 'imagetype': 'PagedImage',
 'incr': array([-1.93925472e-05, 1.93925472e-05, 1.00000000e+00,
 9.76562500e+04]),
 'masks': array([],
 dtype='|S1'),
 'ndim': 4,
 'refpix': array([ 65., 65., 1., 1.]),
 'refval': array([ 6.08129550e+00, 1.07250569e+00, 1.00000000e+00,
 2.36917611e+10]),
 'restoringbeam': {'imagetype': 'Intensity',
 'objectname': '',
 'restoringbeam': {'major': {'unit': 'arcsec',
 'value': 13.205469131469727},
 'minor': {'unit': 'arcsec',
 'value': 12.592819213867188},
 'positionangle': {'unit': 'deg',
 'value': -2.3764669895172119}}},
 'shape': array([128, 128, 1, 55]),
 'tileshape': array([64, 32, 1, 11]),
 'unit': 'Jy/beam'},
'return': []}

```

All Python dictionaries have a range of functions to work with (do a `summary.ġTABġ` to see all the options). For example:


```

summary.keys()          # returns all the keys to the dictionary ['header','return']
summary.get('header')  # returns all the elements of the header
summary['header']      # equivalent command

x=summary.values()[0]  # set x = the header dictionary
x['ndim']              # returns 4 for a standard CASA image

```

7.2 Computing the Moments of an Image Cube (immoments)

For spectral line datasets, the output of the imaging process is an `image cube`, with a frequency or velocity channel axis in addition to the two sky coordinate axes. This can be most easily thought of as a series of image planes stacked along the spectral dimension.

A useful product to compute is to collapse the cube into a *moment* image by taking a linear combination of the individual planes:

$$M_m(x_i, y_i) = \sum_k^N w_m(x_i, y_i, v_k) I(x_i, y_i, v_k) \quad (7.1)$$

for pixel i and channel k in the cube I . There are a number of choices to form the m moment, usually approximating some polynomial expansion of the intensity distribution over velocity mean or sum, gradient, dispersion, skew, kurtosis, etc.). There are other possibilities (other than a weighted sum) for calculating the image, such as median filtering, finding minima or maxima along the spectral axis, or absolute mean deviations. And the axis along which to do these calculation need not be the spectral axis (ie. do moments along Dec for a RA-Velocity image). We will treat all of these as generalized instances of a “moment” map.

The `immoments` task will compute basic moment images from a cube. The default inputs are:

```

# immoments :: Compute moments of an image cube:

imagename   =      ''   # Input image name
moments     =      [0]  # List of moments to compute
axis        =      3    # Axis for moment calculation
planes      =      [''] # Set of planes/channels to use for moment
includepix  =      [-1] # Range of pixel values to include
excludepix  =      [-1] # Range of pixel values to exclude
outfile     =      ''   # Output image file name (or root for multiple moments)
async       =      False # if True run in the background, prompt is freed

```

The choices for the operation mode are:

```

moments=-1  - mean value of the spectrum
moments=0   - integrated value of the spectrum
moments=1   - intensity weighted coordinate;traditionally used to get
              'velocity fields'
moments=2   - intensity weighted dispersion of the coordinate; traditionally

```

```

                                used to get 'velocity dispersion'
moments=3   - median of I
moments=4   - median coordinate
moments=5   - standard deviation about the mean of the spectrum
moments=6   - root mean square of the spectrum
moments=7   - absolute mean deviation of the spectrum
moments=8   - maximum value of the spectrum
moments=9   - coordinate of the maximum value of the spectrum
moments=10  - minimum value of the spectrum
moments=11  - coordinate of the minimum value of the spectrum

```

The meaning of these is described in the CASA Reference Manual (<http://casa.nrao.edu/docs/casaref/image.moments.html>).

7.3 Regridding an Image (regridimage)

It is occasionally necessary to regrid an image onto a new coordinate system. The `regridimage` task will regrid one image onto the coordinate system of another, creating an output image. In this task, the user need only specify the names of the input, template, and output images.

If the user needs to do more complex operations, such as regridding an image onto an arbitrary (but known) coordinate system, changing from Equatorial to Galactic coordinates, or precessing Equinoxes, the CASA toolkit can be used (see sidebox). Some of these facilities will eventually be provided in task form.

The default inputs are:

```

# regridimage :: Regrid imagename to have template image parameters

imagename  =      '' # Name of image to be regridded
template   =      '' # image having the parameters that is wanted in regridded image
output     =      '' # Name of image in which result of regridding is stored
async      =      False # if True run in the background, prompt is freed

```

Inside the Toolkit:

More complex coordinate system and image regridding operation can be carried out in the toolkit. The `cs` (`coordsys`) tool and the `ia.regrid` method are the relevant components.

7.4 Image display in the viewer

The `viewer` is the workhorse for the visual display of images, the analysis of slices through cubes, and the statistics of regions.

7.4.1 Image statistics

You can use the viewer to interactively obtain image statistics on a region:

```
viewer('imagenamename.im')
# Now use the right mouse button (default for region setting)
# create a region and then double click inside to obtain statistics on that region
# Currently this supports a single plane only and the output goes to your casapy
# terminal window as:
```

```
ngc5921_task.image
```

n	Std Dev	RMS	Mean	Variance	Sum
660	0.01262	0.0138	0.005622	0.0001592	3.711
Flux	Med Dev	Quartile	Median	Min	Max
0.3119	0.003758	0.004586	0.001434	-0.009671	0.06105

7.5 Image Import/Export to FITS

To export your images to fits format use the `exportfits` task:

```
exportfits(imagenamename='casa.im', # select CASA format image, casa.im
           fitsimage='image.fits') # write out a fits format image image.fits
```

You can also use the `importfits` task to import fits image into CASA format. Note, the CASA viewer can read fits images so you don't need to do this if you just want to look at the image.

```
importfits(fitsimage='image.fits', # select fits format image
           imagenamename='casa.im') # write out a CASA format image
```

Chapter 8

Single Dish Data Processing

For single-dish spectral calibration and analysis, CASA uses the ATNF Spectral Analysis Package (ASAP). This is imported as the `sd` tool, and forms the basis for a series of tasks (the “SDtasks”) that encapsulate the functionality within the standard CASA task framework. ASAP was developed to support the Australian telescopes such as Mopra, Parkes, and Tidbinbilla, and we have adapted it for use within CASA for GBT and eventually ALMA data also. For details on ASAP, see the ASAP home page at ATNF:

- <http://www.atnf.csiro.au/computing/software/asap/>

You can also download the ASAP User Guide and Reference Manual at this web site. There is also a brief tutorial. Note that within CASA, the ASAP tools are prefaced with `sd.`, e.g. where it says in the ASAP User Guide to use `scantable` you will use `sd.scantable` in CASA. See § 8.3 for more information on the tools.

All of the ASAP functionality is available with a CASA installation. In the following, we outline how to access ASAP functionality within CASA with the tasks and tools, and the data flow for standard use cases.

If you run into trouble, be sure to check the list of known issues and features of ASAP and the SDtasks presented in § 8.5 first.

8.1 Guidelines for Use of ASAP and SDtasks in CASA

8.1.1 Environment Variables

There are a number of environment variables that the ASAP tools (and thus the SDtasks) use to help control their operation. These are described in the ASAP User Guide as being in the `.asaprc` file. Within CASA, these are contained in the Python dictionary `sd.rcParams` and are accessible through its keys and values. For SDtask users, the most important are the `verbose` parameter controlling the display of detailed messages from the tools. By default

```
sd.rcParams['verbose'] = True
```

and you get lots of messages. Also), and the `scantable.storage` parameter controlling whether scantable operations are done in memory or on disk. The default

```
sd.rcParams['scantable.storage'] = 'memory'
```

does it in memory (best choice if you have enough), while to force the scantables to disk use

```
sd.rcParams['scantable.storage'] = 'disk'
```

which might be necessary to allow processing of large datasets. See § 8.3.1 for more details on the ASAP environment variables.

8.1.2 Assignment

Some ASAP methods and function require you to assign that method to a variable which you can then manipulate. This includes `sd.scantable` and `sd.selector`, which make objects. For example,

```
s = sd.scantable('OrionS_rawACSmod', average=False)
```

8.1.3 Lists

For lists of scans or IFs, such as in `scanlist` and `iflist` in the SDtasks, the tasks and functions want a comma-separated Python list, e.g.

```
scanlist = [241, 242, 243, 244, 245, 246]
```

You can use the Python `range` function to generate a list of consecutive numbers, e.g.

```
scanlist = range(241,247)
```

giving the same list as above, e.g.

```
CASA <3>: scanlist=range(241,247)
CASA <4>: print scanlist
[241, 242, 243, 244, 245, 246]
```

You can also combine multiple ranges by summing lists

```
CASA <5>: scanlist=range(241,247) + range(251,255)
CASA <6>: print scanlist
[241, 242, 243, 244, 245, 246, 251, 252, 253, 254]
```

Note that in the future, the `sd` tools and `SDtasks` will use the same selection language as in the synthesis part of the package.

Spectral regions, such as those for setting masks, are pairs of min and max values for whatever spectral axis unit is currently chosen. These are fed into the tasks and tools as a list of lists, with each list element a list with the `[min,max]` for that sub-region, e.g.

```
masklist=[[1000,3000], [5000,7000]].
```

8.1.4 Dictionaries

Currently, the `SDtasks` use the Python dictionary `xstat` as a return variable for the results of line fitting (in `sdfit`) and region statistics (in `sdstat`). You can then access the elements of these through the keywords, e.g.

```
CASA <10>: sdstat()
Current fluxunit = K
No need to convert fluxunits
Using current frequency frame
Using current doppler convention
```

```
CASA <11>: xstat
Out[11]:
{'eqw': 70.861755476162784,
 'max': 1.2750182151794434,
 'mean': 0.35996028780937195,
 'median': 0.23074722290039062,
 'min': -0.20840644836425781,
 'rms': 0.53090775012969971,
 'stddev': 0.39102539420127869,
 'sum': 90.350028991699219}
```

You can then use these values in scripts by accessing this dictionary, e.g.

```
CASA <12>: line_stat = xstat

CASA <13>: print "Line max = %5.3f K" % (line_stat['max'])
Line max = 1.275 K
```

for example.

8.1.5 Line Formatting

The `SDtasks` trap leading and trailing whitespace on string parameters (such as `infile` and `sdfilename`), but `ASAP` does not, so be careful with setting string parameters. `ASAP` is also case-sensitive, with most parameters being upper-case, such as `ASAP` for the `sd.scantable.save` file format. The `SDtasks` are generally more forgiving.

Also, beware Python's sensitivity to indenting.

8.2 Single Dish Analysis Tasks

A set of single dish tasks is available for simplifying basic reduction activities. Currently the list includes:

- **sdcal** — select, calibrate, average, smooth, and fit/remove spectral baselines from SD data
- **sdfit** — line fitting to SD spectra
- **sdlist** — print a summary of a SD dataset
- **sdplot** — plotting of SD spectra, including overlay of line catalog data
- **sdstat** — compute statistics of regions of SD spectra

All of the SDtasks work from a file on disk rather than from a scantable in memory as the ASAP toolkit does (see § 8.3. Inside the tasks we invoke a call to `sd.scantable` to read in the data. The scantable objects do not persist within CASA after completion of the tasks, and are destroyed to free up memory.

The task `sdcal` is the workhorse for the calibration, selection, averaging, baseline fitting, smoothing, and writing of datasets. It is the only SDtask that can write out a dataset. Its operation is controlled by three main "mode" parameters: `calmode` (which selects the type of calibration, if any, to be applied), `kernel` (which selects the smoothing), and `blmode` (which selects baseline fitting). There are also parameters controlling the selection such as `scanlist`, `iflist`, `field`, `scanaverage`, `timeaverage`, and `polaverage`. Note that `sdcal` can be run with `calmode='none'` to allow re-selection or writing out of data that is already calibrated.

There is a "wiring diagram" of the dataflow and control inputs for `sdcal` shown in Figure 8.1. This might help you chart your course through the calibration.

The SDtasks support the import and export file formats supported by ASAP itself. For import, this includes: ASAP (scantables), MS (casa measurement set), RPFITS and SDFITS. For export, this includes: ASAP (scantables), MS (casa measurement set), ASCII (text file), SDFITS (a flavor of SD FITS).

You can get a brief summary of the data in a file using the `sdlist` task.

Plotting of spectra is handled in the `sdplot` task. It also offers some selection, averaging and smoothing options in case you are working from a dataset that has not been split or averaged. Note that there is some rudimentary plotting capability in the `sdcal` and `sdfit` tasks, controlled through the `plotlevel` parameter, to aid in the assessment of the performance of these tasks.

Basic statistics on spectral regions is available in the `sdstat` task. Results are passed in a Python dictionary return variable `xstat`.

Basic Gaussian line-fitting is handled by the `sdfit` task. It can deal with the simpler cases, and offers some automation, but more complicated fitting is best accomplished through the toolkit (`sd.fitter`).

8.2.1 SDtask Summaries

The following are the list of parameters and brief descriptions of each of the SDtasks. These descriptions are also contained in the information produced by `help <taskname>`, once `asap_init` has been invoked. Note that you can use `inp <taskname>` on these as for other tasks.

- `sdcal`

```

Keyword arguments:
infile -- name of input SD dataset
        options: (str) file name
        default: '' (none set) REQUIRED
        example: 'mopra-2005-05-08_0350.rpf'
        Supported formats: ASAP,MS,RPFITS,SDFITS
telescope -- the telescope name or characteristics
            options: (str) name or (list) list of gain info
            default: '' (none set)
            example: telescope='GBT' for the GBT
                   telescope='AT' for one of the default scopes
                   known to ASAP.
                   telescope=[104.9,0.43] diameter(m), ap.eff.
                   telescope=[0.743] gain in Jy/K
                   telescope='FIX' to change default fluxunit
                   see description below
fluxunit -- units for line flux
           options: 'K','Jy',''
           default: '' (keep current fluxunit)
           WARNING: For GBT data, see description below.
specunit -- units for spectral axis
           options: (str) 'channel','km/s','GHz','MHz','kHz','Hz'
           default: 'channel'
           example: this will be the units for masklist
frame -- frequency frame for spectral axis
        options: (str) 'LSRK','REST','TOPO','LSRD','BARY',
                  'GEO','GALACTO','LGROUP','CMB'
        default: currently set frame in scantable
        WARNING: frame='REST' not yet implemented
doppler -- doppler mode
          options: (str) 'RADIO','OPTICAL','Z','BETA','GAMMA'
          default: currently set doppler in scantable
calmode -- calibration mode
          options: 'ps','nod','fs','fsotf','quotient','none'
          default: 'none'
          example: choose mode 'none' if you have
                  already calibrated and want to

```



```

        try baselines or averaging
scanlist -- list of scan numbers to process
    default: [] (use all scans)
    example: [21,22,23,24]
        this selection is in addition to field
        and iflist
field -- selection string for selecting scans by name
    default: '' (no name selection)
    example: 'FLS3a*'
        this selection is in addition to scanlist
        and iflist
iflist -- list of IF id numbers to select
    default: [] (use all IFs)
    example: [15]
        this selection is in addition to scanlist
        and field
scanaverage -- average integrations within scans
    options: (bool) True,False
    default: False
    example: if True, this happens in read-in
        For GBT, set False!
timeaverage -- average times for multiple scan cycles
    options: (bool) True,False
    default: False
    example: if True, this happens after calibration
polaverage -- average polarizations
    options: (bool) True,False
    default: False
kernel -- type of spectral smoothing
    options: 'hanning','gaussian','boxcar','none'
    default: 'none'
kwidth -- width of spectral smoothing kernel
    options: (int) in channels
    default: 5
    example: 5 or 10 seem to be popular for boxcar
        ignored for hanning (fixed at 5 chans)
        (0 will turn off gaussian or boxcar)
tau -- atmospheric optical depth
    default: 0.0 (no correction)
blmode -- mode for baseline fitting
    options: (str) 'auto','list','none'
    default: 'none'
    example: blmode='none' turns off baseline fitting
        blmode='auto' uses AUTOPARS (see below)
        in addition to blpoly to run linefinder

```

```

        to determine line-free regions
        USE WITH CARE! May need to tweak AUTOPARS.
blpoly -- order of baseline polynomial
        options: (int) (<0 turns off baseline fitting)
        default: 5
        example: typically in range 2-9 (higher values
                seem to be needed for GBT)
interactive -- interactive mode for baseline fitting
        options: (bool) True,False
        default: False
        WARNING: Currently this just asks whether you accept
                the displayed fit and if not, continues
                without doing any baseline fit.
masklist -- list of mask regions to INCLUDE in BASELINE fit
        default: [] (entire spectrum)
        example: [[1000,3000],[5000,7000]]
                if blmode='auto' then this mask will be applied
                before fitting
sdfilename -- Name of output file
        default: '' (<infile>_cal)
        example: note that sdfilename is the OUTPUT of sdclean and
                is the INPUT filename param for the other
                sdtasks to streamline processing
        WARNING: output file will be overwritten
outfmt -- format of output file
        options: 'ASCII','SDFITS','MS','ASAP'
        default: 'ASAP'
        example: the ASAP format is easiest for further sd
                processing; use MS for CASA imaging.
                If ASCII, then will append some stuff to
                the sdfilename
plotlevel -- control for plotting of results
        options: (int) 0=none, 1=some, 2=more, <0=hardcopy
        default: 0 (no plotting)
        example: plotlevel<0 as abs(plotlevel), e.g.
                -1 => hardcopy of final plot (will be named
                <sdfilename>_calspec.eps)
        WARNING: be careful plotting in fsof mode!

```

AUTOPARS: the following parameters are used for blmode='auto' ONLY

```

-----
thresh -- S/N threshold for linefinder
        default: 5
        example: a single channel S/N ratio above which the channel is
                considered to be a detection

```

```

avg_limit -- channel averaging for broad lines
            default: 4
            example: a number of consecutive channels not greater than
                    this parameter can be averaged to search for broad lines
edge -- channels to drop at beginning and end of spectrum
            default: 0
            example: [1000] drops 1000 channels at beginning AND end
                    [1000,500] drops 1000 from beginning and 500 from end

```

Note: For bad baselines threshold should be increased, and avg_limit decreased (or even switched off completely by setting this parameter to 1) to avoid detecting baseline undulations instead of real lines.

DESCRIPTION:

Task `sdcal` performs data selection, calibration, and/or spectral baseline fitting for single-dish spectra. By setting `calmode='none'` one can run `sdcal` on already calibrated data, for further selection or baseline fitting. Likewise, one can set `blmode='none'` to bypass baseline fitting.

If you give multiple IFs in `iflist`, then your scantable will have multiple IFs. This can be handled, but there can be funny interactions later on. We recommend you split each IF out into separate files by re-running `sdcal` with each IF in turn.

ASAP recognizes the data of the "AT" telescopes, but currently does not know about the GBT or any other telescope. This task does know about GBT. Therefore, if you wish to change the `fluxunit` (see below), then you need to tell it what to do. If you set `telescope = 'AT'` it will use its internal defaults. If you set `telescope = 'GBT'`, it will use an approximate aperture efficiency conversion. If you give it a list instead of a string, then if the list has a single float it is assumed to be the gain in Jy/K, if two or more elements they are assumed to be telescope diameter (m) and aperture efficiency respectively.

Note that `sdcal` assumes that the `fluxunit` is set correctly in the data already. If not, then set `telescope = 'FIX'` and it will set the default units to fluxunit without conversion. WARNING: If the data in `infile` is an ms from GBT, it will currently say its in 'Jy' but it is really 'K', so set `telescope = 'FIX'` and `fluxunit='K'` to fix this.

- `sdfit`

Keyword arguments:

```

sdfit -- name of input SD dataset
            default: none - must input file name
            example: 'mysd.asap'
            See sdcal for allowed formats.
telescope -- the telescope name or characteristics
            options: (str) name or (list) list of gain info
            default: '' (none set)

```

```

example: telescope='GBT' for the GBT
         telescope='AT' for one of the default scopes
         known to ASAP.
         telescope=[104.9,0.43] diameter(m), ap.eff.
         telescope=[0.743] gain in Jy/K
         telescope='FIX' to change default fluxunit
         see description below
fluxunit -- units for line flux
          options: (str) 'K','Jy',''
          default: '' (keep current fluxunit)
          WARNING: For GBT data, see description below.
specunit -- units for spectral axis
          options: (str) 'channel','km/s','GHz','MHz','kHz','Hz',''
          default: '' (keep current specunit)
frame -- frequency frame for spectral axis
        options: (str) 'LSRK','REST','TOPO','LSRD','BARY',
                  'GEO','GALACTO','LGROUP','CMB'
        default: currently set frame in scantable
        WARNING: frame='REST' not yet implemented
doppler -- doppler mode
          options: (str) 'RADIO','OPTICAL','Z','BETA','GAMMA'
          default: currently set doppler in scantable
scanlist -- list of scan numbers to process
           default: [] (use all scans)
           example: [21,22,23,24]
field -- selection string for selecting scans by name
        default: '' (no name selection)
        example: 'FLS3a*'
           this selection is in addition to scanlist
           and iflist
iflist -- list of IF id numbers to select
         default: [] (use all IFs)
         example: [15]
fitmode -- mode for fitting
          options: (str) 'list','auto'
          default: 'auto'
          example: 'list' will use maskline to define regions to
                  fit for lines with nfit in each
                  'auto' will use the linefinder to fit for lines
                  using autopars (see below)
maskline -- list of mask regions to INCLUDE in LINE fitting
           default: all
           example: maskline=[[3900,4300]] for a single region, or
                  maskline=[[3900,4300],[5000,5400]] for two, etc.
invertmask -- invert mask (EXCLUDE masklist instead)

```

```

options: (bool) True, False
default: False
example: invertmask=True, then will make one region that is
        the exclusion of the maskline regions
nfit -- list of number of gaussian lines to fit in in maskline region
default: 0 (no fitting)
example: nfit=[1] for single line in single region,
        nfit=[2] for two lines in single region,
        nfit=[1,1] for single lines in each of two regions, etc.
fitfile -- name of output file for fit results
default: no output fit file
example: 'mysd.fit'
plotlevel -- control for plotting of results
options: (int) 0=none, 1=some, 2=more
default: 0 (no plotting)
example: plotlevel=1 plots fit and residual
        no hardcopy available for fitter
WARNING: be careful plotting OTF data with lots of fields

```

AUTOPARS: the following parameters are used for fitmode='auto' ONLY

```

-----
thresh -- S/N threshold for linefinder
default: 5
example: a single channel S/N ratio above which the channel is
        considered to be a detection
min_nchan -- minimum number of consecutive channels for linefinder
default: 3
example: minimum number of consecutive channels required to pass threshold
avg_limit -- channel averaging for broad lines
default: 4
example: a number of consecutive channels not greater than
        this parameter can be averaged to search for broad lines
box_size -- running mean box size
default: 0.2
example: a running mean box size specified as a fraction
        of the total spectrum length
edge -- channels to drop at beginning and end of spectrum
default: 0
example: [1000] drops 1000 channels at beginning AND end
        [1000,500] drops 1000 from beginning and 500 from end

```

Note: For bad baselines threshold should be increased, and avg_limit decreased (or even switched off completely by setting this parameter to 1) to avoid detecting baseline undulations instead of real lines.

```
xstat -- RETURN ONLY: a Python dictionary of line statistics
      keys:      'peak','cent','fwhm'
      example:  each value is a list of lists with one list of
                  2 entries [fitvalue,error] per component.
                  e.g. xstat['peak']=[[234.9, 4.8],[234.2, 5.3]]
                  for 2 components.
```

DESCRIPTION:

Task `sdfit` is a basic line-fitter for single-dish spectra. It assumes that the spectra have been calibrated in `sdcal`. Furthermore, it assumes that any selection of scans, IFs, polarizations, and time and channel averaging/smoothing has also already been done (in `sdcal`) as there are no controls for these. Note that you can run `sdcal` with `calmode = 'none'` and do selection, writing out a new scantable.

Note that multiple scans and IFs can in principle be handled, but we recommend that you use `scanlist`, `field`, and `iflist` to give a single selection for each fit.

For complicated spectra, `sdfit` does not do a good job of "auto-guessing" the starting model for the fit. We recommend you use `sd.fitter` in the toolkit which has more options, such as fixing components in the fit and supplying starting guesses by hand.

WARNING: `sdfit` will currently return the fit for the first row in the scantable. Does not handle multiple polarizations.

See the `sdcal` description for information on `fluxunit` conversion and the `telescope` parameter.

- **sdlist**

```
Keyword arguments:
infile -- name of input SD dataset
scanaverage -- average integrations within scans
             options: (bool) True,False
             default: False
             example: if True, this happens in read-in
                       For GBT, set False!
listfile -- Name of output file for summary list
             default: '' (no output file)
             example: 'mysd_summary.txt'
             WARNING: output file will be overwritten
```

DESCRIPTION:

Task `sdlist` lists the scan summary of the dataset after importing as a scantable into ASAP. It will optionally output this summary as file.

Note that if your `PAGER` environment variable is set to 'less' and you have set the 'verbose' ASAP environment variable to True (the default), then the screen version of the summary will page. You can disable this for `sdlist` by setting `sd.rcParams['verbose']=False` before running `sdlist`. Set it back afterwards if you want lots of information.

- `sdplot`

Keyword arguments:

```

sdfile -- name of input SD dataset
        default: none - must input file name
        example: 'mysd.asap'
           See sdcal for allowed formats.
telescope -- the telescope name or characteristics
            options: (str) name or (list) list of gain info
            default: '' (none set)
            example: telescope='GBT' for the GBT
                   telescope='AT' for one of the default scopes
                   known to ASAP.
                   telescope=[104.9,0.43] diameter(m), ap.eff.
                   telescope=[0.743] gain in Jy/K
                   telescope='FIX' to change default fluxunit
                   see description below
fluxunit -- units for line flux
           options: 'K','Jy',''
           default: '' (keep current unit)
           WARNING: For GBT data, see description below.
specunit -- units for spectral axis
           options: (str) 'channel','km/s','GHz','MHz','kHz','Hz',''
           default: '' (keep current specunit)
           example: this will be the units for masklist
frame -- frequency frame for spectral axis
        options: (str) 'LSRK','REST','TOPO','LSRD','BARY',
                  'GEO','GALACTO','LGROUP','CMB'
        default: currently set frame in scantable
        WARNING: frame='REST' not yet implemented
doppler -- doppler mode
           options: (str) 'RADIO','OPTICAL','Z','BETA','GAMMA'
           default: currently set doppler in scantable
scanlist -- list of scan numbers to process
           default: [] (use all scans)
           example: [21,22,23,24]
field -- selection string for selecting scans by name
        default: '' (no name selection)
        example: 'FLS3a*'
           this selection is in addition to scanlist

```

```

                                and iflist
iflist -- list of IF id numbers to select
        default: [] (use all IFs)
        example: [15]
scanaverage -- average integrations within scans
        options: (bool) True,False
        default: False
        example: if True, this happens in read-in
timeaverage -- average times for multiple scans
        options: (bool) True,False
        default: True
polaverage -- average polarizations
        options: (bool) True,False
        default: True
kernel -- type of spectral smoothing
        options: 'hanning','gaussian','boxcar','none'
        default: 'none'
kwidth -- width of spectral smoothing kernel
        options: (int) in channels
        default: 5
        example: 5 or 10 seem to be popular for boxcar
                 ignored for hanning (fixed at 5 chans)
                 (0 will turn off gaussian or boxcar)
stack -- code for stacking on single plot
        options: 'p','b','i','t','s' or
                 'pol','beam','if','time','scan'
        default: 'p'
        example: maximum of 25 stacked spectra
                 stack by pol, beam, if, time, scan
panel -- code for splitting into multiple panels
        options: 'p','b','i','t','s' or
                 'pol','beam','if','time','scan'
        default: 'i'
        example: maximum of 25 panels
                 panel by pol, beam, if, time, scan
flrange -- range for flux axis of plot
        options: (list) [min,max]
        default: [] (full range)
        example: flrange=[-0.1,2.0] if 'K'
                 assumes current fluxunit
sprange -- range for spectral axis of plot
        options: (list) [min,max]
        default: [] (full range)
        example: sprange=[42.1,42.5] if 'GHz'
                 assumes current specunit

```



```

linecat -- control for line catalog plotting
options: (str) 'all', 'none' or by molecule
default: 'none' (no lines plotted)
example: linecat='SiO' for SiO lines
         linecat='*OH' for alcohols
         uses sprange to limit catalog
WARNING: specunit must be in frequency (*Hz)
         to plot from the line catalog!
         and must be 'GHz' or 'MHz' to use
         sprange to limit catalog
linedop -- doppler offset for line catalog plotting
options: (float) doppler velocity (km/s)
default: 0.0
example: linedop=-30.0
plotfile -- file name for hardcopy output
options: (str) filename.eps,.ps,.png
default: '' (no hardcopy)
example: 'specplot.eps', 'specplot.png'
         Note this autodetects the format from
         the suffix (.eps,.ps,.png).

```

DESCRIPTION:

Task `sdplot` displays single-dish spectra. It assumes that the spectra have been calibrated in `sdcal`. It does allow selection of scans, IFs, polarizations, and some time and channel averaging/smoothing options also, but does not write out this data.

Some plot options, like annotation and changing titles, legends, colors, fonts, and the like are not supported in this task. You should use `sd.plotter` from the ASAP toolkit directly for this.

This task uses the JPL line catalog as supplied by ASAP. If you wish to use a different catalog, or have it plot the line IDs from top or bottom (rather than alternating), then you will need to explore the `sd` toolkit also.

Note that multiple scans and IFs can in principle be handled through stacking and panelling, but this is fairly rudimentary at present and you have little control of what happens in individual panels. We recommend that you use `scanlist`, `field`, and `iflist` to give a single selection for each run.

Currently, setting `specunit = 'GHz'` fixes the x-axis span of each IF panel to be the same (an example of the limitations of ASAP plotting at present).

See the `sdcal` description for information on `fluxunit` conversion and the `telescope` parameter.

WARNING: be careful plotting OTF (on-the-fly) mosaic data with lots of fields!

- `sdstat`

Keyword arguments:

sdfile -- name of input SD dataset
 default: none - must input file name
 example: 'mysd.asap'
 See `sdcal` for allowed formats.

telescope -- the telescope name or characteristics
 options: (str) name or (list) list of gain info
 default: '' (none set)
 example: `telescope='GBT'` for the GBT
`telescope='AT'` for one of the default scopes known to ASAP.
`telescope=[104.9,0.43]` diameter(m), `ap.eff.`
`telescope=[0.743]` gain in Jy/K
`telescope='FIX'` to change default fluxunit
 see description below

fluxunit -- units for line flux
 options: 'K','Jy',''
 default: '' (keep current fluxunit)
 WARNING: For GBT data, see description below.

specunit -- units for spectral axis
 options: 'channel','km/s','GHz','MHz','kHz','Hz',''
 default: '' (keep current specunit)
 example: make sure this is the same units of masklist

frame -- frequency frame for spectral axis
 options: (str) 'LSRK','REST','TOPO','LSRD','BARY','
 'GEO','GALACTO','LGROUP','CMB'
 default: currently set frame in scantable

doppler -- doppler mode
 options: (str) 'RADIO','OPTICAL','Z','BETA','GAMMA'
 default: currently set doppler in scantable

scanlist -- list of scan numbers to process
 default: [] (use all scans)
 example: [21,22,23,24]

field -- selection string for selecting scans by name
 default: '' (no name selection)
 example: 'FLS3a*'
 this selection is in addition to scanlist
 and iflist

iflist -- list of IF id numbers to select
 default: [] (use all IFs)
 example: [15]

masklist -- list of mask regions to INCLUDE in stats
 default: [] (whole spectrum)
 example: [4000,4500] for one region
 [[1000,3000],[5000,7000]]

```

                                these must be pairs of [lo,hi] boundaries
invertmask -- invert mask (EXCLUDE masklist instead)
            options: (bool) True,False
            default: false

-----
xstat -- RETURN ONLY: a Python dictionary of line statistics
            keys: 'rms', 'stddev', 'max', 'min', 'sum', 'median', 'mean',
                'eqw'
            example: print "rms = ",xstat['rms']
                    these can be used for testing in scripts or
                    for regression

                    'eqw' is equivalent width (sum/mag) where mag
                    is either max or min depending on which has
                    greater magnitude.

```

DESCRIPTION:

Task `sdstat` computes basic statistics (rms, mean, median, sum) for single-dish spectra. It assumes that the spectra have been calibrated in `sdcal`. Furthermore, it assumes that any selection of scans, IFs, polarizations, and time and channel averaging/smoothing has also already been done (in `sdcal`) as there are no controls for these. Note that you can run `sdcal` with `calmode = 'none'` and do selection, writing out a new scantable.

Note that multiple scans and IFs can in principle be handled, but we recommend that you use `scanlist`, `field`, and `iflist` to give a single selection for each run.

See the `sdcal` description for information on `fluxunit` conversion and the `telescope` parameter.

WARNING: If you do have multiple scantable rows, then `xstat` values will be lists.

8.2.2 A Single Dish Analysis Use Case With SDTasks

As an example, the following illustrates the use of the SDtasks for the Orion data set, which contains the HCCCN line in one of its IFs. This walk-through contains comments about setting parameter values and some options during processing.

```

#####
#
# ORION-S SDtasks Use Case
# Position-Switched data
# Version STM 2007-03-04
#
# This is a detailed walk-through
# for using the SDtasks on a

```

```

# test dataset.
#
#####
import time
import os

# NOTE: you should have already run
# asap_init()
# to import the ASAP tools as sd.<tool>
# and the SDtasks

#
# This is the environment variable
# pointing to the head of the CASA
# tree that you are running
casapath=os.environ['AIPSPATH']

#
# This bit removes old versions of the output files
os.system('rm -rf sdusecase_orions* ')
#
# This is the path to the OrionS GBT ms in the data repository
datapath=casapath+'/data/regression/ATST5/OrionS/OrionS_rawACSmod'
#
# The following will remove old versions of the data and
# copy the data from the repository to your
# current directory. Comment this out if you already have it
# and don't want to recopy
os.system('rm -rf OrionS_rawACSmod')
copystring='cp -r '+datapath+' .'
os.system(copystring)

# This resets all of the CASA task parameters to their
# global defaults. Note that these are not necessarily
# the proper defaults for specific tasks (see below).
restore()

# Now is the time to set some of the more useful
# ASAP environment parameters (the ones that the
# ASAP User Manual claims are in the .asaprc file).
# These are in the Python dictionary sd.rcParams
# You can see whats in it by typing:
#sd.rcParams
# One of them is the 'verbose' parameter which tells
# ASAP whether to spew lots of verbiage during processing

```

```
# or to keep quiet. The default is
#sd.rcParams['verbose']=True
# You can make ASAP run quietly (with only task output) with
#sd.rcParams['verbose']=False

# Another key one is to tell ASAP to save memory by
# going off the disk instead. The default is
#sd.rcParams['scantable.storage']='memory'
# but if you are on a machine with small memory, do
#sd.rcParams['scantable.storage']='disk'

# You can reset back to defaults with
#sd.rcdefaults

#####
#
# ORION-S HC3N
# Position-Switched data
#
#####
startTime=time.time()
startProc=time.clock()

#####
# List data
#####
# List the contents of the dataset
# First reset parameter defaults (safe)
default('sdlist')

# You can see its inputs with
#inp('sdlist')
# or just
#inp
# now that the defaults('sdlist') set the
# taskname='sdlist'
#
# Set the name of the GBT ms file
infile = 'OrionS_rawACSmod'

# Set an output file in case we want to
# refer back to it
listfile = 'sdusecase_orions_summary.txt'
sdlist()
```

```
# You could also just type
#go
```

```
# You should see something like:
```

```
#-----
# Scan Table Summary
#-----
#Beams:          1
#IFs:           26
#Polarisations: 2 (linear)
#Channels:      8192
#
#Observer:      Joseph McMullin
#Obs Date:      2006/01/19/01:45:58
#Project:       AGBT06A_018_01
#Obs. Type:     OffOn:PSWITCHOFF:TPWCAL
#Antenna Name:  GBT
#Flux Unit:     Jy
#Rest Freqs:    [4.5490258e+10] [Hz]
#Abcissa:       Channel
#Selection:     none
#
#Scan Source      Time      Integration
#  Beam  Position (J2000)
#      IF      Frame  RefVal      RefPix  Increment
#-----
# 20 OrionS_psr   01:45:58   4 x      30.0s
#      0   05:15:13.5 -05.24.08.2
#      0   LSRK   4.5489354e+10  4096   6104.233
#      1   LSRK   4.5300785e+10  4096   6104.233
#      2   LSRK   4.4074929e+10  4096   6104.233
#      3   LSRK   4.4166215e+10  4096   6104.233
# 21 OrionS_ps   01:48:38   4 x      30.0s
#      0   05:35:13.5 -05.24.08.2
#      0   LSRK   4.5489354e+10  4096   6104.233
#      1   LSRK   4.5300785e+10  4096   6104.233
#      2   LSRK   4.4074929e+10  4096   6104.233
#      3   LSRK   4.4166215e+10  4096   6104.233
# 22 OrionS_psr   01:51:21   4 x      30.0s
#      0   05:15:13.5 -05.24.08.2
#      0   LSRK   4.5489354e+10  4096   6104.233
#      1   LSRK   4.5300785e+10  4096   6104.233
#      2   LSRK   4.4074929e+10  4096   6104.233
#      3   LSRK   4.4166215e+10  4096   6104.233
```

```

# 23 OrionS_ps      01:54:01    4 x      30.0s
#      0      05:35:13.5 -05.24.08.2
#      0      LSRK    4.5489354e+10    4096    6104.233
#      1      LSRK    4.5300785e+10    4096    6104.233
#      2      LSRK    4.4074929e+10    4096    6104.233
#      3      LSRK    4.4166215e+10    4096    6104.233
# 24 OrionS_psr    02:01:47    4 x      30.0s
#      0      05:15:13.5 -05.24.08.2
#      12     LSRK    4.3962126e+10    4096    6104.2336
#      13     LSRK    4.264542e+10    4096    6104.2336
#      14     LSRK    4.159498e+10    4096    6104.2336
#      15     LSRK    4.3422823e+10    4096    6104.2336
# 25 OrionS_ps      02:04:27    4 x      30.0s
#      0      05:35:13.5 -05.24.08.2
#      12     LSRK    4.3962126e+10    4096    6104.2336
#      13     LSRK    4.264542e+10    4096    6104.2336
#      14     LSRK    4.159498e+10    4096    6104.2336
#      15     LSRK    4.3422823e+10    4096    6104.2336
# 26 OrionS_psr    02:07:10    4 x      30.0s
#      0      05:15:13.5 -05.24.08.2
#      12     LSRK    4.3962126e+10    4096    6104.2336
#      13     LSRK    4.264542e+10    4096    6104.2336
#      14     LSRK    4.159498e+10    4096    6104.2336
#      15     LSRK    4.3422823e+10    4096    6104.2336
# 27 OrionS_ps      02:09:51    4 x      30.0s
#      0      05:35:13.5 -05.24.08.2
#      12     LSRK    4.3962126e+10    4096    6104.2336
#      13     LSRK    4.264542e+10    4096    6104.2336
#      14     LSRK    4.159498e+10    4096    6104.2336
#      15     LSRK    4.3422823e+10    4096    6104.2336

```

```

# The HC3N and CH3OH lines are in IFs 0 and 2 respectively
# of scans 20,21,22,23. We will pull these out in our
# calibration.

```

```
#####
```

```
# Calibrate data
```

```
#####
```

```
# We will use the sdcal task to calibrate the data.
```

```
# Set the defaults
```

```
default('sdcal')
```

```
# You can see the inputs with
```

```
#inp
```

```
# Set our infile (which would have been set from our run of
# sdlist if we were not cautious and reset defaults).
infile = 'OrionS_rawACSm0d'

# Currently, the ASAP scantable filler does not fully recognize
# data from the GBT, and it thinks that the data is in 'Jy'
# (what it does when it doesn't know any better) instead of
# 'K', which is what it really is. So we tell sdcald to fix
# this for us:
telescope = 'FIX'
fluxunit = 'K'

# Lets leave the spectral axis in channels for now
specunit = 'channel'

# This is position-switched data so we tell sdcald this
calmode = 'ps'

# For GBT data, it is safest to not have scantable pre-average
# integrations within scans.
scanaverage = False

# We do want sdcald to average up scans and polarization after
# calibration however.
timeaverage = True
polaverage = True

# Do an atmospheric optical depth (attenuation) correction
# Input the zenith optical depth at 43 GHz
tau = 0.09

# Select our scans and IFs (for HC3N)
scanlist = [20,21,22,23]
iflist = [0]

# We do not require selection by field name (they are all
# the same except for on and off)
field = ''

# We will do some spectral smoothing
# For this demo we will use boxcar smoothing rather than
# the default
#kernel='hanning'
# We will set the width of the kernel to 5 channels
kernel = 'boxcar'
```



```
kwidth = 5

# We wish to fit out a baseline from the spectrum
# The GBT has particularly nasty baselines :(
# We will let ASAP use auto_poly_baseline mode
# but tell it to drop the 1000 edge channels from
# the beginning and end of the spectrum.
# A 2nd-order polynomial will suffice for this test.
# You might try higher orders for fun.
blmode = 'auto'
blpoly = 2
edge = [1000]

# We will not give it regions as an input mask
# though you could, with something like
#masklist=[[1000,3000],[5000,7000]]
masklist = []

# By default, we will not get plots in sdcal (but
# can make them using sdplot).
plotlevel = 0
# But if you wish to see a final spectrum, set
#plotlevel = 1
# or even
#plotlevel = 2
# to see intermediate plots and baselining output.

# Now we give the name for the output file
sdfile = 'sdusecase_orions_hc3n.asap'

# We will write it out in ASAP scantable format
outform = 'asap'

# You can look at the inputs with
#inp

# Before running, lets save the inputs in case we want
# to come back and re-run the calibration.
saveinputs('sdcal','sdcal.orions.save')
# These can be recovered by
#execfile 'sdcal.orions.save'

# We are ready to calibrate
sdcal()
```

```

# Note that after the task ran, it produced a file
# sdcals.last which contains the inputs from the last
# run of the task (all tasks do this). You can recover
# this (anytime before sdcals is run again) with
#execfile 'sdcals.last'

#####
# List data
#####
# List the contents of the calibrated dataset
# Set the input to the just created file
infile = sdcals
listfile = ''
sdlist()

# You should see:
#
#-----
# Scan Table Summary
#-----
#Beams:          1
#IFs:            26
#Polarisations: 1 (linear)
#Channels:       8192
#
#Observer:       Joseph McMullin
#Obs Date:       2006/01/19/01:45:58
#Project:        AGBT06A_018_01
#Obs. Type:      OffOn:PSWITCHOFF:TPWCAL
#Antenna Name:   GBT
#Flux Unit:      K
#Rest Freqs:     [4.5490258e+10] [Hz]
#Abcissa:        Channel
#Selection:      none
#
#Scan Source      Time      Integration
#  Beam  Position (J2000)
#      IF      Frame  RefVal      RefPix  Increment
#-----
#  0 OrionS_ps    01:52:05    1 x    08:00.5
#      0    05:35:13.5 -05.24.08.2
#      0      LSRK    4.5489354e+10    4096    6104.233
#
# Note that our scans are now collapsed (timeaverage=True) but
# we still have our IF 0

```

```
#####  
# Plot data  
#####  
default('sdplot')  
  
# The file we produced after calibration  
# (if we hadn't reset defaults it would have  
# been set - note that sdplot,sdfit,sdstat use  
# sfile as the input file, which is the output  
# file of sdcal).  
sfile = 'sdusecase_orions_hc3n.asap'  
  
# Lets just go ahead and plot it up as-is  
sdplot()  
  
# Looks ok. Plot with x-axis in GHz  
specunit='GHz'  
sdplot()  
  
# Note that the rest frequency in the scantable  
# is set correctly to the HCCCN line at 45.490 GHz.  
# So you can plot the spectrum in km/s  
specunit='km/s'  
sdplot()  
  
# Zoom in  
sprange=[-100,50]  
sdplot()  
  
# Lets plot up the lines to be sure  
# We have to go back to GHz for this  
# (known deficiency in ASAP)  
specunit='GHz'  
sprange=[45.48,45.51]  
linecat='all'  
sdplot()  
  
# Too many lines! Focus on the HC3N ones  
linecat='HCCCN'  
sdplot()  
  
# Finally, we can convert from K to Jy  
# using the aperture efficiencies we have  
# coded into the sdtasks
```

```
telescope='GBT'
fluxunit='Jy'
sdplot()

# Lets save this plot
plotfile='sdusecase_orions_hc3n.eps'
sdplot()

#####
# Off-line Statistics
#####
# Now do some region statistics
# First the line-free region
# Set parameters
default('sdstat')
sdfile = 'sdusecase_orions_hc3n.asap'

# Keep the default spectrum and flux units
# K and channel
fluxunit = ''
specunit = ''

# Pick out a line-free region
# You can bring up a default sdplot again
# to check this
masklist = [[5000,7000]]

# This is a line-free region so we don't need
# to invert the mask
invertmask = False

# You can check with
#inp

sdstat()

# You see that sdstat returns some results in
# the Python dictionary xstat. You can assign
# this to a variable
off_stat = xstat

# and look at it
off_stat
# which should give
# {'eqw': 38.563105620704945,
```

```

# 'max': 0.15543246269226074,
# 'mean': -0.0030361821409314871,
# 'median': -0.0032975673675537109,
# 'min': -0.15754437446594238,
# 'rms': 0.047580458223819733,
# 'stddev': 0.047495327889919281,
# 'sum': -6.0754003524780273}

#You see it has some keywords for the various
#stats. We want the standard deviation about
#the mean, or 'stddev'
print "The off-line std. deviation = ",off_stat['stddev']
# which should give
# The off-line std. deviation = 0.0474953278899

# or better formatted (using Python I/O formatting)
print "The off-line std. deviation = %5.3f K" %\
      (off_stat['stddev'])
# which should give
# The off-line std. deviation = 0.047 K

#####
# On-line Statistics
#####
# Now do the line region
# Continue setting or resetting parameters
masklist = [[3900,4200]]

sdstat()

line_stat = xstat

# look at these
line_stat
# which gives
# {'eqw': 73.335154614280981,
# 'max': 0.92909121513366699,
# 'mean': 0.22636228799819946,
# 'median': 0.10317134857177734,
# 'min': -0.13283586502075195,
# 'rms': 0.35585442185401917,
# 'stddev': 0.27503398060798645,
# 'sum': 68.135047912597656}

```

```
# of particular interest are the max value
print "The on-line maximum = %5.3f K" % (line_stat['max'])
# which gives
# The on-line maximum = 0.929 K

# and the estimated equivalent width (in channels)
# which is the sum/max
print "The estimated equivalent width = %5.1f channels" %\
      (line_stat['eqw'])
# which gives
# The estimated equivalent width = 73.3 channels

#####
# Line Fitting
#####
# Now we are ready to do some line fitting
# Default the parameters
default('sdfit')

# Set our input file
sdfile = 'sdusecase_orions_hc3n.asap'

# Stick to defaults
# fluxunit = 'K', specunit = 'channel'
fluxunit = ''
specunit = ''

# We will try auto-fitting first
fitmode = 'auto'
# A single Gaussian
nfit = [1]
# Leave the auto-parameters to their defaults for
# now, except ignore the edge channels
edge = [1000]

# Lets see a plot while doing this
plotlevel = 1

# Save the fit output in a file
fitfile = 'sdusecase_orions_hc3n.fit'

# Go ahead and do the fit
sdfit()

# If you had verbose mode on, you probably saw something
```

```

# like:
#
# 0: peak = 0.811 K , centre = 4091.041 channel, FWHM = 72.900 channel
#   area = 62.918 K channel
#

# The fit is output in the dictionary xstat
fit_stat = xstat

fit_stat
#
# {'cent': [[4091.04052734375, 0.72398632764816284]],
#  'fwhm': [[72.899894714355469, 1.7048574686050415]],
#  'nfit': 1,
#  'peak': [[0.81080442667007446, 0.016420882195234299]]}
#
# So you can write them out or test them:
print "The line-fit parameters were:"
print "      maximum = %6.3f +/- %6.3f K" %\
      (fit_stat['peak'][0][0],fit_stat['peak'][0][1])
print "      center = %6.1f +/- %6.1f channels" %\
      (fit_stat['cent'][0][0],fit_stat['cent'][0][1])
print "      FWHM = %6.2f +/- %6.2f channels" %\
      (fit_stat['fwhm'][0][0],fit_stat['fwhm'][0][1])
#
# Which gives:
# The line-fit parameters were:
#      maximum = 0.811 +/- 0.016 K
#      center = 4091.0 +/- 0.7 channels
#      FWHM = 72.90 +/- 1.70 channels

# We can do the fit in km/s also
specunit = 'km/s'
# For some reason we need to help it along with a mask
maskline = [-50,0]

fitfile = 'sdusecase_orions_hc3n_kms.fit'
sdfit()
# Should give (if in verbose mode)
# 0: peak = 0.811 K , centre = -27.134 km/s, FWHM = 2.933 km/s
#   area = 2.531 K km/s
#
# or

fit_stat_kms = xstat

```

```

# with
fit_stat_kms
# giving
# {'cent': [[-27.133651733398438, 0.016480101272463799]],
#  'fwhm': [[2.93294358253479, 0.038807671517133713]],
#  'nfit': 1,
#  'peak': [[0.81080895662307739, 0.0092909494414925575]]}

print "The line-fit parameters were:"
print "      maximum = %6.3f +/- %6.3f K" %\
      (fit_stat_kms['peak'][0][0],fit_stat_kms['peak'][0][1])
print "      center = %6.2f +/- %6.2f km/s" %\
      (fit_stat_kms['cent'][0][0],fit_stat_kms['cent'][0][1])
print "      FWHM = %6.4f +/- %6.4f km/s" %\
      (fit_stat_kms['fwhm'][0][0],fit_stat_kms['fwhm'][0][1])

# The line-fit parameters were:
#      maximum = 0.811 +/- 0.009 K
#      center = -27.13 +/- 0.02 km/s
#      FWHM = 2.9329 +/- 0.0388 km/s

#####
#
# End ORION-S Use Case
#
#####

```

8.3 Using The ASAP Toolkit Within CASA

ASAP is included with the CASA installation/build. It is not loaded upon start-up, however, and must be imported as a standard Python package. A convenience function exists for importing ASAP along with a set of prototype tasks for single dish analysis:

```
CASA <1>: asap_init
```

Once this is done, all of the ASAP functionality is now under the Python 'sd' tool. *bf*: Note: This means that if you are following the ASAP cookbook or documentation, all of the commands should be invoked with a 'sd.' before the native ASAP command.

The ASAP interface is essentially the same as that of the CASA toolkit, that is, there are groups of functionality (aka tools) which have the ability to operate on your data. Type:

```
CASA <4>: sd.<TAB>
```


<code>sd.__class__</code>	<code>sd._validate_bool</code>	<code>sd.list_scans</code>
<code>sd.__date__</code>	<code>sd._validate_int</code>	<code>sd.mask_and</code>
<code>sd.__delattr__</code>	<code>sd.asapfitter</code>	<code>sd.mask_not</code>
<code>sd.__dict__</code>	<code>sd.asaplinefind</code>	<code>sd.mask_or</code>
<code>sd.__doc__</code>	<code>sd.asaplog</code>	<code>sd.merge</code>
<code>sd.__file__</code>	<code>sd.asapplotbase</code>	<code>sd.os</code>
<code>sd.__getattr__</code>	<code>sd.asapplotgui</code>	<code>sd.plf</code>
<code>sd.__hash__</code>	<code>sd.asapmath</code>	<code>sd.plotter</code>
<code>sd.__init__</code>	<code>sd.asapplotter</code>	<code>sd.print_log</code>
<code>sd.__name__</code>	<code>sd.asapreader</code>	<code>sd.quotient</code>
<code>sd.__new__</code>	<code>sd.average_time</code>	<code>sd.rc</code>
<code>sd.__path__</code>	<code>sd.calfs</code>	<code>sd.rcParams</code>
<code>sd.__reduce__</code>	<code>sd.calnod</code>	<code>sd.rcParamsDefault</code>
<code>sd.__reduce_ex__</code>	<code>sd.calps</code>	<code>sd.rc_params</code>
<code>sd.__repr__</code>	<code>sd.commands</code>	<code>sd.rcdefaults</code>
<code>sd.__setattr__</code>	<code>sd.defaultParams</code>	<code>sd.reader</code>
<code>sd.__str__</code>	<code>sd.dosigref</code>	<code>sd.scantable</code>
<code>sd.__version__</code>	<code>sd.dototalpower</code>	<code>sd.selector</code>
<code>sd._asap</code>	<code>sd.fitter</code>	<code>sd.simple_math</code>
<code>sd._asap_fname</code>	<code>sd.is_ipython</code>	<code>sd.sys</code>
<code>sd._asaplog</code>	<code>sd.linecatalog</code>	<code>sd.unique</code>
<code>sd._is_sequence_or_number</code>	<code>sd.linefinder</code>	<code>sd.version</code>
<code>sd._n_bools</code>	<code>sd.list_files</code>	<code>sd.welcome</code>
<code>sd._to_list</code>	<code>sd.list_rcparameters</code>	<code>sd.xyplotter</code>

...to see the list of tools.

In particular, the following are essential for most reduction sessions:

- `sd.scantable` - the data structure for ASAP and the core methods for manipulating the data; allows importing data, making data selections, basic operations (averaging, baselines, etc) and setting data characteristics (e.g., frequencies, etc).
- `sd.selector` - selects a subset of data for subsequent operations
- `sd.fitter` - fit data
- `sd.plotter` - plotting facilities (uses `matplotlib`)

The `scantable` functions are used most often and can be applied to both the initial scantable and to any spectrum from that scan table. Type

```
sd.scantable.<TAB>
```

(using TAB completion) to see the full list.

8.3.1 Environment Variables

The `asaprc` environment variables are stored in the Python dictionary `sd.rcParams` in CASA. This contains a number of parameters that control how ASAP runs, for both tools and tasks. You can see what these are set to by typing at the CASA prompt:

```
CASA <2>: sd.rcParams
Out [2]:
{'insitu': True,
 'plotter.colours': '',
 'plotter.decimate': False,
 'plotter.ganged': True,
 'plotter.gui': True,
 'plotter.histogram': False,
 'plotter.linestyles': '',
 'plotter.panelling': 's',
 'plotter.papertype': 'A4',
 'plotter.stacking': 'p',
 'scantable.autoaverage': True,
 'scantable.freqframe': 'LSRK',
 'scantable.save': 'ASAP',
 'scantable.storage': 'memory',
 'scantable.verbosesummary': False,
 'useplotter': True,
 'verbose': True}
```

The use of these parameters is described in detail in the ASAP Users Guide.

You can also change these parameters through the `sd.rc` function. The use of this is described in `help sd.rc`:

```
CASA <3>: help(sd.rc)
Help on function rc in module asap:
```

```
rc(group, **kwargs)
    Set the current rc params. Group is the grouping for the rc, eg
    for scantable.save the group is 'scantable', for plotter.stacking, the
    group is 'plotter', and so on. kwargs is a list of attribute
    name/value pairs, eg
```

```
rc('scantable', save='SDFITS')
```

sets the current rc params and is equivalent to

```
rcParams['scantable.save'] = 'SDFITS'
```

Use `rcdefaults` to restore the default rc params after changes.

8.3.2 Import

Data can be loaded into ASAP by using the `scantable` function which will read a variety of recognized formats (RPFITS, varieties of SDFITS, and the CASA MeasurementSet). For example:

```
CASA <1>: scans = sd.scantable('OrionS_rawACSmod', average=False)
Importing OrionS_rawACSmod...
```

NOTE: It is important to use the `average=False` parameter setting as the calibration routines supporting GBT data require all of the individual times and phases.

NOTE: GBT data may need some pre-processing prior to using ASAP. In particular, the program which converts GBT raw data into CASA MeasurementSets tends to proliferate the number of spectral windows due to shifts in the tracking frequency; this is being worked on by GBT staff. In addition, GBT SDFITS is currently not readable by ASAP (in progress).

NOTE: The MeasurementSet to scantable conversion is able to deduce the reference and source data and assigns an `'r'` to the reference data to comply with the ASAP conventions.

NOTE: GBT observing modes are identifiable in scantable in the name assignment: position switched (`'_ps'`), Nod (`'_nod'`), and frequency switched (`'_fs'`). These are combined with the reference data assignment. (For example, the reference data taken in position switched mode observation are assigned as `'_psr'`.)

Use the `summary` function to examine the data and get basic information:

```
CASA <8>: scans.summary()
```

```
-----
Scan Table Summary
-----
```

```
Beams:          1
IFs:            26
Polarisations: 2  (linear)
Channels:       8192

Observer:       Joseph McMullin
Obs Date:       2006/01/19/01:45:58
Project:        AGBT06A_018_01
Obs. Type:      OffOn:PSWITCHOFF:TPWCAL
Antenna Name:   GBT
Flux Unit:      Jy
Rest Freqs:     [4.5490258e+10] [Hz]
Abcissa:        Channel
Selection:      none
```

```
Scan Source      Time      Integration
  Beam  Position (J2000)
      IF      Frame  RefVal      RefPix  Increment
-----
```

20 OrionS_psr	01:45:58	4 x	30.0s		
0	05:15:13.5	-05.24.08.2			
0	LSRK	4.5489354e+10	4096	6104.233	
1	LSRK	4.5300785e+10	4096	6104.233	
2	LSRK	4.4074929e+10	4096	6104.233	
3	LSRK	4.4166215e+10	4096	6104.233	
21 OrionS_ps	01:48:38	4 x	30.0s		
0	05:35:13.5	-05.24.08.2			
0	LSRK	4.5489354e+10	4096	6104.233	
1	LSRK	4.5300785e+10	4096	6104.233	
2	LSRK	4.4074929e+10	4096	6104.233	
3	LSRK	4.4166215e+10	4096	6104.233	
22 OrionS_psr	01:51:21	4 x	30.0s		
0	05:15:13.5	-05.24.08.2			
0	LSRK	4.5489354e+10	4096	6104.233	
1	LSRK	4.5300785e+10	4096	6104.233	
2	LSRK	4.4074929e+10	4096	6104.233	
3	LSRK	4.4166215e+10	4096	6104.233	
23 OrionS_ps	01:54:01	4 x	30.0s		
0	05:35:13.5	-05.24.08.2			
0	LSRK	4.5489354e+10	4096	6104.233	
1	LSRK	4.5300785e+10	4096	6104.233	
2	LSRK	4.4074929e+10	4096	6104.233	
3	LSRK	4.4166215e+10	4096	6104.233	
24 OrionS_psr	02:01:47	4 x	30.0s		
0	05:15:13.5	-05.24.08.2			
12	LSRK	4.3962126e+10	4096	6104.2336	
13	LSRK	4.264542e+10	4096	6104.2336	
14	LSRK	4.159498e+10	4096	6104.2336	
15	LSRK	4.3422823e+10	4096	6104.2336	
25 OrionS_ps	02:04:27	4 x	30.0s		
0	05:35:13.5	-05.24.08.2			
12	LSRK	4.3962126e+10	4096	6104.2336	
13	LSRK	4.264542e+10	4096	6104.2336	
14	LSRK	4.159498e+10	4096	6104.2336	
15	LSRK	4.3422823e+10	4096	6104.2336	
26 OrionS_psr	02:07:10	4 x	30.0s		
0	05:15:13.5	-05.24.08.2			
12	LSRK	4.3962126e+10	4096	6104.2336	
13	LSRK	4.264542e+10	4096	6104.2336	
14	LSRK	4.159498e+10	4096	6104.2336	
15	LSRK	4.3422823e+10	4096	6104.2336	
27 OrionS_ps	02:09:51	4 x	30.0s		
0	05:35:13.5	-05.24.08.2			
12	LSRK	4.3962126e+10	4096	6104.2336	
13	LSRK	4.264542e+10	4096	6104.2336	
14	LSRK	4.159498e+10	4096	6104.2336	
15	LSRK	4.3422823e+10	4096	6104.2336	

8.3.3 Scantable Manipulation

Within ASAP, data is stored in a `scantable`, which holds all of the observational information and provides functionality to manipulate the data and information. The building block of a `scantable` is an integration which is a single row of a scantable. Each row contains just one spectrum for each beam, IF and polarization.

Once you have a `scantable` in ASAP, you can select a subset of the data based on scan numbers, sources, or types of scan; note that each of these selections returns a new 'scantable' with all of the underlying functionality:

```
CASA <5>: scan27=scans.get_scan(27)           # Get the 27th scan
CASA <6>: scans20to24=scans.get_scan(range(20,25)) # Get scans 20 - 24
CASA <7>: scans_on=scans.get_scan('*_ps')      # Get ps scans on source
CASA <8>: scansOrion=scans.get_scan('Ori*')   # Get all Orion scans
```

To copy a scantable, do:

```
CASA <15>: ss=scans.copy()
```

8.3.3.1 Data Selection

In addition to the basic data selection above, data can be selected based on IF, beam, polarization, scan number as well as values such as `Tsys`. To make a selection you create a `selector` object which you then define with various selection functions, e.g.,

```
sel = sd.selector()           # initialize a selector object
                                # sel.<TAB> will list all options
sel.set_ifs(0)                # select only the first IF of the data
scans.set_selection(sel)     # apply the selection to the data
print scans                   # shows just the first IF
```

8.3.3.2 State Information

Some properties of a scantable apply to all of the data, such as example, spectral units, frequency frame, or Doppler type. This information can be set using the `scantable` `_set_xxxx_` methods. These are currently:

```
CASA <1>: sd.scantable.set_<TAB>
sd.scantable.set_dirframe      sd.scantable.set_fluxunit      sd.scantable.set_restfreqs
sd.scantable.set_doppler       sd.scantable.set_freqframe      sd.scantable.set_selection
sd.scantable.set_feedtype      sd.scantable.set_instrument    sd.scantable.set_unit
```

For example, `sd.scantable.set_fluxunit` sets the default units that describe the flux axis:

```
scans.set_fluxunit('K') # Set the flux unit for data to Kelvin
```

Choices are 'K' or 'Jy'. Note: the `scantable.set_fluxunit` function only changes the **name** of the current fluxunit. To change fluxunits, use `scantable.convert_flux` as described in § 8.3.4.2 instead (currently you need to do some gymnastics for GBT or non-AT telescopes).

Use `sd.scantable.set_unit` to set the units to be used on the spectral axis:

```
scans.set_unit('GHz') # Use GHz as the spectral axis for plots
```

The choices for the units are 'km/s', 'channel', or '*Hz' (e.g. 'GHz', 'MHz', 'kHz', 'Hz'). This does the proper conversion using the current frame and doppler reference as can be seen when the spectrum is plotted.

You can use `sd.scantable.set_freqframe` to set the frame in which the frequency (spectral) axis is defined:

```
CASA <2>: help(sd.scantable.set_freqframe)
Help on method set_freqframe in module asap.scantable:

set_freqframe(self, frame=None) unbound asap.scantable.scantable method
    Set the frame type of the Spectral Axis.
    Parameters:
        frame: an optional frame type, default 'LSRK'. Valid frames are:
               'REST', 'TOPO', 'LSRD', 'LSRK', 'BARY',
               'GEO', 'GALACTO', 'LGROUP', 'CMB'
    Examples:
        scan.set_freqframe('BARY')
```

The most useful choices here are `frame = 'LSRK'` (the default for the function) and `frame = 'TOPO'` (what the GBT actually observes in). Note that the 'REST' option is not yet available. The doppler frame is set with `sd.scantable.set_doppler`:

```
CASA <3>: help(sd.scantable.set_doppler)
Help on method set_doppler in module asap.scantable:

set_doppler(self, doppler='RADIO') unbound asap.scantable.scantable method
    Set the doppler for all following operations on this scantable.
    Parameters:
        doppler: One of 'RADIO', 'OPTICAL', 'Z', 'BETA', 'GAMMA'
```

Finally, there are a number of functions to query the state of the scantable. These can be found in the usual way:

```
CASA <4>: sd.scantable.get<TAB>
sd.scantable.get_abcissa      sd.scantable.get_restfreqs      sd.scantable.getbeamnos
sd.scantable.get_azimuth     sd.scantable.get_scan           sd.scantable.getcycle
sd.scantable.get_column_names sd.scantable.get_selection      sd.scantable.getif
sd.scantable.get_direction   sd.scantable.get_sourcename     sd.scantable.getifnos
```

```
sd.scantable.get_elevation    sd.scantable.get_time        sd.scantable.getpol
sd.scantable.get_fit         sd.scantable.get_tsys       sd.scantable.getpolnos
sd.scantable.get_fluxunit    sd.scantable.get_unit       sd.scantable.getscan
sd.scantable.get_parangle    sd.scantable.getbeam        sd.scantable.getscannos
```

These include functions to get the current values of the states mentioned above, as well as as methods to query the number of scans, IFs, and polarizations in the scantable, and their designations. See the inline help for the individual functions for more information.

8.3.3.3 Masks

Several functions (fitting, baseline subtraction, statistics, etc) may be run on a range of channels (or velocity/frequency ranges). You can create masks of this type using the `create_mask` function:

```
# spave = an averaged spectrum
spave.set_unit('channel')
rmsmask=spave.create_mask([5000,7000]) # create a region over channels 5000-7000
rms=spave.stats(stat='rms',mask=rmsmask) # get rms of line free region

rmsmask=spave.create_mask([3000,4000],invert=True) # choose the region
# *excluding* the specified channels
```

The mask is stored in a simple Python variable (a list) and so may be manipulated using an Python facilities.

8.3.3.4 Scantable Management

scantables can be listed via:

```
CASA <33>: sd.list_scans()
The user created scantables are:
['scans20to24', 's', 'scan27']
```

As every `scantable` will consume memory, if you will not use it any longer, you can explicitly remove it via:

```
del <scantable name>
```

8.3.3.5 Scantable Mathematics

It is possible to do simple mathematics directly on `scantables` from the CASA command line using the `+`, `-`, `*`, `/` operators as well as their cousins `+=`, `-=`, `*=`, `/=`

```
CASA <10>: scan2=scan1+2.0 # add 2.0 to data
CASA <11>: scan *= 1.05    # scale spectrum by 1.05
```

NOTE: mathematics between two scantables is not currently available in ASAP.

8.3.3.6 Scantable Save and Export

ASAP can save scantables in a variety of formats, suitable for reading into other packages. The formats are:

- ASAP – This is the internal format used for ASAP. It is the only format that allows the user to restore the data, fits, etc, without losing any information. As mentioned before, the ASAP scantable is a CASA Table (memory-based table). This function just converts it to a disk-based table. You can access this with the CASA `browsetable` task or any other CASA table tasks.
- SDFITS – The Single Dish FITS format. This format was designed for interchange between packages but few packages can actually read it.
- ASCII – A simple text based format suitable for the user to process using Python or other means.
- MeasurementSet (V2: CASA format) – Saves the data in a MeasurementSet. All CASA tasks which use an MS should work on this.

```
scans.save('output_filename','format'), e.g.,
CASA <19>: scans.save('FLS3a_calfs','MS2')
```

8.3.4 Calibration

For some observatories, the calibration happens transparently as the input data contains the Tsys measurements taken during the observations. The nominal 'Tsys' values may be in Kelvin or Jansky. The user may wish to apply a Tsys correction or apply gain-elevation and opacity corrections.

8.3.4.1 Tsys scaling

If the nominal Tsys measurement at the telescope is wrong due to incorrect calibration, the `scale` function allows it to be corrected.

```
scans.scale(1.05,tsys=True) # by default only the spectra are scaled
                           # (and not the corresponding Tsys) unless tsys=True
```


8.3.4.2 Flux and Temperature Unit Conversion

To convert measurements in Kelvin to Jansky (and vice versa), the `convert_flux` function may be used. This converts and scales the data to the selected units. The user may need to supply the aperture efficiency, telescope diameter or the Jy/K factor

```
scans.convert_flux(eta=0.48, d=35.) # Unknown telescope
scans.convert_flux(jypk=15) # Unknown telescope (alternative)
scans.convert_flux() # known telescope (mostly AT telescopes)
scans.convert_flux(eta=0.48) # if telescope diameter known
```

8.3.4.3 Gain-Elevation and Atmospheric Optical Depth Corrections

At higher frequencies, it is important to make corrections for atmospheric opacity and gain-elevation effects. **NOTE:** Currently, the MS to scantable conversion does not adequately populate the azimuth and elevation in the `scantable`. As a result, one must calculate these via:

```
scans.recalc_azel()
Computed azimuth/elevation using
Position: [882590, -4.92487e+06, 3.94373e+06]
Time: 01:48:38 Direction: 05:35:13.5 -05.24.08.2
=> azel: 154.696 43.1847 (deg)
Time: 01:48:38 Direction: 05:35:13.5 -05.24.08.2
=> azel: 154.696 43.1847 (deg)
Time: 01:48:38 Direction: 05:35:13.5 -05.24.08.2
=> azel: 154.696 43.1847 (deg)
Time: 01:48:38 Direction: 05:35:13.5 -05.24.08.2
=> azel: 154.696 43.1847 (deg)
Time: 01:48:38 Direction: 05:35:13.5 -05.24.08.2
=> azel: 154.696 43.1847 (deg)
...
```

Once you have the correct Az/El, you can correct for a *known* opacity by:

```
scans.opacity(tau=0.09) # Opacity from which the correction factor:
                        # exp(tau*zenith-distance)
```

8.3.4.4 Calibration of GBT data

Data from the GBT is uncalibrated and comes as sets of integrations representing the different phases within a calibration cycle (e.g., on source, calibration on, on source, calibration off, on reference, calibration on; on reference, calibration off). Currently, there are a number of routines emulating the standard GBT calibration (in GBTIDL):

- calps - calibrate position switched data

- calfs - calibrate frequency switched data
- calnod - calibration nod (beam switch) data

All these routines calibrate the spectral data to antenna temperature adopting the GBT calibration method as described in the GBTIDL calibration document available at:

- http://wwwlocal.gb.nrao.edu/GBT/DA/gbtidl/gbtidl_calibration.pdf

There are two basic steps:

First: determine system temperature using a noise tube calibrator (`sd.dototalpower()`)

For each integration, the system temperature is calculated from CAL noise on/off data as:

$$T_{sys} = T_{cal} \times \frac{\langle ref_{caloff} \rangle}{\langle ref_{calon} - ref_{caloff} \rangle} + \frac{T_{cal}}{2}$$

`ref` refers to reference data and the spectral data are averaged across the bandpass. Note that the central 80% of the spectra are used for the calculation.

Second, determine antenna temperature (`sd.dosigref()`)

The antenna temperature for each channel is calculated as:

$$T_a(\nu) = T_{sys} \times \frac{sig(\nu) - ref(\nu)}{ref(\nu)}$$

where $sig = \frac{1}{2}(sig_{calon} + sig_{caloff})$, $ref = \frac{1}{2}(sig_{calon} + sig_{caloff})$.

Each calibration routine may be used as:

```
scans=sd.scantable('inputdata',False)      # create a scantable called 'scans'
calibrated_scans = sd.calps(scans,[scanlist]) # calibrate scantable with position-switched
                                              # scheme
```

Note: For `calps` and `calnod`, the scanlist must be scan pairs in correct order as these routines only do minimum checking.

8.3.5 Averaging

One can average polarizations in a scantable using the `sd.scantable.average_pol` function:

```
averaged_scan = scans.average_pol(mask,weight)
```

where:

Parameters:

`mask`: An optional mask defining the region, where the averaging will be applied. The output will have all specified points masked.

`weight`: Weighting scheme. 'none' (default), 'var' (1/var(spec))

weighted), or 'tsys' (1/Tsys**2 weighted)

Example:

```
spave = stave.average_pol(weight='tsys')
```

One can also average scans over time using `sd.average_time`:

```
sd.average_time(scantable,mask,scanav,weight,align)
```

where:

Parameters:

```

one scan or comma separated scans
mask:      an optional mask (only used for 'var' and 'tsys' weighting)
scanav:    True averages each scan separately.
           False (default) averages all scans together,
weight:    Weighting scheme.
           'none'      (mean no weight)
           'var'       (1/var(spec) weighted)
           'tsys'      (1/Tsys**2 weighted)
           'tint'      (integration time weighted)
           'tintsys'   (Tint/Tsys**2)
           'median'   ( median averaging)
align:     align the spectra in velocity before averaging. It takes
           the time of the first spectrum in the first scantable
           as reference time.
```

Example:

```
stave = sd.average_time(scans,weight='tintsys')
```

Note that alignment of the velocity frame should be done before averaging if the time spanned by the scantable is long enough. This is done through the `align=True` option in `sd.average_time`, or explicitly through the `sd.scantable.freq_align` function, e.g.

```
CASA <62>: sc = sd.scantable('orions_scan20to23_if0to3.asap',False)
```

```
CASA <63>: sc.freq_align()
```

```
Aligned at reference Epoch 2006/01/19/01:49:23 (UTC) in frame LSRK
```

```
CASA <64>: av = sd.average_times(sc)
```

The time averaging can also be applied to multiple scantables. This might have been taken on different days, for example. The `sd.average_time` function takes multiple scantables as input. However, if taken at significantly different times (different days for example) then `sd.scantable.freq_align` must be used to align the velocity scales to the same time, e.g.

```
CASA <65>: sc1 = sd.scantable('orions_scan21_if0to3.asap',False)
```

```
CASA <66>: sc2 = sd.scantable('orions_scan23_if0to3.asap',False)
```

```
CASA <67>: sc1.freq_align()
```

```
Aligned at reference Epoch 2006/01/19/01:49:23 (UTC) in frame LSRK
```

```
CASA <68>: sc2.freq_align(reftime='2006/01/19/01:49:23')
```

```
Aligned at reference Epoch 2006/01/19/01:54:46 (UTC) in frame LSRK
```

```
CASA <69>: scav = sd.average_times(sc1,sc2)
```

8.3.6 Spectral Smoothing

Smoothing on data can be done as follows:

```
scantable.smooth(kernel,      # type of smoothing: 'hanning' (default), 'gaussian', 'boxcar'
                 width,      # width in pixls (ignored for hanning); FWHM for gaussian.
                 insitu)     # if False (default), do smoothing in-situ; otherwise,
                             # make new scantable
```

Example:

```
# spave is an averaged spectrum
spave.smooth('boxcar',5)    # do a 5 pixel boxcar smooth on the spectrum
sd.plotter.plot(spave)     # should see smoothed spectrum
```

8.3.7 Baseline Fitting

The function `sd.scantable.poly_baseline` carries out a baseline fit, given an mask of channels (if desired):

```
mks=scans.create_mask([100,400],[600,900])
scans.poly_baseline(msk,order=1)
```

This will fit a first order polynomial to the selected channels and subtract this polynomial from the full spectrum.

The `auto_poly_baseline` function can be used to automatically baseline your data without having to specify channel ranges for the line free data. It automatically figures out the line-free emission and fits a polynomial baseline to that data. The user can use masks to fix the range of channels or velocity range for the fit as well as mark the band edge as invalid:

```
scans.auto_poly_baseline(mask,edge,order,threshold,chan_avg_limit,plot,insitu):
```

Parameters:

```
mask:          an optional mask retrieved from scantable
edge:          an optional number of channel to drop at
                the edge of spectrum. If only one value is
                specified, the same number will be dropped from
                both sides of the spectrum. Default is to keep
                all channels. Nested tuples represent individual
                edge selection for different IFs (a number of spectral
                channels can be different)
order:         the order of the polynomial (default is 0)
threshold:     the threshold used by line finder. It is better to
                keep it large as only strong lines affect the
                baseline solution.
chan_avg_limit: a maximum number of consecutive spectral channels to
```

```

    average during the search of weak and broad lines.
    The default is no averaging (and no search for weak
    lines). If such lines can affect the fitted baseline
    (e.g. a high order polynomial is fitted), increase this
    parameter (usually values up to 8 are reasonable). Most
    users of this method should find the default value
    sufficient.
plot:    plot the fit and the residual. In this each
         individual fit has to be approved, by typing 'y'
         or 'n'
insitu:  if False a new scantable is returned.
         Otherwise, the scaling is done in-situ
         The default is taken from .asaprc (False)

```

Example:

```
scans.auto_poly_baseline(order=2,threshold=5)
```

8.3.8 Line Fitting

Multi-component Gaussian fitting is available. This is done by creating a fitting object, specifying fit parameters and finally fitting the data. Fitting can be done on a `scantable` selection or an entire `scantable` using the `auto_fit` function.

```

#spave is an averaged spectrum
f=sd.fitter()                # create fitter object
msk=spave.create_mask([3928,4255]) # create mask region around line
f.set_function(gauss=1)      # set a single gaussian component
f.set_scan(spave,msk)       # set the scantable and region
#
# Automatically guess start values
f.fit()                      # fit
f.plot(residual=True)        # plot residual
f.get_parameters()           # retrieve fit parameters
# 0: peak = 0.786 K , centre = 4091.236 channel, FWHM = 70.586 channel
#    area = 59.473 K channel
f.store_fit('orions_hc3n_fit.txt') # store fit
#
# To specify initial guess:
f.set_function(gauss=1)      # set a single gaussian component
f.set_gauss_parameters(0.4,4100,200\ # set initial guesses for Gaussian
    ,component=0)           # for first component (0)
# (peak,center,fwhm)
#
# For multiple components set
# initial guesses for each, e.g.
f.set_function(gauss=2)      # set two gaussian components
f.set_gauss_parameters(0.4,4100,200\ # set initial guesses for Gaussian
    ,component=0)           # for first component (0)
f.set_gauss_parameters(0.1,4200,100\ # set initial guesses for Gaussian

```

```
,component=1) # for second component (1)
```

8.3.9 Plotting

The ASAP plotter uses the same Python matplotlib library as in CASA (for x-y plots). It is accessed via the:

```
sd.plotter<TAB> # see all functions (omitted here)
sd.plotter.plot(scans) # the workhorse function
sd.plotter.set<TAB>
sd.plotter.set_abcissa sd.plotter.set_legend sd.plotter.set_range
sd.plotter.set_colors sd.plotter.set_linestyles sd.plotter.set_selection
sd.plotter.set_colours sd.plotter.set_mask sd.plotter.set_stacking
sd.plotter.set_font sd.plotter.set_mode sd.plotter.set_title
sd.plotter.set_histogram sd.plotter.set_ordinate
sd.plotter.set_layout sd.plotter.set_panelling
```

Spectra can be plotted at any time, and it will attempt to do the correct layout depending on whether it is a set of scans or a single scan.

The details of the plotter display (matplotlib) are detailed in the earlier section.

8.3.10 Single Dish Spectral Analysis Use Case With ASAP Toolkit

Below is a script that illustrates how to reduce single dish data using ASAP within CASA. First a summary of the dataset is given and then the script.

```
# MeasurementSet Name: /home/rohir3/jmcmulli/SD/OrionS_rawACSmod MS Version 2
#
# Project: AGBT06A_018_01
# Observation: GBT(1 antennas)
#
#Data records: 256 Total integration time = 1523.13 seconds
# Observed from 01:45:58 to 02:11:21
#
#Fields: 4
# ID Name Right Ascension Declination Epoch
# 0 OrionS 05:15:13.45 -05.24.08.20 J2000
# 1 OrionS 05:35:13.45 -05.24.08.20 J2000
# 2 OrionS 05:15:13.45 -05.24.08.20 J2000
# 3 OrionS 05:35:13.45 -05.24.08.20 J2000
#
#Spectral Windows: (8 unique spectral windows and 1 unique polarization setups)
# SpwID #Chans Frame Ch1(MHz) Resoln(kHz) TotBW(kHz) Ref(MHz) Corrs
# 0 8192 LSRK 45464.3506 6.10423298 50005.8766 45489.3536 RR LL HC3N
```

```
# 1      8192 LSRK  45275.7825  6.10423298  50005.8766  45300.7854  RR  LL  HN15CO
# 2      8192 LSRK  44049.9264  6.10423298  50005.8766  44074.9293  RR  LL  CH3OH
# 3      8192 LSRK  44141.2121  6.10423298  50005.8766  44166.2151  RR  LL  HCCC15N
# 12     8192 LSRK  43937.1232  6.10423356  50005.8813  43962.1261  RR  LL  HNC0
# 13     8192 LSRK  42620.4173  6.10423356  50005.8813  42645.4203  RR  LL  H15NCO
# 14     8192 LSRK  41569.9768  6.10423356  50005.8813  41594.9797  RR  LL  HNC180
# 15     8192 LSRK  43397.8198  6.10423356  50005.8813  43422.8227  RR  LL  Si0
```

```
# Scans: 21-24 Setup 1 HC3N et al
```

```
# Scans: 25-28 Setup 2 Si0 et al
```

```
casapath=os.environ['AIPSPATH']
```

```
#ASAP script                                # COMMENTS
#-----
import asap as sd                            #import ASAP package into CASA
                                              #Orion-S (Si0 line reduction only)
                                              #Notes:
                                              #scan numbers (zero-based) as compared toGBTIDL

                                              #changes made to get to OrionS_rawACSmod
                                              #modifications to label sig/ref positions
os.environ['AIPSPATH']=casapath              #set this environment variable back - ASAP changes it

s=sd.scantable('OrionS_rawACSmod',False)#load the data without averaging

s.summary()                                  #summary info
s.set_fluxunit('K')                          # make 'K' default unit
scal=sd.calps(s,[20,21,22,23])               # Calibrate HC3N scans

scal.recalc_azel()                           # recalculate az/el to
scal.opacity(0.09)                           # do opacity correction
sel=sd.selector()                             # Prepare a selection
sel.set_ifs(0)                                # select HC3N IF
scal.set_selection(sel)                       # get this IF
stave=sd.average_time(scal,weight='tintsys') # average in time
spave=stave.average_pol(weight='tsys')        # average polarizations;Tsys-weighted (1/Tsys**2) average
sd.plotter.plot(spave)                       # plot

spave.smooth('boxcar',5)                     # boxcar 5
spave.auto_poly_baseline(order=2)            # baseline fit order=2
sd.plotter.plot(spave)                       # plot

spave.set_unit('GHz')
sd.plotter.plot(spave)
sd.plotter.set_histogram(hist=True)          # draw spectrum using histogram
sd.plotter.axhline(color='r',linewidth=2)    # zline
sd.plotter.save('orions_hc3n_reduced.eps')# save postscript spectrum
```

```

spave.set_unit('channel')
rmsmask=spave.create_mask([5000,7000]) # get rms of line free regions
rms=spave.stats(stat='rms',mask=rmsmask)# rms
#-----
#Scan[0] (OrionS_ps) Time[2006/01/19/01:52:05]:
# IF[0] = 0.048
#-----
# LINE
linemask=spave.create_mask([3900,4200])
max=spave.stats('max',linemask) # IF[0] = 0.918
sum=spave.stats('sum',linemask) # IF[0] = 64.994
median=spave.stats('median',linemask) # IF[0] = 0.091
mean=spave.stats('mean',linemask) # IF[0] = 0.210

# Fitting
spave.set_unit('channel') # set units to channel
sd.plotter.plot(spave) # plot spectrum
f=sd.fitter()
msk=spave.create_mask([3928,4255]) # create region around line
f.set_function(gauss=1) # set a single gaussian component
f.set_scan(spave,msk) # set the data and region for the fitter
f.fit() # fit
f.plot(residual=True) # plot residual

f.get_parameters() # retrieve fit parameters
# 0: peak = 0.786 K , centre = 4091.236 channel, FWHM = 70.586 channel
# area = 59.473 K channel

f.store_fit('orions_hc3n_fit.txt') # store fit

# Save the spectrum
spave.save('orions_hc3n_reduced','ASCII',True) # save the spectrum

```

8.4 Single Dish Imaging

Single dish imaging is supported within CASA using standard tasks and tools. The data must be in the MeasurementSet format. Once there, you can use the `sdgrid` task or the `im` (imager) tool to create images:

Tool example:


```

scans.save('outputms','MS2')           # Save your data from ASAP into an MS

im.open('outputms')                    # open the data set
im.selectvis(nchan=901,start=30,step=1, # choose a subset of the dataa
            spwid=0,field=0)           # (just the key emission channels)
dir='J2000 17:18:29 +59.31.23'         # set map center
im.defineimage(nx=150,cellx='1.5arcmin', # define image parameters
              phasecenter=dir,mode='channel',start=30, # (note it assumes symmetry if ny,celly
              nchan=901,step=1)        # aren't specified)

im.setoptions(ftmachine='sd',cache=100000000) # choose SD gridding
im.setsdoptions(convsupport=4)          # use this many pixels to support the
                                        # gridding function used
                                        # (default=prolate spheroidal wave function)

im.makeimage(type='singledish',        # make the image
             image='FLS3a_HI.image')

```

8.4.1 Single Dish Imaging Use Case With ASAP Toolkit

Again, the data summary and then the script is given below.

```

# Project: AGBT02A_007_01
# Observation: GBT(1 antennas)
#
# Telescope Observation Date Observer Project
# GBT [ 4.57539e+09, 4.5754e+09]Lockman AGBT02A_007_01
# GBT [ 4.57574e+09, 4.57575e+09]Lockman AGBT02A_007_02
# GBT [ 4.5831e+09, 4.58313e+09]Lockman AGBT02A_031_12
#
# Thu Feb 1 23:15:15 2007 NORMAL ms::summary:
# Data records: 76860 Total integration time = 7.74277e+06 seconds
# Observed from 22:05:41 to 12:51:56
#
# Thu Feb 1 23:15:15 2007 NORMAL ms::summary:
# Fields: 2
# ID Name Right Ascension Declination Epoch
# 0 FLS3a 17:18:00.00 +59.30.00.00 J2000
# 1 FLS3b 17:18:00.00 +59.30.00.00 J2000
#
# Thu Feb 1 23:15:15 2007 NORMAL ms::summary:
# Spectral Windows: (2 unique spectral windows and 1 unique polarization setups)
# SpwID #Chans Frame Ch1(MHz) Resoln(kHz) TotBW(kHz) Ref(MHz) Corrs
# 0 1024 LSRK 1421.89269 2.44140625 2500 1420.64269 XX YY
# 1 1024 LSRK 1419.39269 2.44140625 2500 1418.14269 XX YY

# FLS3 data calibration
# this is calibration part of FLS3 data
#

```

```

casapath=os.environ['AIPSPATH']
import asap as sd
os.environ['AIPSPATH']=casapath

print '--Import--'

s=sd.scantable('FLS3_all_newcal_SP',false)           # read in MeasurementSet

print '--Split--'

# splitting the data for each field
s0=s.get_scan('FLS3a*')                             # split the data for the field of interest
s0.save('FLS3a_HI.asap')                             # save this scantable to disk (asap format)
del s0                                                # free up memory from scantable

print '--Calibrate--'
s=sd.scantable('FLS3a_HI.asap')                     # read in scantable from disk (FLS3a)
s.set_fluxunit('K')                                  # set the brightness units to Kelvin
scanns = s.getscannos()                              # get a list of scan numbers
sn=list(scanns)                                       # convert it to a list
print "No. scans to be processed:", len(scanns)

res=sd.calfs(s,sn)                                    # calibrate all scans listed using frequency
                                                # switched calibration method

print '--Save calibrated data--'
res.save('FLS3a_calfs', 'MS2')                       # Save the dataset as a MeasurementSet

print '--Image data--'

im.open('FLS3a_calfs')                               # open the data set
im.selectvis(nchan=901,start=30,step=1,              # choose a subset of the dataa
            spwid=0,field=0)                          # (just the key emission channels)
dir='J2000 17:18:29 +59.31.23'                       # set map center
im.defineimage(nx=150,cellx='1.5arcmin',             # define image parameters
              phasecenter=dir,mode='channel',start=30, # (note it assumes symmetry if ny,celly
              nchan=901,step=1)                       # aren't specified)

im.setoptions(ftmachine='sd',cache=1000000000)        # choose SD gridding
im.setsoptions(convsupport=4)                        # use this many pixels to support the
                                                # gridding function used
                                                # (default=prolate spheroidal wave function)
im.makeimage(type='singledish',image='FLS3a_HI.image') # make the image

```

8.5 Known Issues, Problems, Deficiencies and Features

The Single-Dish calibration and analysis package within CASA is still very much under development. Not surprisingly, there are a number of issues with ASAP and the SDtasks that are known

and are under repair. Some of these are non-obvious "features" of the way ASAP or `sd` is implemented, or limitations of the current Python tasking environment. Some are functions that have yet to be implemented. These currently include:

1. `sd.plotter`

Currently you can get hardcopy only after making a viewed plot. Ideally, ASAP should allow you to choose the device for plotting when you set up the plotter.

Multi-panel plotting is poor. Currently you can only add things (like lines, text, etc.) to the first panel. Also, `sd.plotter.set_range()` sets the same range for multiple panels, while we would like it to be able to set the range for each independently, including the default ranges.

The appearance of the plots need to be made a lot better. In principle matplotlib can make "publication quality" figures, but in practice you have to do alot of work to make it do that, and our plots are not good.

The `sd.plotter` object remembers things throughout the session and thus can easily get confused. For example you have to reset the range `sd.plotter.set_range()` if you have ever set it manually. This is not always the expected behavior but is a consequence of having `sd.plotter` be its own object that you feed data and commands to.

Eventually we would like the capability to interactively set things using the plots, like select frequency ranges, identify lines, start fitting.

2. `sd.selector`

The selector object only allows one selection of each type. It would be nice to be able to make a union of selections (without resorting to query) for the `set_name` - note that the others like scans and IFs work off lists which is fine. Should make `set_name` work off lists of names.

3. `sd.scantable`

There is no useful inline help on the scantable constructor when you do `help sd.scantable`, nor in `help sd`.

The inline help for `scantable.summary` claims that there is a `verbose` parameter, but there is not. The `scantable.verbosesummary` `asaprc` parameter (e.g. in `sd.rcParams`) does nothing.

GBT data has incorrect fluxunit ('Jy', should be 'K'), `freqframe` ('LSRK', is really 'TOP0') and reference frequency (set to that of the first IF only).

You cannot set the rest frequencies for GBT data. THIS IS THE MOST SERIOUS BUG RIGHT NOW.

The `sd.scantable.freq_align` does not yet work correctly.

Need to add to `scantable.stats`: 'maxord', 'minord' - the ordinate (channel, vel, freq) of the max/min

4. `sd` general issues

There should be a `sdhelp` equivalent of `toolhelp` and `tasklist` for the `sd` tools and tasks.

The current output of ASAP is verbose, and is controlled by setting `sd.rcParams['verbose']=False` (or `True`). At the least we should make some of the output less cryptic.

Strip off leading and trailing whitespace on string parameters.

5. `SDtasks` general issues

The `SDtasks` work off of files saved onto disk in one of the scantable supported formats. It might be useful to be able to work off of scantables in memory (passing the objects) but this would require changes to the tasking system. Note that this behavior is consistent throughout the `casapy` tasks.

Need interactive region selection, baseline fitting, etc.

6. `sdcal`

Can crash if `timeaverage=True` and/or `polaverage=True` and you give a list of scans that contain a combination of IFs. We need to make the tools smarter about this, but in the meantime you should restrict your scanlist and iflist to scans with the same set of IFs.

7. `sdfit`

Handles multiple IFs poorly (a general problem currently in the package).

No way to input guesses.

8. `sdplot`

Only handles included JPL line catalog.

Also, see `sd.plotter` issues above.

9. `sdstat`

Cannot return the location (channel, frequency, or velocity) of the maximum or minimum.

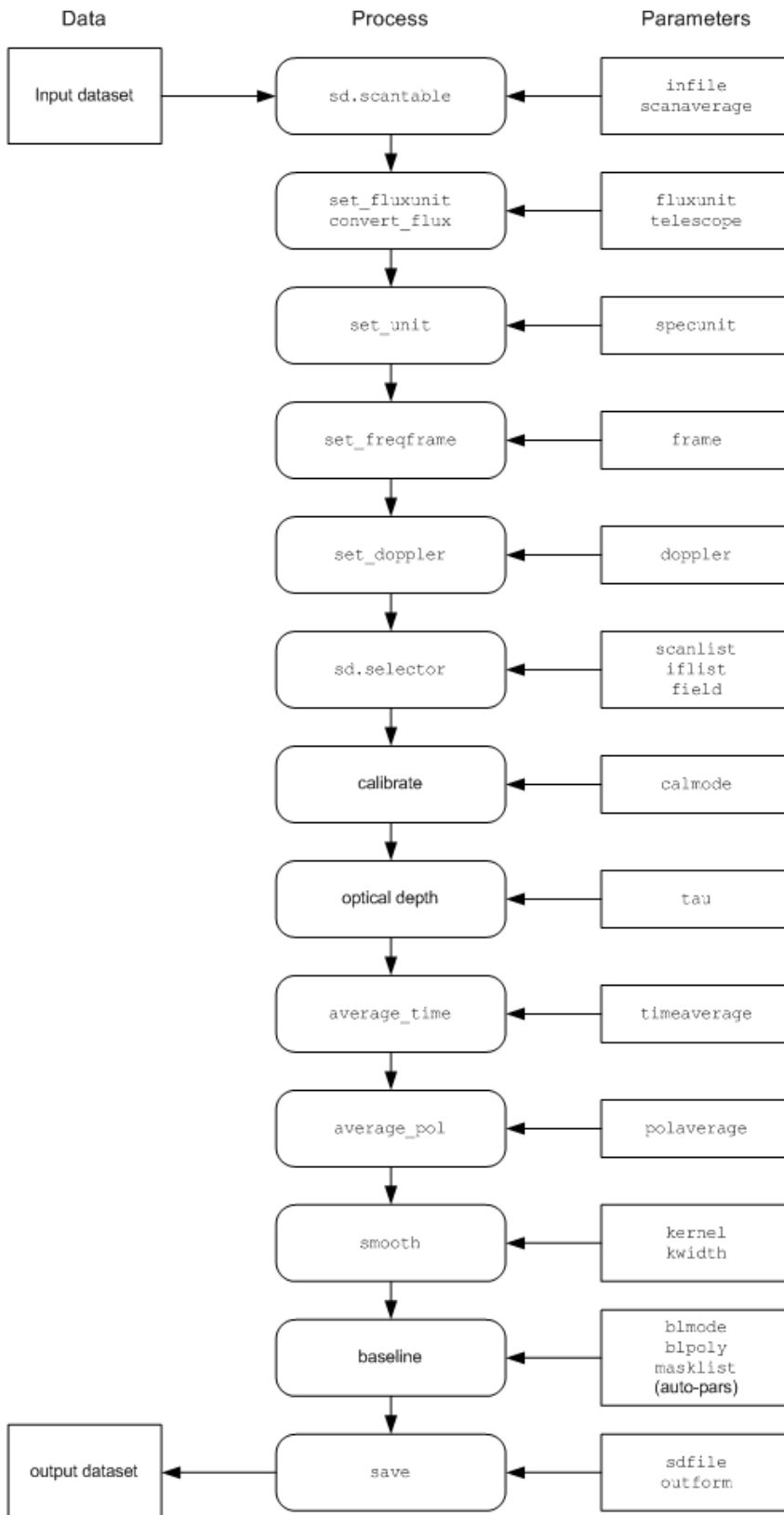


Figure 8.1: Wiring diagram for the SDtask `sdcal`. The stages of processing within the task are shown, along with the parameters that control them.

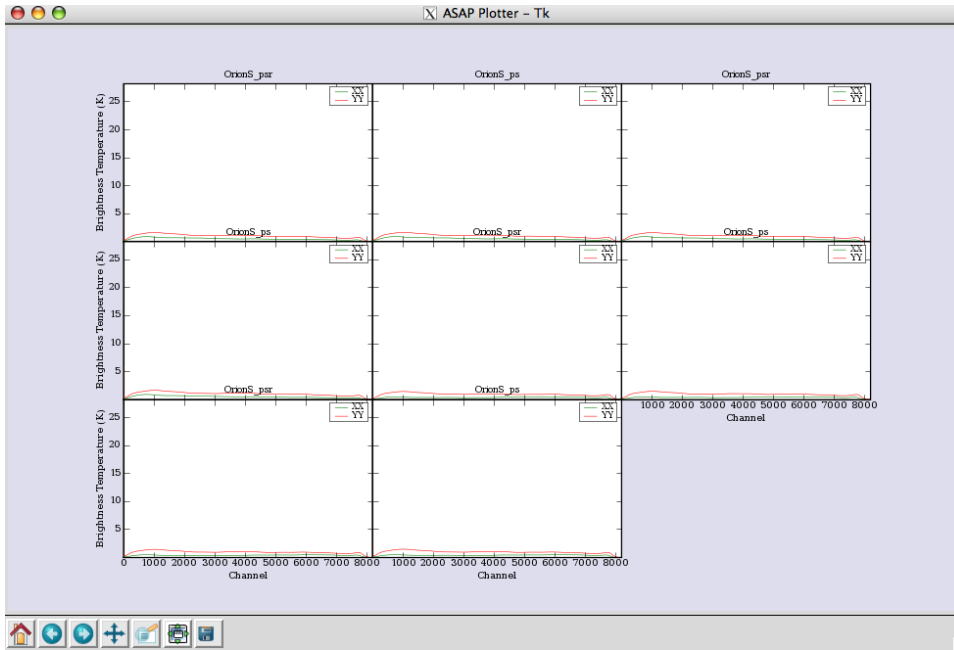


Figure 8.2: Multi-panel display of the scantable. There are two plots per scan indicating the `_psr` (reference position data) and the `_ps` (source data).

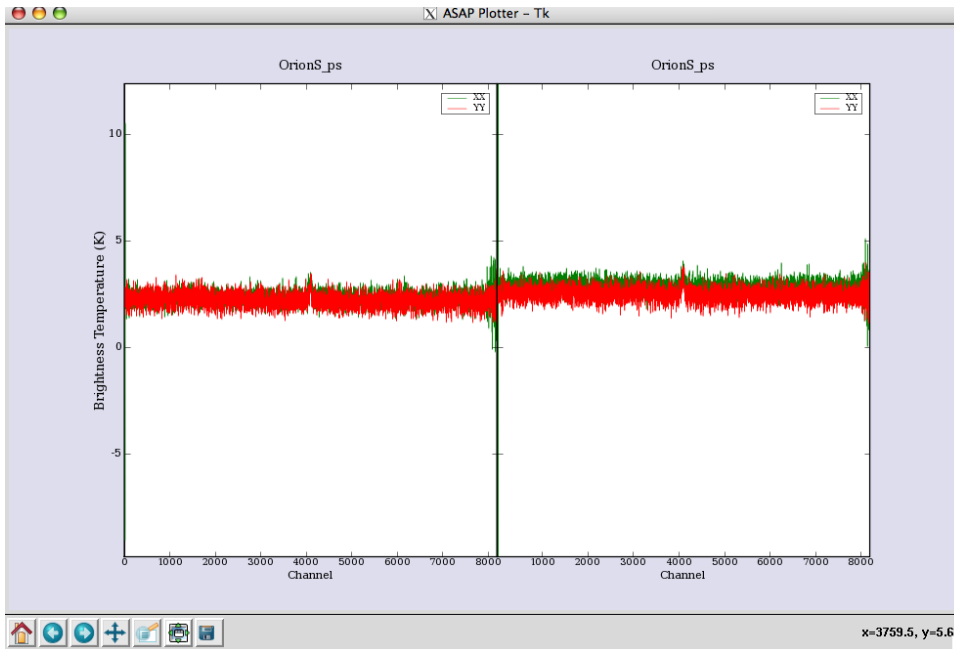


Figure 8.3: Two panel plot of the calibrated spectra. The GBT data has a separate scan for the SOURCE and REFERENCE positions so scans 20,21,22 and 23 result in these two spectra.

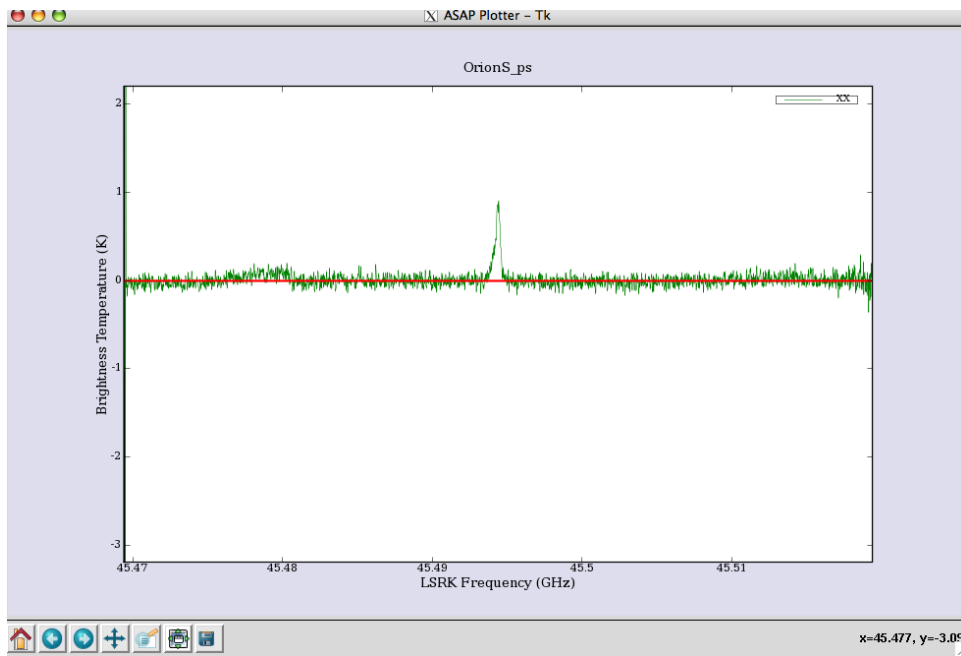


Figure 8.4: Calibrated spectrum with a line at zero (using histograms).

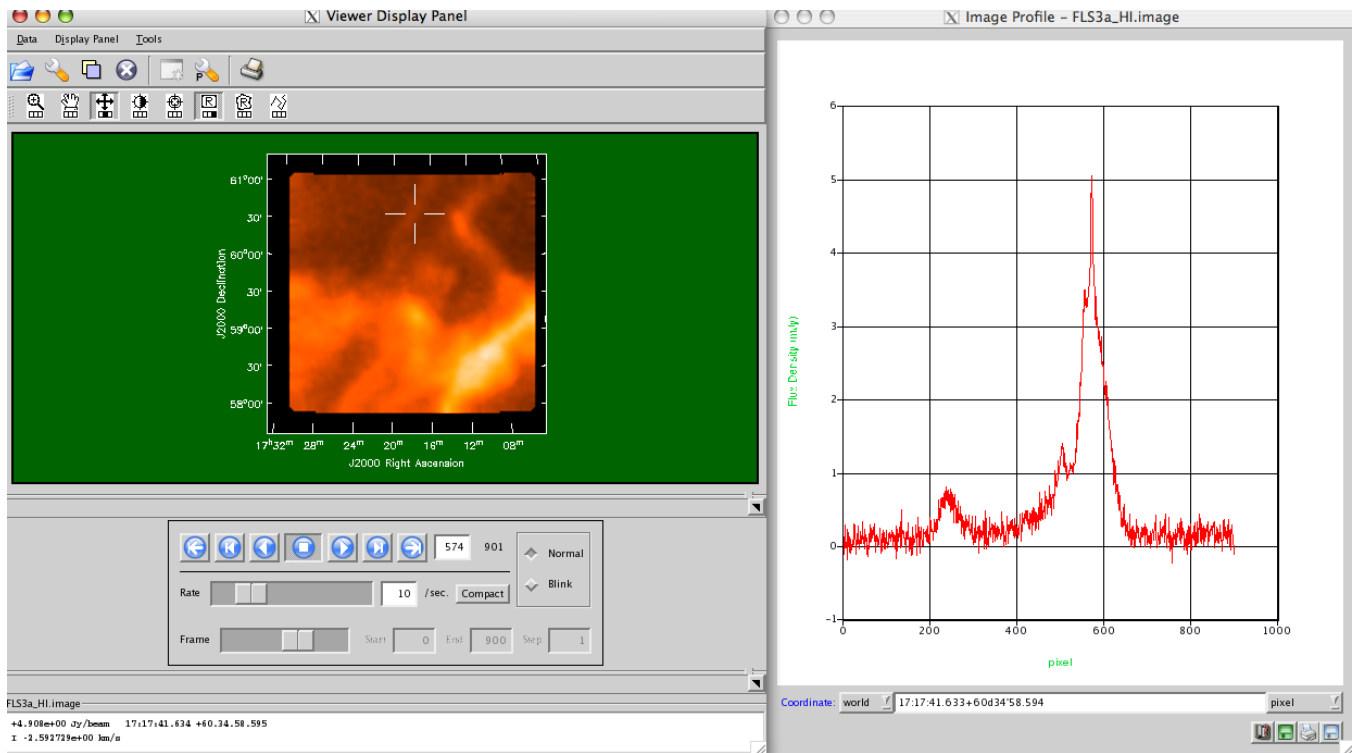


Figure 8.5: FLS3a HI emission. The display illustrates the visualization of the data cube (left) and the profile display of the cube at the cursor location (right); the Tools menu of the Viewer Display Panel has a Spectral Profile button which brings up this display. By default, it grabs the left-mouse button. Pressing down the button and moving in the display will show the profile variations.

Appendix A

Obtaining and Installing CASA

A.1 Installation Script

Currently you must be able to log into your system as the root user or an administrator user to install CASA.

The easiest way to install CASA on a RedHat Enterprise Linux (or compatible) system is to use our installation script, `load-casapy`. This script will ftp the CASA RPMs and install them. To use it, first use the link above to download it to your hard disk. Next, make sure execute permission is set for the file.

Install CASA into `/usr` by logging in as root and running:

```
load-casapy -root
```

This option will install CASA into `/usr`, but it can only be run by the root user.

Alternatively, you can visit our FTP server, download the rpms, and install them by hand. Note: you must be root/administrator to install CASA in this manner.

See the following for more details:

<https://wikio.nrao.edu/bin/view/Software/ObtainingCASA>

A.2 Startup

This section assumes that CASA has been installed on your LINUX or OSX system. *For NRAO-AOC testers, you should do the following on an AOC RHE4 machine:*

```
> . /home/casa/casainit.sh  
or  
> source /home/casa/casainit.csh
```

Appendix B

Python and CASA

CASA uses Python, IPython and matplotlib within the package. IPython is an enhanced, interactive shell to Python which provides many features for efficient command line interaction, while matplotlib is a Python 2-D plotting library for publication quality figures in different hardcopy formats.

From www.python.org: "Python is an interpreted, interactive, object-oriented programming language". Python is used as the underlying command line interface/scripting language to CASA. Thus, CASA inherits the features and the annoyances of Python. For example, since Python is inherently 0-based in its indexing of arrays, vectors, etc, CASA is also 0-based; any Index inputs (e.g., start (for start channel), fieldIndex, antennaID, etc) will start with 0. Another example is that indenting of lines means something to Python, of which users will have to be aware.

Some key links are:

- <http://python.org> – Main Python page
- <http://python.org/doc/2.4.2/ref/ref.html> – Python Reference
- <http://python.org/doc/2.4.2/tut/tut.html> – Python Tutorial
- <http://ipython.scipy.org> – IPython page
- <http://matplotlib.sourceforge.net> – matplotlib page

Each of the features of these components behave in the standard way within CASA . In the following sections, we outline the key elements for analysis interactions; see the Python references and the IPython page for the full suite of functionality.

B.1 Automatic parentheses

Automatic parenthesis is enabled for calling functions with argument lists; this feature is intended to allow less typing for common situations. IPython will display the interpretation of the line,

beneath the one typed, as indicated by the '----->'. Default behavior in CASA is to have automatic parenthesis enabled.

B.2 Indentation

Python pays attention to indentation of lines in scripts or when you enter them interactively. It uses indentation to determine the level of nesting in loops. Be careful when cutting and pasting, if you get the wrong indentation, then unpredictable things can happen (usually it just gives an error).

A blank line can be used to return the indentation to a previous level. For example, expanded parameters in tasks cause indentation in subsequent lines in the interface. For example, the following snippet of inputs from `clean` can be cut and pasted without error due to the blank line after the indented parameters:

```
mode          = 'channel'      # Type of selection
  nchan       =      -1        # Number of channels to select
  start       =         0      # Start channel
  step        =         1      # Increment between channels/velocity
  width       =         1      # Channel width

alg           = 'clark'       # Algorithm to use
```

If the blank line were not there, an error would result if you pasted this at the `casapy` prompt.

B.3 Lists and Ranges

Sometimes, you need to give a task a list of indices. For example, some tasks and tools expect a comma-separated Python list, e.g.

```
scanlist = [241, 242, 243, 244, 245, 246]
```

You can use the Python `range` function to generate a list of consecutive numbers, e.g.

```
scanlist = range(241,247)
```

giving the same list as above, e.g.

```
CASA <1>: scanlist=range(241,247)
CASA <2>: print scanlist
[241, 242, 243, 244, 245, 246]
```

Note that `range` starts from the first limit and goes to one below the second limit (Python is 0-based, and `range` is designed to work in loop functions). If only a single limit is given, the first limit is treated as 0, and the one given is used as the second, e.g.

```
CASA <3>: iflist=range(4)
CASA <4>: print iflist
[0, 1, 2, 3]
```

You can also combine multiple ranges by summing lists

```
CASA <5>: scanlist=range(241,247) + range(251,255)
CASA <6>: print scanlist
[241, 242, 243, 244, 245, 246, 251, 252, 253, 254]
```

B.4 System shell access

Any input line beginning with a `'!` character is passed verbatim (minus the `'!`, of course) to the underlying operating system (*the sole exception to this is the `'cd` command which must be executed without the `'!`*).

Several common commands (`ls`, `pwd`, `cd`, `less`) may be executed with or without the `'!`.

```
CASA [1]: pwd
/export/home/corsair-vml/jmcmulli/data
CASA [2]: ls n*
ngc5921.ms ngc5921.py
CASA [3]: !cp -r ../test.py .
```

In addition, filesystem navigation is aided through the use of bookmarks to simplify access to frequently-used directories:

```
CASA [4]: cd /home/ballista/jmcmulli/other_data
CASA [4]: pwd
/home/ballista/jmcmulli/other_data
CASA [5]: bookmark other_data
CASA [6]: cd /export/home/corsair-vml/jmcmulli/data
CASA [7]: pwd
/export/home/corsair-vml/jmcmulli/data
CASA [8]: cd -b other_data
(bookmark:data) -> /home/ballista/jmcmulli/other_data
```

Output from shell commands can be captured in two ways:

1. `sx shell_command, !!shell_command` - this captures the output to a list

```
CASA [1]: sx pwd # stores output of 'pwd' in a list
Out[1]: ['/home/basho3/jmcmulli/pretest']

CASA [2]: !!pwd # !! is a shortcut for 'sx'
Out[2]: ['/home/basho3/jmcmulli/pretest']
```

```
CASA [3]: sx ls v* # stores output of 'pwd' in a list
```

```
Out[3]:
['vla_calplot.jpg',
'vla_calplot.png',
'vla_msplot_cals.jpg',
'vla_msplot_cals.png',
'vla_plotcal_bpass.jpg',
'vla_plotcal_bpass.png',
'vla_plotcal_fcal.jpg',
'vla_plotcal_fcal.png',
'vla_plotvis.jpg',
'vla_plotvis.png']
```

```
CASA [4]: x=_ # remember '_' is a shortcut for the output from the last command
```

```
CASA [5]: x
```

```
Out[5]:
['vla_calplot.jpg',
'vla_calplot.png',
'vla_msplot_cals.jpg',
'vla_msplot_cals.png',
'vla_plotcal_bpass.jpg',
'vla_plotcal_bpass.png', 'vla_plotcal_fcal.jpg',
'vla_plotcal_fcal.png',
'vla_plotvis.jpg',
'vla_plotvis.png']
```

```
CASA [6]: y=Out[2] # or just refer to the enumerated output
```

```
CASA [7]: y
```

```
Out[7]: ['/home/basho3/jmcmulli/pretest']
```

2. sc - captures the output to a variable; options are '-l' and '-v'

```
CASA [1]: sc x=pwd # capture output from 'pwd' to the variable 'x'
```

```
CASA [2]: x
```

```
Out[2]: '/home/basho3/jmcmulli/pretest'
```

```
CASA [3]: sc -l x=pwd # capture the output from 'pwd' to the variable 'x' but
# split newlines into a list (similar to sx command)
```

```
CASA [4]: x
```

```
Out[4]: ['/home/basho3/jmcmulli/pretest']
```

```
CASA [5]: sc -v x=pwd # capture output from 'pwd' to a variable 'x' and
# show what you get (verbose mode)
```

```
x ==
```

```
'/home/basho3/jmcmulli/pretest'
```

```
CASA [6]: x
Out[6]: '/home/basho3/jmcmulli/pretest'
```

B.5 Logging

There are two components to logging within CASA. Logging of all command line inputs is done via IPython.

Upon startup, CASA will log all commands to a file called `ipython.log`. This file can be changed via the use of the `ipythonrc` file.

The following line sets up the logging for CASA . There are four options following the specification of the logging file: 1) append, 2) rotate (each session of CASA will create a new log file with a counter incrementing `ipython.log.1`, `ipython.log.2` etc, 3) over (overwrite existing file), and 4) backup (renames existing log file to `log_name`).

```
logfile ./ipython.log append
```

The command `logstate` will provide details on the current logging setup:

```
CASA [12]: logstate

File:   ipython.log
Mode:   append
State:  active
```

Logging can be turned on and off using the `logon`, `logoff` commands.

The second component is the output from applications which is directed to the file `./casapy.log`.

Your command line history is automatically maintained and stored in the local directory as `ipython.log`; this file can be edited and re-executed as appropriate using the `execfile 'filename'` feature. In addition, logging of output from commands is sent to the file `casapy.log`, also in your local directory; this is brought up automatically.

The logger has a range of features including the ability to filter messages, sort by Priority, Time, etc, and the ability to insert additional comments.

* CASA logger: `casalogger1.jpg` Figure 1:

* CASA logger - Search facility: Specify a string in the entry box to have all instances of the found string highlighted. `casalogger_select.jpg` Figure 2:

* CASA logger - Filter facility: The log output can be sorted by Priority, Time, Origin. One can also filter for a string found in the Message. `casalogger_filter.jpg` Figure 3:

* CASA logger - Insert facility: The log output can be augmented by adding notes or comments during the reduction. The file should then be saved to disk to retain these changes. `casalogger_insert.jpg` Figure 4:

B.6 History and Searching

Numbered input/output history is provided natively within IPython. Command history is also maintained on-line.

```
CASA [11]: x=1

CASA [12]: y=3*x

CASA [13]: z=x**2+y**2

CASA [14]: x
Out[14]: 1

CASA [15]: y
Out[15]: 3

CASA [16]: z
Out[16]: 10

CASA [17]: Out[14] # Note: The 'Out' vector contains command output
Out[17]: 1

CASA [18]: _15 # Note: The return value can be accessed by _number
Out[18]: 3

CASA [19]: ___ # Note: The last three return values can be accessed as:
Out[19]: 10 # _ , __ , ___
```

Command history can be accessed via the 'hist' command. The history is reset at the beginning of every CASA session, that is, typing 'hist' when you first enter start CASA will not provide any commands from the previous session; however, all of the commands are still available at the command line and can be accessed through the up,down arrow keys, and through searching.

```
CASA [22]: hist
1 : __IP.system("vi temp.py") # Note:shell commands are designated in this way
2 : ipmagic("run -i temp.py") # Note:magic commands are designated in this way
3 : ipmagic("hist ")
4 : more temp.py
5 : __IP.system("more temp.py")
6 : quickhelp() # Note: autoparenthesis are added in the history
7 : im.open('ngc5921.ms')
8 : im.summary()
9 : ipmagic("pdoc im.setdata")
10: im.close()
11: quickhelp()
12: ipmagic("logstate ")
13: x=1
```

```

14: y=3*x
15: z=x**2+y**2
16: x
17: y
18: z
19: Out[16]
20: _17
21: ---

```

The history can be saved as a script or used as a macro for further use:

```

CASA [24]: save script.py 13:16
File 'script.py' exists. Overwrite (y/[N])? y
The following commands were written to file 'script.py':
x=1
y=3*x
z=x**2+y**2
CASA [25]: !more script.py
x=1
y=3*x
z=x**2+y**2

```

Note that the history commands will be saved up to, but not including the last value (i.e., history commands 13-16 saves commands 13, 14, and 15).

There are two mechanisms for searching command history:

1. . Begin typing and use the Ctrl-p (previous,up) and Ctrl-n (next,down) to search through only

The history items that match what you've typed so far. If you use Ctrl-p,Ctrl-n at a blank prompt, they behave just like the normal arrow keys.

2. . Hit Ctrl-r; this opens a search prompt. Begin typing and the system searches your history for

lines that contain what you've typed so far, completing what it can.

For example:

```
CASA [37]:
```

```
(reverse-i-search) '':
```

Typing anything after the colon will provide you with the last command matching the characters, for example, typing 'op' finds:

```
(reverse-i-search) 'op': im.open('ngc5921.ms')
```

Subsequent hitting of Ctrl-r will search for the next command matching the characters.

B.7 Macros

Macros can be made for easy re-execution of previous commands. For example to store the commands 13-15 to the macro 'example':

```
CASA [31]: macro example 13:16
Macro 'example' created. To execute, type its name (without quotes).
Macro contents:
x=1
y=3*x
z=x**2+y**2

CASA [32]: z
Out[32]: 6

CASA [33]: z=10

CASA [34]: example
Out[34]: Executing Macro...

CASA [35]: z
Out[35]: 6

CASA [36]:
```

B.8 On-line editing

You can edit files on-line in two ways:

1. Using the shell access via '!vi'
2. Using the ed function; this will edit the file but upon closing, it will try to execute the file; using the 'script.py' example above:

```
CASA [13]: ed script.py # this will bring up the file in your chosen editor
# when you are finished editing the file,
# it will automatically
# execute it (as though you had done a
# execfile 'script.py'

Editing... done. Executing edited code...

CASA [14]: x
Out[14]: 1

CASA [15]: y
Out[15]: 3
```

```
CASA [16]: z
Out[16]: 6
```

B.9 Executing Python scripts

Python scripts are simple text files containing lists of commands as if typed at the keyboard. Note: the autoparentheses feature of IPython can not be used in scripts, that is, you should make sure all function calls have any opening and closing parentheses.

```
# file is script.py
# My script to plot the observed visibilities
plotxy('ngc5921.ms','uvdist') #yaxis defaults to amplitude
```

This can be done by using the `execfile` command to execute this script. `execfile` will execute the script as though you had typed the lines at the CASA prompt.

```
CASA [5]: execfile 'script.py'
-----> execfile('script.py')
```

B.10 How do I exit from CASA?

Hit <Cntl-d> or type at the CASA command line prompt:

```
CASA>%Exit
```

and press return.

Appendix C

Annotated Example Scripts

Note: These data sets are available with the full CASA rpm distribution. Other data sets can be made available upon request. The scripts are intended to illustrate the types of commands needed for different types of reduction/astronomical observations.

C.1 NGC 5921 — VLA red-shifted HI emission

Note: This script does not include any self-calibration steps.

```
#####  
#                                                                 #  
# Use Case Script for NGC 5921                                  #  
#                                                                 #  
# Converted by STM 2007-05-26                                    #  
# Updated          STM 2007-06-15 (Alpha Patch 1)              #  
# Last Updated STM 2007-09-05 (Alpha Patch 2+)                #  
#                                                                 #  
#####  
  
import time  
import os  
  
#  
# Set up some useful variables  
#  
# Get to path to the CASA home and strip off the name  
pathname=os.environ.get('AIPSPATH').split()[0]  
  
# This is where the NGC5921 UVFITS data will be  
fitsdata=pathname+'/data/demo/NGC5921.fits'  
  
# The prefix to use for all output files  
prefix='ngc5921.usecase'
```

```

# Clean up old files
os.system('rm -rf '+prefix+'*')

#
#=====
#
# Import the data from FITS to MS
#
print '--Import--'

# Safest to start from task defaults
default('importuvfits')

# Set up the MS filename and save as new global variable
msfile = prefix + '.ms'

# Use task importuvfits
fitsfile = fitsdata
vis = msfile
importuvfits()

#
# Note that there will be a ngc5921.usecase.ms.flagversions
# there containing the initial flags as backup for the main ms
# flags.
#
#=====
#
# List a summary of the MS
#
print '--Listobs--'

# Don't default this one and make use of the previous setting of
# vis. Remember, the variables are GLOBAL!

# You may wish to see more detailed information, like the scans.
# In this case use the verbose = True option
verbose = True

listobs()

# You should get in your logger window and in the casapy.log file
# something like:
#
# MeasurementSet Name: /home/sandrock2/smyers/Testing2/Sep07/ngc5921.usecase.ms
# MS Version 2
#
# Observer: TEST      Project:
# Observation: VLA
#

```

```

# Data records: 22653      Total integration time = 5280 seconds
#   Observed from   09:19:00   to   10:47:00
#
#   ObservationID = 0      ArrayID = 0
#   Date            Timerange                Scan  FldId  FieldName        SpwIds
#   13-Apr-1995/09:19:00.0 - 09:24:30.0    1      0 1331+30500002_0 [0]
#                   09:27:30.0 - 09:29:30.0    2      1 1445+09900002_0 [0]
#                   09:33:00.0 - 09:48:00.0    3      2 N5921_2          [0]
#                   09:50:30.0 - 09:51:00.0    4      1 1445+09900002_0 [0]
#                   10:22:00.0 - 10:23:00.0    5      1 1445+09900002_0 [0]
#                   10:26:00.0 - 10:43:00.0    6      2 N5921_2          [0]
#                   10:45:30.0 - 10:47:00.0    7      1 1445+09900002_0 [0]
#
# Fields: 3
#   ID  Code Name           Right Ascension  Declination  Epoch
#   0   C   1331+30500002_013:31:08.29    +30.30.32.96  J2000
#   1   A   1445+09900002_014:45:16.47    +09.58.36.07  J2000
#   2           N5921_2           15:22:00.00    +05.04.00.00  J2000
#
# Spectral Windows: (1 unique spectral windows and 1 unique polarization setups)
#   SpwID #Chans Frame Ch1(MHz)   Resoln(kHz) TotBW(kHz) Ref(MHz) Corrs
#   0           63 LSRK  1412.68608  24.4140625  1550.19688  1413.44902  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.
#   0   1    VLA:N7   25.0 m -107.37.07.2 +33.54.12.9
#   1   2    VLA:W1   25.0 m -107.37.05.9 +33.54.00.5
#   2   3    VLA:W2   25.0 m -107.37.07.4 +33.54.00.9
#   3   4    VLA:E1   25.0 m -107.37.05.7 +33.53.59.2
#   4   5    VLA:E3   25.0 m -107.37.02.8 +33.54.00.5
#   5   6    VLA:E9   25.0 m -107.36.45.1 +33.53.53.6
#   6   7    VLA:E6   25.0 m -107.36.55.6 +33.53.57.7
#   7   8    VLA:W8   25.0 m -107.37.21.6 +33.53.53.0
#   8   9    VLA:N5   25.0 m -107.37.06.7 +33.54.08.0
#   9  10    VLA:W3   25.0 m -107.37.08.9 +33.54.00.1
#  10  11    VLA:N4   25.0 m -107.37.06.5 +33.54.06.1
#  11  12    VLA:W5   25.0 m -107.37.13.0 +33.53.57.8
#  12  13    VLA:N3   25.0 m -107.37.06.3 +33.54.04.8
#  13  14    VLA:N1   25.0 m -107.37.06.0 +33.54.01.8
#  14  15    VLA:N2   25.0 m -107.37.06.2 +33.54.03.5
#  15  16    VLA:E7   25.0 m -107.36.52.4 +33.53.56.5
#  16  17    VLA:E8   25.0 m -107.36.48.9 +33.53.55.1
#  17  18    VLA:W4   25.0 m -107.37.10.8 +33.53.59.1
#  18  19    VLA:E5   25.0 m -107.36.58.4 +33.53.58.8
#  19  20    VLA:W9   25.0 m -107.37.25.1 +33.53.51.0
#  20  21    VLA:W6   25.0 m -107.37.15.6 +33.53.56.4
#  21  22    VLA:E4   25.0 m -107.37.00.8 +33.53.59.7

```

```

# 23 24 VLA:E2 25.0 m -107.37.04.4 +33.54.01.1
# 24 25 VLA:N6 25.0 m -107.37.06.9 +33.54.10.3
# 25 26 VLA:N9 25.0 m -107.37.07.8 +33.54.19.0
# 26 27 VLA:N8 25.0 m -107.37.07.5 +33.54.15.8
# 27 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 273 rows
# OBSERVATION 1 row
# POINTING 168 rows
# POLARIZATION 1 row
# PROCESSOR <empty>
# SOURCE 3 rows
# SPECTRAL_WINDOW 1 row
# STATE <empty>
# SYSCAL <absent>
# WEATHER <absent>
#
#
#=====
#
# Get rid of the autocorrelations from the MS
#
print '--Flagautocorr--'

# Don't default this one either, there is only one parameter (vis)

flagautocorr()

#
#=====
#
# Set the fluxes of the primary calibrator(s)
#
print '--Setjy--'
default('setjy')

vis = msfile

#
# 1331+305 = 3C286 is our primary calibrator
# Use the wildcard on the end of the source name
# since the field names in the MS have inherited the

```

```

# AIPS qualifiers
field = '1331+305*'

# This is 1.4GHz D-config and 1331+305 is sufficiently unresolved
# that we dont need a model image. For higher frequencies
# (particularly in A and B config) you would want to use one.
modimage = ''

# Setjy knows about this source so we dont need anything more

setjy()

#
# You should see something like this in the logger and casapy.log file:
#
# 1331+30500002_0 spwid= 0 [I=14.76, Q=0, U=0, V=0] Jy, (Perley-Taylor 99)
#
# So its using 14.76Jy as the flux of 1331+305 in the single Spectral Window
# in this MS.
#
#=====
#
# Bandpass calibration
#
print '--Bandpass--'
default('bandpass')

# We can first do the bandpass on the single 5min scan on 1331+305
# At 1.4GHz phase stability should be sufficient to do this without
# a first (rough) gain calibration. This will give us the relative
# antenna gain as a function of frequency.

vis = msfile

# set the name for the output bandpass caltable
btable = prefix + '.bcal'
caltable = btable

# No gain tables yet
gaintable = ''

# Use flux calibrator 1331+305 = 3C286 (FIELD_ID 0) as bandpass calibrator
field = '0'
# all channels
spw = ''
# No other selection
selectdata = False

# In this band we do not need a-priori corrections for
# antenna gain-elevation curve or atmospheric opacity
# (at 8GHz and above you would want these)

```

```

gaincurve = False
opacity = 0.0

# Choose bandpass solution type
# Pick standard time-binned B (rather than BPOLY)
bandtype = 'B'

# set solution interval arbitrarily long (get single bpass)
solint = 86400.0

# reference antenna Name 15 (15=VLA:N2) (Id 14)
refant = '15'

bandpass()

# You can use plotcal to examine the solutions
#default('plotcal')
#tablein = btable
#yaxis = 'amp'
#field = '0'
#multiplot = True
#plotcal()
#
#yaxis = 'phase'
#plotcal()
#
# Note the rolloff in the start and end channels. Looks like
# channels 6-56 (out of 0-62) are the best

#=====
#
# Gain calibration
#
print '--Gaincal--'
default('gaincal')

# Armed with the bandpass, we now solve for the
# time-dependent antenna gains

vis = msfile

# set the name for the output gain caltable
gtable = prefix + '.gcal'
caltable = gtable

# Use our previously determined bandpass
# Note this will automatically be applied to all sources
# not just the one used to determine the bandpass
bptable = btable

# Gain calibrators are 1331+305 and 1445+099 (FIELD_ID 0 and 1)

```



```

field = '0,1'

# We have only a single spectral window (SPW 0)
# Choose 51 channels 6-56 out of the 63
# to avoid end effects.
# Channel selection is done inside spw
spw = '0:6~56'

# No other selection
selectdata = False

# In this band we do not need a-priori corrections for
# antenna gain-elevation curve or atmospheric opacity
# (at 8GHz and above you would want these)
gaincurve = False
opacity = 0.0

# scan-based G solutions for both amplitude and phase
gaintype = 'G'
solint = 0.
calmode = 'ap'

# reference antenna 15 (15=VLA:N2)
refant = '15'

gaincal()

# You can use plotcal to examine the gain solutions
#default('plotcal')
#tablein = gtable
#yaxis = 'amp'
#field = '0,1'
#multiplot = True
#plotcal()
#
#yaxis = 'phase'
#plotcal()
#
# The amp and phase coherence looks good

#=====
#
# Bootstrap flux scale
#
print '--Fluxscale--'
default('fluxscale')

vis = msfile

# set the name for the output rescaled caltable
ftable = prefix + '.fluxscale'

```

```

fluxtable = ftable

# point to our first gain cal table
caltable = gtable

# we will be using 1331+305 (the source we did setjy on) as
# our flux standard reference - note its extended name as in
# the FIELD table summary above (it has a VLA seq number appended)
reference = '1331*'

# we want to transfer the flux to our other gain cal source 1445+099
transfer = '1445*'

fluxscale()

# In the logger you should see something like:
# Flux density for 1445+09900002_0 in SpW=0 is:
#   2.48576 +/- 0.00123122 (SNR = 2018.94, nAnt= 27)

# If you run plotcal() on the tablein = 'ngc5921.usecase.fluxscale'
# you will see now it has brought the amplitudes in line between
# the first scan on 1331+305 and the others on 1445+099

#=====
#
# Apply our calibration solutions to the data
# (This will put calibrated data into the CORRECTED_DATA column)
#
print '--ApplyCal--'
default('applycal')

vis = msfile

# We want to correct the calibrators using themselves
# and transfer from 1445+099 to itself and the target N5921

# Start with the fluxscale/gain and bandpass tables
bptable = btable
gaintable = ftable

# all channels
spw = ''
selectdata = False

# as before
gaincurve = False
opacity = 0.0

# select the fields for 1445+099 and N5921
field = '1,2'

```

```

# pick the 1445+099 out of the gain table for transfer
# (NOTE: this currently uses TaQL strings)
gainselect = 'FIELD_ID==1'

applycal()

# Now for completeness apply 1331+305 to itself

field = '0'
gainselect = 'FIELD_ID==0'

# The CORRECTED_DATA column now contains the calibrated visibilities

applycal()

#=====
#
# Split the gain calibrator data
#
print '--Split (cal data)--'
default('split')

vis = msfile

# We first want to write out the corrected data for the calibrator

# Make an output vis file
calsplitms = prefix + '.cal.split.ms'
outputvis = calsplitms

# Select the 1445+099 field, all chans
field = '1445*'
spw = ''

# pick off the CORRECTED_DATA column
datacolumn = 'corrected'

split()

#=====
#
# UV-plane continuum subtraction on the target
# (this will update the CORRECTED_DATA column)
#
print '--UV Continuum Subtract--'
default('uvcontsub')

vis = msfile

# Pick off N5921
field = 'N5921*'

```

```

# Use channels 4-6 and 50-59 for continuum
#spw = '0:4~6;50~59'
# ALPHA ALERT: still does not use standard notation
spw = '0'
channels = range(4,7)+range(50,60)

# Averaging time (none)
solint = 0.0

# Fit only a mean level
fitorder = 0

# Do the uv-plane subtraction
fitmode = 'subtract'

# Let it split out the data automatically for us
splitdata = True

uvcontsub()

# You will see it made two new MS:
# ngc5921.usecase.ms.cont
# ngc5921.usecase.ms.contsub

srcsplitms = msfile + '.contsub'

# Note that ngc5921.usecase.ms.contsub contains the uv-subtracted
# visibilities (in its DATA column), and ngc5921.usecase.ms.contsub
# the pseudo-continuum visibilities (as fit).

# The original ngc5921.usecase.ms now contains the uv-continuum
# subtracted vis in its CORRECTED_DATA column and the continuum
# in its MODEL_DATA column as per the fitmode='subtract'

#=====
#
# Done with calibration
# Now clean an image cube of N5921
#
print '--Clean--'
default('clean')

# Pick up our split source data
vis = srcsplitms

# Make an image root file name
imname = prefix + '.clean'
imagenname = imname

# Set up the output image cube

```

```
mode = 'channel'
nchan = 46
start = 5
step = 1

# This is a single-source MS with one spw
field = '0'
spw = ''

# Set the output image size and cell size (arcsec)
imsize = [256,256]
cell = [15.,15.]

# Do a simple Hogbom clean, standard gain factor 0.1
alg = 'hogbom'
gain = 0.1

# Fix maximum number of iterations
niter = 6000

# Also set flux residual threshold (in mJy)
threshold=8.0

# Set up the weighting
# Use Briggs weighting (a moderate value, on the uniform side)
weighting = 'briggs'
rmode = 'norm'
robust = 0.5

# No clean mask or cleanbox for now
mask = ''
cleanbox = []

# But if you had a cleanbox saved in a file, e.g. "regionfile.txt"
# you could use it:
#cleanbox='regionfile.txt'
#
# and if you wanted to use interactive clean
#cleanbox='interactive'

clean()

# Should find stuff in the logger like:
#
# Fitted beam used in restoration: 51.5643 by 45.6021 (arcsec) at pa 14.5411 (deg)
#
# It will have made the images:
# -----
# ngc5921.usecase.clean.image
# ngc5921.usecase.clean.model
# ngc5921.usecase.clean.residual
```

```

clnimage = imname+'.image'

#=====
#
# Done with imaging
# Now view the image cube of N5921
#
#print '--View image--'
#viewer(clnimage,'image')

#=====
#
# Can do some image statistics if you wish
# Treat this like a regression script
# WARNING: currently requires toolkit
#
print ' NGC5921 results '
print ' ===== '

# Pull the max cal amp value out of the MS
ms.open(calsplitms)
thistest_cal = max(ms.range(["amplitude"]).get('amplitude'))
ms.close()
oldtest_cal = 34.0338668823
print ' Cal Max amplitude should be ',oldtest_cal
print ' Found : Max = ',thistest_cal
diff_cal = abs((oldtest_cal-thistest_cal)/oldtest_cal)
print ' Difference (fractional) = ',diff_cal

print ''
# Pull the max src amp value out of the MS
ms.open(srcsplitms)
thistest_src = max(ms.range(["amplitude"]).get('amplitude'))
ms.close()
#oldtest_src = 1.37963354588 # This was in chans 5-50
oldtest_src = 46.2060050964 # now in all chans
print ' Src Max amplitude should be ',oldtest_src
print ' Found : Max = ',thistest_src
diff_src = abs((oldtest_src-thistest_src)/oldtest_src)
print ' Difference (fractional) = ',diff_src

print ''
# Pull the max and rms from the clean image
ia.open(clnimage)
statistics=ia.statistics()
thistest_immax=statistics['max'][0]
oldtest_immax = 0.052414759993553162
print ' Clean image max should be ',oldtest_immax
print ' Found : Image Max = ',thistest_immax
diff_immax = abs((oldtest_immax-thistest_immax)/oldtest_immax)

```

```

print ' Difference (fractional) = ',diff_immax

print ''
# note thistest_imrms will be a list with one value
thistest_imrms=statistics['rms'][0]
oldtest_imrms = 0.0020218724384903908
print ' Clean image rms should be ',oldtest_imrms
print ' Found : Image rms = ',thistest_imrms
diff_imrms = abs((oldtest_imrms-thistest_imrms)/oldtest_imrms)
print ' Difference (fractional) = ',diff_imrms
ia.close()

print ''
print '--- Done ---'

```

C.1.1 NGC 5921 data summary

Summary created with `listobs('ngc5921.usecase.ms',verbose=True)`: This is written to the logger and the `casapy.log` file.

```

Observer: TEST      Project:
Observation: VLA

```

```

Data records: 22653      Total integration time = 5280 seconds
Observed from 09:19:00 to 10:47:00

```

```

ObservationID = 0      ArrayID = 0
Date           Timerange           Scan  FldId  FieldName      SpwIds
13-Apr-1995/09:19:00.0 - 09:24:30.0    1     0  1331+30500002_0 [0]
                09:27:30.0 - 09:29:30.0    2     1  1445+09900002_0 [0]
                09:33:00.0 - 09:48:00.0    3     2  N5921_2          [0]
                09:50:30.0 - 09:51:00.0    4     1  1445+09900002_0 [0]
                10:22:00.0 - 10:23:00.0    5     1  1445+09900002_0 [0]
                10:26:00.0 - 10:43:00.0    6     2  N5921_2          [0]
                10:45:30.0 - 10:47:00.0    7     1  1445+09900002_0 [0]

```

Fields: 3

ID	Name	Right Ascension	Declination	Epoch
0	1331+30500002_0	13:31:08.29	+30.30.32.96	J2000
1	1445+09900002_0	14:45:16.47	+09.58.36.07	J2000
2	N5921_2	15:22:00.00	+05.04.00.00	J2000

Spectral Windows: (1 unique spectral windows and 1 unique polarization setups)

SpwID	#Chans	Frame	Ch1(MHz)	Resoln(kHz)	TotBW(kHz)	Ref(MHz)	Corrs
0	63	LSRK	1412.68608	24.4140625	1550.19688	1413.44902	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.
0	1	VLA:N7	25.0 m	-107.37.07.2	+33.54.12.9
1	2	VLA:W1	25.0 m	-107.37.05.9	+33.54.00.5
2	3	VLA:W2	25.0 m	-107.37.07.4	+33.54.00.9
3	4	VLA:E1	25.0 m	-107.37.05.7	+33.53.59.2
4	5	VLA:E3	25.0 m	-107.37.02.8	+33.54.00.5
5	6	VLA:E9	25.0 m	-107.36.45.1	+33.53.53.6
6	7	VLA:E6	25.0 m	-107.36.55.6	+33.53.57.7
7	8	VLA:W8	25.0 m	-107.37.21.6	+33.53.53.0
8	9	VLA:N5	25.0 m	-107.37.06.7	+33.54.08.0
9	10	VLA:W3	25.0 m	-107.37.08.9	+33.54.00.1
10	11	VLA:N4	25.0 m	-107.37.06.5	+33.54.06.1
11	12	VLA:W5	25.0 m	-107.37.13.0	+33.53.57.8
12	13	VLA:N3	25.0 m	-107.37.06.3	+33.54.04.8
13	14	VLA:N1	25.0 m	-107.37.06.0	+33.54.01.8
14	15	VLA:N2	25.0 m	-107.37.06.2	+33.54.03.5
15	16	VLA:E7	25.0 m	-107.36.52.4	+33.53.56.5
16	17	VLA:E8	25.0 m	-107.36.48.9	+33.53.55.1
17	18	VLA:W4	25.0 m	-107.37.10.8	+33.53.59.1
18	19	VLA:E5	25.0 m	-107.36.58.4	+33.53.58.8
19	20	VLA:W9	25.0 m	-107.37.25.1	+33.53.51.0
20	21	VLA:W6	25.0 m	-107.37.15.6	+33.53.56.4
21	22	VLA:E4	25.0 m	-107.37.00.8	+33.53.59.7
23	24	VLA:E2	25.0 m	-107.37.04.4	+33.54.01.1
24	25	VLA:N6	25.0 m	-107.37.06.9	+33.54.10.3
25	26	VLA:N9	25.0 m	-107.37.07.8	+33.54.19.0
26	27	VLA:N8	25.0 m	-107.37.07.5	+33.54.15.8
27	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	353 rows
OBSERVATION	1 row
POINTING	168 rows
POLARIZATION	1 row
PROCESSOR	<empty>
SOURCE	3 rows
SPECTRAL_WINDOW	1 row
STATE	<empty>
SYSCAL	<absent>
WEATHER	<absent>

C.2 Jupiter — VLA continuum polarization

Note: This script includes interactive flagging and cleaning and self-calibration loops. Polarization calibration and imaging is still missing.

```
#####
#                                                                 #
# Use Case Script for Jupiter 6cm VLA                            #
#                                                                 #
# Last Updated STM 2007-09-04 (Alpha Patch 2+)                  #
#                                                                 #
#####

import time
import os

#
#=====
#
# This script has some interactive commands: scriptmode = True
# if you are running it and want it to stop during interactive parts.

scriptmode = True

#=====
#
# Set up some useful variables - these will be set during the script
# also, but if you want to restart the script in the middle here
# they are in one place:

pathname=os.environ.get('AIPSPATH').split()[0]
prefix='jupiter6cm.usecase'

msfile = prefix + '.ms'

gtable = prefix + '.gcal'
ftable = prefix + '.fluxscale'
atable = prefix + '.accum'

srcsplitms = prefix + '.split.ms'

clnimage1 = [288,288]
clncell = [4.,4.]

imname1 = prefix + '.clean1'
clnimage1 = imname1+'.image'
clnmodel1 = imname1+'.model'
clnresid1 = imname1+'.residual'
clnmask1 = imname1+'.clean_interactive.mask'
```

```

selfcaltab1 = srcsplitms + '.selfcal1'

imname2 = prefix + '.clean2'
clnimage2 = imname2+'.image'
clnmodel2 = imname2+'.model'
clnresid2 = imname2+'.residual'
clnmask2 = imname2+'.clean_interactive.mask'

selfcaltab2 = srcsplitms + '.selfcal2'
smoothcaltab2 = srcsplitms + '.smoothcal2'

imname3 = prefix + '.clean3'
clnimage3 = imname3+'.image'
clnmodel3 = imname3+'.model'
clnresid3 = imname3+'.residual'
clnmask3 = imname3+'.clean_interactive.mask'

#
#=====
#
# Get to path to the CASA home and strip off the name
pathname=os.environ.get('AIPSPATH').split()[0]

# This is where the UVFITS data will be
#fitsdata=pathname+'/data/demo/jupiter6cm.fits'
fitsdata='/home/sandrock2/smyers/NAUG2/Data/VLA_CONT/FLUX99-6CM.CBAND'

# The prefix to use for all output files
prefix='jupiter6cm.usecase'

# Clean up old files
os.system('rm -rf '+prefix+'*')

#
#=====
# Data Import and List
#=====
#
# Import the data from FITS to MS
#
print '--Import--'

# Safest to start from task defaults
default('importuvfits')

# Set up the MS filename and save as new global variable
msfile = prefix + '.ms'

# Use task importuvfits
fitsfile = fitsdata
vis = msfile

```

```

importuvfits()

#=====
#
# List a summary of the MS
#
print '--Listobs--'

# Don't default this one and make use of the previous setting of
# vis. Remember, the variables are GLOBAL!

# You may wish to see more detailed information, in this case
# use the verbose = True option
verbose = True

listobs()

# You should get in your logger window and in the casapy.log file
# something like:
#
#   Observer: FLUX99      Project:
# Observation: VLA
#
# Data records: 2021424      Total integration time = 85133.2 seconds
#   Observed from  23:15:27   to   22:54:20
#
#   ObservationID = 0      ArrayID = 0
#   Date           Timerange           Scan  FldId  FieldName      SpwIds
#   15-Apr-1999/23:15:26.7 - 23:16:10.0    1     0  0137+331      [0, 1]
#                   23:38:40.0 - 23:48:00.0    2     1  0813+482      [0, 1]
#                   23:53:40.0 - 23:55:20.0    3     2  0542+498      [0, 1]
#   16-Apr-1999/00:22:10.1 - 00:23:49.9    4     3  0437+296      [0, 1]
#                   00:28:23.3 - 00:30:00.1    5     4  VENUS         [0, 1]
#                   00:48:40.0 - 00:50:20.0    6     1  0813+482      [0, 1]
#                   00:56:13.4 - 00:57:49.9    7     2  0542+498      [0, 1]
#                   01:10:20.1 - 01:11:59.9    8     5  0521+166      [0, 1]
#                   01:23:29.9 - 01:25:00.1    9     3  0437+296      [0, 1]
#                   01:29:33.3 - 01:31:10.0   10     4  VENUS         [0, 1]
#                   01:49:50.0 - 01:51:30.0   11     6  1411+522      [0, 1]
#                   02:03:00.0 - 02:04:30.0   12     7  1331+305      [0, 1]
#                   02:17:30.0 - 02:19:10.0   13     1  0813+482      [0, 1]
#                   02:24:20.0 - 02:26:00.0   14     2  0542+498      [0, 1]
#                   02:37:49.9 - 02:39:30.0   15     5  0521+166      [0, 1]
#                   02:50:50.1 - 02:52:20.1   16     3  0437+296      [0, 1]
#                   02:59:20.0 - 03:01:00.0   17     6  1411+522      [0, 1]
#                   03:12:30.0 - 03:14:10.0   18     7  1331+305      [0, 1]
#                   03:27:53.3 - 03:29:39.9   19     1  0813+482      [0, 1]
#                   03:35:00.0 - 03:36:40.0   20     2  0542+498      [0, 1]
#                   03:49:50.0 - 03:51:30.1   21     6  1411+522      [0, 1]
#                   04:03:10.0 - 04:04:50.0   22     7  1331+305      [0, 1]
#                   04:18:49.9 - 04:20:40.0   23     1  0813+482      [0, 1]

```

#	04:25:56.6 - 04:27:39.9	24	2 0542+498	[0, 1]
#	04:42:49.9 - 04:44:40.0	25	8 MARS	[0, 1]
#	04:56:50.0 - 04:58:30.1	26	6 1411+522	[0, 1]
#	05:24:03.3 - 05:33:39.9	27	7 1331+305	[0, 1]
#	05:48:00.0 - 05:49:49.9	28	1 0813+482	[0, 1]
#	05:58:36.6 - 06:00:30.0	29	8 MARS	[0, 1]
#	06:13:20.1 - 06:14:59.9	30	6 1411+522	[0, 1]
#	06:27:40.0 - 06:29:20.0	31	7 1331+305	[0, 1]
#	06:44:13.4 - 06:46:00.0	32	1 0813+482	[0, 1]
#	06:55:06.6 - 06:57:00.0	33	8 MARS	[0, 1]
#	07:10:40.0 - 07:12:20.0	34	6 1411+522	[0, 1]
#	07:28:20.0 - 07:30:10.1	35	7 1331+305	[0, 1]
#	07:42:49.9 - 07:44:30.0	36	8 MARS	[0, 1]
#	07:58:43.3 - 08:00:39.9	37	6 1411+522	[0, 1]
#	08:13:30.0 - 08:15:19.9	38	7 1331+305	[0, 1]
#	08:27:53.4 - 08:29:30.0	39	8 MARS	[0, 1]
#	08:42:59.9 - 08:44:50.0	40	6 1411+522	[0, 1]
#	08:57:09.9 - 08:58:50.0	41	7 1331+305	[0, 1]
#	09:13:03.3 - 09:14:50.1	42	9 NGC7027	[0, 1]
#	09:26:59.9 - 09:28:40.0	43	6 1411+522	[0, 1]
#	09:40:33.4 - 09:42:09.9	44	7 1331+305	[0, 1]
#	09:56:19.9 - 09:58:10.0	45	9 NGC7027	[0, 1]
#	10:12:59.9 - 10:14:50.0	46	8 MARS	[0, 1]
#	10:27:09.9 - 10:28:50.0	47	6 1411+522	[0, 1]
#	10:40:30.0 - 10:42:00.0	48	7 1331+305	[0, 1]
#	10:56:10.0 - 10:57:50.0	49	9 NGC7027	[0, 1]
#	11:28:30.0 - 11:35:30.0	50	10 NEPTUNE	[0, 1]
#	11:48:20.0 - 11:50:10.0	51	6 1411+522	[0, 1]
#	12:01:36.7 - 12:03:10.0	52	7 1331+305	[0, 1]
#	12:35:33.3 - 12:37:40.0	53	11 URANUS	[0, 1]
#	12:46:30.0 - 12:48:10.0	54	10 NEPTUNE	[0, 1]
#	13:00:29.9 - 13:02:10.0	55	6 1411+522	[0, 1]
#	13:15:23.3 - 13:17:10.1	56	9 NGC7027	[0, 1]
#	13:33:43.3 - 13:35:40.0	57	11 URANUS	[0, 1]
#	13:44:30.0 - 13:46:10.0	58	10 NEPTUNE	[0, 1]
#	14:00:46.7 - 14:01:39.9	59	0 0137+331	[0, 1]
#	14:10:40.0 - 14:12:09.9	60	12 JUPITER	[0, 1]
#	14:24:06.6 - 14:25:40.1	61	11 URANUS	[0, 1]
#	14:34:30.0 - 14:36:10.1	62	10 NEPTUNE	[0, 1]
#	14:59:13.4 - 15:00:00.0	63	0 0137+331	[0, 1]
#	15:09:03.3 - 15:10:40.1	64	12 JUPITER	[0, 1]
#	15:24:30.0 - 15:26:20.1	65	9 NGC7027	[0, 1]
#	15:40:10.0 - 15:45:00.0	66	11 URANUS	[0, 1]
#	15:53:50.0 - 15:55:20.0	67	10 NEPTUNE	[0, 1]
#	16:18:53.4 - 16:19:49.9	68	0 0137+331	[0, 1]
#	16:29:10.1 - 16:30:49.9	69	12 JUPITER	[0, 1]
#	16:42:53.4 - 16:44:30.0	70	11 URANUS	[0, 1]
#	16:54:53.4 - 16:56:40.0	71	9 NGC7027	[0, 1]
#	17:23:06.6 - 17:30:40.0	72	2 0542+498	[0, 1]
#	17:41:50.0 - 17:43:20.0	73	3 0437+296	[0, 1]
#	17:55:36.7 - 17:57:39.9	74	4 VENUS	[0, 1]

```

#           18:19:23.3 - 18:20:09.9   75      0 0137+331      [0, 1]
#           18:30:23.3 - 18:32:00.0   76      12 JUPITER      [0, 1]
#           18:44:49.9 - 18:46:30.0   77      9 NGC7027      [0, 1]
#           18:59:13.3 - 19:00:59.9   78      2 0542+498      [0, 1]
#           19:19:10.0 - 19:21:20.1   79      5 0521+166      [0, 1]
#           19:32:50.1 - 19:34:29.9   80      3 0437+296      [0, 1]
#           19:39:03.3 - 19:40:40.1   81      4 VENUS         [0, 1]
#           20:08:06.7 - 20:08:59.9   82      0 0137+331      [0, 1]
#           20:18:10.0 - 20:19:50.0   83      12 JUPITER      [0, 1]
#           20:33:53.3 - 20:35:40.1   84      1 0813+482      [0, 1]
#           20:40:59.9 - 20:42:40.0   85      2 0542+498      [0, 1]
#           21:00:16.6 - 21:02:20.1   86      5 0521+166      [0, 1]
#           21:13:53.4 - 21:15:29.9   87      3 0437+296      [0, 1]
#           21:20:43.4 - 21:22:30.0   88      4 VENUS         [0, 1]
#           21:47:26.7 - 21:48:20.1   89      0 0137+331      [0, 1]
#           21:57:30.0 - 21:59:10.0   90      12 JUPITER      [0, 1]
#           22:12:13.3 - 22:14:00.1   91      2 0542+498      [0, 1]
#           22:28:33.3 - 22:30:19.9   92      4 VENUS         [0, 1]
#           22:53:33.3 - 22:54:19.9   93      0 0137+331      [0, 1]
#
# Fields: 13
#   ID   Name           Right Ascension  Declination   Epoch
#   0    0137+331       01:37:41.30    +33.09.35.13 J2000
#   1    0813+482       08:13:36.05    +48.13.02.26 J2000
#   2    0542+498       05:42:36.14    +49.51.07.23 J2000
#   3    0437+296       04:37:04.17    +29.40.15.14 J2000
#   4    VENUS         04:06:54.11    +22.30.35.91 J2000
#   5    0521+166       05:21:09.89    +16.38.22.05 J2000
#   6    1411+522       14:11:20.65    +52.12.09.14 J2000
#   7    1331+305       13:31:08.29    +30.30.32.96 J2000
#   8    MARS          14:21:41.37    -12.21.49.45 J2000
#   9    NGC7027       21:07:01.59    +42.14.10.19 J2000
#  10    NEPTUNE       20:26:01.14    -18.54.54.21 J2000
#  11    URANUS        21:15:42.83    -16.35.05.59 J2000
#  12    JUPITER       00:55:34.04    +04.45.44.71 J2000
#
# Spectral Windows: (2 unique spectral windows and 1 unique polarization setups)
#   SpwID #Chans Frame Ch1(MHz)   Resoln(kHz) TotBW(kHz) Ref(MHz)   Corrs
#   0      1   TOPO  4885.1     50000      50000     4885.1   RR RL LR LL
#   1      1   TOPO  4835.1     50000      50000     4835.1   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.
#   0    1     VLA:W9   25.0 m  -107.37.25.1 +33.53.51.0
#   1    2     VLA:N9   25.0 m  -107.37.07.8 +33.54.19.0
#   2    3     VLA:N3   25.0 m  -107.37.06.3 +33.54.04.8
#   3    4     VLA:N5   25.0 m  -107.37.06.7 +33.54.08.0

```

```

# 4 5 VLA:N2 25.0 m -107.37.06.2 +33.54.03.5
# 5 6 VLA:E1 25.0 m -107.37.05.7 +33.53.59.2
# 6 7 VLA:E2 25.0 m -107.37.04.4 +33.54.01.1
# 7 8 VLA:N8 25.0 m -107.37.07.5 +33.54.15.8
# 8 9 VLA:E8 25.0 m -107.36.48.9 +33.53.55.1
# 9 10 VLA:W3 25.0 m -107.37.08.9 +33.54.00.1
# 10 11 VLA:N1 25.0 m -107.37.06.0 +33.54.01.8
# 11 12 VLA:E6 25.0 m -107.36.55.6 +33.53.57.7
# 12 13 VLA:W7 25.0 m -107.37.18.4 +33.53.54.8
# 13 14 VLA:E4 25.0 m -107.37.00.8 +33.53.59.7
# 14 15 VLA:N7 25.0 m -107.37.07.2 +33.54.12.9
# 15 16 VLA:W4 25.0 m -107.37.10.8 +33.53.59.1
# 16 17 VLA:W5 25.0 m -107.37.13.0 +33.53.57.8
# 17 18 VLA:N6 25.0 m -107.37.06.9 +33.54.10.3
# 18 19 VLA:E7 25.0 m -107.36.52.4 +33.53.56.5
# 19 20 VLA:E9 25.0 m -107.36.45.1 +33.53.53.6
# 21 22 VLA:W8 25.0 m -107.37.21.6 +33.53.53.0
# 22 23 VLA:W6 25.0 m -107.37.15.6 +33.53.56.4
# 23 24 VLA:W1 25.0 m -107.37.05.9 +33.54.00.5
# 24 25 VLA:W2 25.0 m -107.37.07.4 +33.54.00.9
# 25 26 VLA:E5 25.0 m -107.36.58.4 +33.53.58.8
# 26 27 VLA:N4 25.0 m -107.37.06.5 +33.54.06.1
# 27 28 VLA:E3 25.0 m -107.37.02.8 +33.54.00.5

```

```
#
```

```
# Tables:
```

```

# MAIN 2021424 rows
# ANTENNA 28 rows
# DATA_DESCRIPTION 2 rows
# DOPPLER <absent>
# FEED 28 rows
# FIELD 13 rows
# FLAG_CMD <empty>
# FREQ_OFFSET <absent>
# HISTORY 7058 rows
# OBSERVATION 1 row
# POINTING 2604 rows
# POLARIZATION 1 row
# PROCESSOR <empty>
# SOURCE <empty> (see FIELD)
# SPECTRAL_WINDOW 2 rows
# STATE <empty>
# SYSCAL <absent>
# WEATHER <absent>

```

```
#
```

```
#=====
```

```
# Data Examination and Flagging
```

```
#=====
```

```
#
```

```
# Get rid of the autocorrelations from the MS
```

```
#
```

```

print '--Flagautocorr--'

# Don't default this one either

flagautocorr()

#
#=====
#
# Use Flagmanager to save a copy of the flags
#
print '--Flagmanager--'
default('flagmanager')

vis = msfile

# Save a copy of the MAIN table flags

mode = 'save'
versionname = 'flagautocorr'
comment = 'flagged autocorr'
merge = 'replace'

flagmanager()

# If you look in the 'jupiter6cm.usecase.ms.flagversions/'
# you'll see flags.flagautocorr there along with the
# flags.Original that importuvfits made for you
# Or use

mode = 'list'

flagmanager()

# In the logger you will see something like:
#
# MS : /home/sandro2/smyers/Testing2/Aug07/jupiter6cm.usecase.ms
#
# main : working copy in main table
# Original : Original flags at import into CASA
# flagautocorr : flagged autocorr
# See logger for flag versions for this file

#
#=====
#
# Use Plotxy to interactively flag the data
#
print '--Plotxy--'
default('plotxy')

```

```
vis = msfile

# The fields we are interested in: 1331+305,JUPITER,0137+331
selectdata = True

# First we do the primary calibrator
field = '1331+305'

# Plot only the RR and LL for now
correlation = 'RR LL'

# Plot amplitude vs. uvdist
xaxis = 'uvdist'
yaxis = 'amp'
multicolor = 'both'

# The easiest thing is to iterate over antennas
iteration = 'antenna'

plotxy()

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

# You'll see lots of low points as you step through RR LL RL LR
# A basic clip at 0.75 for RR LL and 0.055 for RL LR will work
# If you want to do this interactively, set
iteration = ''

plotxy()

# You can also use flagdata to do this non-interactively
# (see below)

# Now look at the cross-polar products
correlation = 'RL LR'

plotxy()

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

#-----
# Now do calibrator 0137+331
field = '0137+331'
correlation = 'RR LL'
xaxis = 'uvdist'
spw = ''
iteration = ''
```



```

antenna = ''

plotxy()

# You'll see a bunch of bad data along the bottom near zero amp
# Draw a box around some of it and use Locate
# Looks like much of it is Antenna 9 (ID=8) in spw=1

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

xaxis = 'time'
spw = '1'
correlation = ''

# Note that the strings like antenna='9' first try to match the
# NAME which we see in listobs was the number '9' for ID=8.
# So be careful here (why naming antennas as numbers is bad).
antenna = '9'

plotxy()

# YES! the last 4 scans are bad. Box 'em and flag.

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

# Go back and clean up
xaxis = 'uvdist'
spw = ''
antenna = ''
correlation = 'RR LL'

plotxy()

# Box up the bad low points (basically a clip below 0.52) and flag

# Note that RL,LR are too weak to clip on.

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

#-----
# Finally, do JUPITER
field = 'JUPITER'
correlation = ''
iteration = ''
xaxis = 'time'

```

```
plotxy()

# Here you will see that the final scan at 22:00:00 UT is bad
# Draw a box around it and flag it!

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

# Now look at whats left
correlation = 'RR LL'
xaxis = 'uvdist'
spw = '1'
antenna = ''
iteration = 'antenna'

plotxy()

# As you step through, you will see that Antenna 9 (ID=8) is often
# bad in this spw. If you box and do Locate (or remember from
# 0137+331) its probably a bad time.

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

# The easiset way to kill it:

antenna = '9'
iteration = ''
xaxis = 'time'
correlation = ''

plotxy()

# Draw a box around all points in the last bad scans and flag 'em!

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

# Now clean up the rest
xaxis = 'uvdist'
correlation = 'RR LL'
antenna = ''
spw = ''

# You will be drawing many tiny boxes, so remember you can
# use the ESC key to get rid of the most recent box if you
# make a mistake.
```

```

plotxy()

# Note that the end result is we've flagged lots of points
# in RR and LL. We will rely upon imager to ignore the
# RL LR for points with RR LL flagged!

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

#
#=====
#
# Use Flagmanager to save a copy of the flags so far
#
print '--Flagmanager--'
default('flagmanager')

vis = msfile
mode = 'save'
versionname = 'xyflags'
comment = 'Plotxy flags'
merge = 'replace'

flagmanager()

#
#=====
#
# You can use Flagdata to explicitly clip the data also
#
print '--Flagdata--'
default('flagdata')

vis = msfile

# Set some clipping regions
mode = 'manualflag'
clipcolumn = 'DATA'
clipoutside = False

# Clip calibraters
field = '1331+305'
clipexpr = 'ABS RR'
clipminmax = [0.0,0.75]
flagdata()

clipexpr = 'ABS LL'
clipminmax = [0.0,0.75]

```

```

flagdata()

clipexpr = 'ABS RL'
clipminmax = [0.0,0.055]
flagdata()

clipexpr = 'ABS LR'
clipminmax = [0.0,0.055]
flagdata()

field = '0137+331'
clipexpr = 'ABS RR'
clipminmax = [0.0,0.55]
flagdata()

clipexpr = 'ABS LL'
clipminmax = [0.0,0.55]
flagdata()

# You can also do the antenna edits on 0137+331 and JUPITER
# with flagdata

#
#=====
# Calibration
#=====
#
# Set the fluxes of the primary calibrator(s)
#
print '--Setjy--'
default('setjy')

vis = msfile

#
# 1331+305 = 3C286 is our primary calibrator
field = '1331+305'

# Setjy knows about this source so we dont need anything more

setjy()

#
# You should see something like this in the logger and casapy.log file:
#
# 1331+305 spwid= 0 [I=7.462, Q=0, U=0, V=0] Jy, (Perley-Taylor 99)
# 1331+305 spwid= 1 [I=7.51, Q=0, U=0, V=0] Jy, (Perley-Taylor 99)
#
#
#=====

```

```

#
# Initial gain calibration
#
print '--Gaincal--'
default('gaincal')

vis = msfile

# set the name for the output gain caltable
gtable = prefix + '.gcal'
caltable = gtable

# Gain calibrators are 1331+305 and 0137+331 (FIELD_ID 7 and 0)
# We have 2 IFs (SPW 0,1) with one channel each

# selection is via the field and spw strings
field = '1331+305,0137+331'
spw = ''

# a-priori calibration application
# atmospheric optical depth (turn off)
gaincurve = True
#opacity = False
#tau=0.0
opacity = 0.0

# scan-based G solutions for both amplitude and phase
gaintype = 'G'
solint = 0.
calmode = 'ap'

# reference antenna 11 (11=VLA:N1)
refant = '11'

# minimum SNR 3
minsnr = 3

gaincal()

#
#=====
#
# Bootstrap flux scale
#
print '--Fluxscale--'
default('fluxscale')

vis = msfile

# set the name for the output rescaled caltable
ftable = prefix + '.fluxscale'

```

```

fluxtable = ftable

# point to our first gain cal table
caltable = gtable

# we will be using 1331+305 (the source we did setjy on) as
# our flux standard reference
reference = '1331+305'

# we want to transfer the flux to our other gain cal source 0137+331
# to bring its gain amplitues in line with the absolute scale
transfer = '0137+331'

fluxscale()

# You should see in the logger something like:
#Flux density for 0137+331 in SpW=0 is: 5.42575 +/- 0.00285011 (SNR = 1903.7, nAnt= 27)
#Flux density for 0137+331 in SpW=1 is: 5.46569 +/- 0.00301326 (SNR = 1813.88, nAnt= 27)

#=====
#
# Interpolate the gains onto Jupiter (and others)
#
print '--Accum--'
default('accum')

vis = msfile

tablein = ''
incrtable = ftable
calfield = '1331+305, 0137+331'

# set the name for the output interpolated caltable
atable = prefix + '.accum'
caltable = atable

# linear interpolation
interp = 'linear'

# make 10s entries
accumtime = 10.0

accum()

#=====
#
# Correct the data
# (This will put calibrated data into the CORRECTED_DATA column)
#
print '--ApplyCal--'
default('applycal')

```

```

vis = msfile

# Start with the interpolated fluxscale/gain table
bptable = ''
gaintable = atable

# Since we did gaincurve=True in gaincal, we need it here also
gaincurve = True
#opacity = False
#tau=0.0
opacity=0.0

# select the fields
field = '1331+305,0137+331,JUPITER'
spw = ''
selectdata = False

# do not need to select subset since we did accum
# (note that correct only does 'nearest' interp)
gainselect = ''

applycal()

#
#=====
#
# Now split the Jupiter target data
#
print '--Split Jupiter--'
default('split')

vis = msfile

# Now we write out the corrected data for the calibrator

# Make an output vis file
srcsplitms = prefix + '.split.ms'
outputvis = srcsplitms

# Select the Jupiter field
field = 'JUPITER'
spw = ''

# pick off the CORRECTED_DATA column
datacolumn = 'corrected'

split()

#
#=====

```

```

# FIRST CLEAN / SELFCAL CYCLE
#=====
#
# Now clean an image of Jupiter
#
print '--Clean 1--'
default('clean')

# Pick up our split source data
vis = srcsplitms

# Make an image root file name
imname1 = prefix + '.clean1'
imagenname = imname1

# Set up the output continuum image (single plane mfs)
mode = 'mfs'
stokes = 'I'

# NOTE: current version field='' doesnt work
field = '*'

# Combine all spw
spw = ''

# This is D-config VLA 6cm (4.85GHz) obs
# Check the observational status summary
# Primary beam FWHM = 45'/f_GHz = 557"
# Synthesized beam FWHM = 14"
# RMS in 10min (600s) = 0.06 mJy (thats now, but close enough)

# Set the output image size and cell size (arcsec)
# 4" will give 3.5x oversampling
# 280 pix will cover to 2xPrimaryBeam
# clean will say to use 288 (a composite integer) for efficiency
clnalg = 'clark'
clnimsize = [288,288]

# double for CS Clean
#clnalg = 'csclean'
#clnimsize = [576,576]

clncell = [4.,4.]

alg = clnalg
imsize = clnimsize
cell = clncell

# NOTE: will eventually have an imadvise task to give you this
# information

```



```

# Standard gain factor 0.1
gain = 0.1

# Fix maximum number of iterations
niter = 10000

# Also set flux residual threshold (0.04 mJy)
# From our listobs:
# Total integration time = 85133.2 seconds
# With rms of 0.06 mJy in 600s ==> rms = 0.005 mJy
# Set to 10x thermal rms
threshold=0.05

# Note - we can change niter and threshold interactively
# during clean

# Set up the weighting
# Use Briggs weighting (a moderate value, on the uniform side)
weighting = 'briggs'
rmode = 'norm'
robust = 0.5

# No clean mask
mask = ''

# Use interactive clean mode
cleanbox = 'interactive'

# Moderate number of iter per interactive cycle
npercycle = 100

clean()

# When the interactive clean window comes up, use the right-mouse
# to draw rectangles around obvious emission double-right-clicking
# inside them to add to the flag region. You can also assign the
# right-mouse to polygon region drawing by right-clicking on the
# polygon drawing icon in the toolbar. When you are happy with
# the region, click 'Done Flagging' and it will go and clean another
# 100 iterations. When done, click 'Stop'.

# Set up variables
clnimage1 = imname1+'.image'
clnmodel1 = imname1+'.model'
clnresid1 = imname1+'.residual'
clnmask1 = imname1+'.clean_interactive.mask'

#
#-----
#
# Look at this in viewer

```

```
viewer(clnimage1,'image')

# You can use the right-mouse to draw a box in the lower right
# corner of the image away from emission, the double-click inside
# to bring up statistics. Use the right-mouse to grab this box
# and move it up over Jupiter and double-click again. You should
# see stuff like this in the terminal:
#
# jupiter6cm.usecase.clean1.image      (Jy/beam)
#
# n          Std Dev      RMS          Mean          Variance      Sum
# 4712       0.003914     0.003927   0.0003205     1.532e-05     1.510
#
# Flux       Med |Dev|     IntQtlRng   Median         Min           Max
# 0.09417    0.002646     0.005294   0.0001885     -0.01125     0.01503
#
#
# On Jupiter:
#
# n          Std Dev      RMS          Mean          Variance      Sum
# 3640       0.1007       0.1027     0.02023       0.01015       73.63
#
# Flux       Med |Dev|     IntQtlRng   Median         Min           Max
# 4.592      0.003239     0.007120   0.0001329     -0.01396     1.060
#
# Estimated dynamic range = 1.060 / 0.003927 = 270 (poor)
#
# Note that the exact numbers you get will depend on how deep you
# take the interactive clean and how you draw the box for the stats.
#
#-----
#
# Self-cal using clean model
#
# Note: clean will have left FT of model in the MODEL_DATA column
# If you've done something in between, can use the ft task to
# do this manually.
#
print '--SelfCal 1--'
default('gaincal')

vis = srcsplitms

# New gain table
selfcaltab1 = srcsplitms + '.selfcal1'
caltab1 = selfcaltab1

# Don't need a-priori calcs
selectdata = False
gaincurve = False
opacity = 0.0
```

```

# This choice seemed to work
refant = '11'

# Lets do phase-only first time around
gaintype = 'G'
calmode = 'p'

# Do scan-based solutions with SNR>3
solint = 0.0
minsnr = 3.0

# Do not need to normalize (let gains float)
solnorm = False

gaincal()

#
#-----
#
# Correct the data (no need for interpolation this stage)
#
print '--ApplyCal--'
default('applycal')

vis = srcsplitms

gaintable = selfcaltab1

gaincurve = False
opacity = 0.0
field = ''
spw = ''
selectdata = False

calwt = True

applycal()

# Self-cal is now in CORRECTED_DATA column of split ms
#
#=====
# SECOND CLEAN / SELFCAL CYCLE
#=====
#
print '--Clean 2--'
default('clean')

vis = srcsplitms

inname2 = prefix + '.clean2'

```

```

imagenname = imname2

field = '*'
spw = ''
mode = 'mfs'
gain = 0.1
niter = 10000
threshold=0.04

alg = clnalg
imsize = clnimsize
cell = clncell

weighting = 'briggs'
rmode = 'norm'
robust = 0.5

cleanbox = 'interactive'
npercycle = 100

clean()

# Set up variables
clnimage2 = imname2+'.image'
clnmodel2 = imname2+'.model'
clnresid2 = imname2+'.residual'
clnmask2 = imname2+'.clean_interactive.mask'

#
#-----
#
# Look at this in viewer
viewer(clnimage2,'image')

# jupiter6cm.usecase.clean2.image      (Jy/beam)
#
# n          Std Dev      RMS          Mean          Variance      Sum
# 5236       0.001389      0.001390     3.244e-05     1.930e-06     0.1699
#
# Flux       Med |Dev|     IntQtlRng    Median         Min           Max
# 0.01060    0.0009064    0.001823    -1.884e-05    -0.004015    0.004892
#
#
# On Jupiter:
#
# n          Std Dev      RMS          Mean          Variance      Sum
# 5304       0.08512       0.08629     0.01418       0.007245     75.21
#
# Flux       Med |Dev|     IntQtlRng    Median         Min           Max
# 4.695      0.0008142    0.001657    0.0001557     -0.004526    1.076
#

```

```

# Estimated dynamic range = 1.076 / 0.001389 = 775 (better)
#
# Note that the exact numbers you get will depend on how deep you
# take the interactive clean and how you draw the box for the stats.
#
#-----
#
# Next self-cal cycle
#
print '--SelfCal 2--'
default('gaincal')

vis = srcsplitms

selfcaltab2 = srcsplitms + '.selfcal2'
caltable = selfcaltab2

selectdata = False
gaincurve = False
opacity = 0.0
refant = '11'

# This time amp+phase on 10s timescales SNR>1
gaintype = 'G'
calmode = 'ap'
solint = 10.0
minsnr = 1.0
solnorm = False

gaincal()

#
# It is useful to put this up in plotcal
#
#-----
#
print '--PlotCal--'
default('plotcal')

tablein = selfcaltab2
multiplot = True
yaxis = 'amp'

plotcal()

# Use the Next button to iterate over antennas

# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

```

```

yaxis = 'phase'

plotcal()

#
# You can see it is not too noisy.
# Lets do some smoothing anyway.
#
#-----
#
# Smooth calibration solutions
#
print '--Smooth--'
default('smoothcal')

vis = srcsplitms

tablein = selfcaltab2

smoothcaltab2 = srcsplitms + '.smoothcal2'
caltable = smoothcaltab2

# Do a 30s boxcar average
smoothtype = 'mean'
smoothtime = 30.0

smoothcal()

# If you put into plotcal you'll see the results
# For example, you can grap the inputs from the last
# time you ran plotcal, set the new tablename, and plot!
#run plotcal.last
#tablein = smoothcaltab2
#plotcal()

#
#-----
#
# Correct the data
#
print '--ApplyCal--'
default('applycal')

vis = srcsplitms

gaintable = smoothcaltab2

gaincurve = False
opacity = 0.0
field = ''
spw = ''

```

```

selectdata = False
calwt = True

applycal()

#
#=====
# THIRD CLEAN / SELFCAL CYCLE
#=====
#
print '--Clean 3--'
default('clean')

vis = srcsplitms

imname3 = prefix + '.clean3'
imagenname = imname3

field = '*'
spw = ''
mode = 'mfs'
gain = 0.1
niter = 10000
threshold=0.04

alg = clnalg
imsize = clnimsize
cell = clncell

weighting = 'briggs'
rmode = 'norm'
robust = 0.5

cleanbox = 'interactive'
npercycle = 100

clean()

# Cleans alot deeper
# You can change the npercycle to larger numbers
# (like 250 or so) as you get deeper also.

# Set up variables
clnimage3 = imname3+'.image'
clnmodel3 = imname3+'.model'
clnresid3 = imname3+'.residual'
clnmask3 = imname3+'.clean_interactive.mask'

#
#-----
#

```

```

# Look at this in viewer
viewer(clnimage3,'image')

# jupiter6cm.usecase.clean3.image      (Jy/beam)
#
# n          Std Dev      RMS          Mean          Variance      Sum
# 5848       0.001015     0.001015    -4.036e-06    1.029e-06    -0.02360
#
# Flux       Med |Dev|     IntQtlRng    Median         Min           Max
# -0.001470  0.0006728  0.001347    8.245e-06     -0.003260    0.003542
#
#
# On Jupiter:
#
# n          Std Dev      RMS          Mean          Variance      Sum
# 6003       0.08012     0.08107     0.01245       0.006419     74.72
#
# Flux       Med |Dev|     IntQtlRng    Median         Min           Max
# 4.653      0.0006676  0.001383    -1.892e-06    -0.002842    1.076
#
# Estimated dynamic range = 1.076 / 0.001015 = 1060 (even better!)
#
# Note that the exact numbers you get will depend on how deep you
# take the interactive clean and how you draw the box for the stats.
#
# Greg Taylor got 1600:1 so we still have some ways to go
# This will probably take several more careful self-cal cycles.

# Set up final variables
clnimage = clnimage3
clnmodel = clnmodel3
clnresid = clnresid3
clnmask  = clnmask3

#
#=====
# Image Analysis
#=====
#
# Can do some image statistics if you wish
# Treat this like a regression script
# WARNING: currently requires toolkit
#
print ' Jupiter results '
print ' ===== '

print ''
# Pull the max src amp value out of the MS
ms.open(srcsplitms)
thistest_src = max(ms.range(["amplitude"]).get('amplitude'))
oldtest_src = 4.92000198364

```



```

print ' MS max amplitude should be ',oldtest_src
print ' Found : Max in MS = ',thistest_src
diff_src = abs((oldtest_src-thistest_src)/oldtest_src)
print ' Difference (fractional) = ',diff_src

ms.close()

print ''
# Pull the max and rms from the clean image
ia.open(clnimage)
on_statistics=ia.statistics()
thistest_immax=on_statistics['max'][0]
oldtest_immax = 1.07732224464
print ' Clean image ON-SRC max should be ',oldtest_immax
print ' Found : Max in image = ',thistest_immax
diff_immax = abs((oldtest_immax-thistest_immax)/oldtest_immax)
print ' Difference (fractional) = ',diff_immax

print ''
# Now do stats in the lower right corner of the image
box = ia.setboxregion([0.75,0.00],[1.00,0.25],frac=true)
off_statistics=ia.statistics(region=box)
thistest_imrms=off_statistics['rms'][0]
oldtest_imrms = 0.0010449
print ' Clean image OFF-SRC rms should be ',oldtest_imrms
print ' Found : rms in image = ',thistest_imrms
diff_imrms = abs((oldtest_imrms-thistest_imrms)/oldtest_imrms)
print ' Difference (fractional) = ',diff_imrms

print ''
print ' Final Clean image Dynamic Range = ',thistest_immax/thistest_imrms
print ''
print ' ===== '

ia.close()

print ''
print '--- Done ---'

#
#=====

```

Appendix D

CASA Dictionaries

D.1 AIPS – CASA dictionary

Please see:

- <https://wikio.nrao.edu/bin/view/Software/CASA-AIPSDictionary>

D.2 MIRIAD – CASA dictionary

Table D.1 provides a list of common Miriad tasks, and their equivalent CASA tool or tool function names. The two packages differ in both their architecture and calibration and imaging models, and there is often not a direct correspondence. However, this index does provide a scientific user of CASA who is familiar with MIRIAD, with a simple translation table to map their existing data reduction knowledge to the new package.

D.3 CLIC – CASA dictionary

Table D.2 provides a list of common CLIC tasks, and their equivalent CASA tool or tool function names. The two packages are very similar since the CASA software to reduce IRAM data is based on the CLIC reduction procedures.

Table D.1: MIRIAD – CASA dictionary

MIRIAD Task	Description	CASA tool/function
atlod	load ATCA data	atcafiller
blflag	Interactive baseline based editor/flagger	mp raster displays
cgcurs	Interactive image analysis	qtvviewer
cgdisp	Image display, overlays	qtvviewer
clean	Clean an image	im
fits	FITS image filler	ia.imagefromfits
gpboot	Set flux density scale	cb.fluxscale
gpcal	Polarization leakage and gain calibration	cb with 'G' and 'D'
gpcopy	copy calibration tables	<i>not needed</i>
gpplt	Plot calibration solutions	cp.plot
imcomb	Image combination	im
imfit	Image-plane component fitter	ia.imagefitter
impol	Create polarization images	ia.imagepol
imstat	Image statistics	ia.statistics
imsub	Extract sub-image	ia.subimage
invert	Synthesis imaging	im
linmos	linear mosaic combination of images	im
maths	Calculations involving images	ia.imagecalc, ia.calc
mfcalf	Bandpass and gain calibration	cb with 'G' and 'B'
prthd	Print header of image or uvdata	ia.summary, ms.summary
restor	Restore a clean component model	im
selfcal	selfcalibration of visibility data	im, cb
tvclip	automated flagging based on clip levels	af
tvdisp	Load image to TV display	qtvviewer
tvflag	Interactive TB data editing	mp
uvaver	Average/select data, apply calibration	ms.split
uvfit	uv-plane component fitter	cb
uvflag	Command-based flagging	af
uvgen	Simulator	sm
uvlist	List uv-data	tb
uvmodel	Source model computation	im.ft
uvplt	uv-data plotting	ms
uvsplit	split uv file in sources and spectral windows	ms.split

Table D.2: CLIC–CASA dictionary

CLIC Function	Description	CASA tool/function
load	Load data	almatifiller
print	Print text summary of data	ms.summary
flag	Flag data	mp, af, qtvviewer
phcor	Atmospheric phase correction	almatifiller
rf	Radio frequency bandpass	cb.setsolvebandpoly, cb.solve
phase	Phase calibration	cb.setsolvegainspline,cb.solve
flux	Absolute flux calibration	cb.fluxscale
ampl	Amplitude calibration	cb.setsolvegainspline,cb.solve
table	Split out calibrated data (uv table)	ms.split