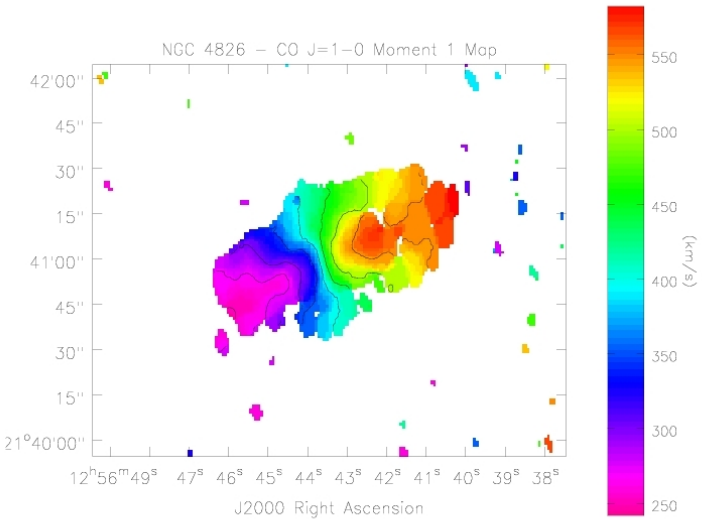
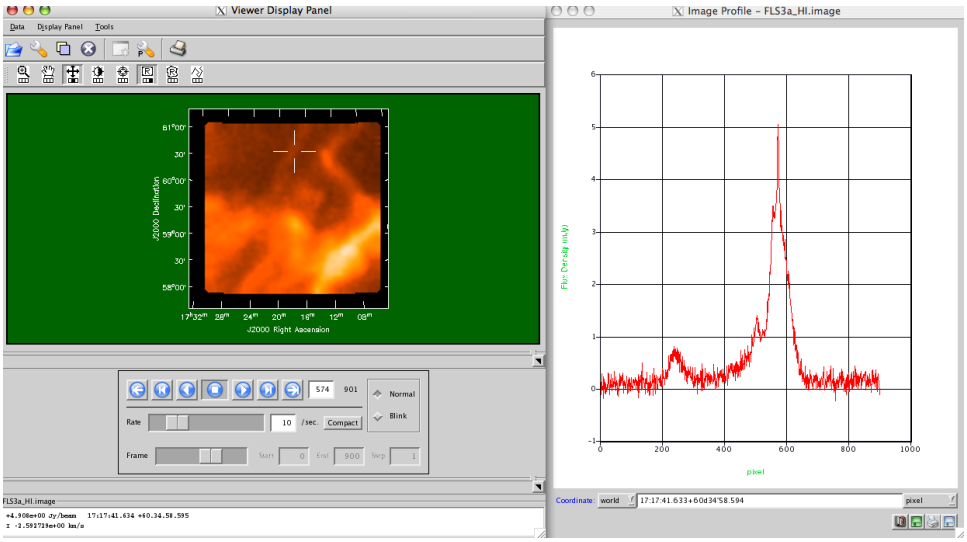
# CASA Synthesis & Single Dish Reduction Cookbook

## Beta Release Edition




**Version: November 10, 2007**

# CASA Synthesis & Single Dish Reduction Cookbook

## Beta Release Edition

**Chef Editor:** Steven T. Myers
CASA Project Scientist

**Sous-Chef:** Joe McMullin
CASA Project Manager

CASA Synthesis & Single Dish Reduction Cookbook, Beta Release Edition,
Version November 10, 2007,


## This tome was scribed by:

The CASA Developers and the NRAO Applications User Group (NAUG)
`http://casa.nrao.edu/`
`http://www.aoc.nrao.edu/~smyers/naug/`

Do you dare to enter CASA Stadium and join battle with the Ironic Chefs?
Let us see who's cuisine reigns supreme ...

# Contents

# List of Tables

# List of Figures

# 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.

You have in your hands the **Beta Release** of CASA. This means that there are a number of caveats and limitations for the use of this package. See § 1.1 below for more information, and pay heed to the numerous **BETA ALERT**s placed throughout this cookbook. You can expect regular updates and patches, as well as increasing functionality. But you can also expect interface changes. The goals of this Beta Release are to get the package out into the hands of real users so you can take it for a spin. Please knock it about a bit, but remember it is not a polished, finished product. Beware!

This cookbook is a task-based walk-through 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 the **CASA User Reference Manual** 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.

**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.

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

**BETA ALERT:** For the Beta Release, there are also special chapters in the Appendix on

- single dish data analysis (Chapter A), and
- simulation (Chapter B).

These are included for users that will be doing EVLA and ALMA telescope commissioning and software development. They will become part of the main cookbook in later releases.

The general 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 C),
- more details about Python and CASA (Appendix D),
- a discussion of the Hamaker-Bregman-Sault Measurement Equation (Appendix E),
- annotated scripts for typical data reduction cases (Appendix F), and
- CASA dictionaries to AIPS, MIRIAD, and CLIC (Appendix G).

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 User Reference Manual** — details on a specific task or tool does and how to use it. **BETA 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. You will also find information on how to obtain the latest release and receive user support.

## 1.1 Limitations of This Beta Release

Currently, CASA is in the **Beta Release** stage. This means that much, but not all, of the eventual functionality is available. Furthermore, the package is still under 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 Beta release 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 **Beta Alert** boxes, or in **BETA ALERT** notes in the text.

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

Not only is the software in Beta Release, but this cookbook is also a living document. You can expect this document, as well as other on-line and in-line user support guides, to be updated regularly. Also, feel free to send us comments and suggestions on the contents of our documentation.

Please check the CASA Home page (`http://casa.nrao.edu`) regularly to look for updates to the release and to the documentation, and to check the list of known problems. You can find the contact information for feedback here also.

We also note here that we are also in the process of commissioning our User Support system for CASA. Thus, we can only support a limited number of official Beta Release Users at this time. See the CASA Home Page for more information on the policies and conditions on obtaining and getting support for this Beta Release.

## 1.2 CASA Basics — Information for First-Time Users

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

### 1.2.1 Before Starting CASA

If you have done a default installation under Linux using rpms, or on the Mac with the CASA application, then there should be a `sh` script called `casapy` in the `/usr/bin` area which is in your path. This shell will set up its environment and run the version of `casapy` that it points to. If this is how you set up the system, then you need to nothing further and can run `casapy`.

On some systems, particularly if you have multiple versions installed, 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. For a default installation this will likely be in `/usr/bin`. Sometimes, you will have multiple versions (for example, various released versions). For example at the NRAO AOC, the "stable build" is in `/home/casa`.

For example, if you have done a default installation under Linux or OSX, then:

```
In bash:
> . /usr/bin/casainit.sh
or for csh:
> source /usr/bin/casainit.csh
```

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

**BETA ALERT:** If you want to run the `qcasabrowser` (see § 3.6) outside of the `casapy` shell, then you will need to put the CASA root in your path using one of the above mechanisms.

#### 1.2.1.1 Environment Variables

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 § 1.2.7.3). 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 § 1.2.7.3 for more information on these choices. We recommend using the `cat` option for most users, as this works smoothly both interactively and in scripts.

BETA ALERT: There is currently no way within CASA to change these environment variables.

#### 1.2.1.2 Where is CASA?

Note that the path to the CASA installation, which contains the scripts and data repository, will also depend upon the installation. With a default installation under Linux this will probably be in

```
/usr/lib/casapy/
```

while in a Mac OSX default install it will likely be

```
/opt/casa/
```

You can find the location after initialized by looking at the `AIPSPATH` environment variable. You can find it within `casapy` by

```
pathname=os.environ.get('AIPSPATH').split()[0]
print pathname
```

### 1.2.2 Starting CASA

After having run the appropriate casainit script, CASA is started by typing
`casapy`
on the UNIX command line, e.g.

```
casapy
```

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. You will also see a `logger` GUI appear on your Desktop (usually near the upper left).

You also have the option of starting CASA without the `logger`, for example if you are running remotely in a terminal window without an X11 connection, or if you just do not want to see the logger GUI. In this case use the `--nolog` option:

```
casapy --nolog
```

See § 1.4.2 for more on this, and the logger in general.

### 1.2.3 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.

### 1.2.4 What happens if something goes wrong?

**BETA ALERT:** This is a Beta Release, and there are still ways to cause CASA to crash. Please check the CASA Home Page for Beta Release information including a list of known problems. If you think you have encountered an unknown problem, please consult the CASA HelpDesk (contact information on the CASA Home Page). See also the caveats to this Beta Release (§ 1.1 for pointers to our policy on User Support.

First, always check that your inputs are correct; use the

```
help <taskname>
```

(§ 1.2.7.2) or

```
help par.<parameter\_name>
```

(§ 1.2.7.4) to review the inputs/output.

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

If the problem causes CASA to crash, see the next sub-section.

### 1.2.5 What happens if CASA crashes?

Usually, restarting `casapy` is sufficient to get you going again after a crash takes you out of the Python interface. Note that there may be spawned subprocesses still running, such as the `casaviewer` or the `logger`. These can be dismissed manually in the usual manner. After a crash, there may also be hidden processes. You can find these by listing processes, e.g. in linux:

```
ps -elf | grep casa
```

or on MacOSX (or other BSD Unix):

```
ps -aux | grep casa
```

You can then kill these, for example using the Unix `kill` or `killall` commands. This may be necessary if you are running remotely using `ssh`, as you cannot logout until all your background processes are terminated. For example,

```
killall ipcontroller
```

or

```
killall Python
```

will terminate the most common post-crash zombies.

### 1.2.6 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 D.

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

#### 1.2.6.1 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 § 1.2.7.1) should help you avoid this.

#### 1.2.6.2 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 D.3 for more information.

#### 1.2.6.3 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 be 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 be indexed as `field='0'` in most tasks, as well as by name `field='0137+331'` (assuming thats the name of the first field). You will see these indices in the MS summary from the task `listobs`.

#### 1.2.6.4 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 D.2 for more information.

#### 1.2.6.5 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 D.4 for more information.

#### 1.2.6.6 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 § 1.3.3.7).

See Appendix D.9 for more information.

### 1.2.7 Getting Help in CASA

#### 1.2.7.1 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
```

#### 1.2.7.2 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, splitd
    Continuum fitting and subtraction in the uv plane:

    A polynomial of the desired order is fit across the specified
    channels that define the continuum emission.  The data may be
    averaged in time to increase the signal to noise.  This fit
    represents a model of the continuum in all channels.

    For fitmode='subtract', the fitted continuum spectrum is
    subtracted from all channels and the result (presumably only
    line emission) is stored in the CORRECTED_DATA.  The
    continuum fit is stored in the MODEL_DATA.

    For fitmode='model' the continuum model is stored in the
    MODEL_DATA; but the CORRECTED_DATA is unaffected.

    For fitmode='replace' the continuum model is stored in
```

```
the CORRECTED_DATA; this is useful to image the continuum model
result.

Keyword arguments:
vis -- Name of input visibility file
        default: none; example: vis='ngc5921.ms'
field -- Field selection
        default: field = '' means select all fieldsx
        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'
        field = '13*' will retrieve '1328+307' and any other fields
           beginning with '13'
spw -- Spectral selection
        default: spw='', means select all spws;
        example: spw=1
channels -- List of channels to fit for the continuum
        default: (all); example: channels=range(4,7)+range(50,60)
solint -- Averaging time for per-baseline fit (seconds)
        default: 0.0 (scan-based averaging for fit); example: solint=10
fitorder -- Polynomial order for the fit of the continuum
        default: 0 (constant); example: fitorder=1
fitmode -- Use of the continuum fit model
        default: 'subtract'; example: fitmode='replace'
        Options:
        'subtract'-store fitted continuum model in MODEL and
           subtract this continuum from data in CORRECTED to
           produce line-emission in CORRECTED.
        'model'-store fit continuum model in MODEL, but
           do not change data in CORRECTED.
        'replace'-replace CORRECTED with continuum mode fit.
splitdata -- Split out continuum and continuum subtracted line data
        default: 'False'; example: splitdata=True
        The continuum data will be placed in: vis.cont
        The continuum subtracted data will be placed in: vis.contsub
async -- Run task in a separate process (return CASA prompt)
        default: False; example: async=True
```

#### 1.2.7.3 `help` and `PAGER`

Your `PAGER` environment variable (§ 1.2.1) 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.

#### 1.2.7.4 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)
```

#### 1.2.7.5 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.

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

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 D. **BETA ALERT:** The User Reference Manual currently covers only tools, not tasks.

## 1.3 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.

### 1.3.1 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 § 1.2.7.2, `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`, e.g.

```
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         makemask                        browsetable
fluxscale       mosaic                          clearplot
(fringecal)                                     clearstat
clearcal                                        concat
listcal                                         filecatalog
smoothcal                                       startup
                                                split


Image Analysis  Simulation
--------------  ----------
imhead          (almasimmos)
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)      : ATF: Calculate a baseline-based calibration solution (prototype)
browsetable  : Browse a visibility data set or calibration table
casalogger   : FUNCTION - invoke to call up the logger gui
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
find         : Find a string in the task help
```

```
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 calibraters
(fringecal)  : ATF:Calculate a baseline-based fringe-fitting soln (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) : ATF:Convert an ALMA Science Data Model directory to a CASA 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
regridimage  : Grid image to same shape and coordinates as template
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)
tget         : Recover/set parameters for a specified task
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. For example,

```
CASA <6>: startup()
--------------------------------------------------------------
Available tasks:

   accum        exportuvfits    immoments       plotcal
   applycal     feather         importfits      plotxy
   bandpass     filecatalog     importuvfits    regridimage
   browsetable  find            importvla       setjy
   clean        flagautocorr    invert          smoothcal
   clearcal     flagdata        listcal         split
   clearplot    flagmanager     listhistory     tget
   clearstat    fluxscale       listobs         uvcontsub
   concat       ft              makemask        uvmodelfit
   deconvolve   gaincal         mosaic          viewer
   exportfits   imhead          plotants

Additional tasks are available for ALMA commissioning use
(still alpha code as of Beta 0 release):
```

```
   almasimmos   blcal           fringecal       importasdm

Available tools:

   cb (calibrater)     cp (cal plot)     fg (flagger)
   ia (image analysis) im (imager)       me (measures)
   mp (MS plot)        ms (MS)           qa (quanta)
   sm (simulation      tb (table)        tp (table plot)

   pl (pylab functions)
   sd (ASAP functions - run asap_init() to import into CASA)

   casalogger             - Call up the casalogger (if it goes away)

------------------------------------------------------------------

Help :
   help taskname          - Full help for task
   help par.parametername - Full help for parameter name
   find string            - Find occurances of string in doc
   tasklist               - Task list organized by catagory
   taskhelp               - One line summary of available tasks
   toolhelp               - One line summary of available tools
   startup                - The start up screen

------------------------------------------------------------------
```

### 1.3.2 Running Tasks Asynchronously

By default, most tasks run synchronously in the foreground. Many tasks, particularly those that can take a long time to execute, have the `async` parameter. This allows the user to send the task to the background for execution.

**BETA ALERT:** A few tasks, such as the `exportuvfits` and `exportfits` tasks, have `async=True` by default. This is a workaround for a known problem where they can trample on other tasks and tools if they use the default global tools underneath.

#### 1.3.2.1 Monitoring Asynchronous Tasks

**BETA ALERT:** Currently, this is only available with the `tm` tool. We are working on a `taskmanager` task.

There is a "taskmanager" tool `tm` that allows the user to retrieve the status of, and to abort the execution of, tasks running with `async=True` in the background. There are two methods of interest for the user, `tm.retrieve` and `tm.abort`.

**BETA ALERT:**
You should not use the `go` command to run a task asynchronously, as the "handle" will be swallowed by the Python task wrapper and you will not be able to access it with `tm`. This is also true if you run in a Python script.

If you run a task with `async=True` then several things will happen. First of all, the task returns a "handle" that is a number used to identify the process. This is printed to the screen, e.g.

```
CASA <5>: inp()
#  mosaic :: Calculate a multi-field deconvolved image with selected clean algorithm:
...
async               =       True        #   if True run in the background, prompt is freed

CASA <6>: mosaic()
Connecting to controller:  ('127.0.0.1', 60775)
  Out[6]: 0
```

where the output value `0` is the handle id.

You can also catch the return value in a variable, e.g.

```
CASA <7>: handle = mosaic()
...
CASA <8>: print handle
1
```

You should also see the usual messages from the task in the `logger`, with some extra lines of information

```
################################################
###  Begin Task: mosaic  ###
Tue Oct 2 17:58:16 2007    NORMAL ::mosaic:
""
"Use: "
tm.abort(return_value)    # to abort the asynchronous task
tm.retrieve(return_value) # to retrieve the status
""
... usual messages here ...

###  End Task: mosaic  ###
################################################
""
```

for the example above.

To show the current state of an asynchronous task, use the `tm.retrieve` method using the handle id as the argument. For example,

```
CASA <9>: tm.retrieve(handle)
  Out[9]: {'result': None, 'state': 'pending'}
```

or

```
CASA <10>: tm.retrieve(1)
  Out[10]: {'result': None, 'state': 'pending'}
```

which means its still running. You should be seeing output in the `logger` also while the task is running.

When a task is finished, you will see:

```
CASA <11>: tm.retrieve(1)
  Out[11]: {'result': None, 'state': 'done'}
```

which indicates completion.

To abort a task while it is running in the background, use the `tm.abort` method, again with the task handle id as the argument. For example,

```
CASA <12>: handle = mosaic()
...
CASA <13>: tm.abort(handle)
```

will abort the task if it is running.

### 1.3.3 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.

**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.

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'
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,uvdis
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)
xlabels             =         ''        #   Label for x-axis
ylabels             =         ''        #   Label for y-axis
fontsize            =       10.0        #   Font size for labels
windowsize          =        1.0        #   Window size
```

See § 1.3.3.3 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).

**Beta 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 (§ 1.3.3.1). 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 § 1.3.3.5) 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 § 1.3.3.7).

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

#### 1.3.3.1 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.

**Important Note:** The `default` command resets the values of the task parameters to a set of "defaults" as specified in the task code. Some defaults are blank strings `''` or empty lists `[]`, others are specific numerical values, strings, or lists. It is important to understand that just setting a string parameter to an empty string `''` is not setting it to its default! Some parameters do not have a blank as an allowed value. See the `help` for a particular task to find out its default. If `''` is the default or an allowed value, it will say so explicitly.

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.

**BETA ALERT:** You currently can only reset ALL of the parameters for a given task to their defaults. In an upcoming update we will allow the `default` command to take a second argument with a specific parameter to default its value.

#### 1.3.3.2 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()
...
```

The `go` command can also be used to launch a different task without changing the current `taskname`, without disrupting the `inp` process on the current task you are working on. For example

```
default 'gaincal' # set current task to gaincal and default
vis = 'n5921.ms'  # set the working ms
...               # set some more parameters
go listobs        # launch listobs w/o chaning current task
inp               # see the inputs for gaincal (not listobs!)
```

**BETA ALERT:** Doing `go listobs(vis='foo.ms')` will currently change the `taskname`, and will change `vis`, which might not be what is desired.

#### 1.3.3.3 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')
#  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]; symmetric for single value
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
phasecenter    =         ''   #   Field Identifier or direction of the image phase center
spw            =         ''   #   spectral window:channels: ''=>all
weighting      =  'natural'   #   Weighting to apply to visibilities
uvfilter       =      False   #   Apply additional filtering/uv tapering of the visibilities
timerange      =         ''   #   range of time to select from data
restfreq       =         ''   #   restfrequency to use in image
async          =      False   #   if True run in the background, prompt is freed
```

Figure 1.1 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.

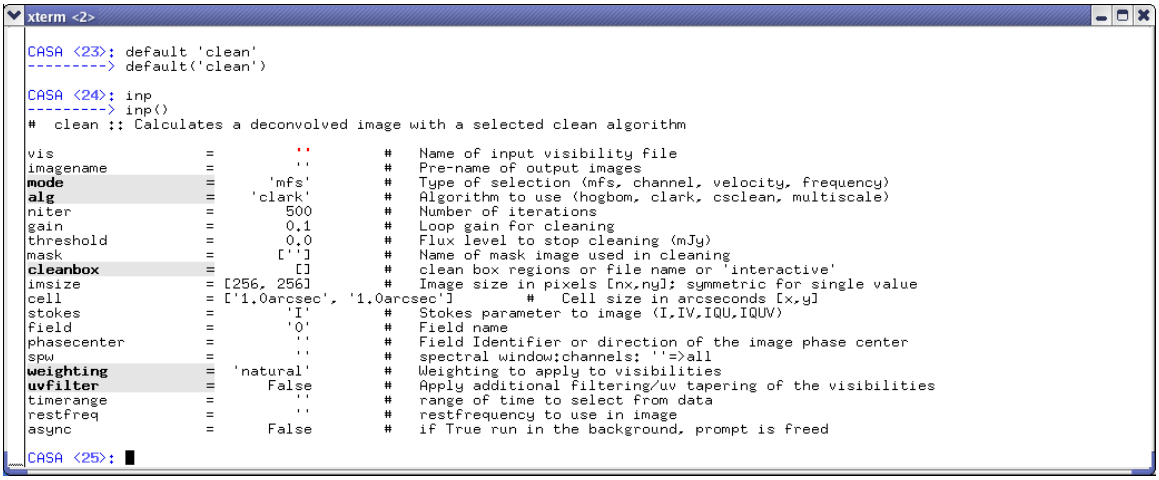



Figure 1.1: Screen shot of the default CASA inputs for task `clean`.

---

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 1.2 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 1.3 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'`).

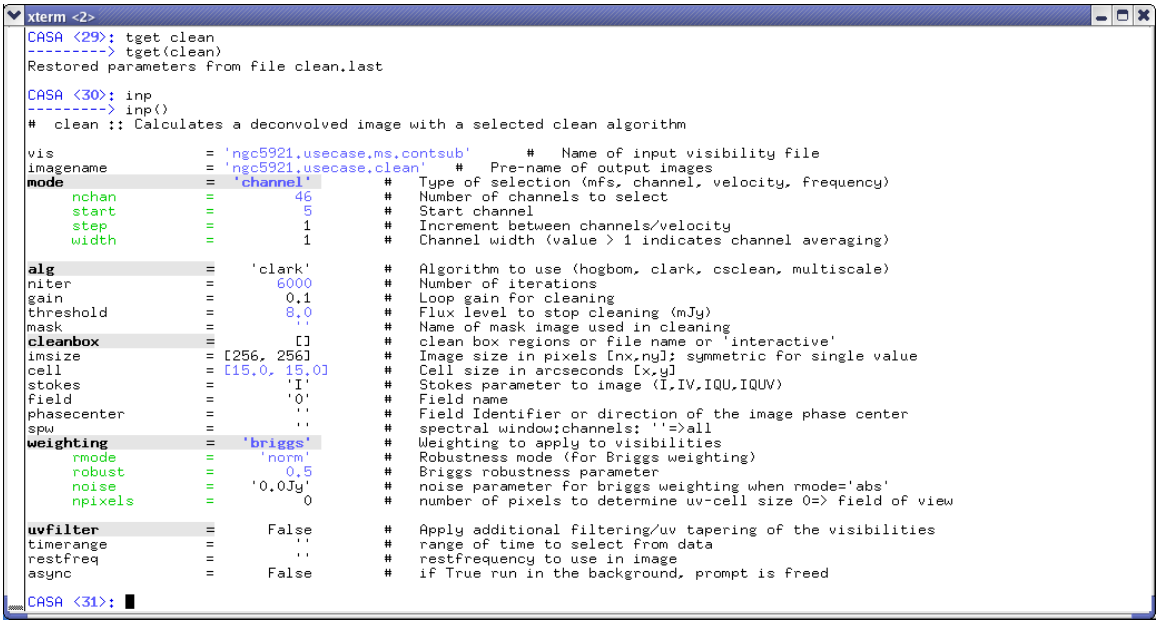



Figure 1.2: The `clean` inputs after setting values away from their defaults (blue text). Note that some of the boldface ones have opened up new dependent sub-parameters (indented and green).

---

#### 1.3.3.4 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.

**Beta 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`.

#### 1.3.3.5 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

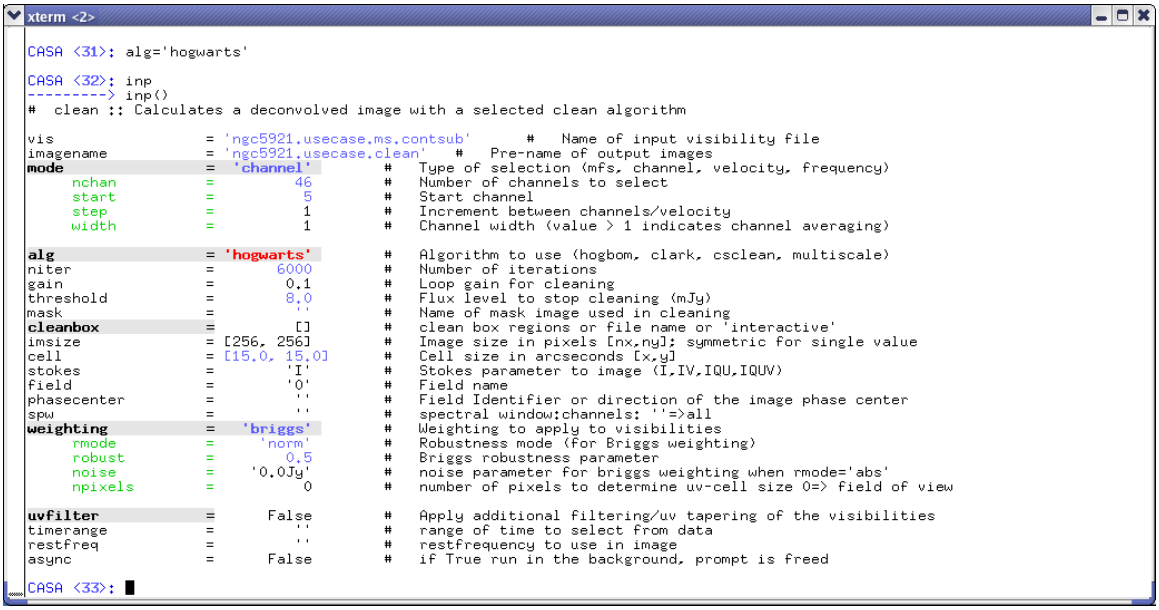



Figure 1.3: The `clean` inputs where one parameter has been set to an invalid value. This is drawn in red to draw attention to the problem. This hapless user probably confused the `'hogbom'` clean algorithm with Harry Potter.

---

```
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)
```

You can also use the CASA `tget` command (see § 1.3.3.6 below) instead of the Python `execfile` to restore your inputs.

#### 1.3.3.6 The `tget` Command

The `tget` command will recover saved values of the inputs of tasks. This is a convenient alternative to using the Python `execfile` command (see above).

Typing `tget` without a taskname will recover the saved values of the inputs for the current task as given in the current value of the `taskname` parameter.

Adding a task name, e.g. `tget <taskname>` will recover values for the specified task. This is done by searching for 1) a `<taskname>.last` file (see § 1.3.3.7 below), then for 2) a `<taskname>.saved` file (see § 1.3.3.5 above), and then executing the Python in these files.

For example,

```
default('clean')     # set current task to clean and default
tget                 # read saved inputs from clean.last (or clean.saved)
inp                  # see these inputs!
tget mosaic          # now get from mosaic.last (or mosaic.saved)
inp                  # task is now mosaic, with recovered inputs
```

#### 1.3.3.7 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 in 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.

## 1.4 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.

### 1.4.1 Your command line history

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

You can also use the "up-arrow" and "down-arrow" keys for command line recall in the `casapy` interface. If you start typing text, and then use "up-arrow", you will navigate back through commands matching what you typed.

### 1.4.2 Logging your session

The output from CASA commands is sent to the file `casapy.log`, also in your local directory. Whenever you start up `casapy`, the previous `casapy.log` is renamed (based on the date and time) and a new log file is started.

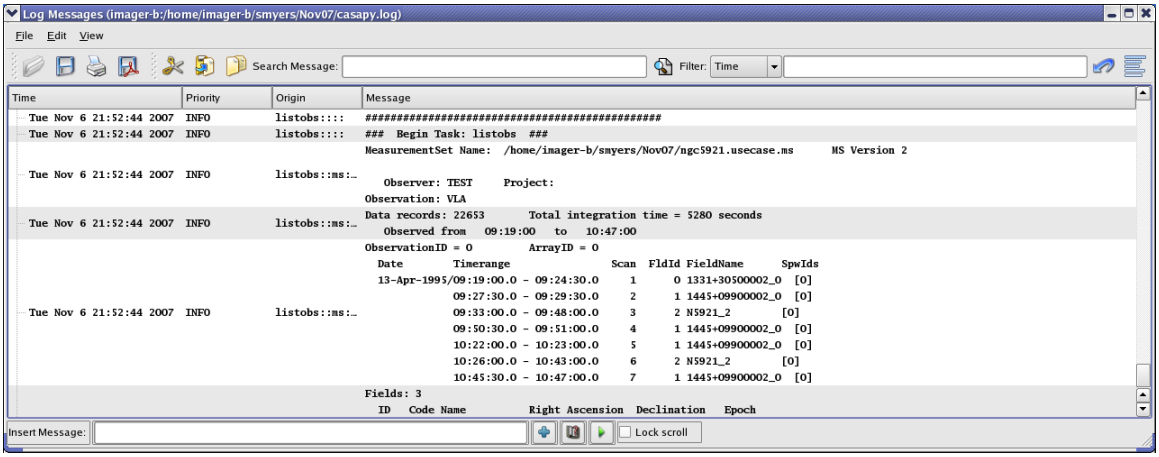



Figure 1.4: The CASA Logger GUI window.

---

The output contained in `casapy.log` is also displayed in a separate window using the *casalogger*. 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.

The CASA logger window is shown in Figure 1.4. The main feature is the display area for the log text, which is divided into columns. The columns are:

- `Time` — the time that the message was generated. Note that this will be in local computer time (usually UT) for casapy generated messages, and may be different for user generated messages;

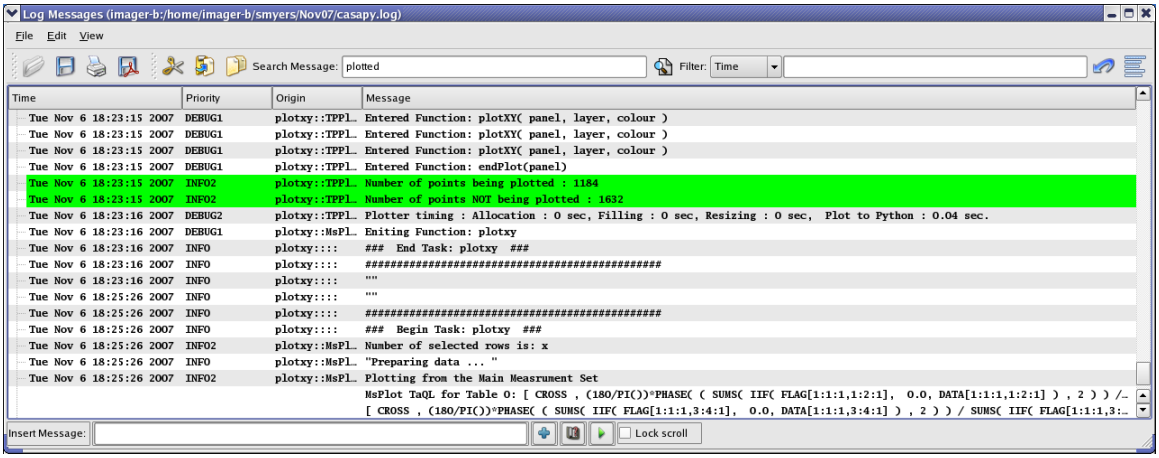



Figure 1.5: Using the Search facility in the `casalogger`. Here we have specified the string `'plotted'` and it has highlighted all instances in green.

---

- `Priority` — the *Priority Level* (see below) of the message;
- `Origin` — where within CASA the message came from. This is in the format `Task::Tool::Method` (one or more of the fields may be missing depending upon the message);
- `Message` — the actual text.

The `casalogger` GUI 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 1.5 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 1.6 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 to enter a longer message. The entered message will appear with a priority of "NOTE" with the Origin as your username. See Figure 1.7 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

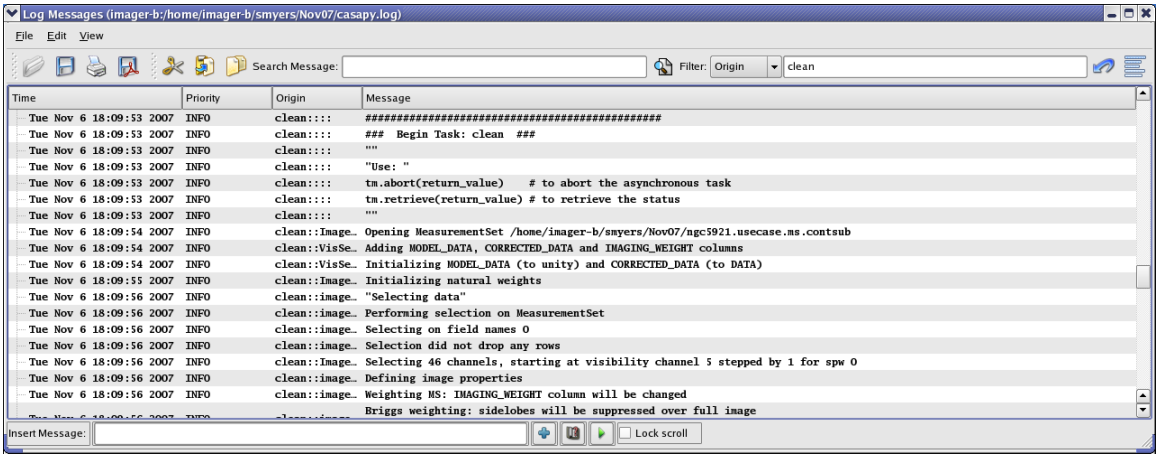



Figure 1.6: Using the `casalogger` Filter facility. The log output can be sorted by Priority, Time, Origin, and Message. In this example we are filtering by Origin using `'clean'`, and it now shows all the log output from the `clean` task.

---

the clipboard. You can then (usually) paste this where you wish. **BETA 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** — **BETA ALERT:** there is an Open function in the **File** menu, and an Open button, but these are "grayed-out" in the beta. Sorry!

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

#### 1.4.2.1 Setting priority levels in the `logger`

Logger messages are assigned a *Priority Level* when generated within CASA. The current levels of Priority are:

1. **SEVERE** — errors;
2. **WARN** — warnings;
3. **INFO** — basic information every user should be aware of or has requested;
4. **INFO1** — information possibly helpful to the user;
5. **INFO2** — details the power user might want to see;
6. **INFO3** — even more details;

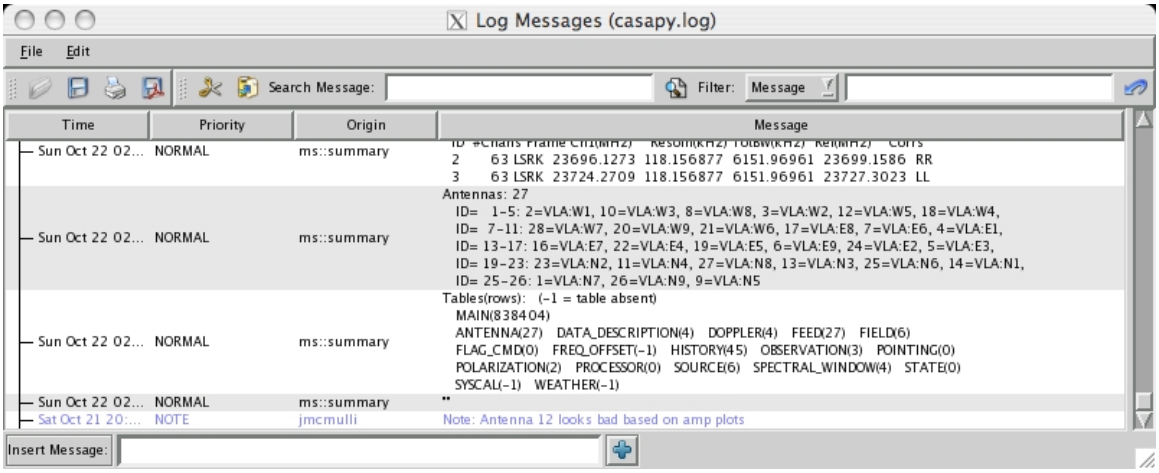



Figure 1.7: CASA Logger - Insert facility: The log output can be augmented by adding notes or comments during the reduction. The file should then be saved to disk to retain these changes.

---

7. **INFO4** — lowest level of non-debugging information;
8. **DEBUGGING** — most "important" debugging messages;
9. **DEBUG1** — more details;
10. **DEBUG2** — lowest level of debugging messages.

The "debugging" levels are intended for the developers use.

There is a threshold for which these messages are written to the `casapy.log` file and are thus visible in the `logger`. By default, only messages at level **INFO** and above are logged. The user can change the threshold using the `casalog.filter` method. This takes a single string argument of the `level` for the threshold. The `level` sets the lowest priority that will be generated, and all messages of this level or higher will go into the `casapy.log` file.

**Inside the Toolkit:**
The `casalog` tool can be used to control the logging. In particular, the `casalog.filter` method sets the priority threshold. This tool can also be used to change the output log file, and to post messages into the logger.

Some examples:

```
casalog.filter('INFO')          # the default
casalog.filter('INFO2')         # should satisfy even advanced users
casalog.filter('INFO4')         # all INFOx messages
casalog.filter('DEBUG2')        # all messages including debugging
```

**WARNING:** Setting the threshold to **DEBUG2** will put lots of messages in the log!

**BETA ALERT:** We are transitioning to the new Priority Level system, and not all tasks and tools obey the guidelines uniformly. This will be improved as we progress through the Beta patches. Also, the `casalog` tool is the only way to set the threshold currently.

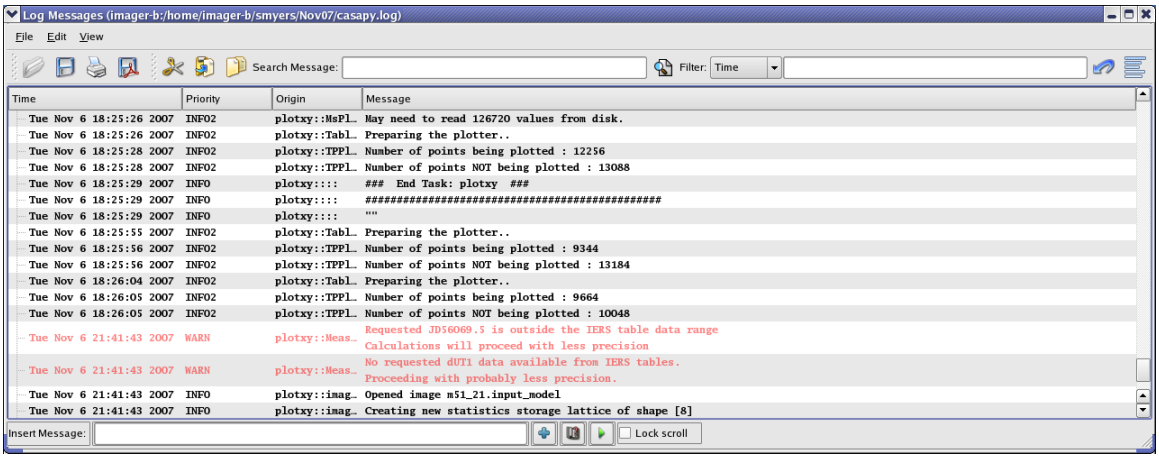



Figure 1.8: Different message priority levels as seen in the `casalogger` window. These can also be Filtered upon.

---

### 1.4.3 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 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 § 1.2.6.5), 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.

### 1.4.4 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).

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.

### 1.4.5 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.

## 1.5 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 6.

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.

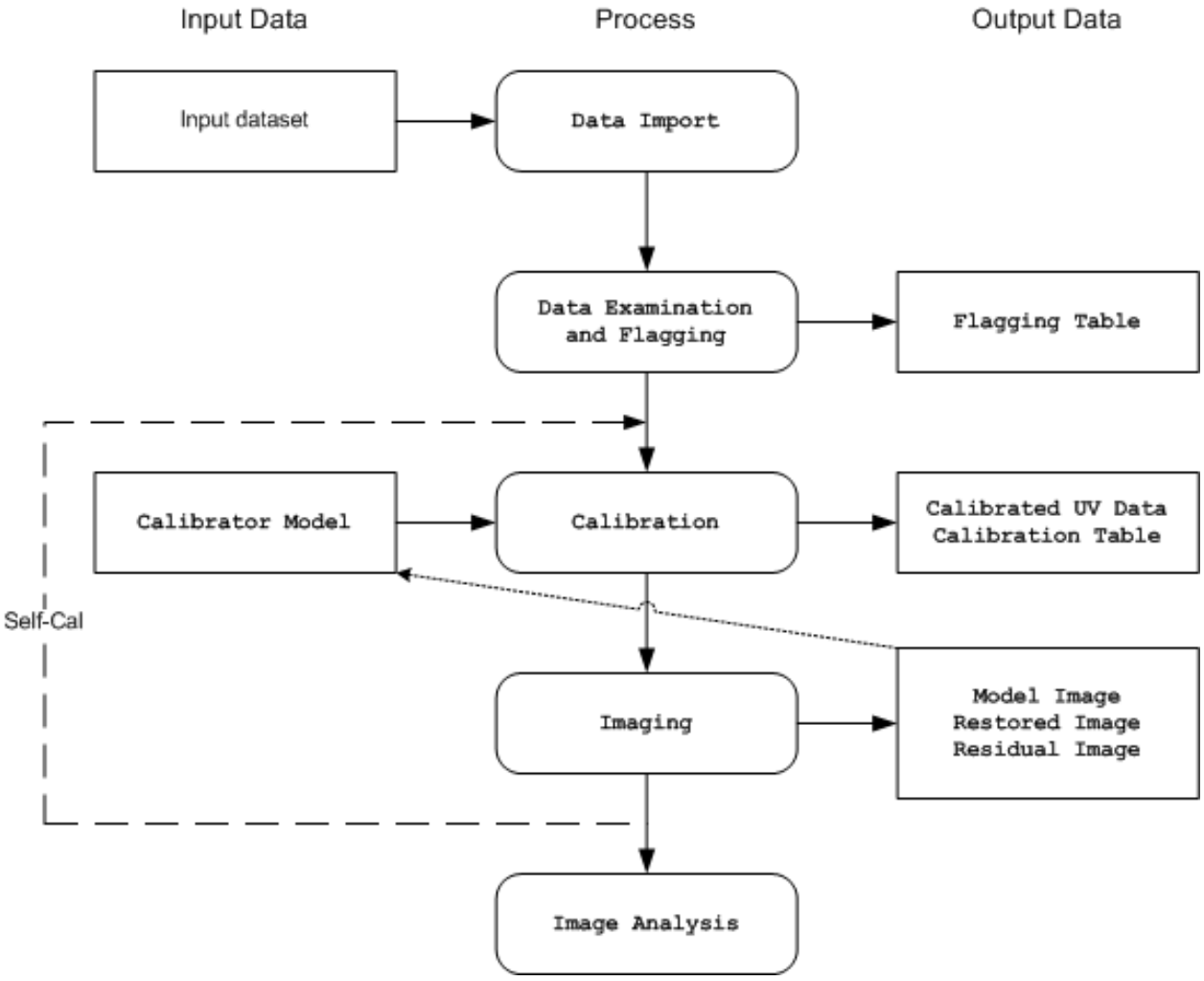



Figure 1.9: Flow chart of the data processing operations that a general user will carry out in an end-to-end CASA reduction session.

---

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

### 1.5.1 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 (§ 6.4).

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 § 1.4.3 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.

#### 1.5.1.1 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.

#### 1.5.1.2 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.

#### 1.5.1.3 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 § 6.4 for more information.

#### 1.5.1.4 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.

### 1.5.2 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 (§ 7).

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.

Visualization and editing of an MS using the `casaviewer` is described in Chapter 7.

#### 1.5.2.1 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 (§ 3.4.4). 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.

#### 1.5.2.2 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.

#### 1.5.2.3 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 § 7.4.

### 1.5.3 Calibration

The major calibration tasks are:

- `setjy` — Computes the model visibilities for a specified source given a flux density or model image, knows about standard calibrator sources (§ 4.3.4);
- `bandpass` — Solves for frequency-dependent (bandpass) complex gains (§ 4.4.2);
- `gaincal` — Solves for time-dependent (frequency-independent) complex gains (§ 4.4.3);
- `fluxscale` — Bootstraps the flux density scale from standard calibrators (§ 4.4.4);
- `accum` — Accumulates incremental calibration solutions into a cumulative calibration table (§ 4.5.4);
- `smoothcal`— Smooths calibration solutions derived from one or more sources (§ 4.5.3);
- `applycal` — Applies calculated calibration solutions (§ 4.6.1);
- `clearcal` — Re-initializes calibrated visibility data in a given measurement set (§ 4.6.3);
- `listcal` — Lists calibration solutions (§ 4.5.2);
- `plotcal` — Plots (and optionally flags) calibration solutions (§ 4.5.1);
- `uvcontsub` — carry out uv-plane continuum subtraction for spectral-line data (§ 4.7.2);
- `split` — write out a new (calibrated) MS for specified sources (§ 4.7.1).

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 taken 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.

#### 1.5.3.1 Prior Calibration

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.

Antenna gain-elevation curves (e.g. for the VLA antennas) and atmospheric optical depth corrections (applied as an elevation-dependent function) may be pre-applied before solving for the bandpass and gains. This is currently done by setting the `gaincurve` and `opacity` parameters in the various calibration solving tasks.

See § 4.3 for more details.

#### 1.5.3.2 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. The default `'B'` solution mode solves for the gains in frequency slots consisting of channels or averages of channels.

A polynomial fit for the solution (solution type `'BPOLY'`) may be carried out instead of the default frequency-slot based `'B'` solutions. This single solution will span (combine) multiple spectral windows.

Bandpass calibration is discussed in detail in § 4.4.2.

If the gains of the system are changing over the time that the bandpass calibrator is observed, then you may need to do an initial gain calibration (see next step).

#### 1.5.3.3 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. The default `'G'` solution mode solved for gains in specified time solution invervals.

A spline fit for the solution (solution type `'GSPLINE'`) may be carried out instead of the default time-slot based `'G'` solutions. This single solution will span (combine) multiple spectral windows.

See § 4.4.3 for more on gain calibration.

#### 1.5.3.4 Examining Calibration Solutions

The `plotcal` task (§ 4.5.1) will plot the solutions in a calibration table. The `xaxis` choices include time (for `gaincal` solutions) and channel (e.g. 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 (§ 4.5.2) will print out the calibration solutions in a specified table.

#### 1.5.3.5 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 § 4.4.4 for more.

#### 1.5.3.6 Calibration Accumulation

The `accum` task applies an incremental solution, of a given type, from a table to a previous calibration table (of the same type), and writes out a cumulative solution table. Different interpolation schemes may be selected.

A description of this process is given in § 4.5.4.

#### 1.5.3.7 Correcting the Data

The final step in the calibration process, `applycal` may be used to apply several calibration tables (e.g., from `gaincal` or `bandpass`). 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 § 4.6.1 for details.

#### 1.5.3.8 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 (§ 4.7.1).

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

### 1.5.4 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`, a file or list specifying sets of pairs of box corners (§ 5.7);
- `ft` — Fourier transforms the specified model (or component list) and inserts this 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 indicated sections for details.

See Chapter 5 for more on synthesis imaging.

#### 1.5.4.1 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.

#### 1.5.4.2 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 iteratively removes at each step a fraction of the flux in the brightest pixel in a defined region of the current "dirty" image, and places 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.

#### 1.5.4.3 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.

#### 1.5.4.4 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.

### 1.5.5 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.

### 1.5.6 Data and Image Analysis

The key data and image analysis tasks are:

- `imhead` — summarize and manipulate the "header" information in a CASA image (§ 6.1);
- `immoments` — compute the moments of an image cube (§ 6.2);
- `regridimage` — regrid an image onto the coordinate system of another image (§ 6.3);
- `viewer` — there are useful region statistics and image cube plotting capabilities in the viewer (§ 7).

#### 1.5.6.1 What's in an image?

The `imhead` task will print out a summary of image "header" keywords and values. This task can also be used to change the header values, and to compute image statistics.

See § 6.1 for more.

#### 1.5.6.2 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 § 6.2 for details.

#### 1.5.6.3 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 § 6.3 for a description of this task.

#### 1.5.6.4 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 for more on viewing images.

### 1.5.7 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 (§ 6.4).

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 (§ 2.2.1.1)
- `importvla` — import data from VLA that is in *export* format (§ 2.2.2)
- `importasdm` — import data in ALMA ASDM format (§ 2.2.3)
- `exportuvfits` — export a CASA MS in UVFITS format (§ 2.2.1.2)
- `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 § 2.5.

There are also tasks for the import and export of image data using FITS:

- `importfits` — import a FITS image into a CASA *image* format table (§ 6.4)
- `exportfits` — export a CASA image table as FITS (§ 6.4)

## 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 § A).

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.

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

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.

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. You may skip ahead to subsequent sections if you like!

**Inside the Toolkit:**
Generic CASA tables are handled in the `tb` tool. You have direct access to keywords, rows and columns of the tables with the methods of this tool.

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 tables are logical constructs, with contents located in the physical `table.*` files on disk. The `MAIN` table consists of the `table.*` files in the main directory of the ms-file itself, and the other tables are in the respective subdirectories. 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 (§ 3.6).

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 the `table.*` files in this directory. 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
```

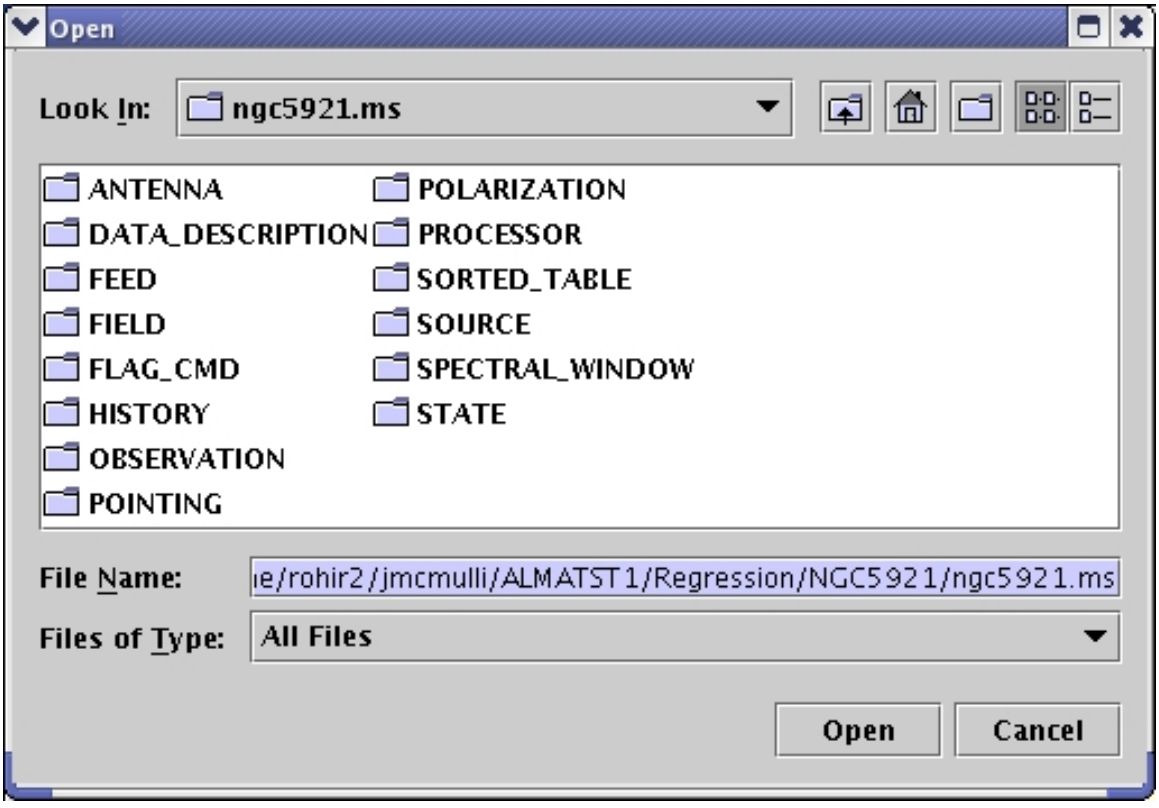



Figure 2.1: The contents of a Measurement Set. These tables compose a Measurement Set named `ngc5921.ms` on disk. This display is obtained by using the **File:Open** menu in `browsetable` and left double-clicking on the `ngc5921.ms` directory.

---

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). These columns are visible in browsetable and are accessible from the toolkit in the ms tool (e.g. the ms.getdata method) and from the tb "table" tool (e.g. using tb.getcol).

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 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).

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** | **Contents** |
|---|---|---|
| DATA | Complex($N_c$, $N_f$) | complex visibility data 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($N_c$, $N_f$) | individual weights for a data matrix |
| ALMA_PHASE_CORR | Complex($N_c$, $N_f$) | on-line phase corrected data (*Not in VLA data*) |
| ALMA_NO_PHAS_CORR | Bool($N_c$, $N_f$) | data that has not been phase corrected (*Not in VLA data*) |
| ALMA_PHAS_CORR_FLAG_ROW | Bool($N_c$, $N_f$) | flag to use phase-corrected data or not (*not in VLA data*) |
| MODEL_DATA | Complex($N_c$, $N_f$) | Scratch: created by calibrater or imager tools |
| CORRECTED_DATA | Complex($N_c$, $N_f$) | Scratch: created by calibrater or imager tools |
| IMAGING_WEIGHT | Float($N_c$) | Scratch: created by calibrater or imager tools |

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`)

### 2.2.1 UVFITS Import and Export

The UVFITS format is not exactly a standard, but is a popular archive and transport format nonetheless. CASA supports UVFITS files written by the AIPS `FITTP` task, and others.

UVFITS is supported for both import and export.

#### 2.2.1.1 Import using `importuvfits`

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.

For example:

```
importuvfits(fitsfile='NGC5921.fits',vis='ngc5921.ms')
```

**BETA ALERT:** We cannot currently fill CARMA data exported via Miriad UVFITS.

#### 2.2.1.2 Export using `exportuvfits`

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

```
#  exportuvfits :: Convert a CASA visibility data set (MS) to a UVFITS file

vis             =         ''    #   Name of input visibility file
fitsfile        =         ''    #   Name of output UVFITS file)
datacolumn      = 'corrected'   #   which data to write (data, corrected, model)
field           =         ''    #   Field name list
spw             =         ''    #   Spectral window and channel selection
antenna         =         ''    #   antenna list to select
time            =         ''    #   time range selection
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    =       True    #   Write station name instead of antenna name
async           =      False    #   if True run in the background, prompt is freed
```

For example:

```
exportuvfits(vis='ngc5921.split.ms',
             fitsfile='NGC5921.split.fits',
             multisource=False)
```

The MS selection parameters `field`, `spw`, `antenna`, and `timerange` follow the standard selection syntax described in § 2.5.

**BETA ALERT:** The `nchan`, `start`, and `width` parameters will be superseded by channel selection in `spw`. Currently, there is a `time` parameter rather than `timerange`.

The `datacolumn` parameter chooses which data-containing column of the MS (see § 2.1.1) is to be written out to the UV FITS file. Choices are: `'data'`, `'corrected'`, and `'model'`.

There are a number of special parameters that control what is written out. These are mostly here for compatibility with AIPS.

The `writesyscal` parameter toggles whether `GC` and `TY` extension tables are written. These are important for VLBA data, and for EVLA data. **BETA ALERT:** Not yet available.

The `multisource` parameter determines whether the UV FITS file is a multi-source file or a single-source file, if you have a *single-source* MS or choose only a single source. Note: the difference between a single-source and multi-source UVFITS file here is whether it has a source (`SU`) table and the source ID in the random parameters. If you select more than one source in `fields`, then the `multisource` parameter will be overridden to be `True` regardless.

The `combinespw` parameter allows combination of all spectral windows at one time. If `True`, then all spectral windows must have the same shape. For AIPS to read an exported file, then set `combinespw=True`.

The `writestation` parameter toggles the writing of the station name instead of antenna name.

### 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:

```
#  importvla :: import VLA archive file(s) to a measurement set:

archivefiles  =         ''   #   Name of input VLA archive file(s)
vis           =         ''   #   Name of output visibility file
bandname      =         ''   #   VLA frequency band name:''=>obtain all bands in archive files
frequencytol  =   150000.0   #   Frequency shift to define a unique spectral window (Hz)
project       =         ''   #   Project name:  '' => all projects in file
starttime     =         ''   #   start time to search for data
stoptime      =         ''   #   end time to search for data
autocorr      =      False   #   import autocorrelations to ms, if set to True
antnamescheme =      'new'   #    'old' or 'new'; 'VA04' or '4' for ant 4
async         =      False   #
```

The main parameters are `archivefiles` to specify the input VLA Archive format file names, and `vis` to specify the output MS name.

The NRAO Archive is located at:

- `https://archive.nrao.edu`

Note that `archivefiles` takes a string or list of strings, as there are often multiple files for a project in the archive.

For example:

```
archivefiles = ['AP314_A950519.xp1','AP314_A950519.xp2']
vis = 'NGC7538.ms'
```

The `importvla` task allows selection on the 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,

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

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']
```

**Important Note:** `importvla` will import the on-line flags (from the VLA system) along with the data. These will be put in the `MAIN` table and thus available to subsequent tasks and tools. If you wish to revert to unflagged data, use `flagmanager` (§ 3.2) to save the flags (if you wish), and then use `flagdata` (§ 3.5) with `mode='manualflag'` and `unflag=True` to toggle off the flags.

The other parameters are:

#### 2.2.2.1 Parameter `bandname`

The `bandname` indicates the VLA Frequency band(s) to load, using the traditional bandname codes. These are:

- `'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
- '' = all bands (default)

Note that as the transition from the VLA to EVLA progresses, the actual frequency ranges covered by the bands will expand, and additional bands will be added (namely 'S' from 1-2 GHz and 'A' from 26.4-40 GHz).

#### 2.2.2.2 Parameter frequencytol

The frequencytol parameter specifies the frequency separation tolerated when assigning data to spectral windows. The default is frequencytol=150000 (Hz). For Doppler tracked data, where the sky frequency changes with time, a frequencytol $<$ 10000 Hz may may produce too many unnecessary spectral windows.

#### 2.2.2.3 Parameter project

You can specify a specific project name to import from archive files. The default '' will import data from all projects in file(s) archivefiles.

For example for VLA Project AL519:

```
project = 'AL519'    # this will work
project = 'al519'    # this will also work
```

while project='AL0519' will NOT work (even though that is what queries to the VLA Archive will print it as - sorry!).

#### 2.2.2.4 Parameters starttime and stoptime

You can specify start and stop times for the data, e.g.:

```
starttime = '1970/1/31/00:00:00'
stoptime = '2199/1/31/23:59:59'
```

Note that the blank defaults will load all data fitting other criteria.

#### 2.2.2.5 Parameter `autocorr`

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.

#### 2.2.2.6 Parameter `antnamescheme`

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).

**BETA 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. Also, currently there are no options for filling selected 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_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 17:23:55 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 Measurement Sets 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.

For example:

```
os.system('cp -r n4826_16apr.split.ms n4826_tboth.ms') # copy
vis = 'n4826_tboth.ms'
concatvis = 'n4826_22apr.split.ms'
freqtol = '50MHz'
concat()
```

combines the two days in `'n4826_16apr.split.ms'` and `'n4826_22apr.split.ms'`.

**BETA ALERT:** You currently need to manually copy the first file to the final output MS, since it appends the `concatvis` to `vis`. We will shortly make it so that optionally a new MS is written.

## 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.

**Beta 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 though 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 accommodate 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 as 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 given 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.

The specification can be via frequency ranges or by indexes. A range of frequencies are used to select all spectral windows which contain channels within the given range. Frequencies can be specified with an optional unit — the default unit being `Hz`. Other common choices for radio and mm/sub-mm data are `kHz`, `MHz`, and `GHz`. You will get the entire spectral windows, not just the channels in the specified range. You will need to do channel selection (see below) to do that.

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.

**BETA ALERT:** In the current Beta Release, `'<ID'` and `'>ID'` are *inclusive* with the `ID` specified included in the selection, e.g. `spw='<2'` is equivalent to `spw='0,1,2'` and not `spw='0,1'` as was intended. This will be fixed in an upcoming release.

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 = '<3,5'               # SPWID 0,1,2,3 and 5
spw = '*'                  # All spectral windows
spw = '1412~1415MHz'       # Spectral windows containing 1412-1415MHz
```

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`s 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:

**Beta 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 `'FSTART~FSTOP^FSTEP'`

The most common selection is via channel ranges `'START~STOP'` or frequency ranges `'FSTART~FSTOP'`:

```
spw = '0:13~53'             # spw 0, channels 13-53, inclusive
spw = '0:1413~1414MHz'      # spw 0, 1413-1414MHz 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 select the spectral window via frequency ranges `'FSTART~FSTOP'`, as described above:

```
spw = '1413~1414MHz:1413~1414MHz'    # channels falling within 1413~1414MHz
spw = '*:1413~1414MHz'               # does the same thing
```

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 2 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`. For example:

   ```
   timerange = '2007/10/09/00:40:00 ~ 2007/10/09/03:30:00'
   ```

   Note that fields missing in `T0` are replaced by the fields in the time stamp of the first valid row in the MS. For example,

   ```
   timerange = '09/00:40:00 ~ 09/03:30:00'
   ```

   where the `YY/MM/` part of the selection has been defaulted to the start of the MS.

   Fields missing in `T1`, such as the date part of the string, are replaced by the corresponding fields of `T0` (after its defaults are set). For example:

   ```
   timerange = '2007/10/09/22:40:00 ~ 03:30:00'
   ```

   does the same thing as above.

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

   ```
   timerange = '2007/10/09/23:41:00'
   ```

   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`. For example,

   ```
   timerange = '23:41:00+01:00:00'
   ```

   picks an hour-long chunk of time.

   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`. For example,

   ```
   timerange = '>2007/10/09/23:41:00'
   ```

   Default settings for `T0` are as above.

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

   ```
   timerange = '<2007/10/09/23:41:00'
   ```

   Default settings for `T1` are as above.

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

$$r_{uv} = \sqrt{u^2 + v^2}. \tag{2.1}$$

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 = '24~35Ml, 40~45Ml'    # two annuli in units of mega-wavelengths
uvrange = '< 45kl'              # less than 45 kilolambda
uvrange = '> 0l'                # greater than zero length (no auto-corrs)
uvrange = '100km'               # baselines of length 100km
uvrange = '100kl'               # visibilities with uv-radius 100 kilolambda
```

#### 2.5.4.5 The msselect Parameter

More complicated selections within the MS structure are possible using the Table Query Language (TaQL). This is accessed through the msselect 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 msselect rather than through field or spw:

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

**BETA ALERT:** The msselect style parameters will be phased out of the tasks. TaQL selection will still be available in the Toolkit.

# 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 (§ 3.2)
- `flagautocorr` — non-interactive flagging of auto-correlations (§ 3.3)
- `plotxy` — create X-Y plots of data in MS, flag data (§ 3.4)
- `flagdata` — non-interactive flagging of data (§ 3.5)
- `browsetable` — browse data in any CASA table (including a MS) (§ 3.6)

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)
- `viewer` — use the `casaviewer` to display the MS as a raster image, and flag it (§ 7)

## 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 on import (by `importvla` or `importuvfits`) or when flagging is first done on an MS without a `.flagversions` (e.g. with `plotxy`).

By default, when the `.flagversions` is created, this directory will contain a `flags.Original` in it containing a copy of the original flags in the `MAIN` table of the MS so that you have a backup. It will also contain a file called `FLAG_VERSION_LIST` that has the information on the various flag versions there.

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()
MS : /home/imager-b/smyers/Oct07/jupiter6cm.usecase.ms

main : working copy in main table
Original : Original flags at import into CASA
flagautocorr : flagged autocorr
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.flagautocorr  flags.Original  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 an 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/`.

**Inside the Toolkit:**
Access to `matplotlib` is also provided through the `pl` tool. See below for a description of the `pl` tool functions.

The `plotxy` plotter is shown in figure 3.1.

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

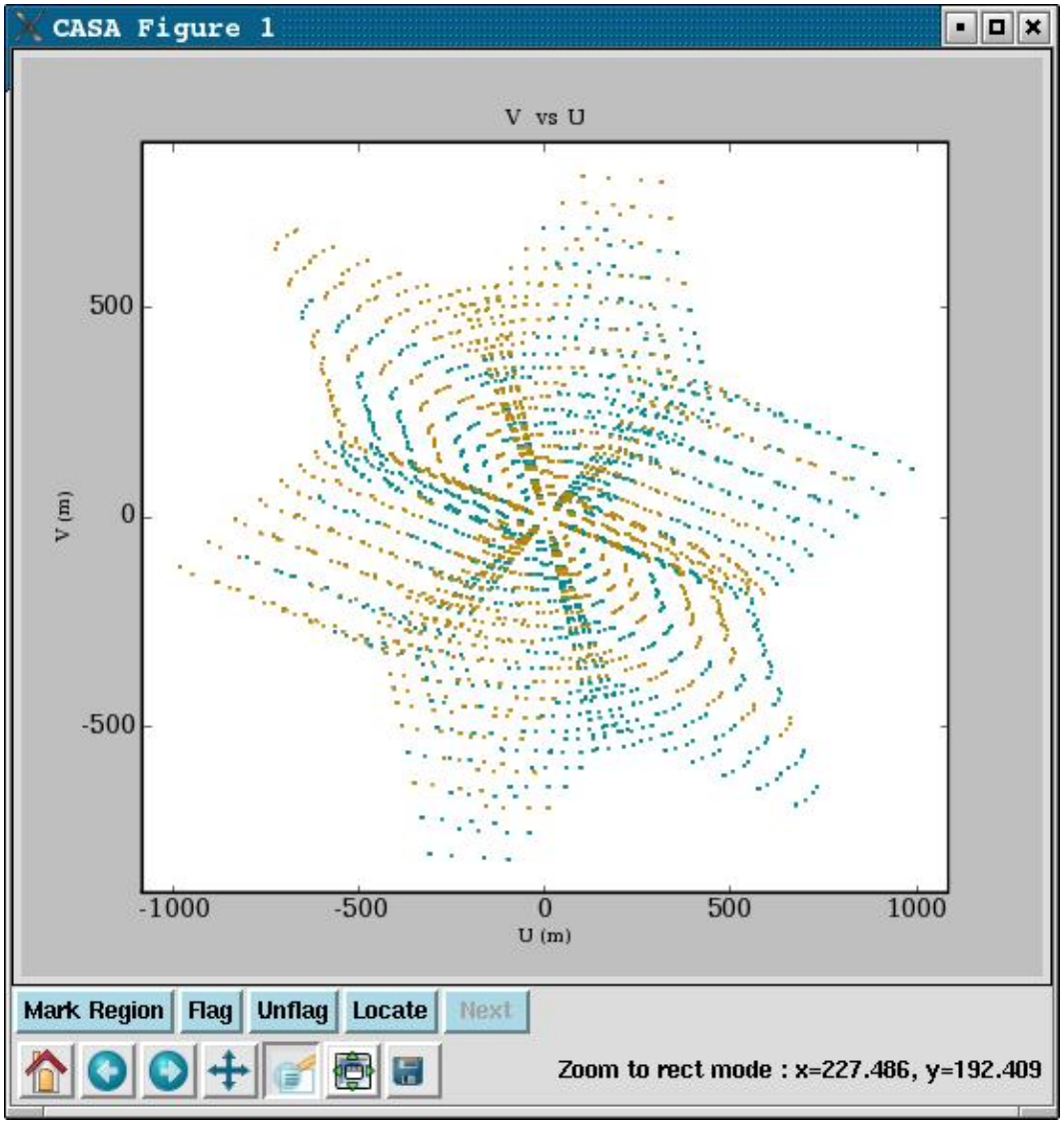



Figure 3.1: The `plotxy` plotter. The **bottom set of buttons** on the lower left are: 1,2,3) **Home, Back, and Forward**. Click to navigate between previously defined views (akin to web navigation). 4) **Pan**. Click and drag to pan to a new position. 5) **Zoom**. Click to define a rectangular region 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.

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

vis              =         ''   #   Name of input visibility
xaxis            =     'time'   #   X-axis: see help for options
yaxis            =      'amp'   #   Y-axis: see help for options
     datacolumn  =     'data'   #   data (raw), corrected, model, residual (corrected - model)

field            =         ''   #   field names or index of calibrators: ''==>all
spw              =         ''   #   spectral window:channels: ''==>all, spw='1:5~57'
selectdata       =      False   #   Other data selection parameters
average          =         ''   #   Select averaging mode: time, chan or both
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
connect          =     'none'   #   Specifies which points are connected with lines
multicolor       =     'corr'   #   Plot in different colors: Options: none,both,chan,corr
selectplot       =      False   #   Select additional plotting options (e.g, fontsize, title,etc)
```

Setting `selectdata=True` opens up the selection sub-parameters:

```
selectdata       =       True   #   Other data selection parameters
     antenna     =         ''   #   antenna/baselines: ''==>all, antenna = '3,VA04'
     timerange   =         ''   #   time range: ''==>all
     correlation =         ''   #   correlations: default = ''
     scan        =         ''   #   scan numbers: Not yet implemented
     feed        =         ''   #   multi-feed numbers: Not yet implemented
     array       =         ''   #   array numbers: Not yet implemented
     uvrange     =         ''   #   uv range''==>all; uvrange = '0~100kl'
```

These are described in § 2.5.

Setting `selectplot=True` will open up a set of plotting control sub-parameters. These are described in § 3.4.2 below.

The `average`, `iteration`, `plotsymbol`, and `subplot` parameters deserve extra explanation, and are described in § 3.4.3 below.

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='corr')                    # Allow plotxy to plot different colors.
```

### 3.4.1 GUI 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 The `selectplot` Parameters

These parameters work in concert with the native matplotlib functionality to enable flexible representations of data displays.

Setting `selectplot=True` will open up a set of plotting control sub-parameters:

```
selectplot           =       True     #   Select additional plotting options
     markersize      =        5.0     #   Size of plotted marks
     linewidth       =        1.0     #   Width of plotted lines
     plotrange       = [-1,-1,-1,-1]  #   The range of data to be plotted
     skipnrows       =          1     #   Plot every nth point
     replacetopplot  =      False     #   Replace the last plot when overplotting
     removeoldpanels =       True     #   Automatic clearing of panels
     title           =         ''     #   Plot title (above plot)
     xlabels         =         ''     #   Label for x-axis
     ylabels         =         ''     #   Label for y-axis
     fontsize        =       10.0     #   Font size for labels
     windowsize      =        1.0     #   Window size
```

The `markersize` parameter will change the size of the plot symbols. Increasing it will help legibility when doing screen shots. Decreasing it can help in congested plots. The `linewidth` parameter will do similar things to the lines.

The `plotrange` parameter can be used to specify the size of the plot. The format is `[xmin, xmax, ymin, ymax]`. The units are those on the plot. For example,

```
plotrange = [-20,100,15,30]
```

Note that if `xmin=xmax` and/or `ymin=ymax`, then the values will be ignored and a best guess will be made to autorange that axis. `BETA ALERT:` Unfortunately, the units for the time axis must be in Julian Days, which are the plotted values.

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

The `skipnrows` parameter, if set to an integer `n` greater than 1, will allow only every `n`th point to be plotted. It does this, as the name suggests, by skipping over whole rows of the MS, so beware (channels are all within the same row for a given `spw`).

The `replacetopplot` toggle lets you choose whether or not the last layer plotted is replaced when `overplot=True`, or whether a new layer is added.

The `removeoldpanels` parameter turns on/off the clearing of plot panels that lie under the current panel layer being plotted.

The `title`, `xlabels`, and `ylabels` parameters can be used to change the plot title and axes labels.

The `fontsize` parameter is useful in order to enlarge the label fonts so as to be visible when making plots for screen capture, or just to improve legibility. Shrinking can help if you have lots of panels on the plot also.

The `windowsize` parameter is supposed to allow adjustments on the window size. **BETA ALERT:** This currently does nothing, unless you set it below 1.0, in which case it will produce an error.

### 3.4.3 Plot Control Parameters

The `average`, `iteration`, `plotsymbol`, and `subplot` parameters deserve extra explanation:

#### 3.4.3.1 `average`

The parameter `average` is used to request averaging either by time or by frequency channel. The choices are: `'time'`, `'chan'`, `'both'`, or `''`.

The default `''` chooses no averaging.

The option `average='time'` leads to the option of setting the sub-parameter `averagenrows`, which sets the number of *rows* to average when plotting. This is somewhat like time averaging, and basically averages `averagenrows` integrations together.

The option `average='chan'` affects the interpretation of the `spw` parameter (§ 2.5.3). For example,

```
average = 'chan'
spw = '0:5~59^5'
```

means to average every 5 channels rather than to show every 5th channel. See Figure 3.2 for an example of channel averaging.

The option `average='both'` allows simultaneous time and channel averaging, with the same controls as each option alone.

**BETA ALERT:** Choosing `average` to be anything other than `''` will currently force `plotxy` to iterate through the MS, even if you set what you think is no averaging (e.g. leaving the default `averagnrows=1` with `average='time'`) and it will run much slower. In fact, the performance for averaging in `plotxy` is a target for improvement. Also, the averaging options are in the process of being refined and extended (for example to specify time averaging in time units).

#### 3.4.3.2 `iteration`

There are currently four iteration options available: `'antenna'`, `'baseline'`, `'field'`, and `'scan'`. 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 'antenna' option will get you 27 displays, one for each antenna).

The will eventually be a `'feed'` option also.

An example use of iteration:

```
default('plotxy')

average = 'chan'                    # choose channel averaging
spw = '0:5~59^5'                    # average every 5 channels

plotxy('n5921.ms','channel',subplot=221,iteration='antenna')
```

The results of this are shown in Figure 3.2. Note that this example combines the use of `average`, `iteration` and `subplot`.

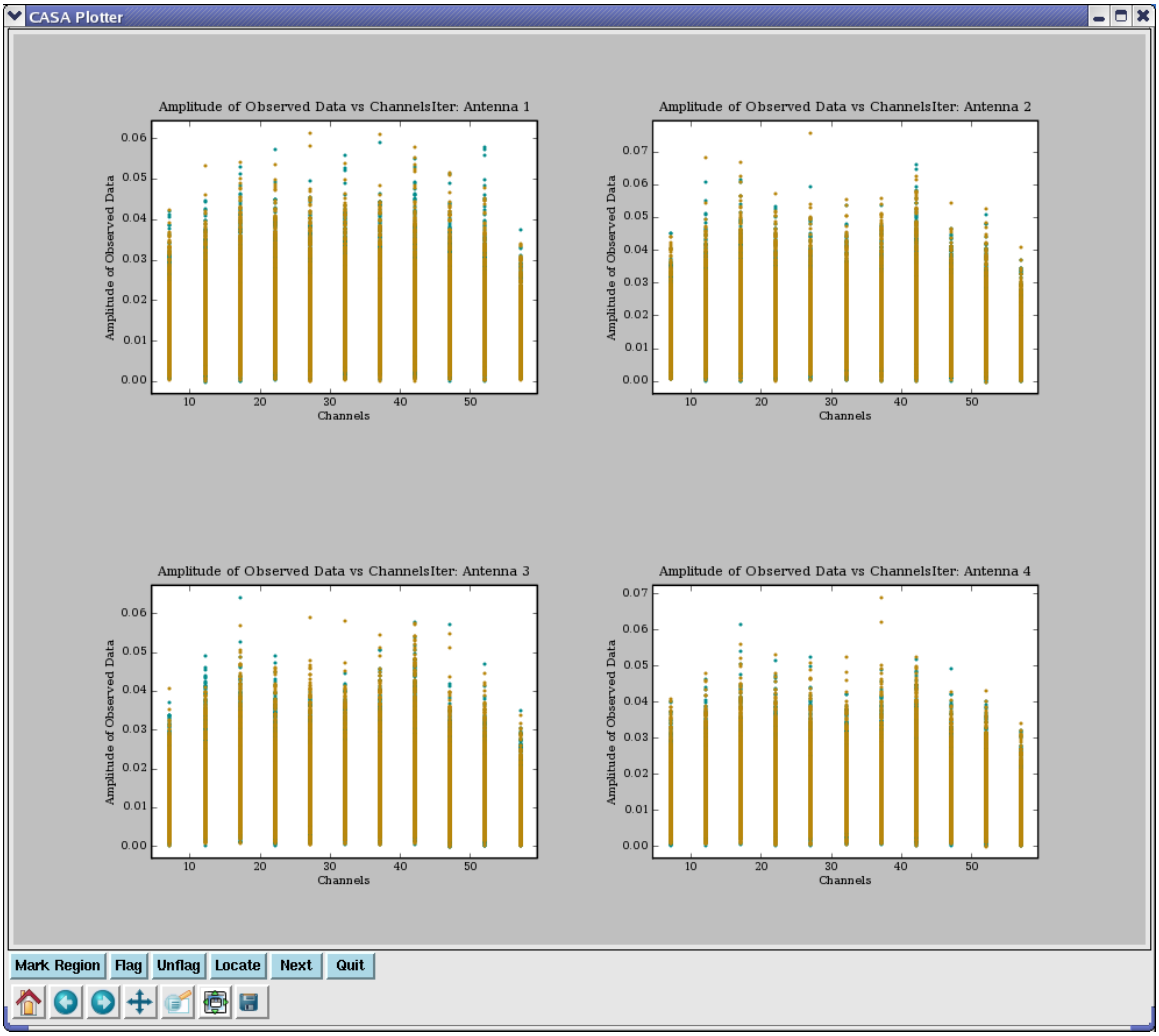



Figure 3.2: The `plotxy` iteration plot. The first set of plots from the example in § 3.4.3.2 with `iteration='antenna'`. Each time you press the **Next** button, you get the next series of plots.

#### 3.4.3.3 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
    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.
```

#### 3.4.3.4 subplot

The subplot parameter 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.
```

See also § 3.4.3.2 above, and Figure 3.2 for an example of channel averaging using iteration and subplot.

### 3.4.4 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 disk-write (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).

### 3.4.5 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 (§ 4.5.1) 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.

### 3.4.6 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.

**BETA ALERT**: Exact syntax may be slightly different in your version as the Beta Release 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 calibrater 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:

```
#  flagdata :: Flag/Clip data based on selections:

vis                 =         ''   #   Name of input visibility file
antenna             =         ''   #   antenna/baseline
spw                 =         ''   #   spectral-window/frequency/channel
correlation         =         ''   #   Select data based on correlation
field               =         ''   #   field names or indices
uvrange             =         ''   #   uv range (def=meters)
timerange           =         ''   #   time range
scan                =         ''   #   scan number
feed                =         ''   #   feed number - NOT ENABLED
array               =         ''   #   array
mode                = 'manualflag' #   Mode (manualflag,autoflag,summary,quack)
     autocorr       =      False   #   Flag autocorrelations
     unflag         =      False   #   Unflag the data specified
     clipexpr       =   'ABS RR'   #   Expression to clip on
     clipminmax     =         []   #   Range to use for clipping
     clipcolumn     =     'DATA'   #   Data column to use for clipping
     clipoutside    =       True   #   Clip outside the range, or within it
```

**BETA ALERT:** the modes `'autoflag'` and `'summary'` 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='antenna',subplot=311)
default('flagdata')
flagdata(vis='ngc5921.ms',antenna='0',spw='0:10~15')
default plotxy
plotxy('ngc5921.ms','channel',iteration='antenna',subplot=311)
```

#### 3.5.1.1 Clipping in `flagdata`

For `mode='manualflag''`, clipping is controlled by the sub-parameters:

```
mode                = 'manualflag' #   Mode (manualflag,autoflag,summary,quack)
     clipexpr       =   'ABS RR'   #   Expression to clip on
     clipminmax     =         []   #   Range to use for clipping
     clipcolumn     =     'DATA'   #   Data column to use for clipping
     clipoutside    =       True   #   Clip outside the range, or within it
```

The following commands give the results shown in Figure 3.6:

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

## 3.6 Browse the Data

The `browsetable` task is available for viewing data directly (and handles all CASA tables, including Measurement Sets, 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.

**BETA ALERT:** The `browsetable` task brings up the CASA Java `casabrowser`, which is a separate program. You can launch this from outside `casapy`. You may encounter some crashes, and this browser is being replaced with a new Qt based one. This is distributed with the current Beta as the `qcasabrowser`. This is under development, but if you want to try it you can type `qcasabrowser` outside `casapy` (if it is in your path), or from inside `casapy` using the `!` escape to the shell:

```
CASA <1>: !qcasabrowser
```

which will bring up the new browser (which is the best way if you have not run a `casainit.sh` with a default installation from rpms on Linux or on the Mac).

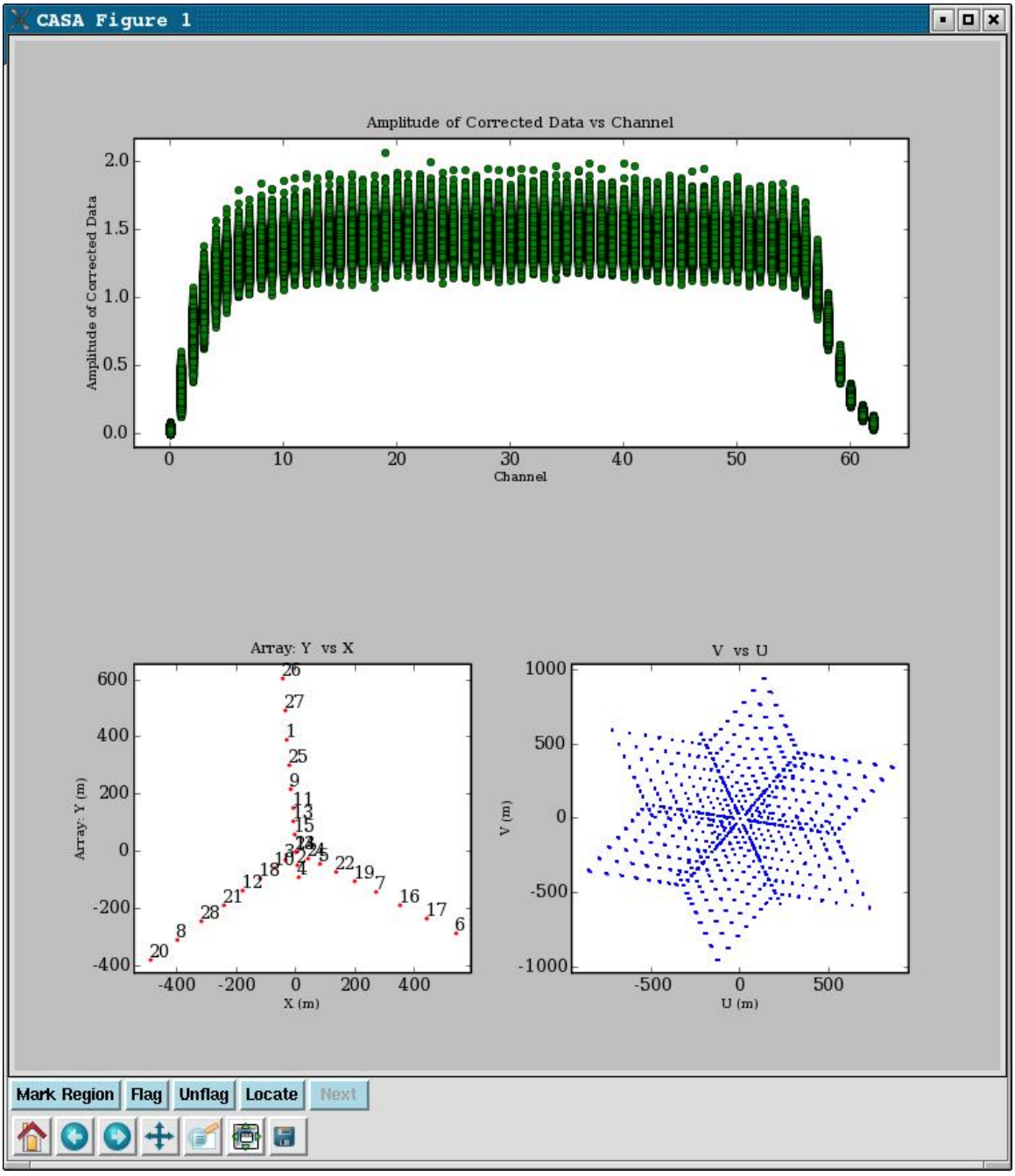



Figure 3.3: Multi-panel display of visibility versus channel (**top**), antenna array configuration (**bottom left**) and the resulting uv coverage (**bottom right**). The commands to make these three panels respectively are: 1) `plotxy('n5921.ms', xaxis='channel', datacolumn='corrected', field='0',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

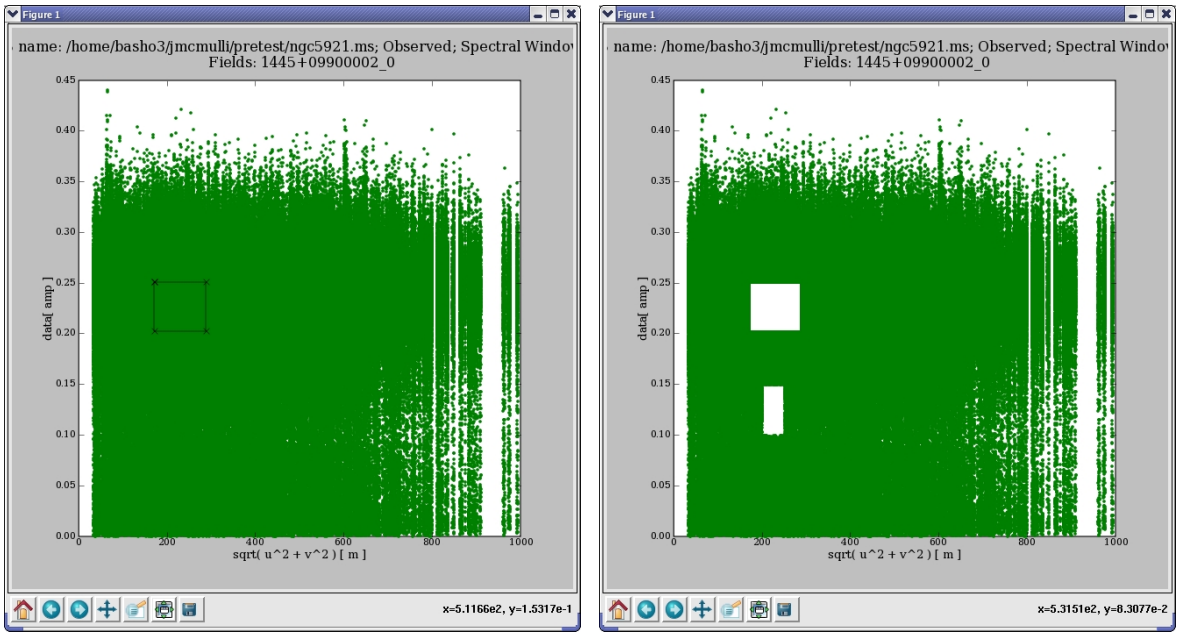



Figure 3.4: Plot of amplitude versus uv distance, before (left) and after (right) flagging two marked regions. The call was: `plotxy(vis='n5921.ms',xaxis='uvdist', plotsymbol='b,', subplot=111, datacolumn='data', field='1445*')`.

---

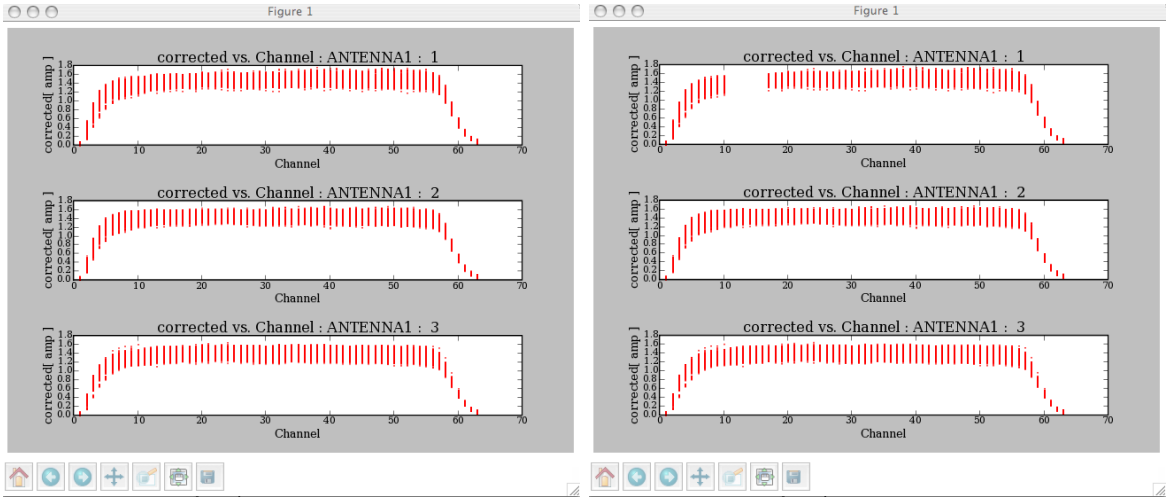



Figure 3.5: `flagdata`: Example showing before and after displays using a selection of one antenna and a range of channels. Note that each invocation of the flagdata task represents a cumulative selection, i.e., running antenna='0' will flag all data with antenna 0, while antenna='0', spw='0:10 15' will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

---

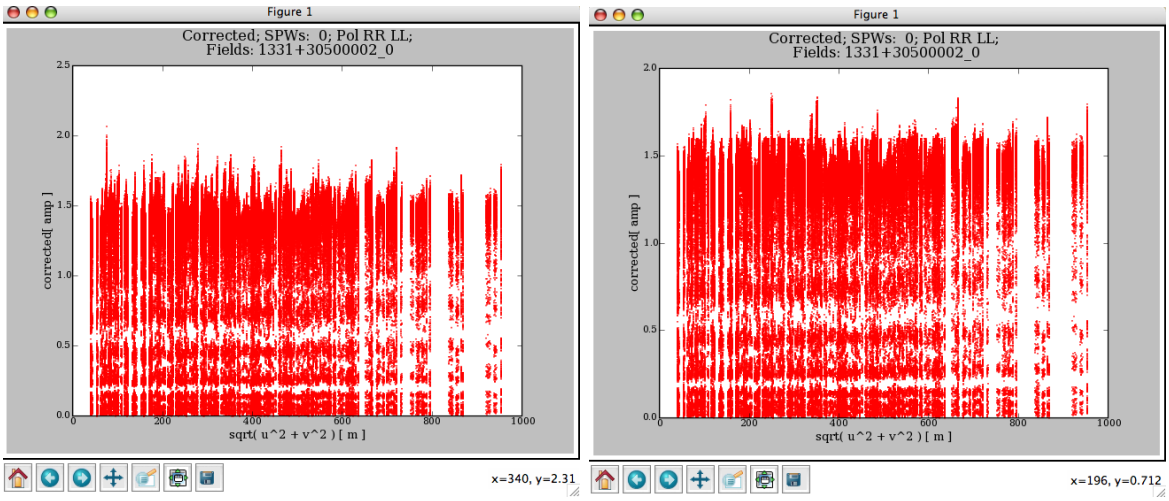



Figure 3.6: `flagdata`: Flagging example using the clip facility.

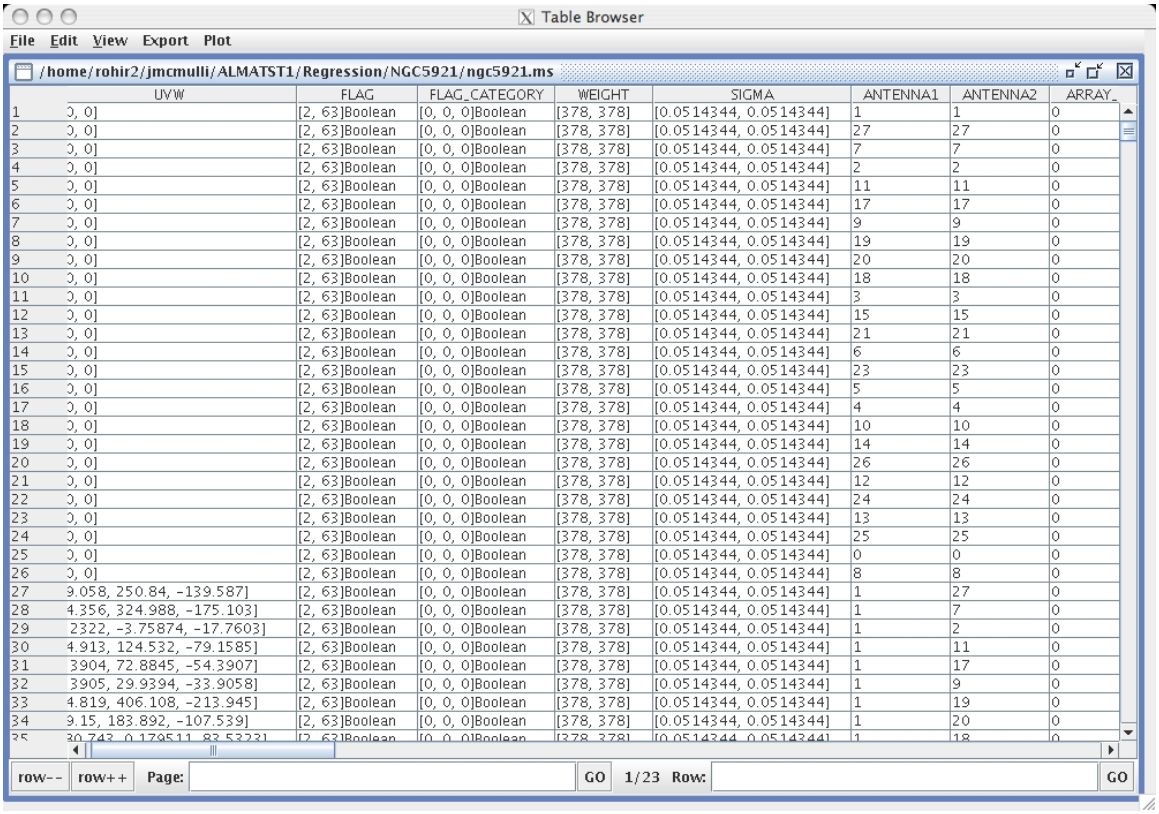



Figure 3.7: `browsetable`: The browser displays the main table within a frame. Hit the expand button to fill the browser frame (this has been done for this figure). You can scroll through the data (x=columns of the `MAIN` table, and y=the rows) or select a specific page or row as desired.

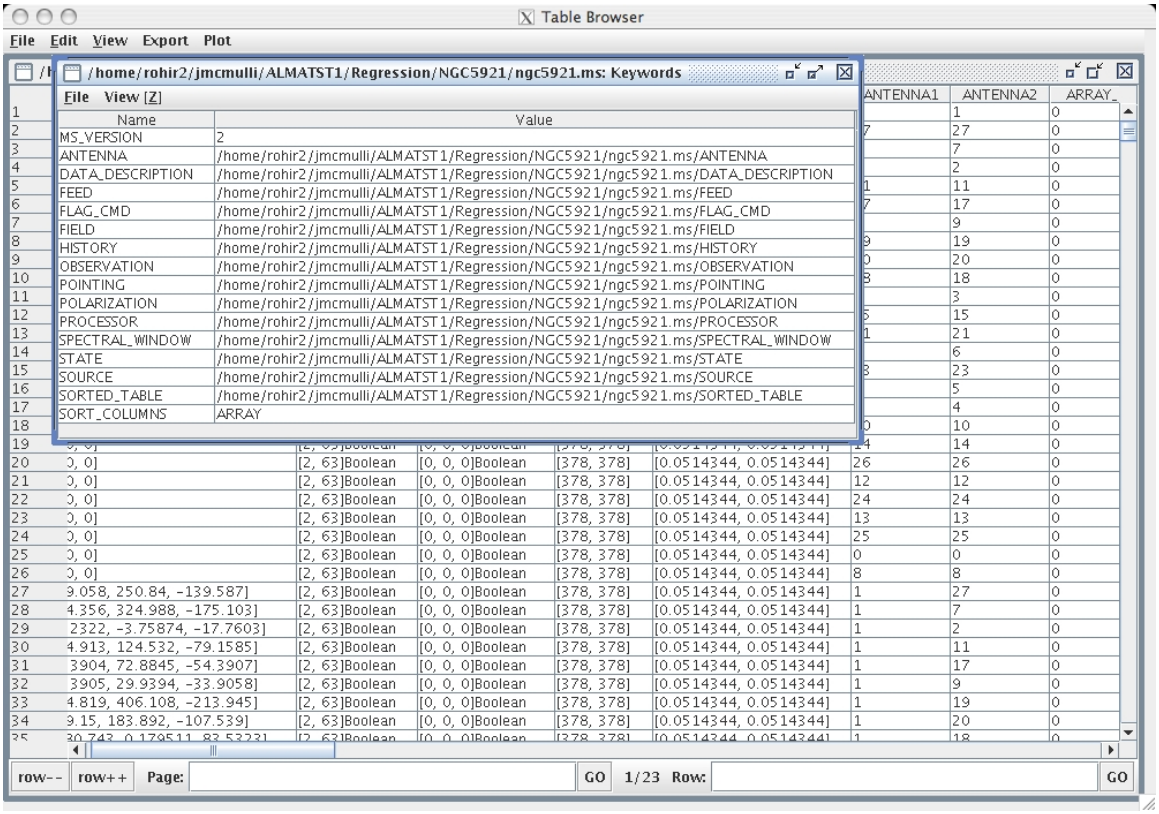



Figure 3.8: `browsetable`: You can use the Menu option **View** to look at other tables within an MS. If you select on **View:Table Keywords** you get the image displayed. You can then select on a table to view its contents.

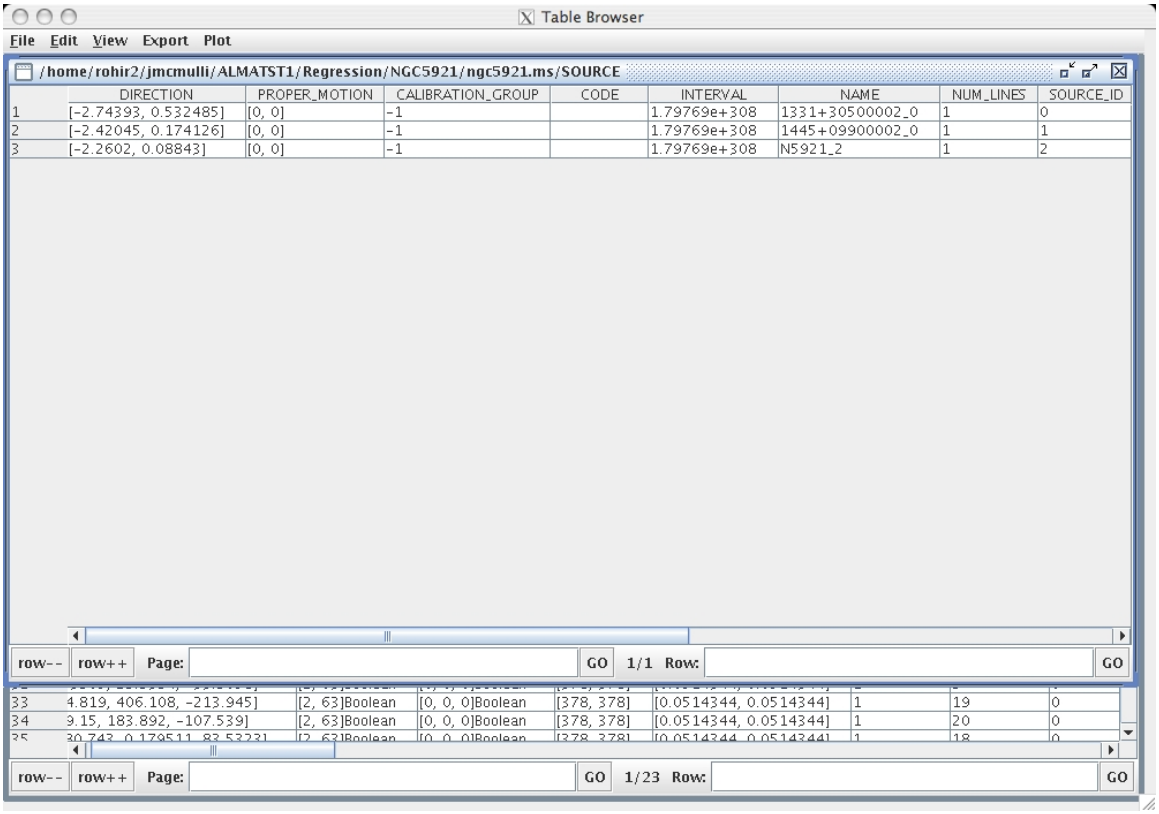



Figure 3.9: `browsetable`: View the `SOURCE` table of the MS.

# 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 philosophy behind the CASA calibration methodology (see § 4.2.1 for more on this). For the most part, calibration in CASA using the tasks is not too different than calibration in other packages such as AIPS or Miriad, so the user should not be alarmed by cosmetic differences such as task and parameter names!

**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 (§ 4.5.4),
- `applycal` — Apply calculated calibration solutions (§ 4.6.1),
- `bandpass` — B calibration solving; supports pre-apply of other calibrations (§ 4.4.2),
- `clearcal` — Re-initialize visibility data set calibration data (§ 4.6.3),
- `fluxscale` — Bootstrap the flux density scale from standard calibration sources (§ 4.4.4),
- `gaincal` — G calibration solving; supports pre-apply of other calibrations (§ 4.4.3),
- `listcal` — list calibration solutions (§ 4.5.2),
- `plotcal` — Plot calibration solutions (§ 4.5.1),
- `setjy` — Compute the model visibility for a specified source flux density (§ 4.3.4),

- `smoothcal` — Smooth calibration solutions derived from one or more sources (§ 4.5.3),
- `split` — Write out new MS containing calibrated data from a subset of the original MS (§ 4.7.1).

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

- `blcal` — *baseline-based* gain or bandpass calibration; supports pre-apply of other calibrations (§ 4.4.6),
- `fringecal` — *Experimental: baseline-based* fringe-fitting calibration solving; supports pre-apply of other calibrations (§ 4.4.7),
- `uvcontsub` — uv-plane continuum fitting and subtraction (§ 4.7.2),
- `uvmodelfit` — Fit a component source model to the uv data (§ 4.7.3).

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 what is 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 The Calibration Process — Outline and Philosophy

A work-flow diagram for CASA calibration of interferometry data is shown in Figure 4.1. This should help you chart your course through the complex set of calibration steps. In the following sections, we will detail the steps themselves and explain how to run the necessary tasks and tools.

This can be broken down into a number of discrete phases:

- **Prior Calibration** — set up previously known calibration quantities that need to be pre-applied, such as the flux density of calibrators, antenna gain-elevation curves, and atmospheric models. Use the `setjy` task (§ 4.3.4), and set the `gaincurve` (§ 4.3.2) and `opacity` (§ 4.3.3) parameters in subsequent tasks;
- **Bandpass Calibration** — solve for the relative gain of the system over the frequency channels in the dataset (if needed), having pre-applied the prior calibration. Use the `bandpass` task (§ 4.4.2);
- **Gain Calibration** — solve for the gain variations of the system as a function of time, having pre-applied the bandpass (if needed) and prior calibration. Use the `gaincal` task (§ 4.4.3);

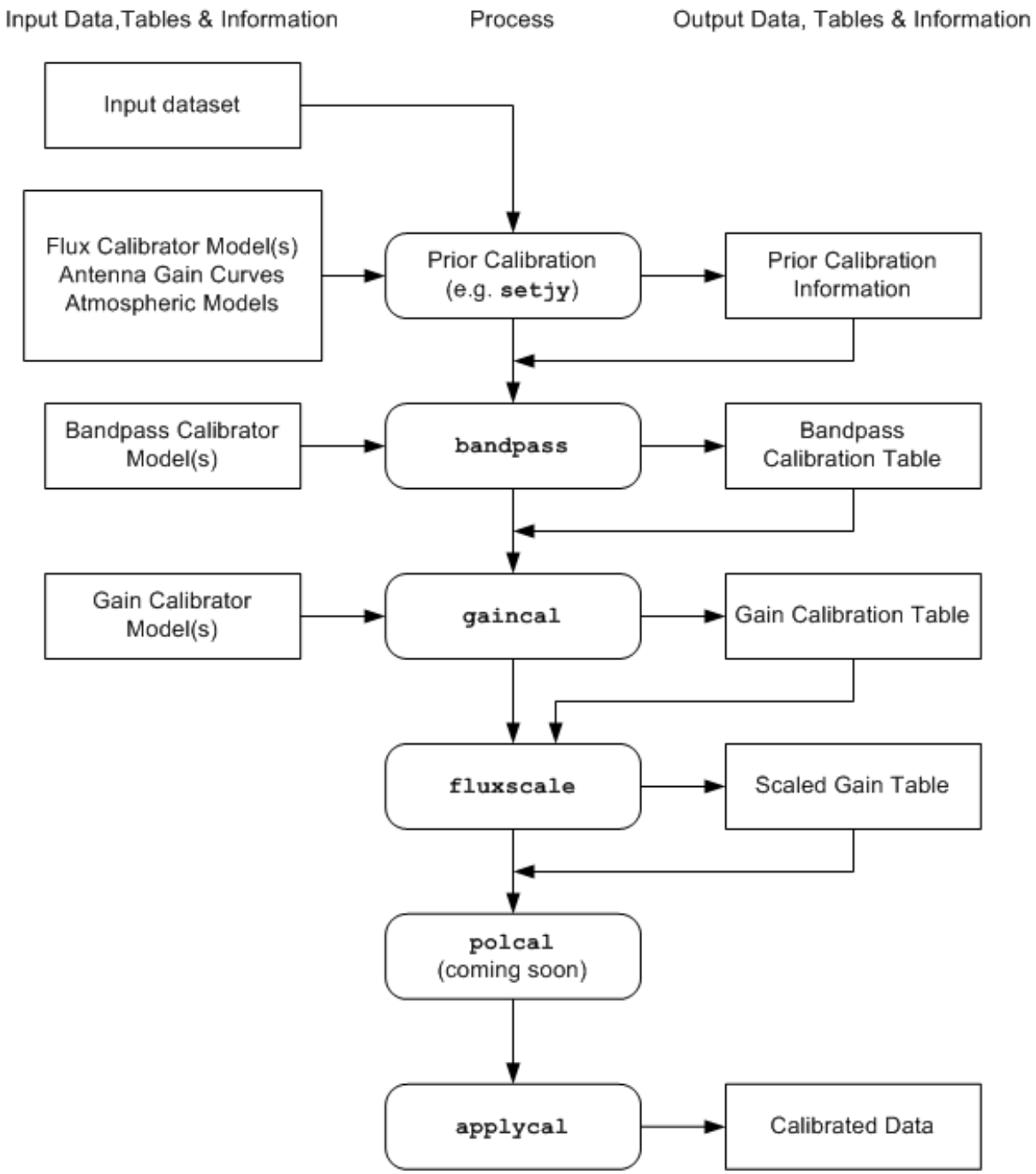



Figure 4.1: Flow chart of synthesis calibration operations. Not shown are use of table manipulation and plotting tasks `accum`, `plotcal`, and `smoothcal` (see Figure 4.2). The polarization calibration task listed here as `polcal` is still under development.

- **Polarization Calibration** — solve for any unknown polarization leakage terms. **BETA ALERT:** Polarization Calibration tasks are not yet available in this Beta Release;
- **Establish Flux Density Scale** — if only some of the calibrators have know flux densities, then rescale gain solutions and derive flux densities of secondary calibrators. Use the `fluxscale` task (§ 4.4.4);
- **Manipulate, Accumulate, and Iterate** — if necessary, accumulate different calibration solutions (tables), smooth, and interpolate/extrapolate onto different sources, bands, and times. Use the `accum` (§ 4.5.4) and `smoothcal` (§ 4.5.3) tasks;
- **Examine Calibration** — at any point, you can (and should) use `plotcal` (§ 4.5.1) and/or `listcal` (§ 4.5.2) to look at the calibration tables that you have created;
- **Apply Calibration to the Data** — this can be forced explicitly by using the `applycal` task (§ 4.6.1), and can be undone using `clearcal` (§ 4.6.3);
- **Post-Calibration Activities** — this includes the determination and subtraction of continuum signal from line data, the splitting of data-sets into subsets (usually single-source), and other operations (such as model-fitting). Use the `uvcontsub` (§ 4.7.2), `split` (§ 4.7.1), and `uvmodelfit` (§ 4.7.3) tasks.

The flow chart and the above list are in a suggested order. However, the actual order in which you will carry out these operations is somewhat fluid, and will be determined by the specific data-reduction use cases you are following. For example, you may need to do an initial **Gain Calibration** on your bandpass calibrator before moving to the **Bandpass Calibration** stage. Or perhaps the polarization leakage calibration will be known from prior service observations, and can be applied as a constituent of Prior Calibration.

### 4.2.1 The Philosophy of Calibration in CASA

Calibration 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), described in Appendix E. 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.

There are a number of "physical" components to calibration in CASA:

- **data** — in the form of the Measurement Set (§ 2.1). The MS includes a number of columns that can hold calibrated data, model information, and weights;
- **calibration tables** — these are in the form of standard CASA tables, and hold the calibration solutions (or parameterizations thereof);

- **task parameters** — sometimes the calibration information is in the form of CASA task parameters that tell the calibration tasks to turn on or off various features, contain important values (such as flux densities), or list what should be done to the data.

At its most basic level, Calibration in CASA is the process of taking "uncalibrated" **data**, setting up the operation of calibration tasks using **parameters**, solving for new calibration **tables**, and then applying the calibration tables to form "calibrated" **data**. Iteration can occur as necessary, with the insertion of other non-calibration steps (e.g. "self-calibration" via imaging).

### 4.2.2 Keeping Track of Calibration Tables

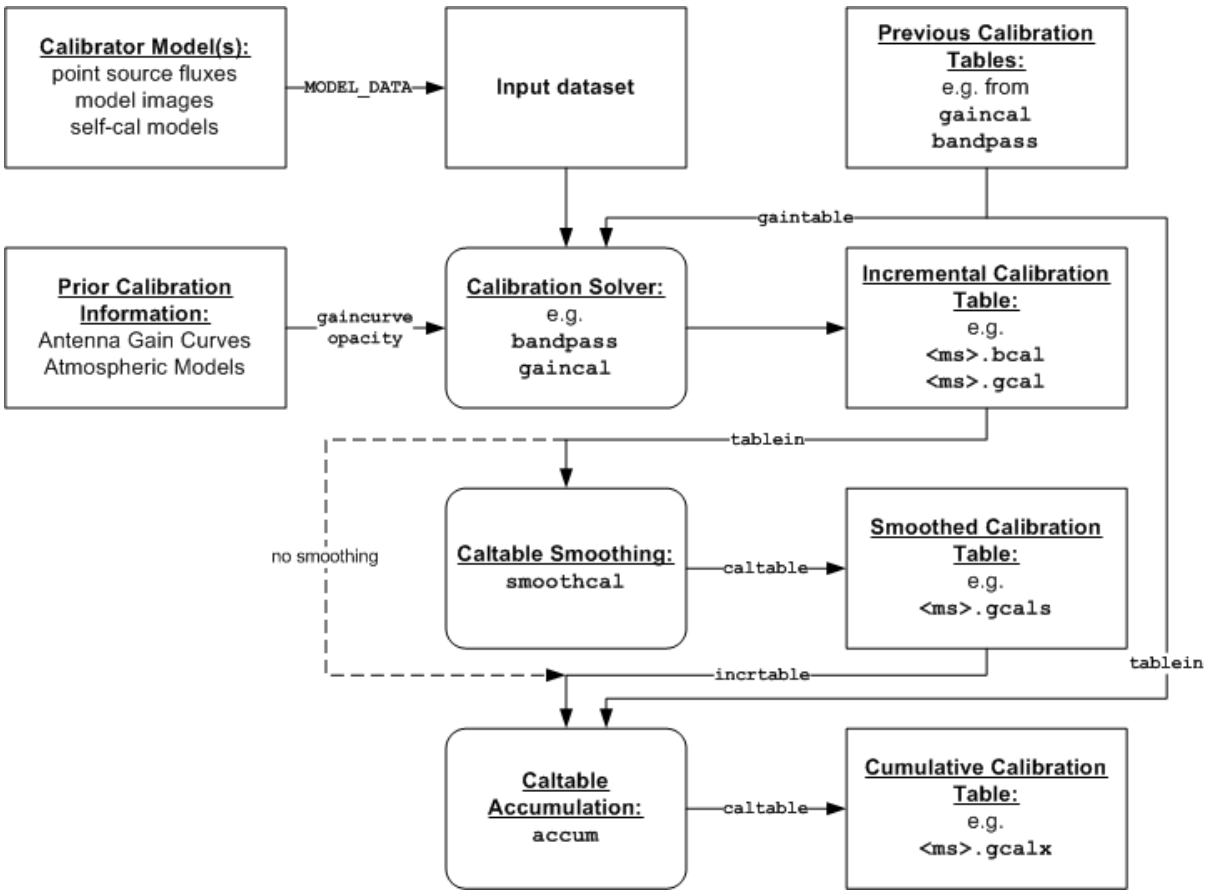



Figure 4.2: Chart of the table flow during calibration. The parameter names for input or output of the tasks are shown on the connectors. Note that from the output solver through the accumulator only a single calibration type (e.g. `'B'`, `'G'`) can be smoothed, interpolated or accumulated at a time. The final set of cumulative calibration tables of all types are then input to `applycal` as shown in Figure 4.1.

The calibration tables are the currency that is exchanged between the calibration tasks. The "solver" tasks (`gaincal`, `bandpass`, `blcal`, `fringecal`) take in the MS (which may have a calibration model in the `MODEL_DATA` column from `setjy` or `ft`) and previous calibration tables, and will output an "incremental" calibration table (it increments the previous calibration, if any). This table can then be smoothed using `smoothcal` if desired.

You can accumulate the incremental calibration onto previous calibration tables with `accum`, which will then output a cumulative calibration table. This task will also interpolate onto a different time scale. See § 4.5.4 for more on accumulation and interpolation.

Figure 4.2 graphs the flow of these tables through the sequence

```
solve   =>   smooth   =>   accumulate
```

Note that this sequence applied to separate *types* of tables (e.g. `'B'`, `'G'`) although tables of other types can be previous calibration input to the solver.

The final set of cumulative calibration tables is what is applied to the data using `applycal`. You will have to keep track of which tables are the intermediate incremental tables, and which are cumulative, and which were previous to certain steps so that they can also be previous to later steps until accumulation. This can be a confusing business, and it will help if you adopt a consistent table naming scheme (see Figure 4.2) for an example naming scheme).

### 4.2.3 The Calibration of VLA data in CASA

CASA supports the calibration of classic VLA data (taken before the Modcomp turn-off in late June 2007) that is imported from the Archive through the `importvla` task. See § 2.2.2 for more information.

You can also import VLA data in UVFITS format with the `importuvfits` task (§ 2.2.1.1). However, in this case, you must be careful during calibration in that some prior or previous calibrations (see below) may or may not have been done in AIPS and applied (or not) before export.

For example, the default settings of AIPS `FILLM` will apply VLA gaincurve and approximate (weather-based) atmospheric optical depth corrections when it generates the extension table `CL 1`. If the data is exported immediately using `FITTP`, then this table is included in the UVFITS file. However, CASA is not able to read or use the AIPS `SN` or `CL` tables, so that prior calibration information is lost and must be applied during calibration here (ie. using `gaincurve=True` and setting the `opacity` parameter).

On the other hand, if you apply calibration in AIPS by using the `SPLIT` or `SPLAT` tasks to apply the `CL` tables before exporting with `FITTP`, then this calibration will be in the data itself. In this case, you do not want to re-apply these calibrations when processing in CASA.

**BETA ALERT:** We do not recommend importing using `importvla` data obtained with the VLA after June 2007 (after the VLA Modcomp was turned off and the array was switched to the new EVLA system). We do not yet support the correct scaling of the data and weights by $T_{sys}$ in this case. You should import into AIPS first (see § 2.2.2).

## 4.3 Preparing for Calibration

There are a number of "a priori" calibration quantities that may need to be applied to the data before further calibration is carried out. These include

- **system temperature correction** — turn correlation coefficient into correlated flux density (necessary for some telescopes),
- **gain curves** — antenna gain-elevation dependence,
- **atmospheric optical depth** — attenuation of the signal by the atmosphere, correcting for its elevation dependence.
- **flux density models** — establish the flux density scale using "standard" calibrator sources, with models for resolved calibrators,

These are pre-determined effects and should be applied (if known) before solving for other calibration terms. If unknown, then they will need to be solved for as one of the standard calibration types (gain or bandpass).

We now deal with these in turn.

### 4.3.1 System Temperature Correction

Some telescopes, including the EVLA and the VLBA, record the visibilities in the form of raw *correlation coefficient* with weights proportional to the number of bits correlated. The correlation coefficient is the fraction of the total signal that is correlated, and thus multiplication by the system temperature and the antenna gain (in Jy/K) will produce visibilities with units of correlated flux density. Note that the old VLA system did this initial calibration on-line, and ALMA will also provide some level of on-line calibration (TBD).

**BETA ALERT:** There is as yet no mechanism available in `importvla` or in the calibration tasks to use the system temperature information provided by the VLA/EVLA on-line system to calibrate EVLA or VLBA data in raw form. This includes VLA data taken after the Modcomp turn-over in late June 2007. You may pass the data through AIPS first. You can also just forge ahead with standard calibration. The drawback to this is that short-term changes in $T_{sys}$ which are not tracked by calibrator observations or self-calibration will remain in the data.

### 4.3.2 Antenna Gain-Elevation Curve Calibration

Large antennas (such as the 25-meter antennas used in the VLA and VLBA) have a forward gain and efficiency that changes with elevation. 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.

Gain curve calibration has been implemented in CASA for the VLA (only), with gain curve polynomial coefficients available directly from the CASA data repository. To make gain curve corrections for VLA data, set `gaincurve=True` for any of the calibration tasks.

**BETA ALERT:** The `gaincurve` parameter must be supplied to any calibration task that allows pre-application of the prior calibration (e.g. `bandpass`, `gaincal`, `applycal`). This should be done consistently through the calibration process. In future updates we will likely move to a separate task to calibrate the gain curve.

For example, to pre-apply the gaincurve during gain calibration:

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

**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`, `applycal`, etc.), the gain curve data appropriate to the observing frequencies will be automatically retrieved from the data repository and applied.

**BETA ALERT:** Currently, gain-curves are available for VLA data only. The application of the gain-curves, if `gaincurve=True`, is allowed only if the VLA is set as the telescope of observation in the MS, otherwise an error will be generated. Set `gaincurve=False` if you are not using VLA data. A general mechanism for incorporating gaincurve information for other arrays will be made available in future releases. Also note that the VLA gain-curves are the most recent ones (that are also supplied in AIPS). Caution should be used in applying these gaincurve corrections to VLA data taken before 2001, as antenna changes were poorly tracked previous to this time. We will include gain curves for EVLA antennas when those are measured and become available.

### 4.3.3 Atmospheric Optical Depth 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). This is then supplied to the `opacity` parameter in the calibration tasks.

**BETA ALERT:** The `opacity` parameter must be supplied to any calibration task that allows pre-application of the prior calibration (e.g. `bandpass`, `gaincal`, `applycal`). This should be done consistently through the calibration process. In future updates we will likely move to a separate task to calibrate the atmospheric optical depth.

For example, if the zenith optical depth is 0.1 nepers, then use the following parameters:

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

**BETA ALERT:** Currently, you can only supply a single value of `opacity`, which will then be pre-applied to whatever calibration task that you set it in. 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.

#### 4.3.3.1 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 optical depth value is `opacity=0.060` for `gaincal`.

### 4.3.4 Setting the Flux Density Scale using (`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).

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 |

The inputs for setjy are:

```
#  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
```

**BETA ALERT:** At this time, all that setjy does is to fill the MODEL_DATA column of the MS with the Fourier transform of a source model. The ft task (§ 5.8) will do the same thing, although it does not offer the options for flux rescaling that setjy does.

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
```

**Note:** The `setjy` (or `ft`) operation is different than the antenna gain-elevation and atmospheric opacity Prior Calibrations (§ 4.3.2–4.3.3) in that it is applied to (and carried with) the MS itself, rather than via other tables or parameters to the subsequent tasks. It is more like the Tsys correction (§ 4.3.1) in this regard.

#### 4.3.4.1 Using Calibration Models for Resolved Sources

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:

- Red Hat Linux RPMs (RHE4, Fedora 6): 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).

Note that `setjy` will rescale the flux in the models for known sources (e.g. those in Table 4.1) to match those it would have calculated. It will thus extrapolated the flux out of the frequency band of the model image to whatever spectral windows in the MS are specified (but will use the structure of the source in the model image).

**BETA ALERT:** The reference position in the `modimage` is currently used by `setjy` when it does the Fourier transform, thus differences from the positions for the calibrator in the MS will show

up as phase gradients in the uv-plane. If your model image position is significantly different but you don't want this to affect your calibration, then you can doctor either the image header using `imhead` (§ 6.1) or in the MS (using the `ms` tool) as appropriate. In an upcoming Beta patch we will put in a toggle to use or ignore the position of the `modimage`. Note that this will not affect the flux scaling (only put in erroneous model phases); in any event small position differences, such as those arising by changing epoch from B1950 to J2000 using `regridimage` (§ 6.3), will be inconseqential to the calibration.

This illustrates the use of `uvrange` for a slightly resolved calibrator:

```
# 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 uvrange in solve

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

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

# Append phase-calibrator's solutions (no uvrange) to the same table
gaincal(vis='ngc7538_XBAND.ms', caltable='cal.G', field='2',
        solint=60.0, refant=10, uvrange=[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)
```

while the following illustrates the use of of a model:

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

# Copy the model image 3C48_X.im to the working directory first!

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

# Solutions on both calibrators with no uvrange
gaincal(vis='ngc7538_XBAND.ms', caltable='cal.G2', field='0,2',
        solint=60.0, refant=10, uvrange=[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
```

### 4.3.5 Other *a priori* Calibrations and Corrections

Other *a priori* calibrations will be added to the `calibrater` (`cb`) 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.

## 4.4 Solving for Calibration — Bandpass, Gain, Polarization

These tasks actually solve for the unknown calibration parameters, placing the results in a calibration table. They take as input an MS, and a number of parameters that specify any prior calibration or previous calibration tables to pre-apply before computing the solution. These are placed in the proper sequence of the Measurement Equation automatically.

We first discuss the parameters that are in common between many of the calibration tasks. Then we describe each solver in turn.

### 4.4.1 Common Calibration Solver Parameters

There are a number of parameters that are in common between the calibration "solver" tasks. These also appear in some of the other calibration manipulation and application tasks.

#### 4.4.1.1 Parameters for Specification : `vis` and `caltable`

The input measurement set and output table are controlled by the following parameters:

```
vis          =         ''   #   Name of input visibility file
caltable     =         ''   #   Name of output calibration table
```

The MS name is input in `vis`. If it is highlighted red in the inputs (§ 1.3.3.3) then it does not exist, and the task will not execute. Check the name and path in this case.

The output table name is placed in `caltable`. Be sure to give a unique name to the output table, or be careful. If the table exists, then what happens next will depend on the task and the values of other parameters (e.g. § 4.4.1.6). The task may not execute giving a warning that the table already exists, or will go ahead and overwrite the solutions in that table, or append them. Be careful.

#### 4.4.1.2 Selection: field, spw, and selectdata

Selection is controlled by the parameters:

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

Field and spectral window selection are so often used, that we have made these standard parameters `field` and `spw` respectively.

The `selectdata` parameter expands as usual, uncovering other selection sub-parameters:

```
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)
```

Note that if `selectdata=False` these parameters are not used when the task is executed, even if set underneath.

See § 2.5 for more on the selection parameters.

#### 4.4.1.3 Prior Calibration: gaincurve and opacity

There are two control parameters for applying Prior Calibration:

```
gaincurve    =      False   #   Apply VLA antenna gain curve correction
opacity      =        0.0   #   Opacity correction to apply (nepers)
```

See § 4.3 for more on **Prior Calibration**.

#### 4.4.1.4 Previous Calibration: gaintable, gainfield, interp and spwmap

Calibration tables that have already been determined can also be applied before solving for the new table:

```
gaintable    =         ''   #   Prior gain calibration table(s) to apply
gainfield    =         ''   #   Field selection on prior gaintable(s)
interp       =         ''   #   Interpolation mode (in time) for prior gaintable(s)
spwmap       =         []   #   Spectral window mapping for each gaintable (see help)
```

This is controlled by the `gaintable` parameter, which takes a string or list of strings giving one or more calibration tables to pre-apply. For example,

```
gaintable = ['ngc5921.bcal','ngc5921.gcal']
```

specifies two tables, in this case bandpass and gain calibration tables respectively.

The other parameters key off `gaintable`, taking single values or lists, with an entry for each table in `gaintable`. The order is given by that in `gaintable`.

The `gainfield` parameter specifies which fields from the respective `gaintable` to use to apply. This is a list, with each entry a string or list of strings. The default `''` for an entry means to use all in that table. For example,

```
gaintable = ['ngc5921.bcal','ngc5921.gcal']
gainfield = [ '1331+305', ['1331+305','1445+099'] ]
```

or using indices

```
gainfield = [ '0', ['0','1'] ]
```

to specify the field `'1331+305'` from the table `'ngc5921.bcal'` and fields `'1331+305'` and `'1445+099'` from the second table 'ngc5921.gcal'. We could also have wildcarded the selection, e.g.

```
gainfield = [ '0', '*' ]
```

taking all fields from the second table. And of course we could have used the default

```
gainfield = [ '0', '' ]
```

or even

```
gainfield = [ '0' ]
```

which is to take all.

The `interp` parameter chooses the interpolation scheme to be used when pre-applying the solution in the tables. This interpolation is (currently) only in time. The choices are currently `'nearest'`, `'linear'`, and `'aipslin'`:

- `'nearest'` just picks the entry nearest in time to the visibility in question;
- `'linear'` interpolation calibrates each datum with calibration phases and amplitudes linearly interpolated from neighboring time values. In the case of phase, this mode will assume that phase jumps greater than 180° between neighboring points indicate a cycle slip, and the interpolated value will follow this change in cycle accordingly;

- `'aipslin'` emulates the classic AIPS interpolation mode with linearly interpolated amplitudes and phases derived from interpolation of the complex calibration values. While this method avoids having to track cycle slips (which is unstable for solutions with very low SNR), it will yield a phase interpolation which becomes increasingly non-linear as the spanned phase difference increases. The non-linearity mimics the behavior of `interp='nearest'` as the spanned phase difference approaches 180° (the phase of the interpolated complex calibration value initially changes very slowly, then rapidly jumps to the second value at the midpoint of the interval).

If the uncalibrated phase is changing rapidly, a `'nearest'` interpolation is not desirable. Usually, `interp='linear'` is the best choice. For example,

```
interp = [ 'nearest', 'linear' ]
```

uses nearest "interpolation" on the first table, and linear on the second.

The `spwmap` parameter sets the spectral window combinations to form for the `gaintable`(s). This is a list, or a list of lists, of integers giving the `spw` IDs to map. There is one list for each table in `gaintable`, with an entry for each ID in the MS. For example,

```
spwmap=[0,0,1,1]                # apply from spw=0 to 0,1 and 1 to 2,3
```

for an MS with `spw=0,1,2,3`. For multiple `gaintable`, use lists of lists, e.g.

```
spwmap=[ [0,0,1,1], [0,1,0,1] ] # 2nd table spw=0 to 0,2 and 1 to 1,3
```

**BETA ALERT:** This scheme for mapping the pre-apply tables is not particularly elegant, particularly for `spwmap`. This may change in the future.

#### 4.4.1.5 Solving: `solint`, `refant`, and `minsnr`

The parameters controlling common aspects of the solution are:

```
solint       =   864000.0   #   Solution interval (sec); 0=scan,-1=each datum
refant       =         ''   #   Reference antenna name or ID number:''=no explicit reference
minsnr       =        0.0   #   Reject solutions below this SNR: 0==>no rejection
```

The solution interval is given by `solint`. This is in seconds. The special values `0` and `-1` specify an interval of scan and visibility respectively.

The reference antenna is specified by the `refant` parameter. This useful to "lock" the solutions with time, effectively rotating (after solving) the phase of the gain solution for the reference antenna to be zero (the exact effect depends on the type of solution). You can also run without a reference antenna, but in this case the solutions will float with time, with a phase that rotates around with the relative weights of the antennas in the solution (its more or less like setting the weighted sum of the antenna phases to zero). It is usually prudent to select an antenna in the center of the

array that is known to be particularly stable, as any gain jumps or wanders in the `refant` will be transferred to the other antenna solutions.

The minimum signal-to-noise ratio allowed for an acceptable solution is specified in the `minsnr` parameter. Not all tasks have this one.

#### 4.4.1.6 Action: `append` and `solnorm`

The following parameters control some things that happen after solutions are obtained:

```
solnorm      =      False   #   Normalize solution amplitudes post-solve.
append       =      False   #   Append solutions to (existing) table.  False will overwrite.
```

The `solnorm` parameter toggles on the option to normalize the solution amplitudes after the solutions are obtained. The exact effect of this depends upon the type of solution. Not all tasks include this parameter.

One should be aware when using `solnorm` that if this is done in the last stage of a chain of calibration, then the part of the calibration that is "normalized" away will be lost. It is best to use this in early stages (for example in a first bandpass calibration) so that later stages (such as final gain calibration) can absorb the lost normalization scaling. It is not strictly necessary to use `solnorm=True` at all, but is sometimes helpful if you want to have a normalized bandpass for example.

The `append` parameter, if set to `True`, will append the solutions from this run to existing solutions in `caltable`. Of course, this only matters if the table already exists. If `append=False` and `caltable` exists, it will overwrite.

### 4.4.2 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 the gain 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:

```
#  bandpass :: Calculate a bandpass solution

vis          =         ''   #   Name of input visibility file
caltable     =         ''   #   Name of output bandpass calibration table
field        =         ''   #   field names or index of calibrators: ''==>all
spw          =         ''   #   spectral window:channels: ''==>all
selectdata   =      False   #   Other data selection parameters
solint       =   864000.0   #   Solution interval (sec), default=10days
refant       =         ''   #   Reference antenna name or ID number:''=no explicit reference
```

```
solnorm      =      False   #   Normalize bandpass amplitudes and phases
bandtype     =        'B'   #   Type of bandpass solution (B or BPOLY)
append       =      False   #   Append solutions to (existing) table
gaintable    =         ''   #   Prior gain calibration table(s) to apply
gainfield    =         ''   #   Field selection on prior gaintable(s)
interp       =         ''   #   Interpolation mode (in time) for prior gaintable(s)
spwmap       =         []   #   Spectral window mapping for each gaintable (see help)
gaincurve    =      False   #   Apply VLA antenna gain curve correction
opacity      =        0.0   #   Opacity correction to apply (nepers)
async        =      False   #   if True run in the background, prompt is freed
```

Many of these parameters are in common with the other calibration tasks and are described above in § 4.4.1.

The `bandtype` parameter selects the type of solution used for the bandpass. The choices are `'B'` and `'BPOLY'`. The former solves for a complex gain in each channel in the selected part of the MS. See § 4.4.2.2 for more on `'B'`. The latter uses a polynomial as a function of channel to fit the bandpass, and expands further to reveal a number of sub-parameters See § 4.4.2.3 for more on `'BPOLY'`.

It is usually best to solve for the bandpass in channel data before solving for the gain as a function of time. However, if the gains of the bandpass calibrator observations are fluctuating over the timerange of those observations, then it can be helpful to first solve for the gains of that source with `gaincal` , and input these to `bandpass` via `gaintable`. See more below on this strategy.

We now describe the issue of bandpass normalization, followed by a description of the options `bandtype='B'` and `bandtype='BPOLY'`.

#### 4.4.2.1 Bandpass Normalization

The `solnorm` parameter (§ 4.4.1.6) deserves more explanation in the context of the bandpass. Most users are used to seeing a normalized bandpass, where the vector sum of the antenna-based channel gains sums to unity amplitude and zero phase. The toggle `solnorm=True` allows this. However, the parts of the bandpass solution normalized away will be still left in the data, and thus you should not use `solnorm=True` if the `bandpass` calibration is the end of your calibration sequence (e.g. you have already done all the gain calibration you want to). Note that setting `solnorm=True` will NOT rescale any previous calibration tables that the user may have supplied in `gaintable`.

You can safely use `solnorm=True` if you do the bandpass first (perhaps after a throw-away initial gain calibration) as we suggest above in § 4.2, as later gain calibration stages will deal with this remaining calibration term. This does have the benefit of isolating the overall (channel independent) gains to the following `gaincal` stage. It is also recommended for the case where you have multiple scans on possibly different bandpass calibrators. It may also be preferred when applying the bandpass before doing `gaincal` and then `fluxscale` (§ 4.4.4), as significant variation of bandpass among antennas could otherwise enter the gain solution and make (probably subtle) adjustments to the flux scale.

We finally note that `solnorm=False` at the bandpass step in the calibration chain will in the end produce the correct results. It only means that there will be a part of what we usually think of the gain calibration inside the bandpass solution, particularly if `bandpass` is run as the first step.

#### 4.4.2.2 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 frequency-independent time-scale variations.

The `'B'` solutions are limited by the signal-to-noise ratio 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 an initial `'G'` or `'T'` solve using `gaincal` (§ 4.4.3). 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.

For example, to solve for a `'B'` bandpass using a single short scan on the calibrator, then

```
default('bandpass')

vis = 'n5921.ms'
caltable = 'n5921.bcal'
gaintable = ''                    # No gain tables yet
gainfield = ''
interp = ''
field = '0'                       # Calibrator 1331+305 = 3C286 (FIELD_ID 0)
spw = ''                          # all channels
selectdata = False                # No other selection
gaincurve = False                 # No gaincurve at L-band
opacity = 0.0                     # No troposphere
bandtype = 'B'                    # standard time-binned B (rather than BPOLY)
solint = 86400.0                  # set solution interval arbitrarily long
refant = '15'                     # ref antenna 15 (=VLA:N2) (ID 14)

bandpass()
```

On the other hand, we might have a number of scans on the bandpass calibrator spread over time, but we want a single bandpass solution. In this case, we could solve for and then pre-apply an initial gain calibration,

```
gaintable = 'n5921.init.gcal'     # Our previously determined G table
gainfield = '0'
interp = 'linear'                 # Do linear interpolation
```

Note that we obtained a bandpass solution for all channels in the MS. If explicit channel selection is desired, for example some channels are useless and can be avoided entirely (e.g. edge channels or those dominated by Gibbs ringing), then `spw` can be set to select only these channels, e.g.

```
spw = '0:4~59'                  # channels 4-59 of spw 0
```

This is not so critical for `'B'` solutions as for `'BPOLY'`, as each channel is solved for independently, and poor solutions can be dropped.

If you have multiple time solutions, then these will be applied using whatever interpolation scheme is specified in later tasks.

**BETA ALERT:** The `'B'` solutions will allow you to use multiple `field`s, but in this case `bandpass` will produce different solutions for each source. i.e. you cannot average in time across different fields. Note that this currently provides a safety net of sorts because you should not average across fields unless the phase has already been corrected (e.g. in an initial `gaincal`). In the future it will be possible to do this but then `'BPOLY'` caveat below will then hold for `'B'` solutions obtained using this option as well in the multiple field case.

#### 4.4.2.3 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 `bandtype='BPOLY'` solver. The `'BPOLY'` solver naturally enough fits (Chebychev) 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 selection. 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).

The `'BPOLY'` solver requires a number of unique sub-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
     visnorm    =      False   #   Normalize data prior to BPOLY 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). The `maskedge` drops beginning and end channels. The `visnorm` parameter turns on normalization before the solution is obtained (rather than after for `solnorm`).

**BETA ALERT:** Note that currently, `'BPOLY'` solutions cannot be solved for in a time-dependent manner. Furthermore, `bandpass` will allow you to use multiple `field`s, but will determine a single solution for all specified fields. If you want to use more than one field in the solution it is prudent to use an initial `gaincal` and use this table as an input to bandpass because in general the phase towards two (widely separated) sources will not be sufficiently similar to combine them. If you

do not include amplitude in the initial `gaincal`, you probably want to set `visnorm=True` also to take out the amplitude normalization change. Note also in the case of multiple `field`s, that the `'BPOLY'` solution will be labeled with the field ID of the first `field` used in the `'BPOLY'` solution, so if for example you point `plotcal` at the name or ID of one of the other fields used in the solution, plotcal does not plot.

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 calibrater (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 may also be undertaken for each spectral window separately, or all the tables included in `gaintable` during later operations.

**BETA ALERT:** Once you do a separate `bandpass` run for different fields (making separate tables, you will need to continue keeping the calibration for these fields separate (in `gaincal` etc.) as they cannot be currently recombined later. Because of this complication, we recommend doing `bandpass` with `'BPOLY'` on a single field only at this time.

### 4.4.3 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 polarizations. 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").

It is best to have removed a (slowly-varying) "bandpass" from the frequency channels by solving for the bandpass (see above). Thus, the `bandpass` calibration table would be input to `gaincal` via the `gaintable` parameter (see below).

The `gaincal` task has the following inputs:

```
#  gaincal :: Determine temporal gains from calibrator observations:

vis             =      ''   #   Name of input visibility file
caltable        =      ''   #   Name of output calibration table
field           =      ''   #   field names or index of calibrators ''==>all
spw             =      ''   #   spectral window:channels: ''==>all
selectdata      =   False   #   Other data selection parameters
gaintype        =     'G'   #   Type of solution (G, T, or GSPLINE)
calmode         =    'ap'   #   Type of solution (a,p,ap): amplitude, phase, amp and phase
solint          =     0.0   #   Solution interval (sec); 0 = scan, -1 = each data sample
refant          =      ''   #   Reference antenna name or ID number:''=no explicit reference
minsnr          =     0.0   #   Reject solutions below this SNR: 0==>no rejection
solnorm         =   False   #   Normalize solution amplitudes (G,T) post-solve.
append          =   False   #   Append solutions to (existing) table.  False will overwrite.
gaintable       =      ''   #   Prior gain calibration table(s) to apply
gainfield       =      ''   #   Field selection on prior gaintable(s)
interp          =      ''   #   Interpolation mode (in time) for prior gaintable(s)
spwmap          =      []   #   Spectral window mapping for each gaintable (see help)
gaincurve       =   False   #   Apply VLA antenna gain curve correction
opacity         =     0.0   #   Opacity correction to apply (nepers)
preavg          =    -1.0   #   Sub-solution interval pre-averaging timescale (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). The bulk of the other parameters are the standard solver parameters. See § 4.4.1 above for a description of these.

The `gaintype` parameter selects the type of gain solution to compute. The choices are `'T'`, `'G'`, and `'GSPLINE'`. The `'G'` and `'T'` options solve for independent complex gains in each solution interval (classic AIPS style), with `'T'` enforcing a single polarization-independent gain for each co-polar correlation (e.g. `RR` and `LL`, or `XX` and `YY`) and `'G'` having independent gains for these. See § 4.4.3.1 for a more detailed description of `'G'` solutions, and § 4.4.3.2 for more on `'T'`. The `'GSPLINE'` fits cubic splines to the gain as a function of time. See § 4.4.3.3 for more on this option.

#### 4.4.3.1 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
        gaincurve=True          # Note: gaincurve=False by default
        refant=3)               #

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):

```
applycal('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',datacolum='data',subplot=211)
plotxy('data.ms',xaxis='channel',datacolumn='corrected',subplot=212)
```

#### 4.4.3.2 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.

#### 4.4.3.3 GSPLINE solutions

At high radio frequencies, where tropospheric phase fluctuates rapidly, it is often the case that there is insufficient signal-to-noise ratio to obtain robust `'G'` or `'T'` solutions 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. This fits a time-series of cubic B-splines to the phase and/or amplitude of the calibrator visibilities.

**BETA ALERT:** Unlike ordinary `'G'`, a single common `'GSPLINE'` solution will be determined from data for all selected spectral windows and fields specified in the MS selection parameters, and the resulting solution will be applicable to any field or spectral window in the same Measurement

Set. This behavior is similar to that of the 'BPOLY' in bandpass. If you do want separate spectral window solutions, then you will have to do separate runs of gaincal. 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. The sub-parameters are:

```
gaintype         =  'GSPLINE'   #   Type of solution (G, T, or GSPLINE)
     splinetime  =     3600.0   #   Spline (smooth) timescale (sec), default=1 hours
     npointaver  =          3   #   Points to average for phase wrap (okay)
     phasewrap   =        180   #   Wrap phase when greater than this (okay)
```

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 ($> 180°$) 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).

For example, to solve for 'GSPLINE' phase and amplitudes, with splines of duration 600 seconds,

```
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
```

**BETA ALERT':** The 'GSPLINE' solutions can not yet be used in fluxscale. You should do at least some 'G' amplitude solutions to establish the flux scale, then do 'GSPLINE' in phase before or after to fix up the short timescale variations. Note that the "phase tracking" algorithm in 'GSPLINE' needs some improvement.

### 4.4.4 Establishing the Flux Density Scale (fluxscale)

The 'G' or 'T' solutions obtained from calibrators for which the flux density was unknown and assumed to be 1 Jansky 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:

```
#  fluxscale :: Bootstrap the flux density scale from standard calibrators

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

Before running `fluxscale`, one must have first run `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.

For example, given a `'G'` table, e.g. `'cal.G'`, containing solutions for a flux density calibrator (in this case `'3C286'`) and for one or more gain calibrator sources with unknown flux densities (in this example `'0234+285'` and `'0323+022'`):

```
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 flux-scaled (see § 4.3 and § 4.4.1.3). 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 and one was solved relative to the other (use fluxscale only on the first of the two).

**BETA ALERT:** The `'GSPLINE'` option is not yet supported in `fluxscale` (see § 4.4.3.3).

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 functions similarly to the standard `spwmap` parameter (§ 4.4.1.4), and takes a list 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`.

**Note:** You should be careful when you have a dataset with spectral windows with different bandwidths, and you have observed the calibrators differently in the different `spw`. The flux-scaling will probably be different in windows with different bandwidths.

For example,

```
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.

#### 4.4.4.1 Using Resolved 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 `fluxscale` 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 0) 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 1) 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 uvrange
          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. Note that one can nominally get by only with the `uvrange` selection, but you may find that you get strange effects from some antennas only having visibilities to a subset of the baselines and thus causing problems in the solving.

### 4.4.5 Instrumental Polarization Calibration (D)

**BETA ALERT:** 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.

### 4.4.6 Baseline-based Calibration (`blcal`)

**BETA ALERT:** The `blcal` task has not had extensive testing, and is included as part of our support for the ALMA commissioning effort.

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      =      False   #   Activate data selection details
freqdep         =      False   #   Solve for frequency dependent solutions
solint          =        0.0   #   Solution interval (sec)
gaintable       =         ''   #   Prior gain calibration table(s) to apply
gainfield       =         ''   #   Field selection on prior gaintable(s)
interp          =         ''   #   Interpolation mode (in time) for prior gaintable(s)
spwmap          =         []   #   Spectral window mapping for each gaintable (see help)
gaincurve       =      False   #   Apply VLA antenna gain curve correction
opacity         =        0.0   #   Opacity correction to apply (nepers)
async           =      False   #
```

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. These common calibration parameters are described in § 4.4.1.

### 4.4.7 EXPERIMENTAL: Fringe Fitting (`fringecal`)

**BETA ALERT:** The `fringecal` task has not had extensive testing, and is included as part of our support for the ALMA commissioning effort.

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   =      False   #   Activate data selection details
gaincurve    =      False   #   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
```

All of the `fringecal` parameters are common calibration parameters as described in § 4.4.1.

**BETA ALERT:** Note that `plotcal` cannot currently display `'delay'` or `delayrate` solutions from `fringecal`.

## 4.5 Plotting and Manipulating Calibration Tables

At some point, the user should examine (plotting or listing) the calibration solutions. Calibration tables can also 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.

### 4.5.1 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:

```
#  plotcal :: An all-purpose plotter for calibration results:

caltable     =         ''   #   Name of input calibration table
xaxis        =         ''   #   Value to plot along x axis (time,chan,amp,phase,real,imag,snr)
yaxis        =      'amp'   #   Value to plot along y axis (amp,phase,real,imag,snr)
poln         =         ''   #   Polarization to plot (RL,R,L,XY,X,Y,/)
field        =         ''   #   Field names or index: ''=all, '3C286,P1321*', '0~3'
antenna      =         ''   #   Antenna selection.  E.g., antenna='3~5'
spw          =         ''   #   Spectral window: ''=all, '0,1' means spw 0 and 1
timerange    =         ''   #   Time selection ''=all
subplot      =        111   #   Panel number on display screen (yxn)
overplot     =      False   #   Overplot solutions on existing display
iteration    =         ''   #   Iterate on antenna,time,spw,field
plotrange    =         []   #   plot axes ranges: [xmin,xmax,ymin,ymax]
showflags    =      False   #   If true, show flags
plotsymbol   =        '.'   #   pylab plot symbol
plotcolor    =     'blue'   #   initial plotting color
markersize   =        5.0   #   size of plot symbols
fontsize     =       10.0   #   size of label font
```

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

The `xaxis` and `yaxis` plot options available are:

- `'amp'` — amplitude,
- `'phase'` — phase,
- `'real'` – the real part,
- `'imag'` — the imaginary part,

- 'snr' – the signal-to-noise ratio,

of the calibration solutions that are in the caltable. The xaxis choices also include 'time' and 'channel' which will be used as the sensible defaults (if xaxis='') for gain and bandpass solutions respectively.

The poln parameter determines what polarization or combination of polarization is being plotted. The poln='RL' plots both R and L polarizations on the same plot. The respective XY options do equivalent things. The poln='/' option plots amplitude ratios or phase differences between whatever polarizations are in the MS (R and L. or X and Y).

The field, spw, and antenna selection parameters are available to obtain plots of subsets of solutions. The syntax for selection is given in § 2.5.

The subplot parameter is particularly helpful in making multi-panel plots. 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, ..., xy, in order upper left to right, then down). See § 3.4.3.4 for more details on this option.

The iteration parameter allows you to select an identifier to iterate over when producing multi-panel plots. The choices for iteration are: 'antenna', 'time', 'spw', 'field'. For example, if per-antenna solution plots are desired, use iteration='antenna'. You can then use subplot to specify the number of plots to appear on each page. In this case, set the n to 1 for subplot=yxn. Use the **Next** button on the plotcal window to advance to the next set of plots. 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. Note that iteration can take more than one iteration choice (as a single string containing a comma-separated list of the options). **BETA ALERT:** the iteration order is fixed (independent of the order specified in the iteration string), for example:

```
iteration = 'antenna, time, field'
iteration = 'time, antenna, field'
```

will both iterate over each field (fastest) then time (next) and antenna (slowest). The order is:

```
iteration = 'antenna, time, field, spw'
```

from the slowest (outer loop) to fastest (inner loop).

The markersize and fontsize parameters are especially helpful in making the dot and label sizes appropriate for the plot being made. The screen shots in this section used this feature to make the plots more readable in the cookbook. Adjusting the fontsize can be tricky on multi-panel plots, as the labels can run together if too large. You can also help yourself by manually resizing the Plotter window to get better aspect ratios on the plots.

For example, to plot amplitude or phase as a function of time for 'G' solutions (after rescaling by fluxscale for the NGC5921 "usecase" data (see § 4.8.1 below, and Appendix F.1),

```
default('plotcal')
fontsize = 14.0     # Make labels larger
markersize = 10.0   # Make dots bigger
plotcal('ngc5921.usecase.fluxscale','','amp',subplot=211)
plotcal('ngc5921.usecase.fluxscale','','phase',subplot=212)
```

The results are shown in Figure 4.3. This makes use of the `subplot` option to make multi-panel displays.

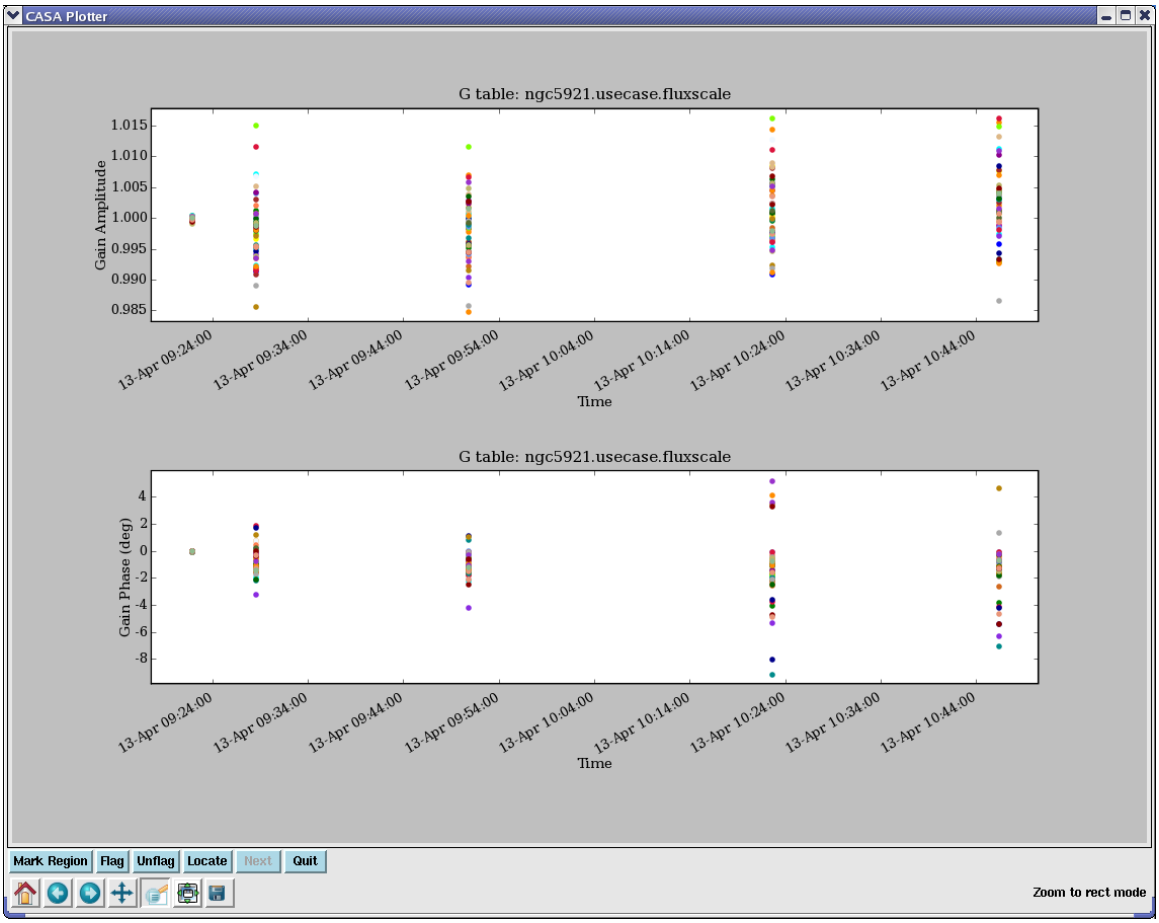



Figure 4.3: Display of the amplitude (upper) and phase (lower) gain solutions for all antennas and polarizations in the `ngc5921` post-`fluxscale` table.

---

Similarly, to plot amplitude or phase as a function of channel for `'B'` solutions for `NGC5921`:

```
default('plotcal')
fontsize = 14.0     # Make labels larger
markersize = 10.0   # Make dots bigger
plotcal('ngc5921.usecase.bcal','','amp',antenna='1',subplot=311)
plotcal('ngc5921.usecase.bcal','','phase',antenna='1',subplot=312)
```

```
plotcal('ngc5921.usecase.bcal','','snr',antenna='1',subplot=313)
```

The results are shown in Figure 4.4. This stacks three panels with amplitude, phase, and signal-to-noise ratio. We have picked `antenna='1'` to show.

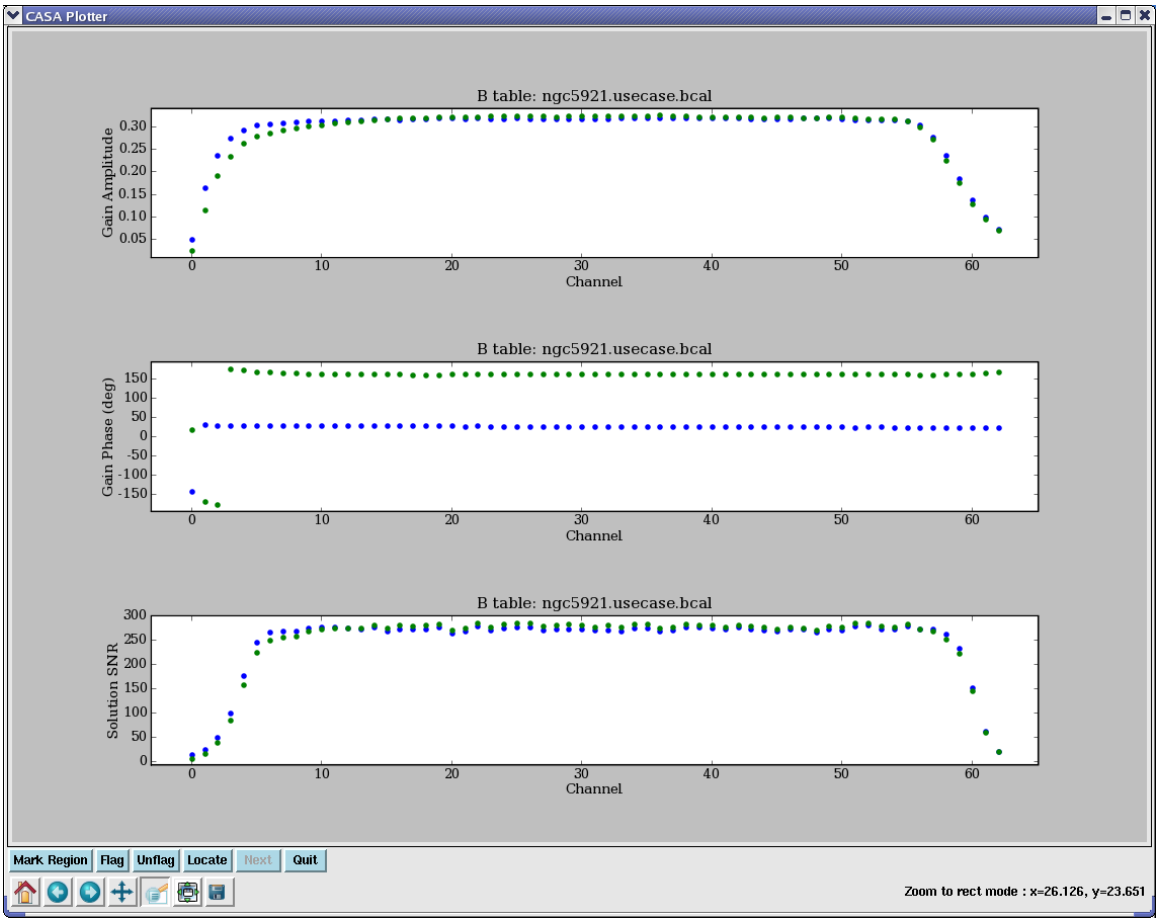



Figure 4.4: Display of the amplitude (upper), phase (middle), and signal-to-noise ratio (lower) of the `bandpass 'B'` solutions for `antenna='0'` and both polarizations for `ngc5921`. Note the falloff of the SNR at the band edges in the lower panel.

---

For example, to show 6 plots per page of `'B'` amplitudes on a $3 \times 2$ grid:

```
default('plotcal')
fontsize = 12.0     # Make labels just large enough
markersize = 10.0   # Make dots bigger
plotcal('ngc5921.usecase.bcal','','amp',subplot=231,iteration='antenna')
```

See Figure 4.5 for this example. This uses the `iteration` parameter.

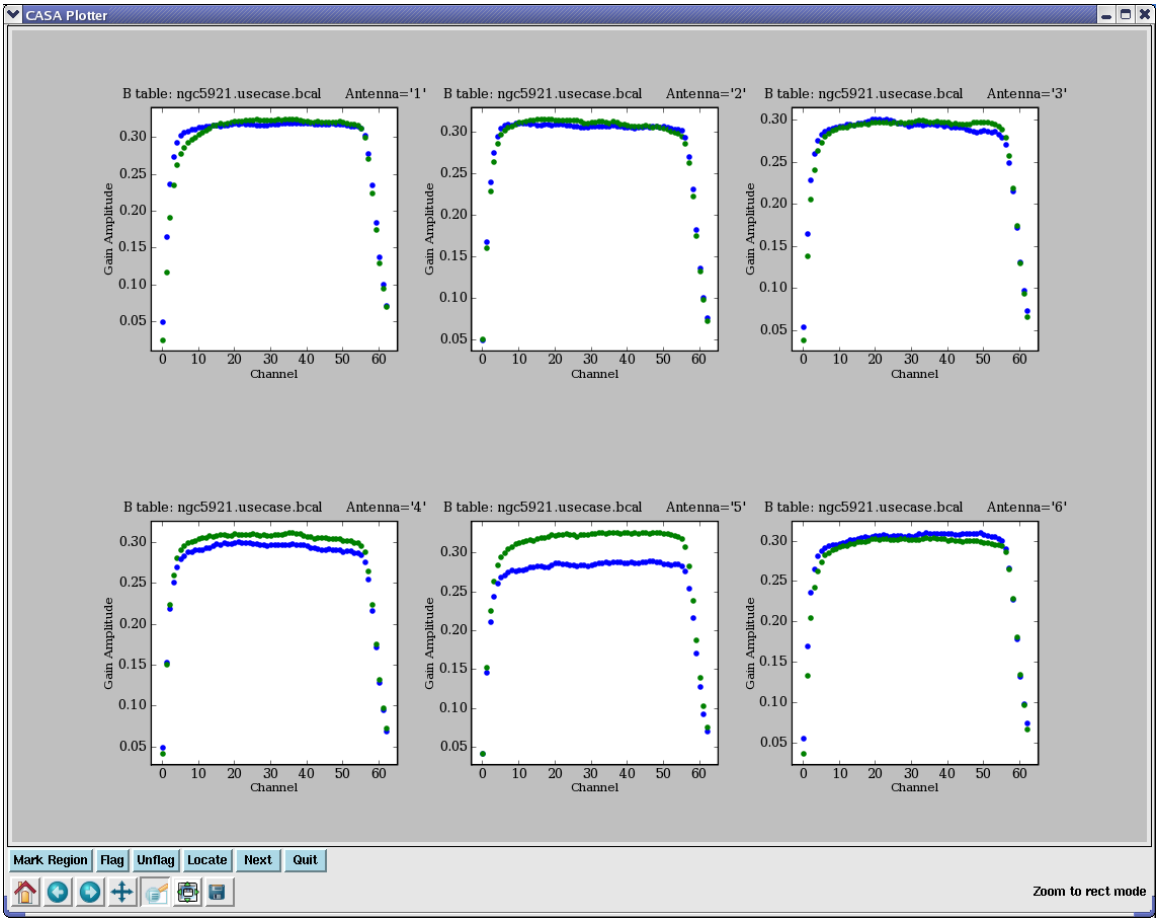



Figure 4.5: Display of the amplitude of the `bandpass 'B'` solutions. Iteration over antennas was turned on using `iteration='antenna'`. The first page is shown. The user would use the **Next** button to advance to the next set of antennas.

---

**BETA ALERT:** Note that `plotcal` cannot currently display `'delay'` or `delayrate` solutions from `fringecal`.

### 4.5.2 Listing calibration solutions with (`listcal`)

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

The inputs are:

```
#  listcal :: List data set summary in the logger:

vis         =         ''    #   Name of input visibility file (MS)
caltable    =         ''    #   Input calibration table to list
```

```
field      =         ''   #   Select data based on field name or index
antenna    =         ''   #   Select data based on antenna name or index
spw        =         ''   #   Spectral window, channel to list
listfile   =         ''   #   Disk file to write, else to terminal
pagerows   =          0   #   Rows listed per page
```

An example listing is:

```
Listing CalTable: jupiter6cm.usecase.split.ms.smoothcal2   (G Jones)
-----------------------------------------------------------------

SpwId = 0,  channel = 0.
Time                   Field      Ant       :   Amp    Phase      Amp    Phase
---------------------- ---------- --------    ---------------   ---------------
1999/04/16/14:10:43.5 'JUPITER'  '1'       :  1.016   -11.5     1.016    -9.2
                                 '2'       :  1.013    -5.3     0.993    -3.1
                                 '3'       :  0.993    -0.8     0.990    -5.1
                                 '4'       :  0.997   -10.7     0.999    -8.3
                                 '5'       :  0.985    -2.7     0.988    -4.0
                                 '6'       :  1.005    -8.4     1.009    -5.3
                                 '7'       :  0.894    -8.7     0.897    -6.8
                                 '8'       :  1.001    -0.1     0.992    -0.7
                                 '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
```

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

### 4.5.3 Calibration Smoothing (smoothcal)

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:

```
#  smoothcal :: Smooth calibration solution(s) derived from one or more sources:

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:

- `'mean'` — use the mean of the points within the window defined by `smoothtime` (a "boxcar" average),
- `'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.

`BETA ALERT:` Note that `smoothcal` currently smooths by `field` and `spw`, and thus you cannot smooth solutions from different sources or bands together into one solution.

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

```
default('smoothcal')
smoothcal('n4826_16apr.ms',
       tablein='n4826_16apr.gcal',
       caltable='n4826_16apr.smoothcal',
       smoothtime=7200.,
       smoothtype='mean')

# Plot up before and after tables
default('plotcal')
plotcal('n4826_16apr.gcal','','amp',antenna='1',subplot=211)
plotcal('n4826_16apr.smoothcal','','amp',antenna='1',subplot=212)
```

This example uses 2 hours (7200 sec) for the smoothing time and `smoothtype='mean'`. The `plotcal` results are shown in Figure 4.6.

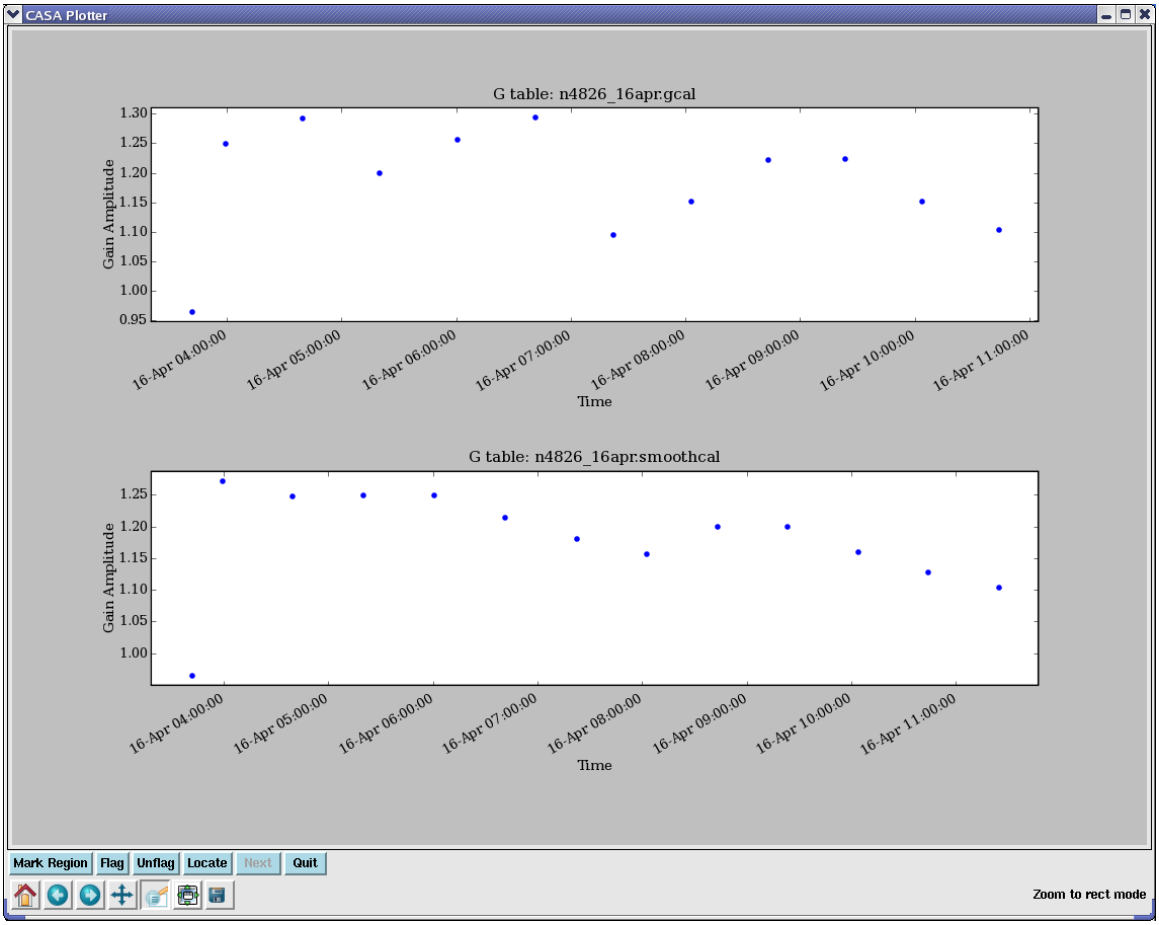



Figure 4.6: The `'amp'` of gain solutions for `NGC4826` before (top) and after (bottom) smoothing with a 7200 sec `smoothtime` and `smoothtype='mean'`. Note that the first solution is in a different `spw` and on a different source, and is not smoothed together with the subsequent solutions.

---

### 4.5.4 Calibration Interpolation and Accumulation (`accum`)

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

Its inputs are:

```
#  accum :: Accumulate incremental calibration solutions

vis             =         ''   #   Name of input visibility file
tablein         =         ''   #   Input (cumulative) calibration table; use '' on first run
     accumtime  =        1.0   #   Timescale on which to create cumulative table

incrtable       =         ''   #   Input incremental calibration table to add
```

```
caltable        =         ''   #   Output (cumulative) calibration table
field           =         ''   #   List of field names to process from tablein.
calfield        =         ''   #   List of field names to use from incrtable.
interp          =   'linear'   #   Interpolation mode to use for resampling incrtable solutions
spwmap          =       [-1]   #   Spectral window combinations to apply
```

The *mapping* implied here is

```
tablein + incrtable => caltable
```

(mathematically the cal solutions are multiplied as complex numbers as per the Measurement Equation). The `tablein` is optional (see below). You must specify an `incrtable` and a `caltable`.

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 if `tablein=''` then accumulate will generate one from scratch (on-the-fly), using the timescale (in seconds) specified by the sub-parameter `accumtime`. These nominal solutions will be unit-amplitude, zero-phase 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. If `tablein` is specified, then the entries in that table 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 (`''`), 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 `spwmap` parameter gives the mapping of the spectral windows in the `incrtable` onto those in `tablein` and `caltable`. The syntax is described in § 4.4.1.4.

The `interp` parameter controls the method used for interpolation. The options are (currently): `'nearest'`, `'linear'`, and `'aipslin'`. These are described in § 4.4.1.4. For most purposes, the `'linear'` option should suffice.

We now describe the two uses of `accum`.

#### 4.5.4.1 Interpolation using (accum)

Calibration solutions (most notably $G$ or $T$) can be interpolated onto the timestamps of the science target observations using `accum`.

The following example uses `accum` to interpolate an existing table onto a new time grid:

```
default('accum')
accum(vis='n4826_16apr.ms',
      tablein='',
      accumtime=20.0,
      incrtable='n4826_16apr.gcal',
      caltable='n4826_16apr.20s.gcal',
      interp='linear',
      spwmap=[0,1,1,1,1,1])

default('plotcal')
plotcal('n4826_16apr.gcal','','phase',antenna='1',subplot=211)
plotcal('n4826_16apr.20s.gcal','','phase',antenna='1',subplot=212)
```

See Figure 4.7 for the `plotcal` results. The data used in this example is BIMA data (single polarization `YY`) where the calibrators were observed in single continuum spectral windows (`spw='0,1'`) and the target NGC4826 was observed in 64-channel line windows (`spw='2,3,4,5'`). Thus, it is necessary to use `spwmap=[0,1,1,1,1,1]` to map the bandpass calibrator in `spw='0'` onto itself, and the phase calibrator in `spw='1'` onto the target source in `spw='2,3,4,5'`.

#### 4.5.4.2 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), a solution *of the same type* and incremental *relative to the first* is required, and it is not possible or convenient 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

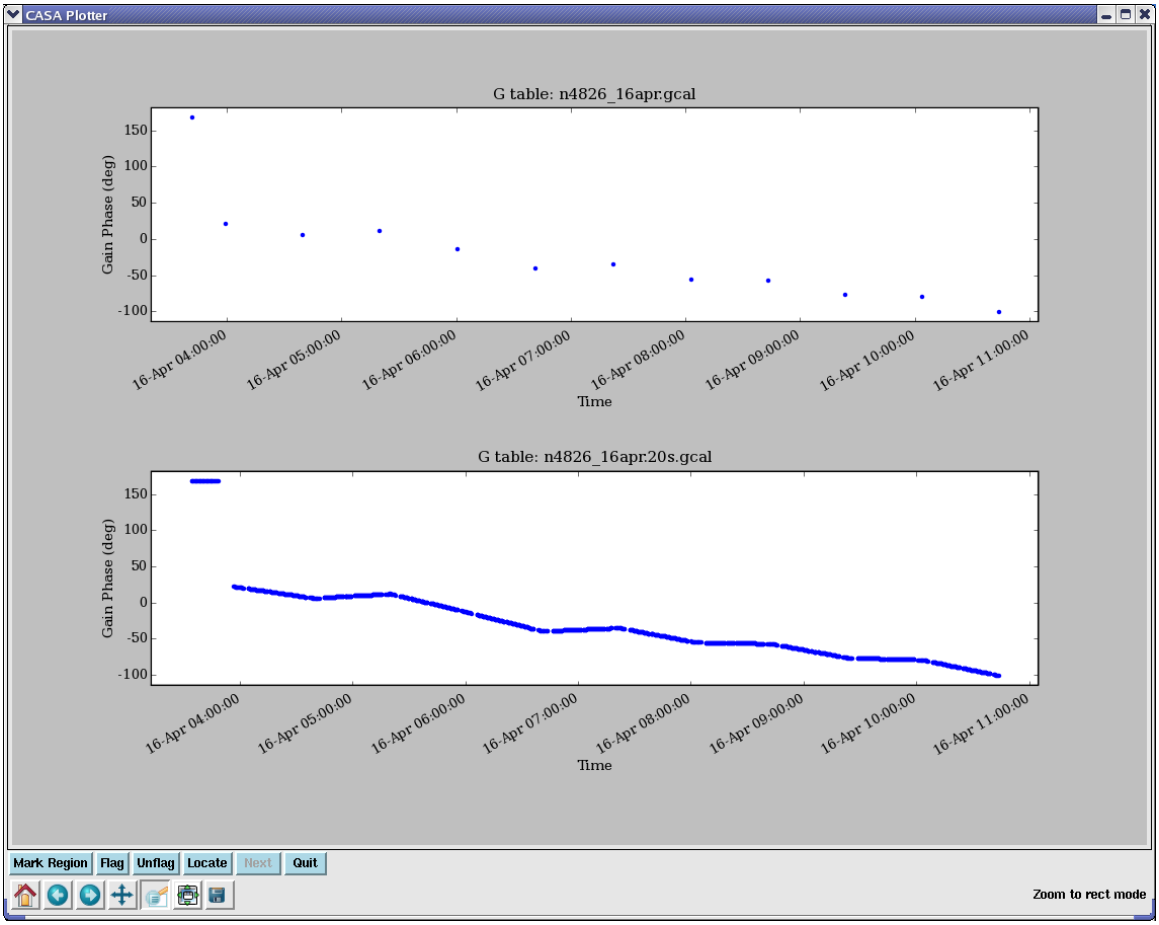



Figure 4.7: The `'phase'` of gain solutions for NGC4826 before (top) and after (bottom) `'linear'` interpolation onto a 20 sec `accumtime` grid. The first scan was 3C273 in `spw='0'` while the calibrator scans on 1331+305 were in `spw='1'`. The use of `spwmap` was necessary to transfer the interpolation correctly onto the NGC4826 scans.

---

the net effect of both. In terms of the Measurement Equation, the net calibration is the product of the initial and incremental solutions.

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 `applycal`. 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 Packages:**
The analog of `accum` in classic AIPS is the use of `CLCAL` to combine a series of (incremental) `SN` calibration tables to form successive (cumulative) `CL` calibration tables. AIPS `SN/CL` tables are the analog of `'G'` tables in CASA.

other respects (internal format, application to data with `applycal`, 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` (see sidebox) model of cumulative calibration in that its application is not limited to accumulation of `'G'` solutions. 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), delay-rate, 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'` self-cal (with no prior solution applied), as long as the timescale of this solution is at least as short as that of the existing solution.

One obvious application is to calibrate the amplitudes and phases on different timescales during self-calibration. Here is an example, using the Jupiter VLA 6m continuum imaging example (see § 4.8.2 below):

```
# Put clean model into MODEL_DATA column
default('ft')
ft(vis='jupiter6cm.usecase.split.ms',
   model='jupiter6cm.usecase.clean1.model')

# Phase only self-cal on 10s timescales
default('gaincal')
gaincal(vis='jupiter6cm.usecase.split.ms',
        caltable='jupiter6cm.usecase.phasecal1',
        gaintype='G',
        calmode='p',
        refant='6',
        solint=10.0,
        minsnr=1.0)

# Plot up solution phase and SNR
default('plotcal')
plotcal('jupiter6cm.usecase.phasecal1','','phase',antenna='1',subplot=211)
plotcal('jupiter6cm.usecase.phasecal1','','snr',antenna='1',subplot=212)

# Amplitude and phase self-cal on scans
```

```
default('gaincal')
gaincal(vis='jupiter6cm.usecase.split.ms',
        caltable='jupiter6cm.usecase.scancal1',
        gaintable='jupiter6cm.usecase.phasecal1',
        gaintype='G',
        calmode='ap',
        refant='6',
        solint=0,
        minsnr=1.0)

# Plot up solution amp and SNR
default('plotcal')
plotcal('jupiter6cm.usecase.scancal1','','amp',antenna='1',subplot=211)
plotcal('jupiter6cm.usecase.scancal1','','snr',antenna='1',subplot=212)

# Now accumulate these - they will be on the 10s grid
default('accum')
accum(vis='jupiter6cm.usecase.split.ms',
      tablein='jupiter6cm.usecase.phasecal1',
      incrtable='jupiter6cm.usecase.scancal1',
      caltable='jupiter6cm.usecase.selfcal1',
      interp='linear')

# Plot this up
default('plotcal')
fontsize = 14.0     # Make the labels a little larger on this one
markersize = 10.0   # Make to dots bigger also.
plotcal('jupiter6cm.usecase.selfcal1','','amp',antenna='1',subplot=211)
plotcal('jupiter6cm.usecase.selfcal1','','phase',antenna='1',subplot=212)
```

The final plot is shown in Figure 4.8

**BETA ALERT:** Only interpolation is offered in `accum`, no smoothing (as in `smoothcal`).

## 4.6 Application of Calibration to the Data

After the calibration solutions are computed and written to one or more calibration tables, one then needs to apply them to the data.

### 4.6.1 Application of Calibration (`applycal`)

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:

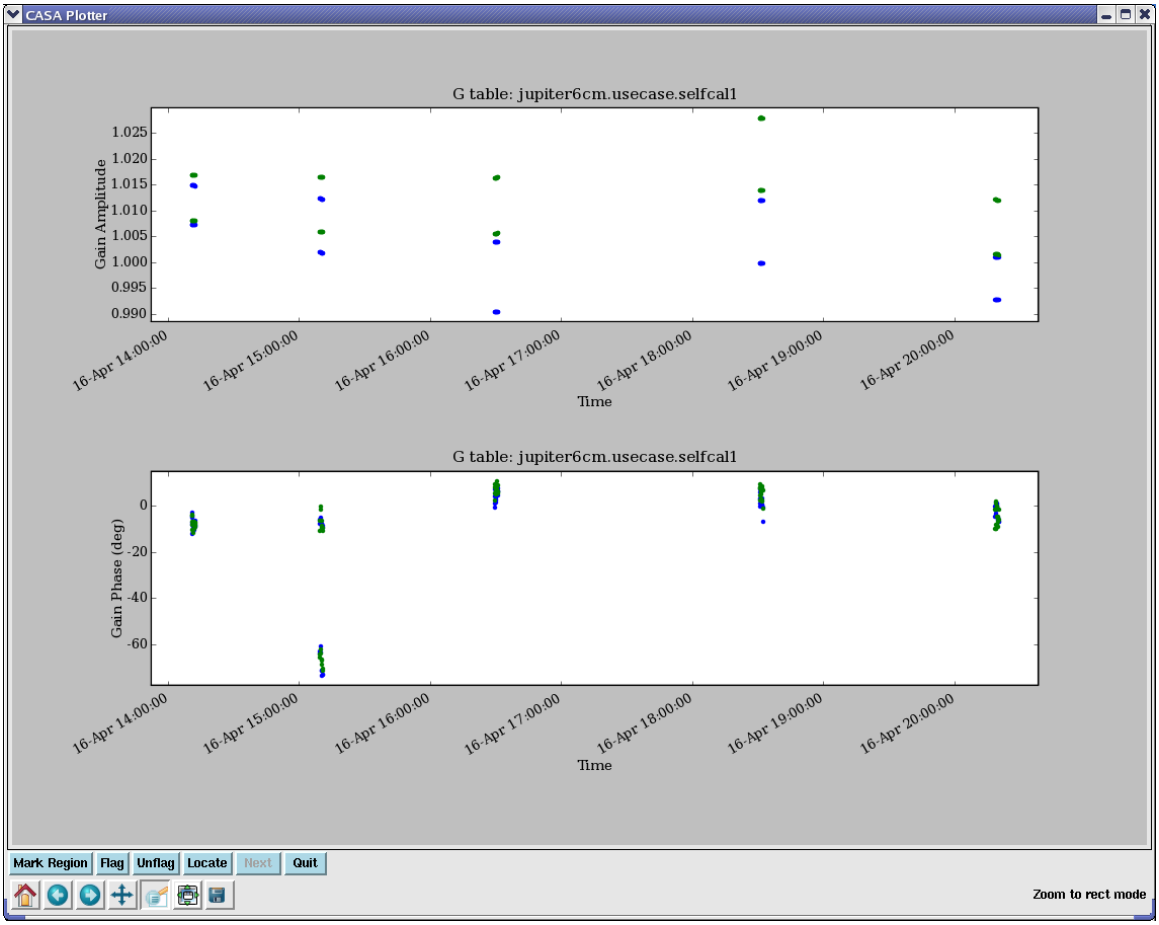



Figure 4.8: The final `'amp'` (top) and `'phase'` (bottom) of the self-calibration gain solutions for Jupiter. An initial phase calibration on 10s `solint` was followed by an incremental gain solution on each scan. These were accumulated into the cumulative solution shown here.

---

```
#  applycal :: Apply calibration solution(s) to data

vis          =         ''   #   Name of input visibility file
field        =         ''   #   Names or indices of data fields to apply calibration ''==>all
spw          =         ''   #   spectral window:channels: ''==>all
selectdata   =      False   #   Other data selection parameters
gaintable    =         ''   #   List of calibration table(s) to apply
gainfield    =         ''   #   Field selection for each gaintable
interp       =         ''   #   Interpolation mode (in time) for each gaintable
spwmap       =         []   #   Spectral window mapping for each gaintable (see help)
gaincurve    =      False   #   Apply VLA antenna gain curve correction
opacity      =        0.0   #   Opacity correction to apply (nepers)
calwt        =       True   #   Apply calibration also to the WEIGHTS
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). Many of the other parameters are the common calibration parameters that are described in § 4.4.1.

The single non-standard parameter is the `calwt` option to toggle the ability to scale the visibility weights by the inverse of the products of the scale factors applied to the amplitude of the antenna gains (for the pair of antennas of a given visibility). This should in *almost all cases* be set to its default (`True`). The weights should reflect the inverse noise variance of the visibility, and errors in amplitude are usually also in the weights.

For `applycal`, the list of final cumulative tables is given in `gaintable`. In this case you will have run `accum` if you have done incremental calibration for any of the types, such as `'G'`. You can also feed `gaintable` the full sets and rely on use of `gainfield`, `interp` and `spwmap` to do the correct interpolation and transfer. It is often more convenient to go through accumulation of each type with `accum` as described above (see § 4.5.4.2), as this makes it easier to keep track of the sequence of incremental calibration as it is solved and applied. You can also do any required smoothing of tables using `smoothcal` (§ 4.5.3), as this is not yet available in `accum` or `applycal`.

For example, to apply the final bandpass and flux-scaled gain calibration tables solutions to the NGC5921 data:

```
default('applycal')

vis='ngc5921.usecase.ms'

# 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
gaintable=['ngc5921.usecase.fluxscale','ngc5921.usecase.bcal']

# pick the 1445+099 (field 1) out of the gain table for transfer
# use all of the bandpass table
gainfield = ['1','*']

# interpolation using linear for gain, nearest for bandpass
interp = ['linear','nearest']

# only one spw, do not need mapping
spwmap = []

# all channels, no other selection
spw = ''
selectdata = False

# no prior calibration
gaincurve = False
opacity = 0.0

# select the fields for 1445+099 and N5921 (fields 1 and 2)
```

```
field = '1,2'

applycal()

# Now for completeness apply 1331+305 (field 0) to itself

field = '0'
gainfield = ['0','*']

applycal()

# The CORRECTED_DATA column now contains the calibrated visibilities
```

In another example, we apply the final cumulative self-calibration of the Jupiter continuum data obtained in the example of § 4.5.4.2:

```
default('applycal')

applycal(vis='jupiter6cm.usecase.split.ms',
         gaintable='jupiter6cm.usecase.selfcal1',
         selectdata=False)
```

Again, 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), as we describe next.

### 4.6.2 Examine the Calibrated 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, to look at the calibrated Jupiter data in the last example given in the previous section:

```
default('plotxy')

fontsize = 14.0
plotxy('jupiter6cm.usecase.split.ms','uvdist','amp','corrected',
       selectdata=True,correlation='RR LL')
```

will show the `CORRECTED_DATA` column. See Figure 4.9.

See § 3.4 for a description of how to display and edit data using `plotxy`, and § 7.4 for use of the `viewer` to visualize and edit a Measurement Set.

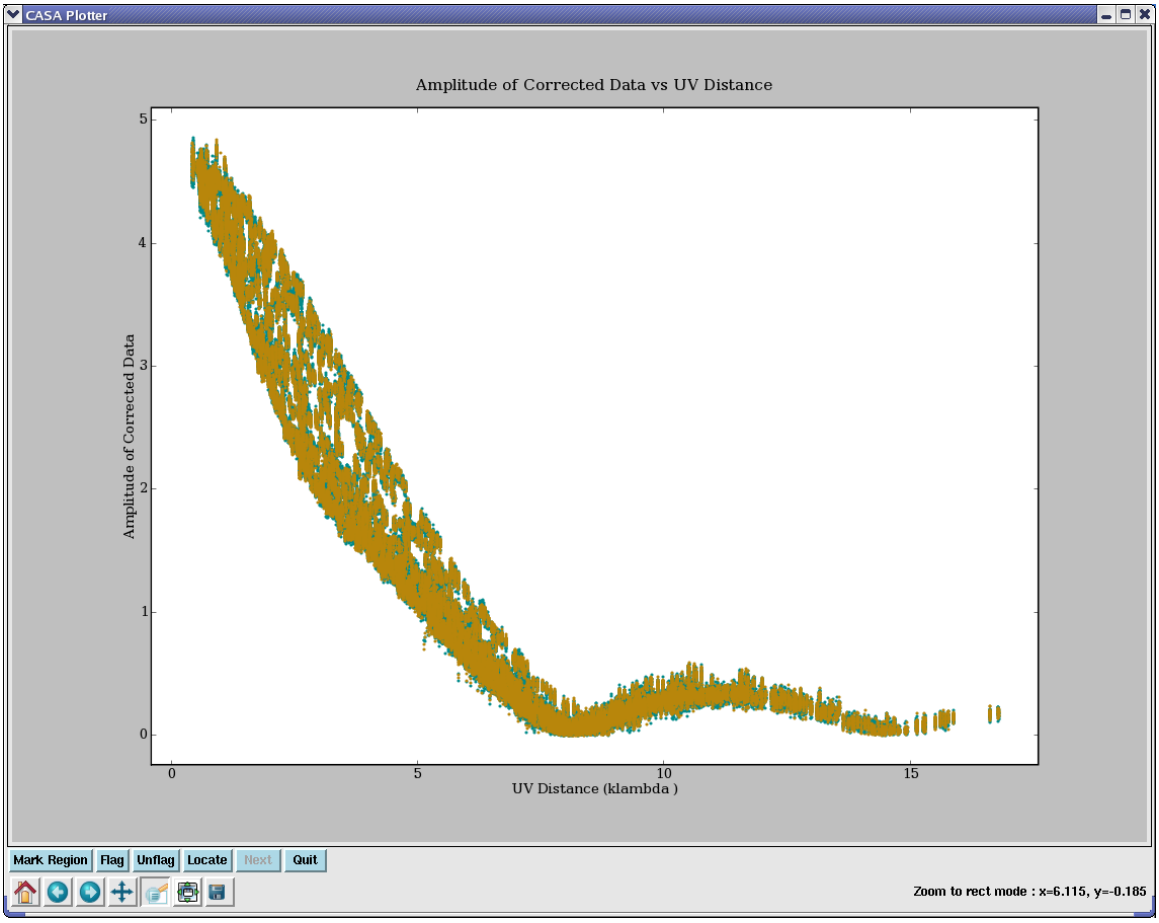



Figure 4.9: The final `'amp'` versus `'uvdist'` plot of the self-calibrated Jupiter data, as shown in `plotxy`. The `'RR LL'` correlations are selected. No outliers that need flagging are seen.

---

### 4.6.3 Resetting the Applied Calibration using (`clearcal`)

The `applycal` task will set the `CORRECTED_DATA` column. The `clearcal` task will reset it to be the same as the `DATA` column. This may or may not be what you really want to do — nominally you will rerun `applycal` to get new calibration if you have changed the tables or want to apply them differently.

There is only a single input to `clearcal`:

```
#  clearcal :: Re-initializes calibration for an ms

vis                 =         ''        #   Name of input visibility file
```

**Note:** `clearcal` also resets the `MODEL_DATA` column to `(1,0)` for all fields and spectral windows.

## 4.7 Other Calibration and UV-Plane Analysis Options

### 4.7.1 Splitting 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
```

Usually you will run split with 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 of our NGC5921 calibrated dataset:

```
default('split')

split(vis='ngc5921.usecase.ms',
      outputvis='ngc5921.split.ms',
      field='2',                      # Output NGC5921 data (field 2)
      spw='0:5~50',                   # Select 46 chans from spw 0
      datacolumn='corrected')         # Take the calibrated data column
```

**BETA ALERT**: The ability to average channels in split is on the way.

### 4.7.2 UV-Plane Continuum Subtraction (uvcontsub)

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 to `uvcontsub` are:

```
#  uvcontsub :: Continuum fitting and subtraction in the uv plane

vis         =          ''   #   Name of input visibility file
field       =          ''   #   Field name selection
spw         =         '0'   #   Spectral window selection
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
```

**BETA ALERT:** The `spw` parameter can currently only be used to specify the Spectral Window, not channelization. For now, we provide the `channels` 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. Usually, one would set `solint=-1.0` which does no averaging and fits each integration. However, if the continuum emission comes from a small region around the phase center, then you can set `solint` larger (as long as it is shorter than the timescale for changes in the visibility function of the continuum). If your scans are short enough you can also use scan averaging `solint=0.0`. Be warned, setting `solint` too large will introduce "time smearing" in the estimated continuum and thus not properly subtracting emission not at the phase center.

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`. **BETA 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.

For example, we perform uv-plane continuum subtraction on our NGC5921 dataset:

```
default('uvcontsub')

# Want to use chans 4-6 and 50-59 for continuum
# Use Python range command for this
channels = range(4,7) + range(50,60)

uvcontsub(vis='ngc5921.usecase.ms',
          field='N5921*',
          spw='0',                    # it will use channels set above
          solint=0.0,                 # scans are short enough
          fitorder=0                  # mean only
          fitmode='subtract'          # uv-plane subtraction
          splitdata=True)             # split the data for us

# You will see it made two new MS:
# ngc5921.usecase.ms.cont
# ngc5921.usecase.ms.contsub
```

### 4.7.3 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 $\chi^2$ minimum, the algorithm will happily converge to an incorrect local minimum. In fact, the $\chi^2$ 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 $\chi^2$ minimum that could conceivable 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 $\chi^2$ 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 4.10):

```
#
# 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 Calibrater::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(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
```

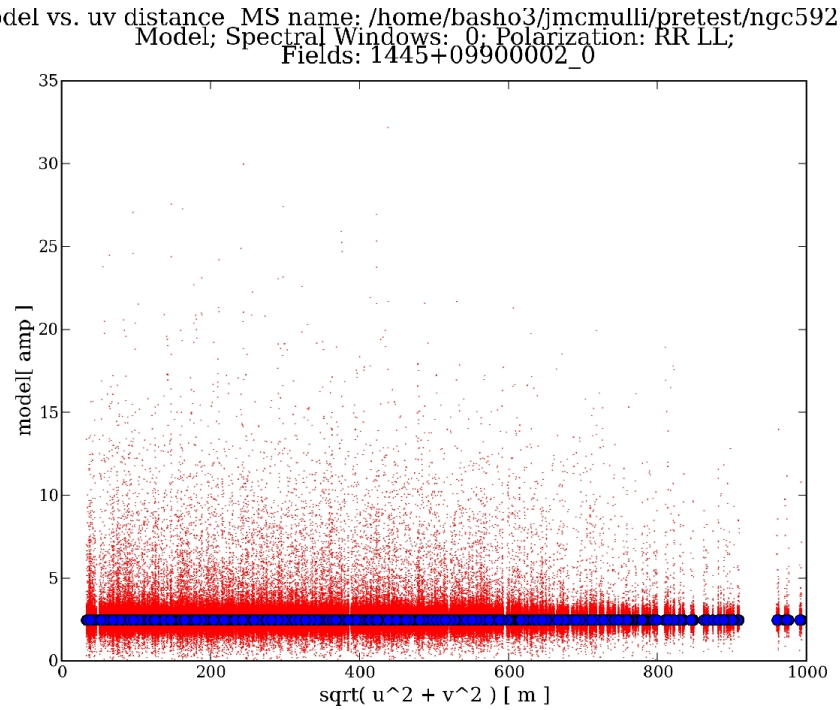



Figure 4.10: Use of plotxy to display corrected data (red points) and uv model fit data (blue circles).

---

## 4.8 Examples of Calibration

Here are two examples of calibration.

**BETA ALERT**: Note that the syntax has been changing recently and these may get out of date quickly!

### 4.8.1 Spectral Line Calibration for NGC5921

The following is an example calibration using the NGC5921 VLA observations as the demonstration. This uses the CASA tasks as of the Beta Release. 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 F.1.

```
###########################################################################
#                                                                         #
# Calibration Script for NGC 5921                                         #
#                                                                         #
# Last Updated STM 2007-11-09 (Beta 0.5)                                  #
#                                                                         #
###########################################################################

# Set up some useful variables
# The prefix to use for all output files
prefix='ngc5921.usecase'

# The MS filename
msfile = prefix + '.ms'

# Use task importuvfits to make an ms.
#
# Note that there will be a ngc5921.usecase.ms.flagversions
# in additon to ngc5921.usecase.ms with the data.
#
#==========================================================================
#
# 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 stablility 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 = ''
gainfield = ''
interp = ''

# 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')
#caltable = btable
#yaxis = 'amp'
#field = '0'
#iteration = 'antenna'
#subplot = 221
#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
gaintable = btable
gainfield = ''

# Use nearest (there is only one bandpass entry)
interp = 'nearest'

# 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'

# minimum SNR allowed
minsnr = 1.0

# reference antenna 15 (15=VLA:N2)
refant = '15'

gaincal()

# You can use plotcal to examine the gain solutions
#default('plotcal')
#caltable = gtable
#yaxis = 'amp'
#field = '0,1'
#iteration = 'antenna'
#subplot = 211
#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
gaintable = [ftable,btable]

# pick the 1445+099 out of the gain table for transfer
# use all of the bandpass table
gainfield = ['1','*']

# interpolation using linear for gain, nearest for bandpass
interp = ['linear','nearest']

# only one spw, do not need mapping
spwmap = []

# all channels
spw = ''
selectdata = False

# as before
gaincurve = False
opacity = 0.0

# select the fields for 1445+099 and N5921
field = '1,2'

applycal()

# Now for completeness apply 1331+305 to itself

field = '0'
gainfield = ['0','*']

# The CORRECTED_DATA column now contains the calibrated visibilities

applycal()

#=============================================================================
#
# Split the gain calibrater data, then the target
#
print '--Split 1445+099 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()

#
# Now split NGC5921 data (before continuum subtraction)
#
print '--Split NGC5921 Data--'

splitms = prefix + '.src.split.ms'
outputvis = splitms

# Pick off N5921
field = 'N5921*'

split()

#=============================================================================
#
# Export the NGC5921 data as UVFITS
# Start with the split file.
#
print '--Export UVFITS--'
default('exportuvfits')

srcuvfits = prefix + '.split.uvfits'

vis = splitms
fitsfile = srcuvfits

# Since this is a split dataset, the calibrated data is
# in the DATA column already.
datacolumn = 'data'

# Write as a multisource UVFITS (with SU table)
# even though it will have only one field in it
multisource = True

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importuvfits)
async = True

exportuvfits()

#=============================================================================
```

```
#
# 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'
# BETA 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.cont
# 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'

#=====================================================================
```

### 4.8.2 Continuum Calibration of Jupiter

The following is an example of continuum calibration on the Jupiter 6cm VLA dataset. This assumes you have already imported and flagged the data, and have the ms file `jupiter6cm.usecase.ms` on disk in your working directory.

The full Jupiter example script can be found in Appendix F.2.

```
###########################################################################
#                                                                         #
# Calibration Script for Jupiter 6cm VLA                                  #
#                                                                         #
# Last Updated STM 2007-10-04 (Beta)                                      #
#                                                                         #
###########################################################################

prefix='jupiter6cm.usecase'
msfile = prefix + '.ms'

#==========================================================================
#
# List a summary of the MS
#
print '--Listobs--'

vis = msfile
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>

#
#============================================================================
# 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 = 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=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()

#=============================================================================
#
# Export the Jupiter data as UVFITS
# Start with the split file.
#
print '--Export UVFITS--'
default('exportuvfits')

srcuvfits = prefix + '.split.uvfits'

vis = srcsplitms
fitsfile = srcuvfits

# Since this is a split dataset, the calibrated data is
# in the DATA column already.
datacolumn = 'data'

# Write as a multisource UVFITS (with SU table)
# even though it will have only one field in it
```

```
multisource = True

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importuvfits)
async = True

exportuvfits()

#============================================================================
```

# 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

The current imaging and deconvolution 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).
- `viewer` — image display including region statistics and image cube slice and profile capabilities (§ 7)

## 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.

**BETA ALERT:** There are still a subset of data selection parameters used in the imaging tasks: `field`, `spw`, `timerange`. In a later patch, we will use 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 Parameter `cell`

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 Parameter `field`

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 Parameter `imagename`

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 Parameter `imsize`

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 Parameter `mode`

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 $\boldsymbol{B}_k$ at wavelength $\lambda_k$ is gridded into the uv-plane at $\boldsymbol{u}_k = \boldsymbol{B}_k/\lambda_k$. 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. See the example in § 5.11.1 for using the 'channel' mode to image a spectral-line 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 Parameter phasecenter

The phasecenter parameter indicates which of the field IDs should be used to define the phase center of the mosaic image, or what that phase center is in RA and Dec. The default action is to use the first one given in the field list.

For example:

```
phasecenter='5'                          # field 5 in multi-src ms
phasecenter='J2000 19h30m00 -40d00m00' # specify position
```

### 5.2.7 Parameter restfreq

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.

### 5.2.8 Parameter spw

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 spw parameter is a string with an integer, list of integers, or a range, e.g.

```
spw = '1'                #  select spw 1
spw = '0,1,2,3'          #  select spw 0,1,2,3
spw = '0~3'              #  same thing using ranges
```

You can select channels in the same string with a : separator, for example

```
spw = '1:10~30'          #  select channels 10-30 of spw 1
spw = '0:5~55,3:5;6;7'   #  chans 5-55 of spw 0 and 5,6,7 of spw 3
```

This uses the standard syntax for spw selection is given in § 2.5.3. See that section for more options.

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.

### 5.2.9 Parameter stokes

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 with the exception of `alg='hogbom'` (see § 5.4.1) the chosen Stokes images will be deconvolved jointly rather than sequentially as in AIPS.

**BETA 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.

### 5.2.10 Parameter `uvfilter`

This controls the radial weighting of visibilities in the uv-plane (see § 5.2.11 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.

### 5.2.11 Parameter `weighting`

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

**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.

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 § 5.2.10). The weighting matrix also includes the convolution 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.

**BETA 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:

#### 5.2.11.1 `'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

$$w_i = \omega_i = \frac{1}{\sigma_k^2} \tag{5.1}$$

where the data weight $\omega_i$ is determined from $\sigma_i$ 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 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

#### 5.2.11.2 `'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.

For uniform weighting, we first grid the inverse variance $\omega_i$ for all selected data onto a grid with uv cell-size given by $2/FOV$ where $FOV$ is the specified field of view (defaults to the image field of view). This forms the gridded weights $W_k$. The weight of the $i$-th sample is then:

$$w_i = \frac{\omega_i}{W_k}. \tag{5.2}$$

#### 5.2.11.3 '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.

#### 5.2.11.4 'radial' weighting

The `weighting = 'radial'` mode is a seldom-used option that increases the weight by the radius in the uv-plane, ie.

$$w_i = \omega_i \cdot \sqrt{u_i^2 + v_i^2}. \tag{5.3}$$

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.

#### 5.2.11.5 '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 trade-off between resolution and sensitivity. The `robust` $R$ takes value between $-2.0$ (close to uniform weighting) to 2.0 (close to natural).

Superuniform weighting can be combined with Briggs weighting using the `npixels` sub-parameter. This works as in `'superuniform'` weighting (§ 5.2.11.3).

Briggs sub-parameter `rmode` controls how the `robust` parameter is used. The choices are `'none'`, `norm`, and `abs`. Specifically:

If `rmode='none'`, Briggs weighting is turned off and `robust` is not used.

If `rmode='norm'`, then

$$w_i = \frac{\omega_i}{1 + W_k f^2} \tag{5.4}$$

where $W_k$ is defined as in `uniform` and `superuniform` weighting, and

$$f^2 = \frac{(5 * 10^{-R})^2}{\frac{\sum_k W_k^2}{\sum_i \omega_i}} \tag{5.5}$$

and $R$ is the robust parameter.

For `rmode='abs'`, a slightly different weighting is used, with

$$w_i = \frac{\omega_i}{W_k R^2 + 2\sigma_R^2} \tag{5.6}$$

where $R$ is the robust parameter and $\sigma_R$ is the `noise` parameter.

### 5.2.12 Parameter `vis`

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.

**Beta Alert!**
Multi-MS handling is not percolated to the tasks yet, as we are still working on this. Use single MS only.

For example,

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

set a single input MS, while

```
vis = ['ngc5921_day1.ms', 'ngc5921_day2.ms', 'ngc5921_day3.ms']
```

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
imagename     =         ''   #   Pre-name of output images
```

```
mode          =      'mfs'   #   Type of selection (mfs, channel, velocity)
imsize        = [256, 256]   #   Image size in pixels; symmetric for single value
cell          = ['1arcsec', '1arcsec']    #   Cell size ; symmetric for single value
stokes        =        'I'   #   Stokes parameter to image (I,IV,IQU,IQUV)
field         =        '0'   #   Field name
spw           =         ''   #   spectral window:channels: ''=>all
weighting     =  'natural'   #   Weighting to apply to visibilities
restfreq      =         ''   #   restfrequency to use in image
phasecenter   =         ''   #   Default to field phase center, or explicit direction
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]; symmetric for single value
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
phasecenter   =         ''   #   Field Identifier or direction of the image phase center
spw           =         ''   #   spectral window:channels: ''=>all
weighting     =  'natural'   #   Weighting to apply to visibilities
uvfilter      =      False   #   Apply additional filtering/uv tapering of the visibilities
timerange     =         ''   #   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:

```
vis           = 'ngc5921.usecase.ms.contsub' #   Name of input visibility file
imagename     = 'ngc5921.usecase.clean'   #   Pre-name of output images
mode          =  'channel'   #   Type of selection (mfs, channel, velocity, frequency)
     nchan    =         46   #   Number of channels to select
     start    =          5   #   Start channel
     step     =          1   #   Increment between channels/velocity
     width    =          1   #   Channel width (value > 1 indicates channel averaging)

alg           =   'hogbom'   #   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 or 'interactive'
imsize        = [256, 256]   #   Image size in pixels [nx,ny]; symmetric for single value
cell          = [15.0, 15.0] #   Cell size in arcseconds [x,y]
stokes        =        'I'   #   Stokes parameter to image (I,IV,IQU,IQUV)
field         =        '0'   #   Field name
phasecenter   =         ''   #   Field Identifier or direction of the image phase center
spw           =         ''   #   spectral window:channels: ''=>all
weighting     =   'briggs'   #   Weighting to apply to visibilities
     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
timerange     =         ''   #   range of time to select from data
restfreq      =         ''   #   restfrequency to use in image
async         =      False   #   if True run in the background, prompt is freed
```

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='ggtau.1mm.split.ms', # Use data in ggtau.1mm.split.ms
      imagename='ggtau.1mm',    # Name output images 'ggtau.1mm.*' on disk
      alg='clark',              # Use the Clark CLEAN algorithm
      niter=500, gain=0.1,      # Iterate 500 times using gain of 0.1
      nchan=1,start=3,width=58, # multi-frequency synthesis (combine channels)
      spw=[0,1,2],field='0',    # Combine channels from 3 spectral windows
      stokes='I',               # Image stokes I polarization
      weighting='briggs',       # Use Briggs robust weighting
      rmode='norm',robust=0.5,  #    with robustness parameter of 0.5
      cell=[0.1,0.1],           # Using 0.1 arcsec pixels
      imsize=[256,256])         # Set image size = 256x256 pixels
```

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.

**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`.

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 use this region:

```
cleanbox=[80,80,120,120])    # Set the deconvolution region as a simple box in the center.
```

The following are the `clean` specific parameters and their allowed values, followed by a description of carrying out interactive cleaning.

### 5.4.1 Parameter `alg`

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 `alg` parameter behaves somewhat differently in `mosaic` than in `clean`, see § 5.5.1 for the differences in the former task.

#### 5.4.1.1 The `clark` algorithm

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. Also, for polarization imaging, Clark searches for the peak in $I^2+Q^2+U^2+V^2$.

#### 5.4.1.2 The `csclean` algorithm

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.

Note that when using the Cotton-Schwab algorithm with a `threshold` (§ 5.4.6), there may be strange behavior when you hit the threshold with a major cycle. In particular, it may be above threshold again at the start of the next major cycle. This is particularly noticeable when cleaning a cube, where different channels will hit the threshold at different times.

**BETA ALERT:** You will see a warning message in the logger, similar to this:

```
Zero Pixels selected with a Flux limit of 0.000551377 and a maximum Residual of 0.00751239
```

whenever it find 0 pixels above the threshold. This is normal, and not a problem, if you've specified a non-zero threshold. On the other hand, if you get this warning with the threshold set to the default of `'0Jy'`, then you should look carefully at your inputs or your data, since this usually means that the masking is bad.

#### 5.4.1.3 The `hogbom` algorithm

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. But in some cases, with poor uv-coverage and/or a PSF with bad sidelobes, the Hogbom algorithm will do better as it uses a smaller beam patch. For polarization cleaning, Hogbom searches for clean peak in $I$, $Q$, $U$, and $V$ independently.

#### 5.4.1.4 The `multiscale` algorithm

**BETA ALERT:** The `multiscale` option is currently under development and should be used with caution and be considered as an "experimental" algorithm. The `multiscale` CLEAN method is known to need careful tuning in order to properly converge. However, currently the only control for `multiscale` in the `clean` and `mosaic` tasks is the setting of the `scales`.

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 for the Gaussians. The sizes of the Gaussians are set using the `scales` sub-parameter.

**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.

The `multiscale` algorithm here uses a Cotton-Schwab clean at each scale size.

Choosing `alg='multiscale'` opens up sub-parameters:

```
alg              = 'multiscale' #   Algorithm to use (hogbom, clark, csclean, multiscale)
     scales      = [0, 3, 10]   #   sizes of component (number of pixel) to use in multiscale clean
```

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, effectivley 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

$$\theta_i = \theta_{bmin}\, 10^{(i-N/2)/2} \tag{5.7}$$

where $N =$`nscales` and $\theta_{bmin}$ is the fitted FWHM of the minor axis of the CLEAN beam.

### 5.4.2 Parameter `cleanbox`

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.

### 5.4.3 Parameter `gain`

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.

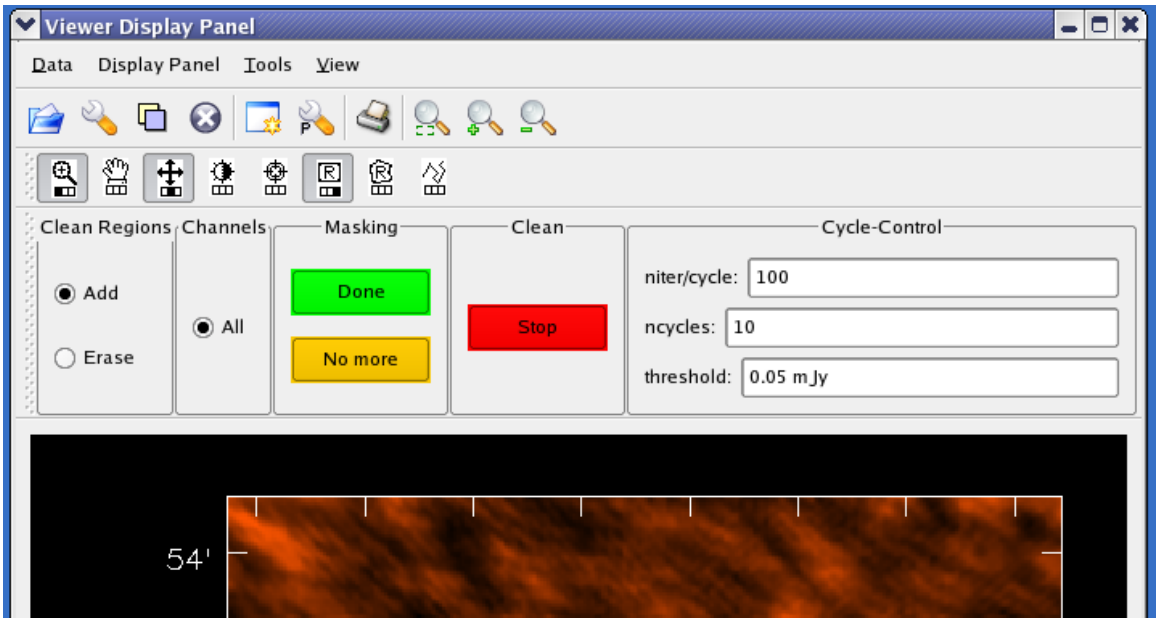



Figure 5.1: Close-up of the top of the interactive `clean` window. Note the boxes at the right (where the `npercycle`, `niter`, and `threshold` can be changed), the buttons that control the masking and whether to continue or stop cleaning, and the row of Mouse-button tool assignment icons.

---

### 5.4.4 Parameter `mask`

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.

### 5.4.5 Parameter `niter`

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.

### 5.4.6 Parameter `threshold`

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.

### 5.4.7 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
```

**BETA 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`. A close-up of the controls are show in Figure 5.1, and an example is shown in Figure 5.2. 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.

While stopped in an interactive step, you can change a number of control parameters in the boxes provided. The main use of this is to control how many iterations before the next breakpoint, and to change the threshold for ending cleaning. Note the boxes at the top right if the interactive panel where the `npercycle`, `niter`, and `threshold` can be changed. Typically, the user would start with a relatively small `npercycle` (50 or 100) to clean the bright emission in tight mask regions, and then increase this as you get deeper and the masking covers more of the emission region. For extended sources, you may end up needing to clean a large number of components (10000 or more) and thus it is useful to set `niter` to a large number to begin with — you can always terminate the clean interactively when you think it is done.

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.

See the example of cleaning and self-calibrating the Jupiter 6cm continuum data given below in § 5.11.2. The sequence of cleaning starting with the "raw" externally calibrated data is shown in Figures 5.2 – 5.4.

For spectral cube images you can use the tapedeck to move through the channels. There is a panel `Channels` with a radio button `All` which toggles the ability of the mask that will be drawn to

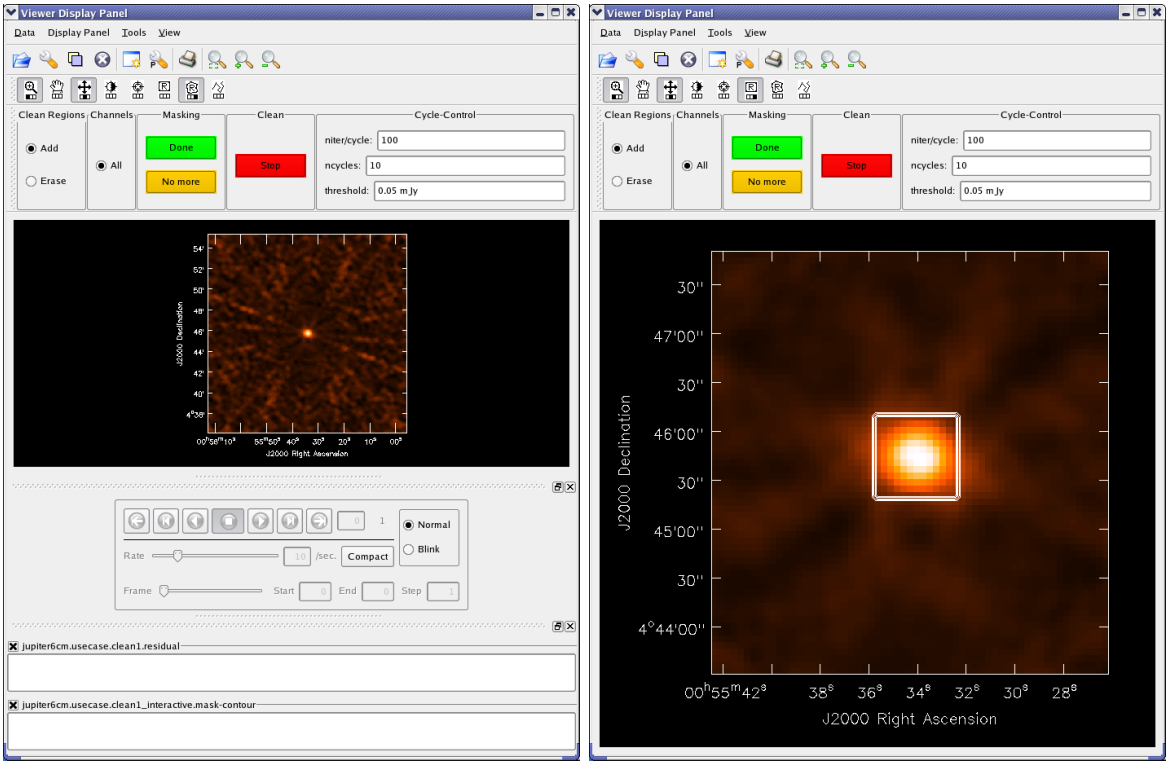



Figure 5.2: Screen-shots of the interactive `clean` window during deconvolution of the VLA 6m Jupiter dataset. We start from the calibrated data, but before any self-calibration. In the initial stage (left), the window pops up and you can see it dominated by a bright source in the center. Next (right), we zoom in and draw a box around this emission. We have also at this stage dismissed the tape deck and Position Tracking parts of the display (§ 7.2.1) as they are not used here. We will now hit the **Done** button to start cleaning.

---

apply of the current or all channels. See Figure 5.5 for an example. Note that the `Channels::All` toggle is currently set (so masks apply to all channels) by default. This toggle is unimportant for single-channel images or `mode='mfs'`.

**BETA ALERT:** Currently, interactive spectral line cleaning is done globally over the cube, with halts for interaction after searching all channels for the requested `npercycle` total iterations. It is more convenient for the user to treat the channels in order, cleaning each in turn before moving on. This will be implemented in an upcoming update.

## 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`

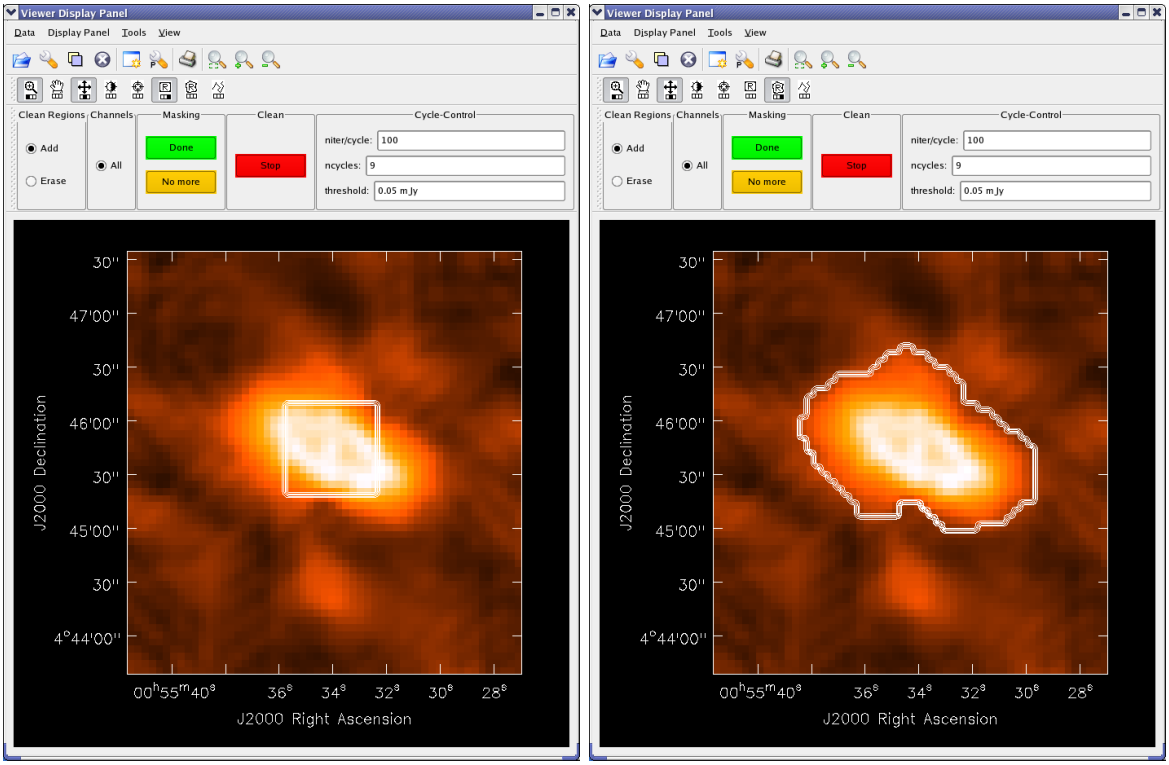



Figure 5.3: We continue in our interactive `cleaning` of Jupiter from where Figure 5.2 left off. In the first (left) panel, we have cleaned 100 iterations in the region previously marked, and are zoomed in again ready to extend the mask to pick up the newly revealed emission. Next (right), we have used the Polygon tool to redraw the mask around the emission, and are ready to hit **Done** to clean another 100 iterations.

---

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       =         ''   #   Pre-name of output images
mode            =      'mfs'   #   image spectral definition (mfs, channel, velocity, frequency)
alg             =    'clark'   #   deconvolution algorithm: clark, hogbom, multiscale, entropy
imsize          = [256, 256]   #   Image size in pixels [nx,ny]; 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
```

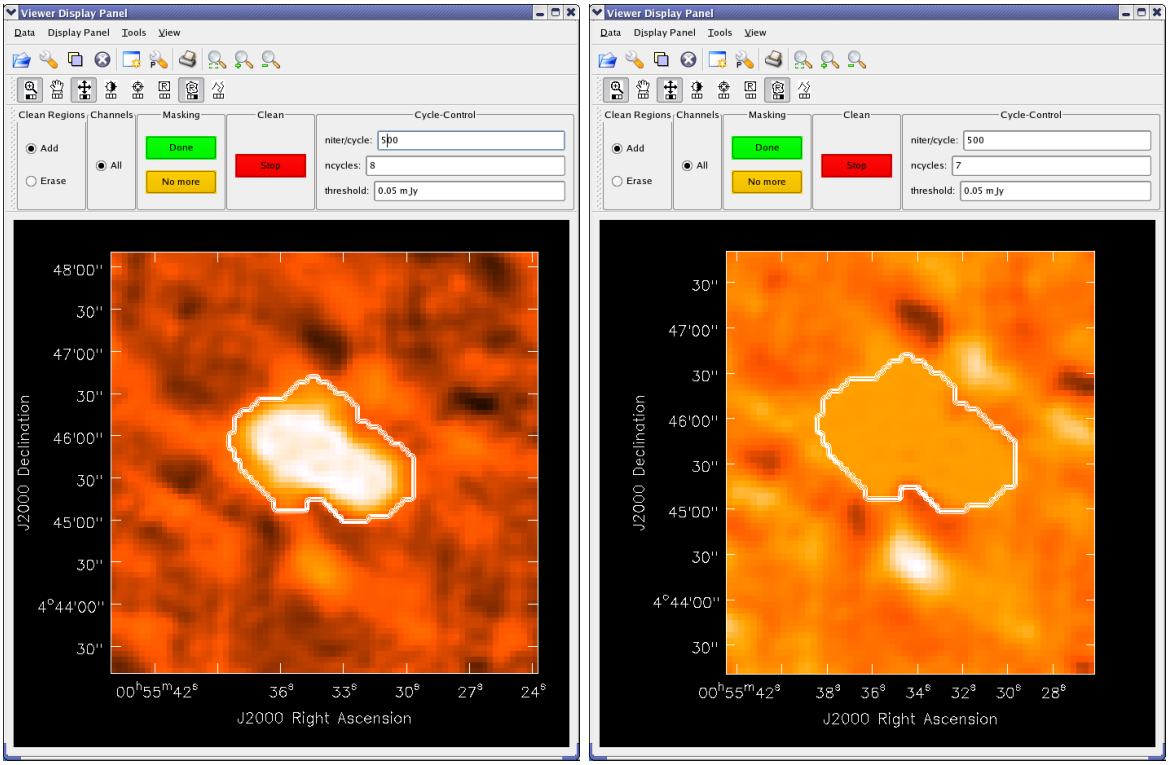



Figure 5.4: We continue in our interactive `clean`ing of Jupiter from where Figure 5.3 left off. In the first (left) panel, it has cleaned deeper, and we come back and zoom in to see that our current mask is good and we should clean further. We change `npercycle` to 500 (from 100) in the box at upper right of the window. In the final panel (right), we see the results after this clean. The residuals are such that we should **Stop** the `clean` and use our model for self-calibration.

---

```
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          =         ''   #   Field ids list to use in mosaic
spw            =         ''   #   Spectral window identifier (0-based)
timerange      =         ''   #   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
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
```

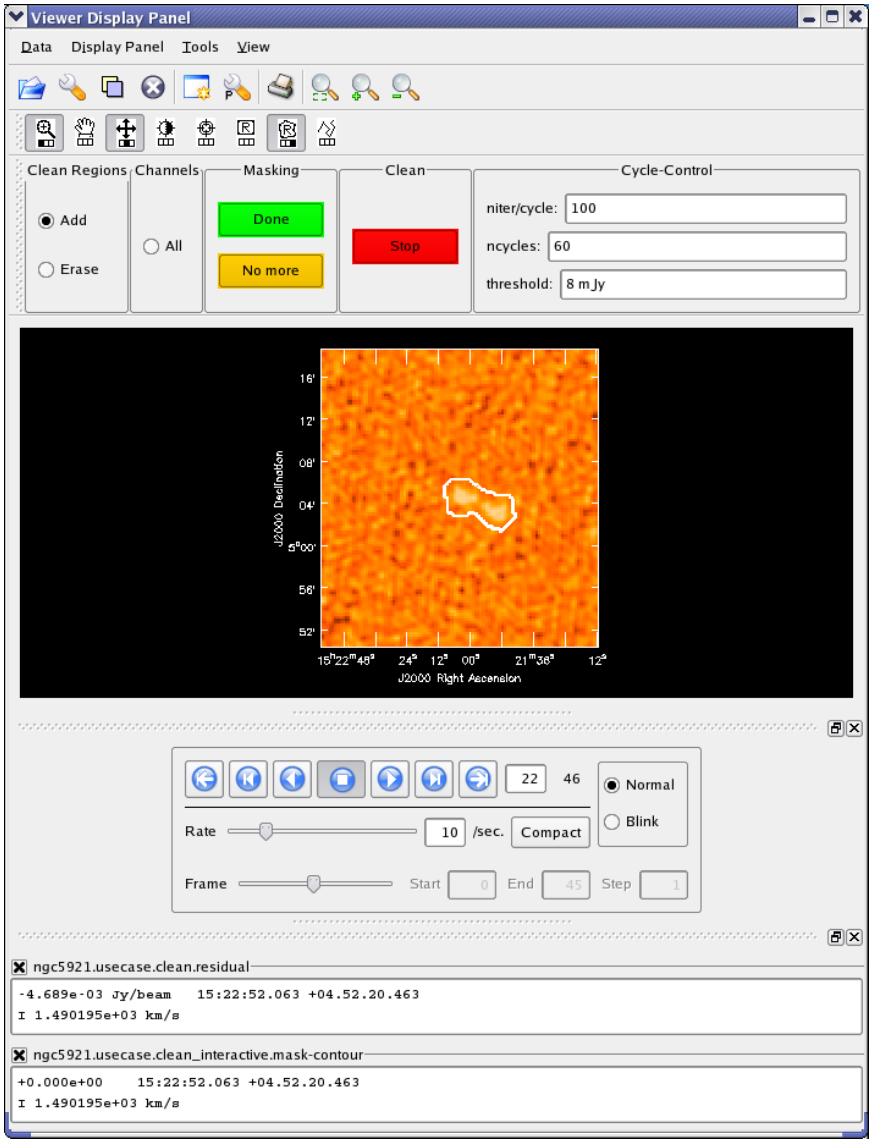



Figure 5.5: Screen-shot of the interactive `clean` window during deconvolution of the NGC5921 spectral line dataset. Note the new box at the top (second from left) where the `Channels::All` toggle can be set/unset. We have just used the Polygon tool to draw a mask region around the emission in this channel. The `Channels::All` toggle is unset, so the mask will apply to this channel only.

---

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:

```
mosaic(vis='n4826_tboth.ms',
       imagename='tmosaic',
       nchan=30,start=46,           # Make the output cube 30 chan
       width=4,                     # start with 46 of spw 0, avg by 4 chans
       spw='0~2',
       field='0~6',
       cell=[1.,1.],
       imsize=[256,256],
       stokes='I',
       mode='channel',
       alg='clark',
       niter=500,
       scaletype='SAULT',
       cyclefactor=0.1)
```

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, and § 5.2 for the general imaging parameters.

### 5.5.1 Parameter `alg`

The general use of `alg` for setting the deconvolution algorithm is described previously in § 5.4.1. The options available here are: `'clark'`, `'hogbom'`, and `'multiscale'`. The default is `alg = 'clark'`.

Here we point out the differences in the `mosaic` task. The main difference is that for `mosaic`, all of the clean algorithms use Cotton-Schwab style major cycles (ie. transforming back to form the residual visibilities, then transforming forward to the image from that, see § 5.4.1.2). The `'clark'` and `'hogbom'` options here only control what image plane beam is used in the minor cycles. There is also a new 'entropy' option.

#### 5.5.1.1 The `clark` algorithm

As stated above, this uses the Cotton-Schwab scheme of returning to the visibility residual on major cycles and the Clark beam (§ 5.4.1.1) on minor cycles, and thus is the same as `alg='csclean'` in the `clean` task (§ 5.4.1.2).

#### 5.5.1.2 The `hogbom` algorithm

This is also a Cotton-Schwab style clean, using the Hogbom beam (§ 5.4.1.3) on minor cycles. This might be better for poor uv-coverage and a PSF with bad sidelobes.

#### 5.5.1.3 The `multiscale` algorithm

This is the same as the `alg='multiscale'` option in `clean` (§ 5.4.1.4). As before this opens the sub-parameters:

```
alg              = 'multiscale' # deconvolution algorithm: clark, hogbom, multiscale, entropy
     scales      = [0, 3, 10]   # Number of scales or a vector of scales in pixel numbers
     negcomponent =         -1   # Stop cycle when a large neg component is found this many times
```

In `mosaic`, the `negcomponent` sub-parameter is here to set the point at which the clean terminates because of negative components. For `negcomponent > 0`, component search will cease when this number of negative components are found at the largest scale. If `negcomponent = -1` then component search will continue even if the largest component is negative.

#### 5.5.1.4 The `entropy` algorithm

There is an option to use the Maximum Entropy deconvolution algorithm instead of CLEAN. The MEM implementation here is fairly basic, and does not include advanced convergence criteria or other improvements.

The sub-parameters are:

```
alg                =  'entropy' # deconvolution algorithm: clark, hogbom, multiscale, entropy
     sigma         =   '0.01Jy' # Image sigma to try to achieve
     targetflux    =    '1.0Jy' # Target flux for final image
     constrainflux =      False # Constrain image to match target flux
     prior         =       [''] # Name of MEM prior images
```

The standard usage is to supply MEM with an estimate of the total flux in the image (`targetflux`), an estimated noise level (`sigma`), and perhaps a "prior" model image (`prior`). For example, the `targetflux` can be measured using autocorrelations, or more commonly guessed using `plotxy` with `xaxis='uvdist'` and `yaxis='amp'`. The `sigma` parameter will control the convergence.

**BETA ALERT:** This is a basic version of MEM, and also an experimental implementation for mosaicing. Beware.

### 5.5.2 Parameter `cyclefactor`

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:

```
cycle threshold = cyclefactor * max sidelobe * max residual
```

### 5.5.3 Parameter `cyclespeedup`

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`).

### 5.5.4 Parameter `ftmachine`

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`.

### 5.5.5 Parameter `minpb`

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 alot of emission near the edge of the mosaic, then set this lower if you want to be able to clean it out.

### 5.5.6 Parameter `modelimage`

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`.

### 5.5.7 Parameter `mosweight`

If `mosweight=True` then individual mosaic fields will receive independent weights, which will give optimum signal to noise ratio.

If `mosweight=False` then the data will be weighted so that the signal-to-noise ratio is as uniform as possible across the mosaic image.

### 5.5.8 Parameter `scaletype`

The `scaletype` parameter controls weighting of pixels in the image plane.

**Inside the Toolkit:**
The `im.setmfcontrol` method gives more options for controlling the primary beam and noise across the image.

The default `scaletype='none'` scales the image to have the correct flux scale across it (out to the beam level cutoff `minpb`). This means that the noise level will vary across the image, being increased by the inverse of the weighted primary beam responses that are used to rescale the fluxes. 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.

If `scaletype='sault'` then the image will be scaled so as to have constant noise across it. This means that the point source response function varies across the image attenuated by the weighted primary beam(s). However, this response is output in the `.flux` image and can be later used to correct this.

Note that this scaling as a function of position in the image occurs after the weighting of mosaic fields specified by `mosweight` and implied by the gridding weights (`ftmachine` and weighting).

### 5.5.9 Parameter `sdimage`

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.

The feathering technique does the following:

1. The single-dish and interferometer images are Fourier transformed.
2. The beam from the single-dish image is Fourier transformed ($FTSDB(u,v)$).
3. The Fourier transform of the interferometer image is multiplied by $(1 - FTSDB(u,v))$. This basically down weights the shorter spacing data from the interferometer image.
4. The Fourier transform of the single-dish image is scaled by the volume ratio of the interferometer restoring beam to the single dish beam.
5. The results from 3 and 4 are added and Fourier transformed back to the image plane.

The term feathering derives from the tapering or down-weighting of the data in this technique; the overlapping, shorter spacing data from the deconvolved interferometer image is weighted down compared to the single dish image while the overlapping, longer spacing data from the single-dish are weighted down compared to the interferometer image.

**Other Packages:**
The `feather` task is analogous to the AIPS `IMERG` task and the MIRIAD `immerge` task with option `'feather'`.

The tapering uses the transform of the low resolution point spread function. This can be specified as an input image or the appropriate telescope beam for the single-dish. The point spread function for a single dish image may also be calculated using `invert`.

Advice: Note that if you are feathering large images, be advised to have the number of pixels along the X and Y axes to be composite numbers and definitely not prime numbers. In general FFTs work much faster on even and composite numbers. You may use subimage function of the image tool to trim the number of pixels to something desirable.

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 § 5.4.2 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
<CNTL-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 helpful for self-calibration (see § 5.10 below).

**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.

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 :: Deconvoving a point spread function from an image

imagename   =         '' #   Name of image to decolvolve
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 alorithm 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.

One important thing to keep in mind is that the self-calibration relies upon having the most recent Fourier transform of the model in the `MODEL_DATA` column of the MS. This is indeed the case if you follow the imaging (using `clean` or `mosaic`) directly by the self-calibration. If you have done something strange in between and have lost or overwritten the `MODEL_DATA` column (for example done some extra cleaning that you do not want to keep), then use the `ft` task (see § 5.8 above), which fills the `MODEL_DATA` column with the Fourier transform of the specified model or model image.

Likewise, during self-calibration (once you have a new calibration solution) the imaging part relies upon having the `CORRECTED_DATA` column contain the self-calibrated data. This is done with the `applycal` task (§ 4.6.1).

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`. If you need to restore the `CORRECTED_DATA` to any previous stage in the self-calibration, use `applycal` again with the appropriate calibration tables.

**BETA ALERT:** 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.

See the example of cleaning and self-calibrating the Jupiter 6cm continuum data given below in § 5.11.2.

## 5.11 Examples of Imaging

Here are two examples of imaging.

**BETA ALERT**: Note that the syntax has been changing recently and these may get out of date quickly!

### 5.11.1 Spectral Line Imaging with NGC5921

The following is an example use of `clean` on the NGC5921 VLA data that we calibrated in the previous Chapter (§ 4.8.1). 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 F.1.

```
############################################################################
#                                                                          #
# Imaging Script for NGC 5921                                              #
#                                                                          #
# Last Updated STM 2007-10-04 (Beta)                                       #
#                                                                          #
############################################################################

# Set up some useful variables
# The prefix to use for all output files
prefix='ngc5921.usecase'

# The split MS filename
msfile = prefix + '.split.ms'

#===========================================================================
#
# 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'
imagename = 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.

#=============================================================================
#
# Export the Final CLEAN Image as FITS
#
print '--Final Export CLEAN FITS--'
default('exportfits')

clnfits = prefix + '.clean.fits'

imagename = clnimage
fitsimage = clnfits

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importfits)
async = True

exportfits()

#=============================================================================
```

### 5.11.2 Continuum Imaging of Jupiter

The following is an example use of interactive `clean` and self-calibration on the Jupiter 6cm VLA dataset. This assumes you have already flagged, calibrated, and split out that data and are ready to image, as well as having the split calibrated ms file `jupiter6cm.usecase.split.ms` on disk in your working directory.

In this script, notice the different self-calibrations that were done each cycle, and how they gradually improved the image.

The full Jupiter example script can be found in Appendix F.2.

```
##############################################################################
#                                                                            #
# Imaging/Self-Calibration Script for Jupiter 6cm VLA                        #
#                                                                            #
# Last Updated STM 2007-10-10 (Beta)                                         #
#                                                                            #
##############################################################################

# Some variables defined
prefix='jupiter6cm.usecase'
srcsplitms = prefix + '.split.ms'

#
#=============================================================================
```

```
# 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'
imagename = 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'
caltable = selfcaltab1

# Don't need a-priori cals
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

imname2 = prefix + '.clean2'
```

```
imagename = 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.
#
# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

# 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'
imagename = 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

#====================================================================
#
# Export the Final CLEAN Image as FITS
#
print '--Final Export CLEAN FITS--'
default('exportfits')

clnfits = prefix + '.clean.fits'

imagename = clnimage
fitsimage = clnfits
```

```
# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importfits)
async = True

exportfits()

#=============================================================================
#
# Export the Final Self-Calibrated Jupiter data as UVFITS
#
print '--Final Export UVFITS--'
default('exportuvfits')

caluvfits = prefix + '.selfcal.uvfits'

vis = srcsplitms
fitsfile = caluvfits

# The self-calibrated data is in the CORRECTED_DATA column
datacolumn = 'corrected'

# Write as a multisource UVFITS (with SU table)
# even though it will have only one field in it
multisource = True

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importuvfits)
async = True

exportuvfits()

#=============================================================================
```

# Chapter 6

# 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.

**Inside the Toolkit:**
Image analysis is handled in the `ia` tool. Many options exist there, including region statistics and image math. See § 6.5 below for more information.

**BETA ALERT:** We have assembled a skeleton of image analysis tasks for this release. Many more are still under development.

The image analysis tasks are:

- `imhead` — summarize and manipulate the "header" information in a CASA image (§ 6.1)
- `immoments` — compute the moments of an image cube (§ 6.2)
- `regridimage` — regrid an image onto the coordinate system of another image (§ 6.3)
- `importfits` — import a FITS image into a CASA *image* format table (§ 6.4)
- `exportfits` — write out an image in FITS format (§ 6.4)

There are other tasks which are useful during image analysis. These include:

- `viewer` — there are useful region statistics and image cube slice and profile capabilities in the viewer (§ 7)

We also give some examples of using the CASA Toolkit to aid in image analysis (§ 6.5).

## 6.1 Summary of an Image and Headers

To summarize and change keywords and values in the "header" of your image, use the `imhead` task. Its inputs are:

```
#  imhead :: Lists/gets/puts/stats of image header properties:

imagename   =         ''   #   Name of input image file
mode        =  'summary'   #   Options: get, put, summary, list, stats
```

The default is `mode='summary'`, which will print out a summary, for example

```
  imhead('ngc5921.usecase.clean.image','summary')
```

prints in the logger:

```
Opened image ngc5921.usecase.clean.image

Image name       : ngc5921.usecase.clean.image
Object name      :
Image type       : PagedImage
Image quantity   : Intensity
Pixel mask(s)    : None
Region(s)        : None
Image units      : Jy/beam
Restoring Beam   : 51.5254 arcsec, 45.5987 arcsec, 14.6417 deg

Direction reference : J2000
Spectral  reference : LSRK
Velocity  type      : RADIO
Rest frequency      : 1.42041e+09 Hz
Pointing center     :  15:22:00.000000  +05.04.00.000000
Telescope           : VLA
Observer            : TEST
Date observation    : 1995/04/13/00:00:00

Axis Coord Type      Name             Proj Shape Tile   Coord value at pixel    Coord incr Units
-------------------------------------------------------------------------------------------------
0    0     Direction Right Ascension   SIN   256   64  15:22:00.000   128.00 -1.500000e+01 arcsec
1    0     Direction Declination       SIN   256   64 +05.04.00.000   128.00  1.500000e+01 arcsec
2    1     Stokes    Stokes                  1    1             I
3    2     Spectral  Frequency               46    8   1.41281e+09     0.00  2.441406e+04 Hz
                     Velocity                                1603.56     0.00 -5.152860e+00 km/s
```

If you choose `mode='list'`, you get the summary in the logger and a listing of keywords and values to the terminal:

```
Available header items to modify:
```

```
General --
         -- object
         -- telescope VLA
         -- observer TEST
         -- epoch "1995/04/13/00:00:00"
         -- restfrequency "1420405752.0Hz"
         -- projection "SIN"
         -- bunit Jy/beam
         -- beam ["51.5253715515arcsec","45.5987281799arcsec","14.6416902542deg"]
axes --
         -- ctype1 Right Ascension
         -- ctype2 Declination
         -- ctype3 Stokes
         -- ctype4 Frequency
crpix --
         -- crpix1 128.0
         -- crpix2 128.0
         -- crpix3 0.0
         -- crpix4 0.0
crval --
         -- crval1 4.02298392585
         -- crval2 0.0884300154344
         -- crval3 1.0
         -- crval4 1412808153.26
cdelt --
         -- cdelt1 -7.27220521664e-05
         -- cdelt2 7.27220521664e-05
         -- cdelt3 1.0
         -- cdelt4 24414.0625
units --
         -- cunit1 rad
         -- cunit2 rad
         -- cunit3
         -- cunit4 Hz
```

The values for these keywords can be queried using `mode='get'`. This opens sub-parameters

```
mode           =      'get'   #   Options: get, put, summary, list, stats
     hditem    =         ''   #   header item to get or put
```

You can set the values for these keywords using `mode='put'`. This opens sub-parameters

```
mode           =      'put'   #   Options: get, put, summary, list, stats
     hditem    =         ''   #   header item to get or put
     hdvalue   =         ''   #   header value to set (for mode=put)
```

For example, continuing the above example:

```
CASA <6>: mode = 'get'
CASA <7>: hditem = 'observer'
CASA <8>: imhead()
***
observer ::  TEST
  Out[8]: {'observer': 'TEST'}

CASA <9>: mode = 'put'
CASA <10>: hdvalue = 'CASA'
CASA <11>: imhead()
  Out[11]: {'observer': 'CASA'}

CASA <12>: mode = 'list'
CASA <13>: imhead()
Available header items to modify:

General --
         -- object
         -- telescope VLA
         -- observer CASA
...
```

Note that these options return a Python dictionary containing the current value of the `hditem` (after updating in the case of `mode='put'`). This dictionary can be manipulated in Python in the usual manner. For example, continuing from the above,

```
CASA <14>: mode = 'get'
CASA <15>: hditem = 'observer'
CASA <16>: myobs = imhead()
***
observer ::  CASA

CASA <17>: print myobs
{'observer': 'CASA'}
CASA <18>: print myobs['observer']
CASA

CASA <19>: mode = 'put'
CASA <20>: hdvalue = 'VLA'
CASA <21>: newobs = imhead()

CASA <22>: print newobs
{'observer': 'VLA'}
```

Finally, `mode = 'stats'` can be used to obtain statistics over the whole image, again returned as a dictionary. For example,

```
CASA <23>: mode = 'stats'
CASA <24>: imhead()
```

```
  Out[24]:
{'blc': array([0, 0, 0, 0]),
 'blcf': '15:24:08.404, +04.31.59.181, I, 1.41281e+09Hz',
 'flux': array([ 4.07744818]),
 'max': array([ 0.05234427]),
 'maxpos': array([134, 134,   0,  38]),
 'maxposf': '15:21:53.976, +05.05.29.998, I, 1.41374e+09Hz',
 'mean': array([  1.60033267e-05]),
 'min': array([-0.01049265]),
 'minpos': array([160,   1,   0,  30]),
 'minposf': '15:21:27.899, +04.32.14.923, I, 1.41354e+09Hz',
 'npts': array([ 3014656.]),
 'rms': array([ 0.00202178]),
 'sigma': array([ 0.00202172]),
 'sum': array([ 48.24452486]),
 'sumsq': array([ 12.32271508]),
 'trc': array([255, 255,   0,  45]),
 'trcf': '15:19:52.390, +05.35.44.246, I, 1.41391e+09Hz'}

CASA <25>: mystats = imhead()

CASA <26>: print mystats['rms']
[ 0.00202178]
```

**TOOLKIT NOTE:** The `mode='stats'` option in the `imhead` task uses the `ia.summary` and `ia.statistics` method (with a mask set to the entire image) — see § 6.5 below.

## 6.2 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:

$$M_m(x_i, y_i) = \sum_k^N w_m(x_i, y_i, v_k)\, I(x_i, y_i, v_k) \tag{6.1}$$

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(e.g,"3~20,21")
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`).

For example, using the NGC5921 example (§ F.1):

```
default('immoments')

imagename = 'ngc5921.usecase.clean.image'

# Do first and second moments
moments = [0,1]

# Need to mask out noisy pixels, currently done
# using hard global limits
excludepix = [-100,0.009]

# Include all planes
planes = ''

# Output root name
outfile = 'ngc5921.usecase.moments'
```

```
immoments()

# It will have made the images:
# -----------------------------------
# ngc5921.usecase.moments.integrated
# ngc5921.usecase.moments.weighted_coord
```

In order to make a usable moment image, it is critical to set a reasonable threshold to exclude noise from being added to the moment maps. Something like a few times the rms noise level in the usable planes seems to work (put into `includepix` or `excludepix` as needed. Also use `planes` to ignore channels with bad data.

**BETA ALERT:** We are working on improving the thresholding of planes beyond the global cutoffs in `includepix` and `excludepix`.

## 6.3 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.

**Inside the Toolkit:**
More complex coordinate system and image regridding operation can be carried out in the toolkit. The `coordsys` (`cs`) tool and the `ia.regrid` method are the relevant components.

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
```

## 6.4 Image Import/Export to FITS

To export your images to fits format use the `exportfits` task. The inputs are:

```
#  exportfits :: Convert a CASA image to a FITS file

imagename    =         ''   #   Name of input CASA image
fitsimage    =         ''   #   Name of output FITS image
velocity     =      False   #   Prefer velocity for spectral axis
optical      =       True   #   Prefer optical velocity definition
bitpix       =        -32   #   Bits per pixel (-32 (floating point), 16 (integer))
minpix       =          0   #   Minimum pixel value
maxpix       =          0   #   Maximum pixel value
overwrite    =      False   #   Overwrite pre-existing output file
dropdeg      =      False   #   Drop degenerate axes
deglast      =      False   #   Put degenerate axes last in header
async        =      False   #   if True run in the background, prompt is freed
```

For example,

```
exportfits('ngc5921.usecase.clean.image','ngc5921.usecase.image.fits')
```

You can also use the `importfits` task to import a FITS image into CASA image table format. Note, the CASA `viewer` can read fits images so you don't need to do this if you just want to look a the image. The inputs for `importfits` are:

```
#  importfits :: Convert an image FITS file into a CASA image:

fitsimage    =         ''   #   Name of input image FITS file
imagename    =         ''   #   Name of output CASA image
whichrep     =          0   #   Which coordinate representation (if multiple)
whichhdu     =          0   #   Which image (if multiple)
zeroblanks   =       True   #   If blanked fill with zeros (not NaNs)
overwrite    =      False   #   Overwrite pre-existing imagename
async        =      False   #   if True run in the background, prompt is freed
```

For example, we can read the above image back in

```
importfits('ngc5921.usecase.image.fits','ngc5921.usecase.image.im')
```

## 6.5 Using the CASA Toolkit for Image Analysis

Although this cookbook is aimed at general users employing the tasks, we include here a more detailed description of doing image analysis in the CASA toolkit. This is because there are currently only a few tasks geared towards image analysis, as well as due to the breadth of possible manipulations that the toolkit allows that more sophisticated users will appreciate.

**Inside the Toolkit:**
The image analysis tool (`ia`) is the workhorse here. It appears in the User Reference Manual as the `image` tool. Other relevant tools for analysis and manipulation include `measures` (`me`), `quanta` (`qa`) and `coordsys` (`cs`).

To see a list of the `ia` methods available, use the CASA `help` command:

```
CASA <1>: help ia
--------> help(ia)
Help on image object:

class image(__builtin__.object)
 |  image object
 |
 |  Methods defined here:
 |
 |  __init__(...)
 |      x.__init__(...) initializes x; see x.__class__.__doc__ for signature
 |
 |  __str__(...)
 |      x.__str__() <==> str(x)
 |
 |  adddegaxes(...)
 |      Add degenerate axes of the specified type to the image‘  :
 |        outfile
 |        direction = false
 |        spectral  = false
 |        stokes
 |        linear    = false
 |        tabular   = false
 |        overwrite = false
 |      ----------------------------------------
 |
 |  addnoise(...)

...

 |
 |  unlock(...)
 |      Release any lock on the image‘  :
 |      ----------------------------------------
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object at 0x55d0f20>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

or for a compact listing use <TAB> completion on ia., e.g.

```
CASA <2>: ia.
Display all 101 possibilities? (y or n)
ia.__class__                ia.fitsky                   ia.newimagefromshape
ia.__delattr__              ia.fromarray                ia.open
ia.__doc__                  ia.fromascii                ia.outputvariant
ia.__getattribute__         ia.fromfits                 ia.pixelvalue
ia.__hash__                 ia.fromforeign              ia.putchunk
```

```
ia.__init__                 ia.fromimage              ia.putregion
ia.__new__                  ia.fromshape              ia.rebin
ia.__reduce__               ia.getchunk               ia.regrid
ia.__reduce_ex__            ia.getregion              ia.remove
ia.__repr__                 ia.getslice               ia.removefile
ia.__setattr__              ia.hanning                ia.rename
ia.__str__                  ia.haslock                ia.replacemaskedpixels
ia.adddegaxes               ia.histograms             ia.restoringbeam
ia.addnoise                 ia.history                ia.rotate
ia.boundingbox              ia.imagecalc              ia.sepconvolve
ia.brightnessunit           ia.imageconcat            ia.set
ia.calc                     ia.insert                 ia.setboxregion
ia.calcmask                 ia.isopen                 ia.setbrightnessunit
ia.close                    ia.ispersistent           ia.setcoordsys
ia.continuumsub             ia.lock                   ia.sethistory
ia.convertflux              ia.makearray              ia.setmiscinfo
ia.convolve                 ia.makecomplex            ia.setrestoringbeam
ia.convolve2d               ia.maketestimage          ia.shape
ia.coordmeasures            ia.maskhandler            ia.statistics
ia.coordsys                 ia.maxfit                 ia.subimage
ia.decompose                ia.miscinfo               ia.summary
ia.deconvolvecomponentlist  ia.modify                 ia.toASCII
ia.done                     ia.moments                ia.tofits
ia.echo                     ia.name                   ia.topixel
ia.fft                      ia.newimage               ia.toworld
ia.findsources              ia.newimagefromarray      ia.twopointcorrelation
ia.fitallprofiles           ia.newimagefromfile       ia.type
ia.fitpolynomial            ia.newimagefromfits       ia.unlock
ia.fitprofile               ia.newimagefromimage
```

A common use of the `ia` tool is to do region statistics on an image. The `imhead` task has `mode='stats'` to do this quickly over the entire image cube. The tool can do this on specific planes or sub-regions. For example, in the Jupiter 6cm example script (§ F.2), the `ia` tool is used to get on-source and off-source statistics for regression:

```
# The variable clnimage points to the clean image name

# 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()
```

**BETA ALERT:** Bad things can happen if you open some tools, like `ia`, in the Python command line on files and forget to close them before running scripts that use the `os.system('rm -rf <filename>)` call to clean up. We are in the process of cleaning up cases like this where there can be stale handles on files that have been manually deleted, but for the meantime be warned that you might get exceptions (usually of the "`SimpleOrderedMap-remove`" flavor, or even Segmentation Faults and core-dumps!

# Chapter 7

# Visualization With The CASA Viewer

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 Measurement Sets.

## 7.1 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.usecase.ms','ms')

  CASA <6>: viewer('ngc5921.usecase.clean.image')
```

The first of these starts an empty `viewer`, which will bring up an empty **Viewer Display Panel** (§ 7.2.1) and a **Load Data** panel (§ 7.2.3) . The second starts the `viewer` loaded with a Measurement Set. The last of these examples starts the `viewer` with an image cube (see Figure 7.1).

**BETA 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.

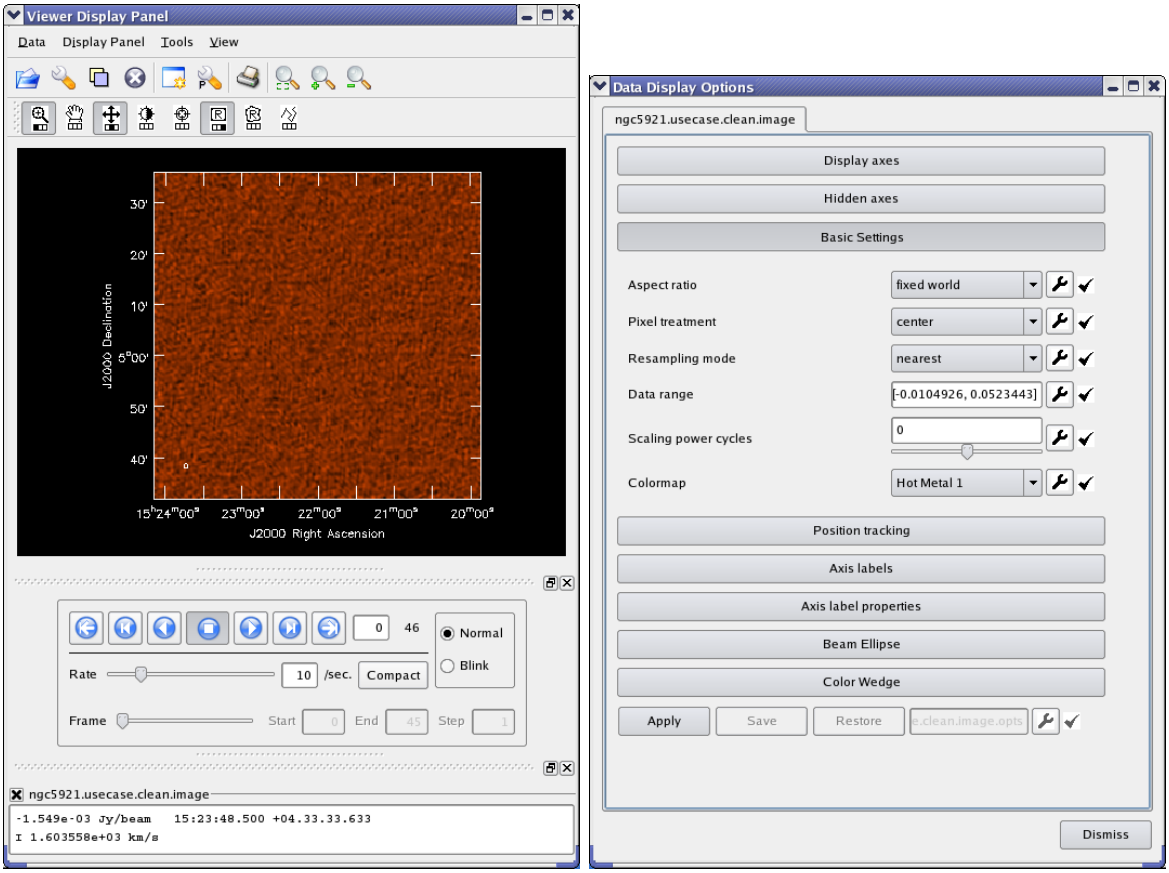



Figure 7.1: The **Viewer Display Panel** (left) and **Data Display Options** (right) panels that appear when the `viewer` is called with the image cube from NGC5921 (`viewer('ngc5921.usecase.clean.image')`). The initial display is of the first channel of the cube.

---

### 7.1.1 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 (§ 7.2.3) and a blank, standard **Viewer Display Panel** (§ 7.2.1).

```
> 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.

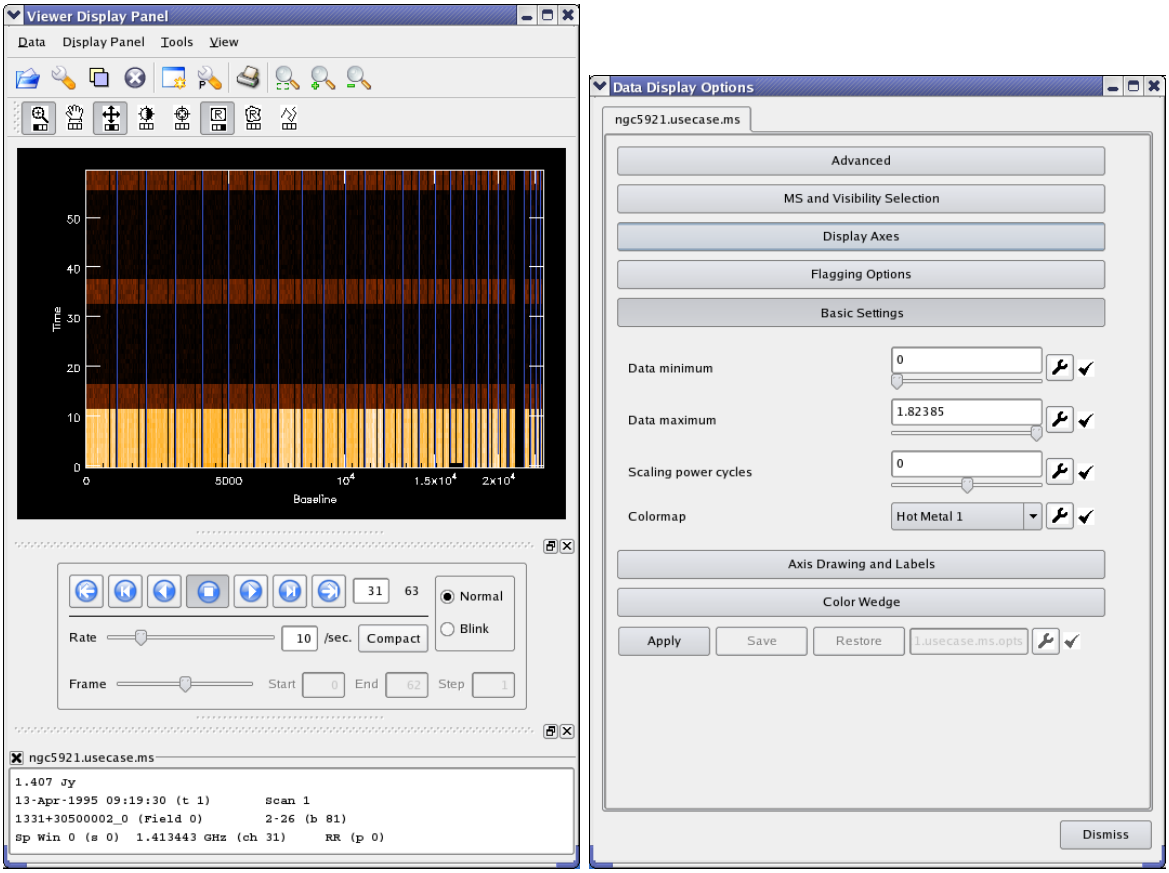



Figure 7.2: The **Viewer Display Panel** (left) and **Data Display Options** (right) panels that appear when the `viewer` is called with the NGC5921 Measurement Set (`viewer('ngc5921.usecase.ms','ms')`).

---

```
> casaviewer image_filename &
```

Start the `casaviewer` with the selected Measurement Set; note the additional parameter indicating that it is an ms; the default is 'image'.

```
> casaviewer ms_filename ms &
```

## 7.2 The `viewer` GUI

The CASA `viewer` application consists of a number of graphical user interfaces (GUIs) that are mouse cursor and button controlled. There are a number of panels to this GUI.

We describe the **Viewer Display Panel** (§ 7.2.1) and the **Load Data - Viewer** (§ 7.2.3) below, as these are common to whether you are viewing and image or MS. The other panels are context specific and described in the following sections on viewing images (§ 7.3) and Measurement Sets (§ 7.4).

### 7.2.1 The Viewer Display Panel

The Viewer Display Panel GUI is the the panel that contains the image or MS display. This is shown in the left panels of Figures 7.1 and 7.2. Note that this panel is the same whether an image or MS is being displayed.

At the top of the Viewer Display Panel GUI are the menus:

- **Data**
  - `Open` — open an image from disk
  - `Register` — register/unregister selected image (menu expands to the right containing all loaded images)
  - `Close` — close selected image (menu expands to the right)
  - `Adjust` — open the Data Display Options ('Adjust') panel
  - `Print` — print the displayed image
  - `Close Panel` — close the Viewer Display Panel (will exit if this is the last display panel open)
  - `Quit Viewer` — close all display panels and exit
- **Display Panel**
  - `New Panel` — create a new Viewer Display Panel
  - `Panel Options` — open the Display Panel's options window
  - `Print` — print displayed image
  - `Close Panel` — close the Viewer Display Panel (will exit if this is the last display panel open)
- **Tools**
  - `Annotations` — not yet available (greyed out)
  - `Spectral Profile` — plot frequency/velocity profile of point or region of image
- **View**
  - `Main Toolbar` — show/hide top row of icons
  - `Mouse Toolbar` — show/hide second row of mouse-button action selection icons
  - `Animator` — show/hide tapedeck control panel
  - `Position Tracking` — show/hide bottom position tracking report box

Below this is the **Main Toolbar**, the top row of 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 ('Adjust') 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 the Display Panel's options window
- **print** (`Display Panel:Print`) — print data
- **magnifier box** — Zoom out all the way
- **magnifier plus** — Zoom in (by a factor of 2)
- **magnifier minus** — Zoom out (by a factor of 2)

Below this are the eight **Mouse Tool** buttons. These allow assignment of *each* of the three mouse buttons to a different operation on the display area. Clicking a mouse tool icon will [re-]assign **the mouse button that was clicked** to that tool. The icons show which mouse button is currently assigned to which tool.

The 'escape' key can be used to cancel any mouse tool operation that was begun but not completed, and to erase any tool showing in the display area.

- **Zooming (magnifying glass icon):** To zoom into a selected area, press the Zoom tool's mouse button (the **left** button by default) on one corner of the desired rectangle and drag to the desired opposite corner. Once the button is released, the zoom rectangle can still be moved or resized by dragging. To complete the zoom, double-click inside the selected rectangle (double-clicking *outside* it will zoom *out* instead).
- **Panning (hand icon):** Press the tool's mouse button on a point you wish to move, drag it to the position where you want it moved, and release. *Note: The arrow keys, Page Up, Page Down, Home and End keys can also be used to scroll through your data any time you are zoomed in. (Click on the main display area first, to be sure the keyboard is 'focused' there).*
- **Stretch-shift colormap fiddling (crossed arrows):** This is usually the handiest color adjustment; it is assigned to the **middle** mouse button by default.
- **Brightness-contrast colormap fiddling (light/dark sun)**

- **Positioning (bombsight):** This tool can place a 'crosshair' marker on the display to select a position. It is used to flag Measurement Set data or to select an image position for spectral profiles. Click on the desired position with the tool's mouse button to place the crosshair; once placed you can drag it to other locations. Double-click is not needed for this tool. See § 7.2.2 for more detail.
- **Rectangle and Polygon region drawing:** The rectangle region tool is assigned to the **right** mouse button by default. As with the zoom tool, a rectangle region is generated by dragging with the assigned mouse button; the selection is confirmed by double-clicking within the rectangle. Polygon regions are created by clicking the assigned mouse button at the desired vertices, clicking the final location twice to finish. Once created, a polygon can be moved by dragging from inside, or reshaped by dragging the handles at the vertices. Double-click inside to confirm region selection. See § 7.2.2 for the uses of this tool.
- **Polyline drawing:** A polyline can be created by selecting this tool. It is manipulated similarly to the polygon region tool: create segments by clicking at the desired positions and then double-click to finish the line. [Uses for this tool are still to be implemented].

The main **Display Area** lies below the toolbars.

Underneath the display area is an **Animator** panel. The most prominent feature is the "tape deck" which provides movement between image planes along a selected third dimension of an image cube. This set of buttons is only enabled when a registered image reports that it has more than one plane along its 'Z axis'. In the most common case, the animator selects the frequency channel. 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 (channel) number and a label showing the total number of frames. Below that is a slider for controlling the (nominal) animation speed. To the right is a 'Full/Compact' toggle. In 'Full' mode (the default), a slider controlling frame number and a 'Blink mode' control are also available.

'Blink' mode is useful when more than one raster image is registered. In that mode, the tapedeck controls *which image* is displayed at the moment rather than the particular image plane (set that in 'Normal' mode first). The registered images must cover the same portion of the sky and use the same coordinate projection.

**Note:** *In 'Normal' mode, it is advisable to have only ONE raster image registered at a time, to avoid confusion. Unregister (or close) the others).*

At the bottom of the Display Panel is the **Position Tracking** panel. As the mouse moves over the main display, this panel shows information such as flux density, position (e.g. RA and Dec), Stokes, and frequency (or velocity), for the point currently under the cursor. Each registered image/MS displays its own tracking information. Tracking can be 'frozen' (and unfrozen again) with the space bar. (Click on the main display area first, to be sure the keyboard is 'focused' there).

The Animator or Tracking panels can be hidden or detached (and later re-attached) by using the boxes at upper right of the panels; this is useful for increasing the size of the display area. (Use the 'View' menu to show a hidden panel again). The individual tracking areas (one for each registered image) can be hidden using the checkbox at upper left of each area.

### 7.2.2 Region Selection and Positioning

You can draw regions or select positions on the display with the mouse, once you have selected the appropriate tool(s) on the `Mouse Toolbar` (see above).

The `Rectangle Region` drawing tool currently works for the following:

- Region statistics reporting for images,
- Region spectral profiles for images, via the `Tools:Spectral Profile` menu,
- Flagging of Measurement Sets

The `Polygon Region` drawing tool works for region statistics and spectral profiles (polygon region flagging of an MS is not supported).

The `Positioning` crosshair tool works for the last two of the above.

The `Spectral Profile` display (see § 7.3.4), when active, updates on *each change* of the rectangle, polygon, or crosshair. Flagging with the crosshair also responds to single click or drag.

Region statistics are printed in the terminal window (not the logger) by double-clicking the completed region. The `Rectangle Region` tool's mouse button must also be double-clicked to confirm an MS flagging edit.

Here is an example of region statistics from the viewer:

```
ngc5921.usecase.clean.image-contour     (Jy/beam)

n           Std Dev     RMS         Mean        Variance    Sum
52          0.01067     0.02412     0.02168     0.0001139   1.127

Flux        Med |Dev|   IntQtlRng   Median      Min         Max
0.09526     0.009185    0.01875     0.02076     0.003584    0.04181
```

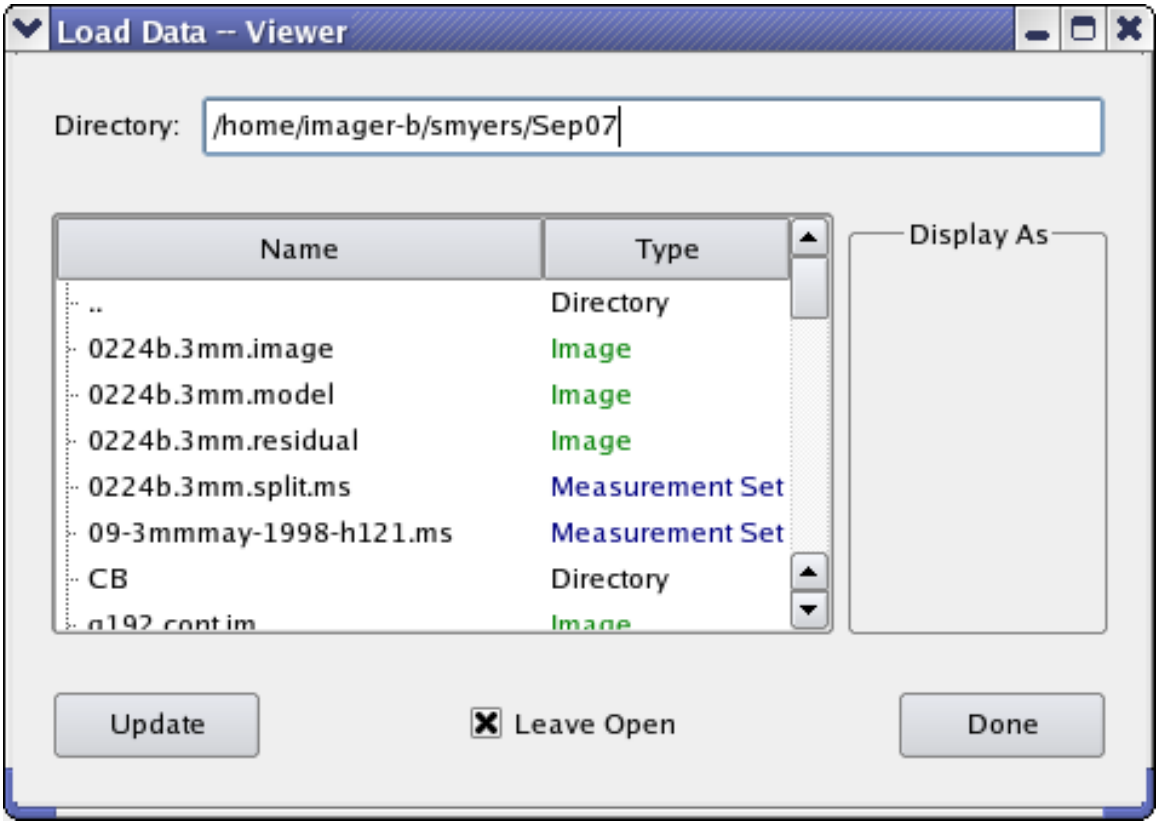



Figure 7.3: The **Load Data - Viewer** panel that appears if you open the `viewer` without any `infile` specified, or if you use the `Data:Open` menu or Open icon. You can see the images and MS available in your current directory, and the options for loading them.

---

### 7.2.3 The Load Data Panel

You can use the **Load Data - Viewer** GUI to interactively choose images or MS to load into the viewer. An example of this panel is shown in Figure 7.3. This panel is accessed through the `Data:Open` menu or Open icon of the **Viewer Display Panel**. It also appears if you open the `viewer` without any `infile` specified.

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.

A MS can only be displayed as a raster.

#### 7.2.3.1 Registered vs. Open Datasets

When you 'load' data as described above, it is first *opened*, and then *registered* on all existing `Display Panels`. The distinction is subtle. An 'open' dataset has been prepared in memory from disk; it may be registered (enabled for drawing) on one `Display Panel` and not on another. All open datasets will have a tab in the `Data Options` window, whether currently registered or not. On the other hand, only those datasets registered on a particular panel will show in its `Tracking` area.

At present, it is useful to have more than one image registered on a panel *only* if you are displaying a contour image over a raster image (§ 7.3.3) or 'blinking' between images (see **Animator** in § 7.2.1). (In future we also hope to provide transparent overlay of raster images).

It is the user's responsibility – and highly advisable – to unregister (or close) datasets that are no longer in use, using the `Register` or `Close` toolbutton or menu. In future the viewer will attempt to aid in unregistering datasets which are not 'compatible' with a newly-loaded one (different sky area, e.g., or MS vs. image).

If you close a dataset, you must reload it from disk as described above to see it again. That can take a little time for MSs, especially. If you unregister a dataset, it is set to draw immediately when you re-register it, with its options as you have previously set them. In general, close unneeded datasets but unregister those you'll be working with again.

## 7.3 Viewing Images

You have several options for viewing an image. These are seen at the right of the **Load Data - Viewer** panel described in § 7.2.3 and shown in Figure 7.4 when an image is selected. They are:

- `Raster Image` — a greyscale or color image,
- `Contour Map` — contours of intensity as a line plot,
- `Vector Map` — vectors (as in polarization) as a line plot,
- `Marker Map` — a line plot with symbols to mark positions.

The `Raster Image` is the default image display, and is what you get if you invoke the `viewer` from `casapy` with an image file name. In this case, you will need to use the `Open` menu to bring up the **Load Data** panel to choose a different display.

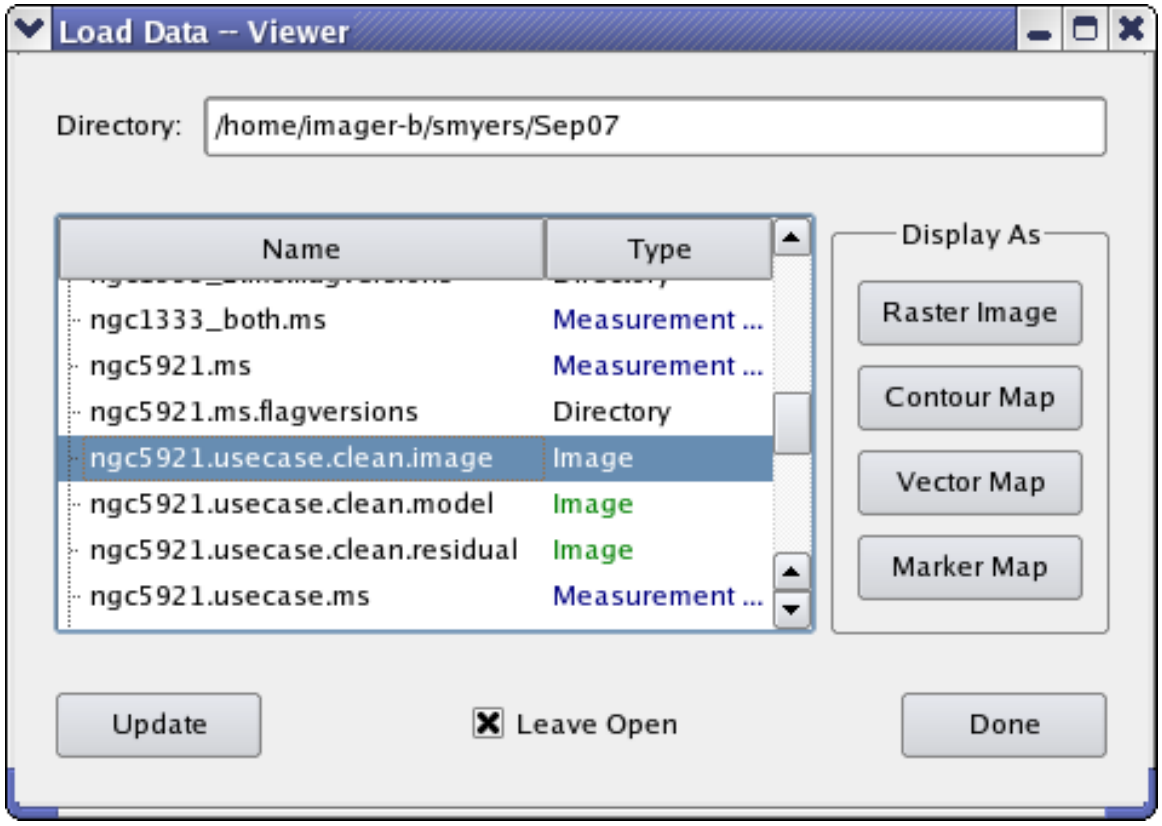



Figure 7.4: The **Load Data - Viewer** panel as it appears if you select an image. You can see all options are available to load the image as a `Raster Image`, `Contour Map`, `Vector Map`, or `Marker Map`. In this example, clicking on the `Raster Image` button would bring up the displays shown in Figure 7.1.

---

### 7.3.1 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 the example in Figure 7.1.

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 viewable files in the current working directory along with their type (Image, Measurement Set, etc). After selecting a file, you are presented with the available display types (raster, contour, vector, marker) for these data. Clicking on the button `Raster Map` will create a display as above.

The data display can be adjusted by the user as needed. This is done through the **Data Display Options** panel. This window appears when you choose the `Data:Adjust` menu or use the wrench

icon from the **Main Toolbar**. This also comes up by default along with the **Viewer Display Panel** when the data is loaded.

The **Data Display Options** window is shown in the right panel of Figure 7.1. It consists of a tab for each image or MS loaded, under which are a cascading series of expandable categories. For an image, these are:

- **Display axes**
- **Hidden axes**
- **Basic Settings**
- **Position tracking**
- **Axis labels**
- **Axis label properties**
- **Beam Ellipse**
- **Color Wedge**

The **Basic Settings** category is expanded by default. To expand a category to show its options, click on it with the left mouse button.

#### 7.3.1.1 Raster Image — Basic Settings

This roll-up is open by default. It has some commonly-used parameters that alter the way the image is displayed; three of these affect the colors used. An example of this part of the panel is shown in Figure 7.5.

The options available are:

- `Basic Settings:Aspect ratio`

  This option controls the horizontal-vertical size ratio of data pixels on screen. `Fixed world` (the default) means that the aspect ratio of the pixels is set according to the coordinate system of the image (i.e., true to the projected sky). `Fixed lattice` means that data pixels will always be square on the screen. Selecting `flexible` allows the map to stretch independently in each direction to fill as much of the display area as possible.

- `Basic Settings:Pixel treatment`

  This option controls the precise alignment of the edge of the current 'zoom window' with the data lattice. `edge` (the default) means that whole data pixels are always drawn, even on the edges of the display. For most purposes, `edge` is recommended. `center` means that data pixels on the edge of the display are drawn only from their centers inwards. (Note that a data pixel's center is considered its 'definitive' position, and corresponds to a whole number in 'data pixel' or 'lattice' coordinates).

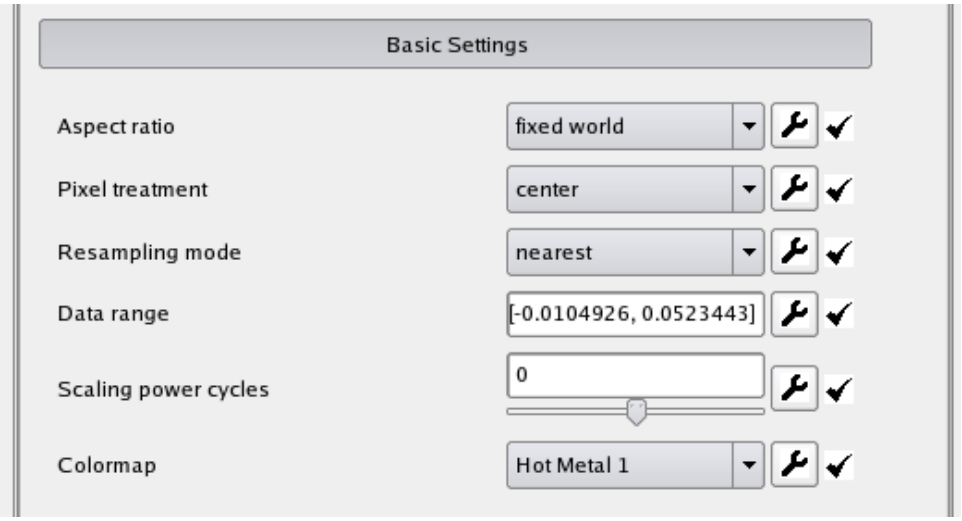



Figure 7.5: The `Basic Settings` category of the **Data Display Options** panel as it appears if you load the image as a `Raster Image`. This is a zoom-in for the data displayed in Figure 7.1.

---

- `Basic Settings: Resampling mode`

  This setting controls how the data are resampled to the resolution of the screen. `nearest` (the default) means that screen pixels are colored according to the intensity of the nearest data point, so that each data pixel is shown in a single color. `bilinear` applies a bilinear interpolation between data pixels to produce smoother looking images when data pixels are large on the screen. `bicubic` applies an even higher-order (and somewhat slower) interpolation.

- `Basic Settings: Data Range`

  You can use the entry box provided to set the minimum and maximum data values mapped to the available range of colors as a list `[min, max]`. For very high dynamic range images, you will probably want to enter a `max` less than the data maximum in order to see detail in lower brightness-level pixels. The next setting also helps very much with high dynamic range data.

- `Basic settings: Scaling power cycles`

  This option allows logarithmic scaling of data values to colormap cells.

  The color for a data value is determined as follows: first, the value is clipped to lie within the data range specified above, then mapped to an index into the available colors, as described in the next paragraph. The color corresponding to this index is determined finally by the current colormap and its 'fiddling' (shift/slope) and brightness/contrast settings (see **Mouse Toolbar**, above). Adding a **Color Wedge** to your image can help clarify the effect of the various color controls.

  The `Scaling power cycles` option controls the mapping of clipped data values to colormap indices. Set to zero (the default), a straight linear relation is used. For negative scaling

values, a logarithmic mapping assigns an larger fraction of the available colors to lower data values (this is usually what you want). Setting `dataMin` to something around the noise level is often useful/appropriate in conjunction with a negative 'Power cycles' setting.

For positive values, an larger fraction of the colormap is used for the high data values. [1]

See Figure 7.6 for sample curves.

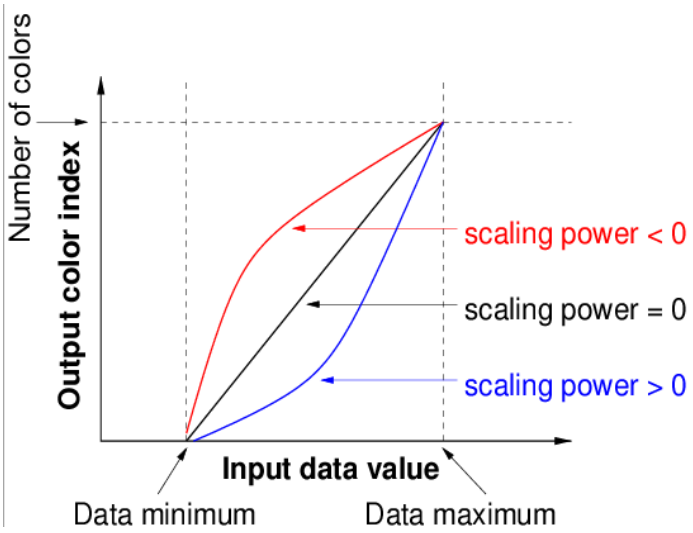



Figure 7.6: Example curves for scaling power cycles.

---

- `Basic settings: Colormap`

  You can select from a variety of colormaps here. `Hot Metal`, `Rainbow` and `Greyscale` colormaps are the ones most commonly used.

#### 7.3.1.2 Raster Image — Other Settings

Many of the other settings on the `Data Options` panel for raster images are self-explanatory, such as those which affect `Beam ellipse` drawing (only available if your image provides beam data), or the form of the `Axis labeling` and `Position tracking` information. You can also give your image a `Color wedge`, a key to the current mapping from data values to colors.

[1] The actual functions are computed as follows:

For negative scaling values (say $-p$), the data is scaled linearly from the range (`dataMin` – `dataMax`) to the range $(1 - 10^p)$. Then the program takes the log (base 10) of that value (arriving at a number from 0 to $p$) and scales that linearly to the number of available colors. Thus the data is treated as if it had $p$ decades of range, with an equal number of colors assigned to each decade.

For positive scaling values, the inverse (exponential) functions are used. If $p$ is the (positive) value chosen, The data value is scaled linearly to lie between 0 and $p$, and 10 is raised to this power, yielding a value in the range $(1 - 10^p)$. Finally, that value is scaled linearly to the number of available colors.

You can control which of your image's axes are on the vertical and horizontal display axes and which on the animation or 'movie' axis, within the `Display axes` drop-down. You must set the X, Y and Z (animation) axes so that each shows a *different* image axis, in order for your choice to take effect.

If your image has a fourth axis (typically Stokes), it can be controlled by a slider within the `Hidden axes` drop-down.

### 7.3.2 Viewing a contour map

Viewing a contour image is similar the process above. A contour map shows lines of equal data value (e.g., flux density) for the selected plane of gridded data (Figure 7.7). Several `Basic Settings` options control the contour levels used. 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.

### 7.3.3 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 Data Options Panel will have multiple tabs which allow adjusting each overlay individually (Note tabs along the top). **Beware:** it's easy to forget which tab is active! Also note that `axis labeling` is controlled by the *first-registered* image overlay that has labeling turned on (whether raster or contour), so make label adjustments within that tab.

To add a Contour overlay, open the **Load Data** panel (Use the **Data** menu or click on the Folder icon), select the data set and select `Contour`. See Figure 7.8 for an example using NGC5921.

### 7.3.4 Spectral Profile Plotting

From the **Tools** menu, the `Spectral Profile` plotting tool can be selected. This will pop up a new **Image Profile** window containing an x-y plot of the intensity versus spectral axis (usually velocity). You can then select a region with the **Rectangle** or **Polygon Region** drawing tools, or pinpoint a position using the **Crosshair** tool. The profile for the region or position selected will then appear in the **Image Profile** window. This profile will update in real time to track changes to the region or crosshair, which can be moved by click-dragging the mouse. See Figure 7.9.

### 7.3.5 Adjusting Canvas Parameters/Multi-panel displays

The display area can also be manipulated with the following controls in the **Panel Options** (or 'Viewer Canvas Manager') window. Use the wrench icon with a 'P' (or the 'Display Panel' menu) to show this window.

- Margins - specify the spacing for the left, right, top, and bottom margins

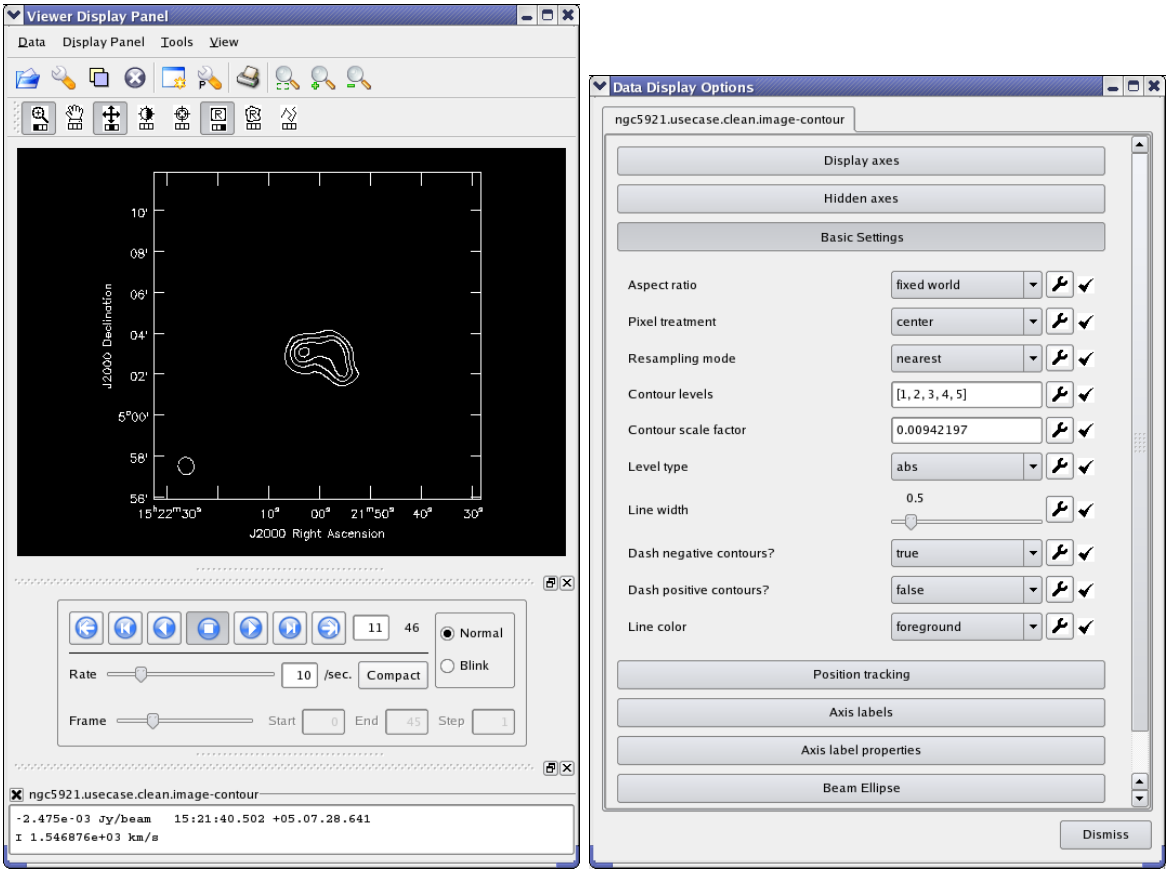



Figure 7.7: The **Viewer Display Panel** (left) and **Data Display Options** panel (right) after choosing `Contour Map` from the **Load Data** panel. The image shown is for channel 11 of the NGC5921 cube, selected using the **Animator** tape deck, and zoomed in using the tool bar icon. Note the different options in the open `Basic Settings` category of the **Data Display Options** panel.

---

- Number of panels - specify the number of panels in x and y and the spacing between those panels.
- Background Color - white or black (more choices to come)

******

#### 7.3.5.1 Setting up multi-panel displays

Figure 7.10 illustrates a multi-panel display along with the Viewer Canvas Manager settings which created it.

******

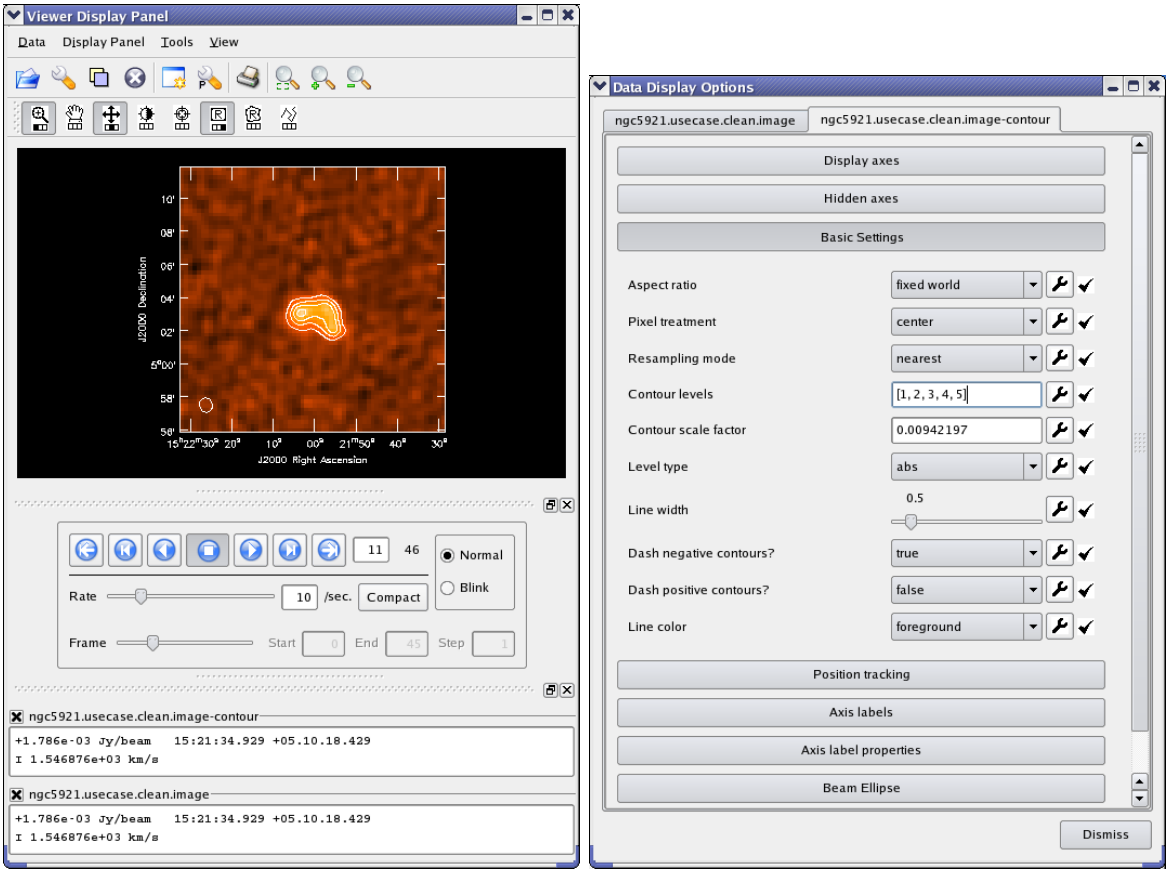



Figure 7.8: The **Viewer Display Panel** (left) and **Data Display Options** panel (right) after overlaying a `Contour Map` on a `Raster Image` from the same image cube. The image shown is for channel 11 of the NGC5921 cube, selected using the **Animator** tape deck, and zoomed in using the tool bar icon. The tab for the contour plot is open in the **Data Display Options** panel.

---

#### 7.3.5.2 Background Color

The **Background Color** selection can be used to change the background color from its default of `black`. Currently, the only other choice is `white`, which is more appropriate for printing or inclusion in documents.

## 7.4 Viewing Measurement Sets

Visibility data can also be displayed and flagged directly from the viewer. For Measurement Set files the only option for display is 'Raster' (similar to AIPS task `TVFLG`). An example of MS display is shown in Figure 7.2; loading of an MS is shown in Figure 7.11.

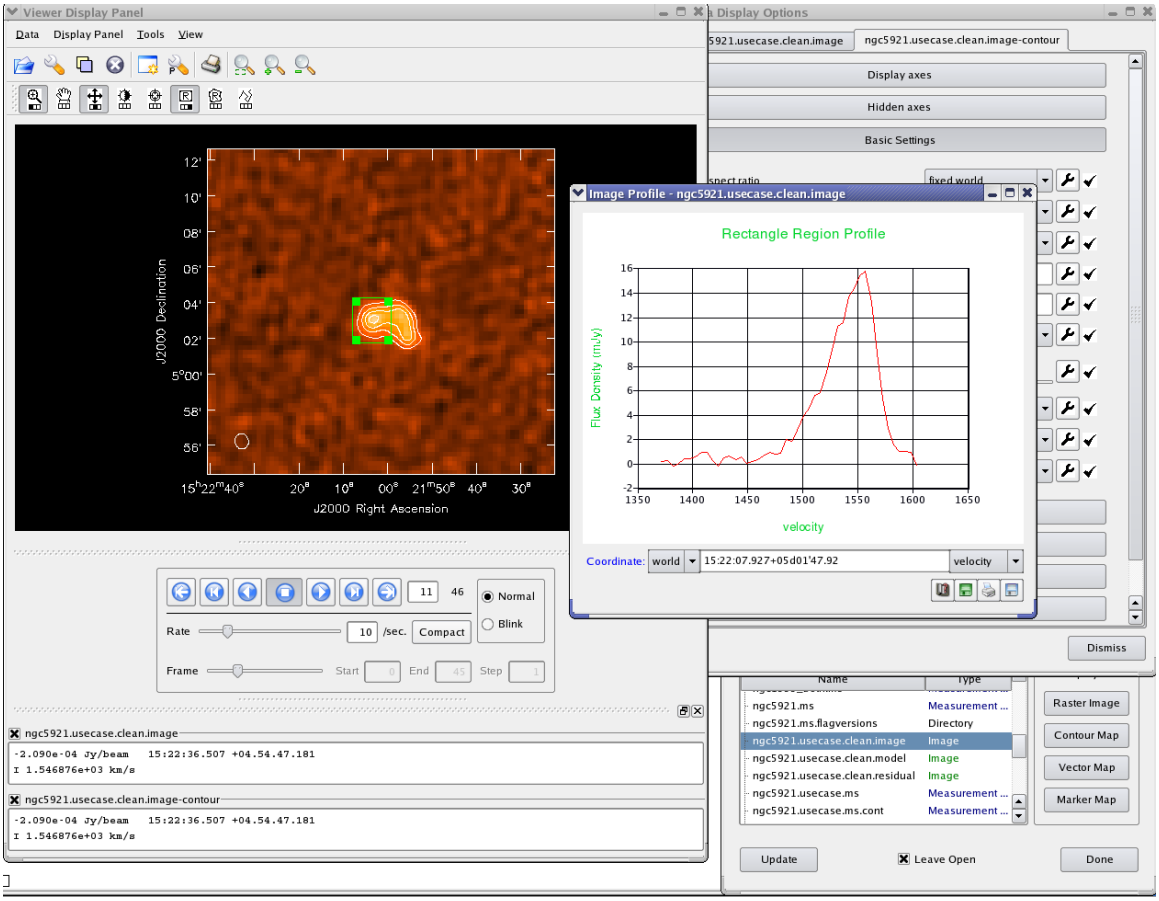



Figure 7.9: The **Image Profile** panel that appears if you use the `Tools:Spectral Profile` menu, and then use the rectangle or polygon tool to select a region in the image. You can also use the crosshair to get the profile at a single position in the image. The profile will change to track movements of the region or crosshair if moved by dragging with the mouse.

---

**Warning:** *Only one MS should be registered at a time on a Display Panel.* Only one MS can be shown in any case. You do not have to close other images/MSs, but you should at least 'unregister' them from the Display Panel used for viewing the MS. If you wish to see other images or MSs at the same time, create multiple Display Panel windows.

### 7.4.1 Data Display Options Panel for Measurement Sets

The **Data Display Options** panel provides adjustments for MSs similar to those for images, and also includes flagging options. As with images, this window appears when you choose the `Data:Adjust` menu or use the wrench icon from the **Main Toolbar**. It is also shown by default when an MS is loaded. The right panel of Figure 7.2 shows a `Data Options` window. It has a tab

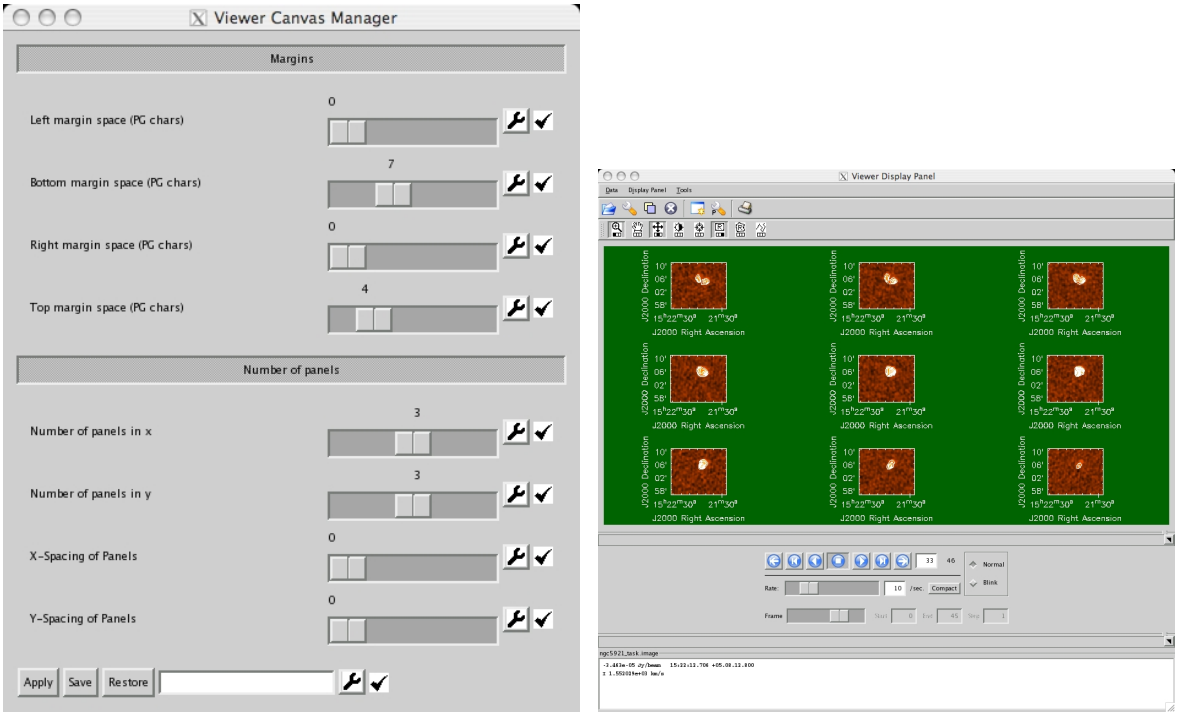



Figure 7.10: A multi-panel display set up through the **Viewer Canvas Manager**.

---

for each open MS, containing a set of categories. The options within each category can be either 'rolled up' or expanded by clicking the category label.

For a Measurement Set, the categories are:

- **Advanced**
- **MS and Visibility Selection**
- **Display Axes**
- **Flagging Options**
- **Basic Settings**
- **Axis Drawing and Labels**
- **Color Wedge**

#### 7.4.1.1 MS Options — Basic Settings

The **Basic Settings** roll-up is expanded by default. It contains entries similar to those for a raster image (§ 7.3.1.1). Together with the brightness/contrast and colormap adjustment icons on the

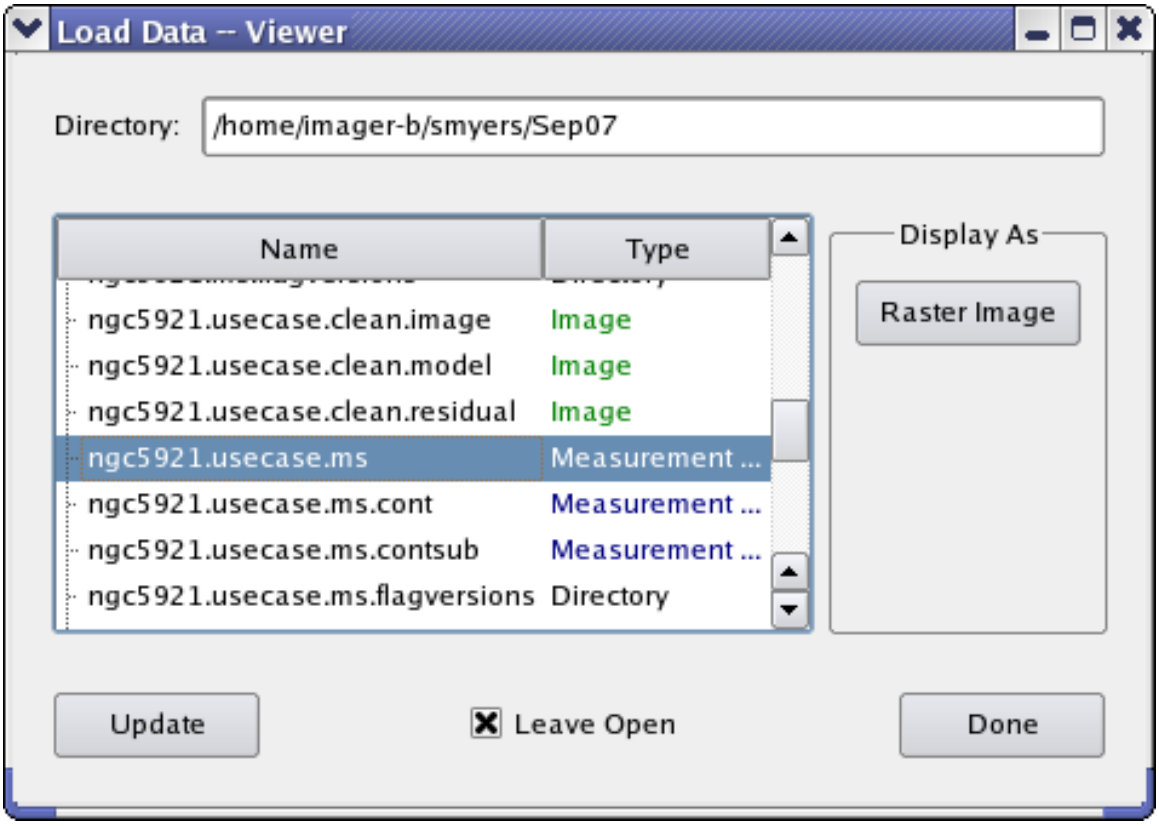



Figure 7.11: The **Load Data - Viewer** panel as it appears if you select an MS. The only option available is to load this as a `Raster Image`. In this example, clicking on the `Raster Image` button would bring up the displays shown in Figure 7.2.

---

`Mouse Toolbar` of the Display Panel, they are especially important for adjusting the color display of your MS.

The available Basic options are:

- `Data minimum/maximum`

  This has the same usage as for raster images. Lowering the data maximum will help brighten weaker data values.

- `Scaling power cycles`

  This has exactly the same usage as for raster images (see § 7.3.1.1). Again, lowering this value often helps make weaker data visible. If you want to view several fields with very different amplitudes simultaneously, this is typically one of the best adjustments to make early, together with the `Colormap fiddling` mouse tool, which is on the middle mouse button by default.

- `Colormap`

`Greyscale` or `Hot Metal` colormaps are generally good choices for MS data.

#### 7.4.1.2 MS Options— MS and Visibility Selections

- `Visibility Type`
- `Visibility Component`
- `Moving Average Size`

This roll-up provides choice boxes for Visibility Type (Observed, Corrected, Model, Residual) and Component (Amplitude, Phase, Real, or Imaginary).

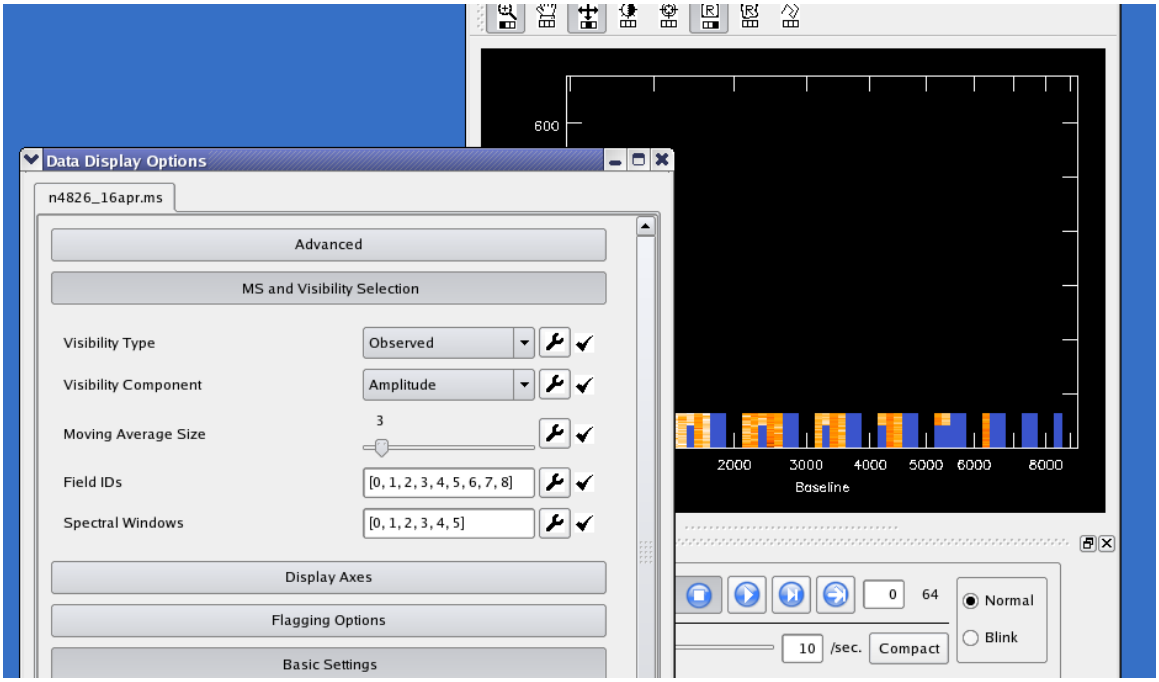



Figure 7.12: The MS for NGC4826 BIMA observations has been loaded into the viewer. We see the first of the `spw` in the Display Panel, and have opened up `MS and Visibility Selections` in the **Data Display Options** panel. The display panel raster is not full of visibiltiies because `spw 0` is continuum and was only observed for the first few scans. This is a case where the different spectral windows have different numbers of channels also.

---

Changes to Visibility Type or Component (changing from Phase to Amplitude, for example) require the data to be retrieved again from the disk into memory, which can be a lengthy process. When a large MS is first selected for viewing, the user must trigger this retrieval manually by pressing the **Apply** button (located below all the options), after selecting the data to be viewed (see `Field IDs` and `Spectral Windows`, below).

**Tip:** Changing visibility type between 'Observed' and 'Corrected' can also be used to assure that data and flags are reloaded from disk. You should do this if you're using another flagging tool such as autoflag simultaneously, so that the viewer sees the other tool's new edits and doesn't overwrite them with obsolete flags. The **Apply** button alone won't reload unless something within the viewer itself requires it; in the future, a button will be provided to reload flags from the disk unconditionally.

You can also choose to view the difference from a running mean or the local RMS deviation of either Phase or Amplitude. There is a slider for choosing the nominal number of time slots in the 'local neighborhood' for these displays.

(Note: **Insufficient Data** is shown in the tracking area during these displays when there is no other unflagged data in the local neighborhood to compare to the point in question. The moving time windows will not extend across changes in either field ID or scan number boundaries, so you may see this message if your scan numbers change with every time stamp. An option will be added later to ignore scan boundaries).

- `Field IDs`
- `Spectral Windows`

You can retrieve and edit a selected portion of the MS data by entering the desired Spectral Window and Field ID numbers into these boxes. **Important:** Especially with large MSs, often the first thing you'll want to do is to select **spectral windows** which all have the **same number of channels** and the **same polarization setup**. It also makes sense to edit only a few fields at a time. Doing this will also greatly reduce data retrieval times and memory requirements.

You can separate the ID numbers with spaces or commas; you do not need to enter enclosing brackets. Changes to either entry box will cause the selected MS data to be reloaded from disk.

If you select, say, spectral windows 7, 8, 23, and 24, the animator, slice position sliders, and axis labeling will show these as 0, 1, 2, and 3 (the 'slice positions' or 'pixel coordinates' of the chosen spectral windows). Looking at the position tracking display is the best way to avoid confusion in such cases. It will show something like: `Sp Win 23 (s 2)` when you are viewing spectral window 23 (plane 2 of the selected spectral windows).

Changes to MS selections will not be allowed until you have saved (or discarded) any previous edits you have made (see `Flagging Options -- Save Edits`, below). A warning is printed on the console (not the logger).

Initially, all fields and spectral windows are selected. To revert to this 'unselected' state, choose 'Original' under the wrench icons next to the entry boxes.

See Figure 7.12 for an example showing the use of the `MS and Visibility Selections` controls when viewing an MS.

#### 7.4.1.3 MS Options — Display Axes

This roll-up is very similar to that for images: it allows the user to choose which axes (from Time, Baseline, Polarization, Channel, and Spectral Window) are are on the display and the animator. There are also sliders here for choosing positions on the remaining axes. (It's useful to note that the data *is* actually stored internally in memory as an array with these five axes).

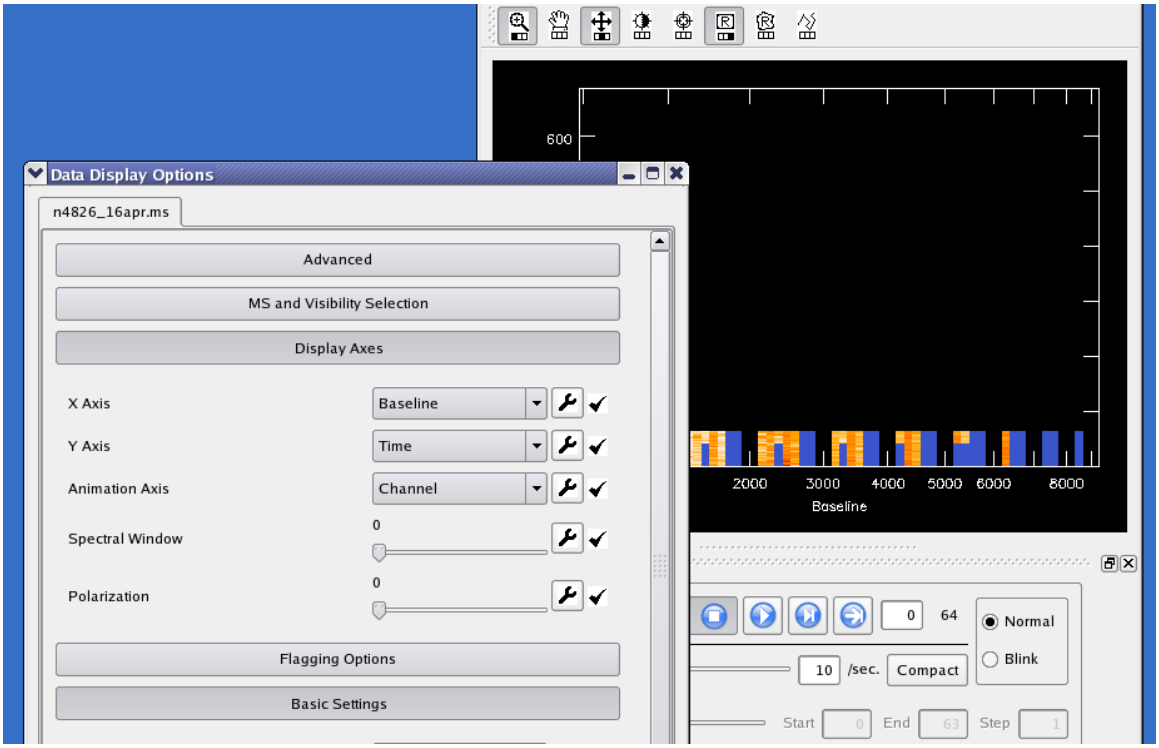



Figure 7.13: The MS for NGC4826 from Figure 7.12, now with the `Display Axes` open in the **Data Display Options** panel. By default, `channels` are on the **Animation Axis** and thus in the tapedeck, while `spectral window` and `polarization` are on the `Display Axes` sliders.

---

For MSs, changing the choice of axis on one control will automatically swap axes, maintaining different axes on each control. Changing axes or slider/animator positions does not normally require pressing **Apply** — the new slice is shown immediately. However, the display may be partially or completely grey in areas if the required data is not currently in memory, either because no data has been loaded yet, or because not all the selected data will fit into the allowed memory. Press the **Apply** button in this case to load the data (see § 7.4.1.6 and `Max. Visibility Memory` at the end of § 7.4.1.5).

See Figures 7.13–7.14 showing the use of the `Display Axes` controls to change the axes on the animation and sliders.


Figure 7.14: The MS for NGC4826, continuing from Figure 7.13. We have now put `spectral window` on the **Animation Axis** and used the tapedeck to step to `spw 2`, where we see the data from the rest of the scans. Now `channels` is on a `Display Axes` slider, which has been dragged to show `Channel 33`.

---

#### 7.4.1.4 MS Options — Flagging Options

These options allow you to edit (flag or unflag) MS data. The Crosshair and Rectangle Region **Mouse Tools** (§ 7.2.2) are used on the display to select the area to edit. When using the Rectangle Region tool, double-click inside the selected rectangle to confirm the edit.

The options below determine how edits will be applied.

- `Show Flagged Regions...`

  You have the option to display flagged regions in the background color (as in `TVFLG`) or to highlight them with color. In the former case, flagged regions look just like regions of no data. With the (default) color option, flags are shown in shades of blue: darker blue for flags already saved to disk, lighter blue for new flags not yet saved; regions with no data will be shown in black.

- `Flag or Unflag`

  This setting determines whether selected regions will be flagged or unflagged. This does *not*

affect previous edits; it only determines the effect which later edits will have. Both flagging and unflagging edits can be accumulated and then saved in one pass through the MS.

- `Flag/Unflag All...`

  These flagging extent checkboxes allow you to extend your edit over any of the five data axes. For example, to flag *all* the data in a given time range, you would check all the axes *except* Time, and then select the desired time range with the `Rectangle Region` mouse tool. Such edits will extend along the corresponding axes over the entire selected MS (whether loaded into memory or not) and optionally over unselected portions of the MS as well (`Use Entire MS`, below). Use care in selecting edit extents to assure that you're editing all the data you wish to edit.

- `Flag/Unflag Entire Antenna?`

  This control can be used to extend subsequent edits to all baselines which include the desired antenna[s]. For example, if you set this item to 'Yes' and then click the crosshair on a visibility point with baseline 3-19, the edit would extend over baselines 0-3, 1-3, 2-3, 3-3, 3-4, ... 3-`nAntennas-1`. Note that the second antenna of the selection (19) is irrelevant here – you can click anywhere within the 'Antenna 3 block', i.e., where the *first* antenna number is 3, to select all baselines which include antenna 3.

  This item controls the edit extent only along the baseline axis. If you wish to flag *all* the data for a given antenna, you must still check the boxes to flag all Times, Channels, Polarizations and Spectral Windows. There would be no point, however, in activating *both* this item and the 'Flag All Baselines' checkbox. You can flag an antenna in a limited range of times, etc., by using the appropriate checkboxes and selecting a rectangular region of visibilities with the mouse.

  **Note:** You do not need to include the entire 'antenna block' in your rectangle (and you may stray into the next antenna if you try). Anywhere within the block will work. To flag higher-numbered antennas, it often helps to zoom in.

- `Undo Last Edit`

- `Undo All Edits`

  The 'Undo' buttons do the expected thing: completely undo the effect of the last edit (or all unsaved edits). Please note, however, that only unsaved edits can be undone here; there is no ability to revert to the flagging state at the start of the session once flags have been saved to disk (unless you have previously saved a 'flag version'. The flag version tool is not available through the viewer directly).

- `Use Entire MS When Saving Edits?`

  "Yes" means that saving the edits will flag/unflag over the entire MS, *including* fields (and possibly spectral windows) which are not currently selected for viewing. Specifically, data within time range(s) you swept out with the mouse (even for unselected fields) will be edited.

  In addition, if "Flag/Unflag All..." boxes were checked, such edits will extend throughout the MS. Note that only unselected *times* (fields) can be edited *without* checking extent boxes for

the edits as well. Unselected spectral windows, e.g., will *not* be edited unless the edit also has "Flag/Unflag All Spectral Windows" checked.

Warning: Beware of checking "All Spectral Windows" unless you have also checked "All Channels" or turned "Entire MS" off; channel edits appropriate to the selected spectral windows may not be appropriate to unselected ones. Set "Use Entire MS" to"No" if your edits need to apply only to the portion of the MS you have selected for viewing. *Edits can often be saved significantly faster this way as well.*

Also note that checkboxes apply to individual edits, and must be checked before making the edit with the mouse. "Use Entire MS", on the other hand, applies to all the edits saved at one time, and must be set as desired before pressing "Save Edits".

- `Save Edits`

  MS editing works like a text editor in that you see all of your edits immediately, but nothing is committed to disk until you press 'Save Edits'. Feel free to experiment with all the other controls; nothing but 'Save Edits' will alter your MS on disk. As mentioned previously, however, there is no way to undo your edits once they are saved, except by manually entering the reverse edits (or restoring a previously-saved 'flag version').

  Also, *you must save* (or discard) *your edits before changing the MS selections.* If edits are pending, the selection change will not be allowed, and a warning will appear on the console.

  If you close the MS in the viewer, *unsaved edits are simply discarded*, without prior warning. It's important, therefore, to remember to save them yourself. You can distinguish unsaved flags (when using the 'Flags In Color' option), because they are in a lighter shade of blue.

  The program must make a pass through the MS on disk to save the edits. This can take a little time; progress is shown in the console window.

#### 7.4.1.5 MS Options— Advanced

These settings can help optimize your memory usage, especially for large MSs. A rule of thumb is that they can be increased until response becomes sluggish, when they should be backed down again.

You can run the unix 'top' program and hit 'M' in it (to sort by memory usage) in order to examine the effects of these settings. Look at the amount of RSS (main memory) and SWAP used by the X server and 'casaviewer' processes. If that sounds familiar and easy, then fiddling with these settings is for you. Otherwise, the default settings should provide reasonable performance in most cases.

- `Cache size`

  The value of this option specifies the maximum number of different views of the data to save so that they can be redrawn quickly. If you run an animation or scroll around zoomed data, you will notice that the data displays noticeably faster the second time through because of this feature. Often, setting this value to the number of animation frames is ideal Note, however, that on multi-panel displays, each panel counts as one cached image.

Large images naturally take more room than small ones. The memory used for these images will show up in the X server process. If you need more Visibility Memory (below) for a really large ms, it is usually better to forgo caching a large number of views.

- `Max. Visibility Memory`

  This option specifies how many megabytes of memory may be used to store visibility data from the measurement set internally. *Even if you do not adjust this entry, it is useful to look at it to see how many megabytes are required to store your entire (selected) MS in memory.* If the slider setting is above this, the whole selected MS will fit into the memory buffer. Otherwise, some data planes will be 'greyed out' (see `Apply Button`, § 7.4.1.6 below), and the selected data will have to be viewed one buffer at a time, which is somewhat less convenient. In most cases, this means you should **select fewer fields or spectral windows** – see § 7.4.1.2. The 'casaviewer' process contains this buffer memory (it contains the entire viewer, but the memory buffer can take most of the space).

#### 7.4.1.6 MS Options — Apply Button

When viewing large MSs the display may be partially or completely grey in areas where the required data is not currently in memory, either because no data has been loaded yet, or because not all the selected data will fit into the allowed memory (see `Max. Visibility Memory` above). When the cursor is over such an area, the following message shows in the position tracking area:

```
press 'Apply' on Adjust panel to load data
```

Pressing the **Apply** button (which lies below all the options) will reload the memory buffer so that it includes the slice you are trying to view.

The message **No Data** has a different meaning; in that case, there simply *is* no data in the selected MS at the indicated position.

For large measurement sets, loading visibility data into memory is the most time-consuming step. Progress feedback is provided in the console window. Again, careful selection of the data to be viewed can greatly speed up retrieval.

## 7.5 Printing from the Viewer

You can use the `Data:Print` menu or the **Print** button to bring up the **Viewer Print Manager**. From this panel, you can print a hardcopy of what is in the Display Panel, or save it in a variety of formats.

Figure 7.15 shows an example of printing to a file. The key to making acceptable hardcopies (particularly for printing or inclusion in documents) is to set the background color and line widths to appropriate values so the plot and labels show up in the limited resolution of the hardcopy.

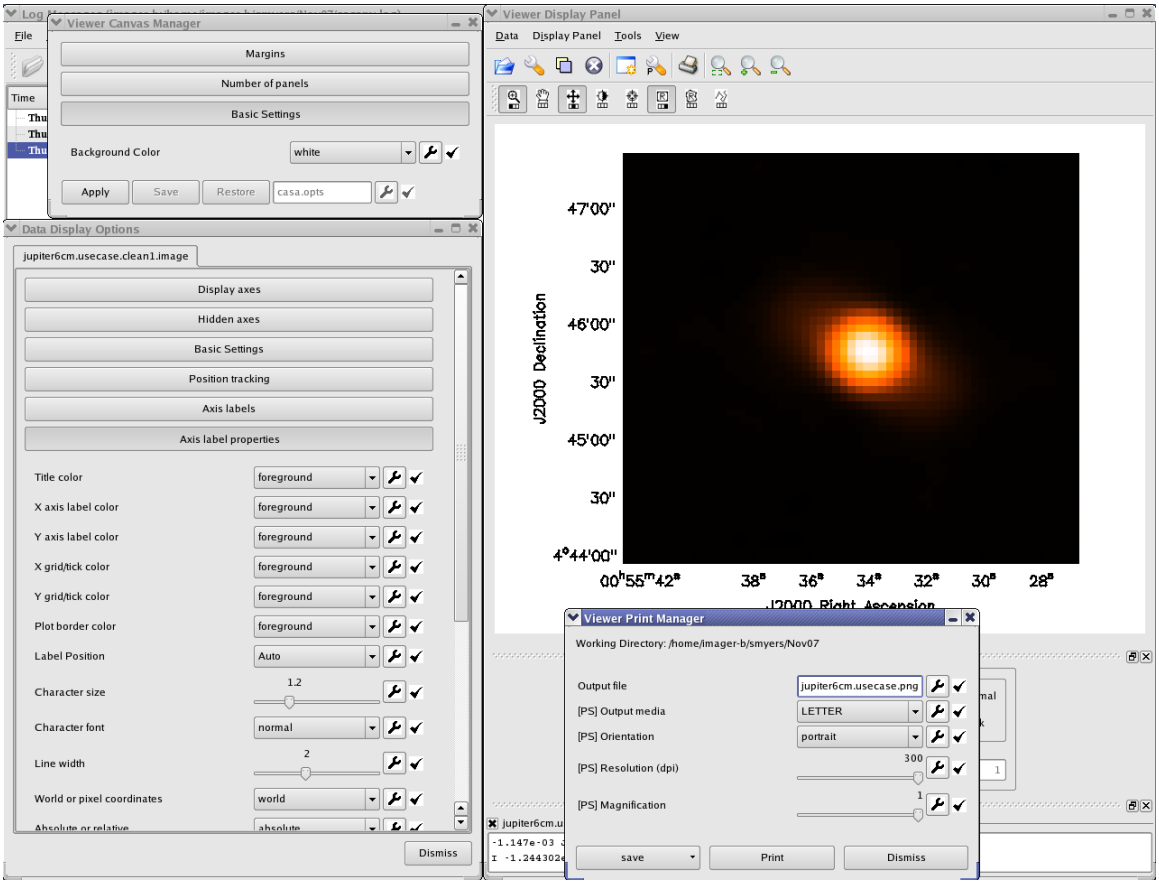



Figure 7.15: Setting up to print to a file. The background color has been set to `white`, the line width to `2`, and the print resolution to `300` dpi (for a postscript plot). A name has been given in preparation for saving as a PNG raster. To make the plot, use the **Save** button on the **Viewer Print Manager** panel (positioned by the user below the display area) and select a format with the drop-down, or use the **Print** button to send directly to a printer.

---

Use the **Viewer Canvas Manager** (§ 7.3.5) to change the **Background Color** from its default of `black` to `white` if you are making plots for printing or inclusion in a document. You might also want to change the **colormap** accordingly.

Adjust the **Line Width** of the **Axis Label Properties** options in the **Data Display Options** panel so that the labels will be visible when printed. Increasing from the default of `1.4` to a value around `2` seems to work well.

You can choose an output file name in the panel. Be sure to make it a new name, otherwise it will not overwrite a previous file (and will not say anything about it).

If you will be printing to a postscript printer or to a PS file, dial up the **[PS] Resolution (dpi)** to its maximum of `300`. This will increase the size of the PS file unfortunately, but will make a much

better plot. Use `gzip` to compress the PS file if necessary. Be sure to choose the desired Output Media and Orientation for PS also.

**BETA ALERT:** The postscript printing capabilities of the `casaviewer` are currently fairly poor, due to some limitations in Qt and the way we do axis labels. This will be upgraded in the future, but for now you will need to follow the suggestions above to get a useable plot. Note that `ghostview` may show a poorer version of the PS than you will get when you print. You may also wish to consider outputting as PNG and then using another program such as `convert` to turn into PS.

# Appendix A

# Single Dish Data Processing

**BETA ALERT:** The single-dish analysis package within CASA is still largely toolkit-based, with a few experimental basic tasks thrown in. It is included in the Beta release for the use of the ALMA computing and commissioning groups, and is not intended for general users. Therefore, this is included in this Cookbook as an appendix.

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 § A.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 § A.5 first.

## A.1 Guidelines for Use of ASAP and SDtasks in CASA

### A.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 § A.3.1 for more details on the ASAP environment variables.

### A.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)
```

### A.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]].
```

### A.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.

### A.1.5 Line Formatting

The SDtasks trap leading and trailing whitespace on string parameters (such as `infile` and `sdfile`), 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.

## A.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 § A.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 A.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`).

### A.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
sdfile -- Name of output file
        default: '' (<infile>_cal)
        example: note that sdfile is the OUTPUT of sdcal and
                 is the INPUT filename param for the other
                 sdtasks to steamline processing
        WARNING: output file will be overwritten
outform -- 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 sdfile name
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
                 <sdfile>_calspec.eps)
        WARNING: be careful plotting in fsotf 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 consequtive 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:
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: (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 consequtive 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 afterward 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 paneling, 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.

### A.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 follwing 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_rawACSmod'

# 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 sdcal 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 sdcal this
calmode = 'ps'

# For GBT data, it is safest to not have scantable pre-average
# integrations within scans.
scanaverage = False

# We do want sdcal 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
# sdcal.last which contains the inputs from the last
# run of the task (all tasks do this). You can recover
# this (anytime before sdcal is run again) with
#execfile 'sdcal.last'

##########################
# List data
##########################
# List the contents of the calibrated dataset
# Set the input to the just created file
infile = sdfile
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
# sdfile as the input file, which is the output
# file of sdcal).
sdfile = '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
#
##########################
```

## A.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>
sd.__class__                sd._validate_bool          sd.list_scans
sd.__date__                 sd._validate_int           sd.mask_and
sd.__delattr__              sd.asapfitter              sd.mask_not
sd.__dict__                 sd.asaplinefind            sd.mask_or
sd.__doc__                  sd.asaplog                 sd.merge
sd.__file__                 sd.asaplotbase             sd.os
sd.__getattribute__         sd.asaplotgui              sd.plf
sd.__hash__                 sd.asapmath                sd.plotter
sd.__init__                 sd.asapplotter             sd.print_log
sd.__name__                 sd.asapreader              sd.quotient
sd.__new__                  sd.average_time            sd.rc
sd.__path__                 sd.calfs                   sd.rcParams
sd.__reduce__               sd.calnod                  sd.rcParamsDefault
sd.__reduce_ex__            sd.calps                   sd.rc_params
sd.__repr__                 sd.commands                sd.rcdefaults
sd.__setattr__              sd.defaultParams           sd.reader
sd.__str__                  sd.dosigref                sd.scantable
sd.__version__              sd.dototalpower            sd.selector
sd._asap                    sd.fitter                  sd.simple_math
sd._asap_fname              sd.is_ipython              sd.sys
sd._asaplog                 sd.linecatalog             sd.unique
sd._is_sequence_or_number   sd.linefinder              sd.version
sd._n_bools                 sd.list_files              sd.welcome
sd._to_list                 sd.list_rcparameters       sd.xyplotter
```

...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.

### A.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.
```

### A.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 Measurement Set). 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 Measurement Sets 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 Measurement Set 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
```

### A.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()
```

#### A.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
```

#### A.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 § A.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.

#### A.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.

#### A.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>
```

#### A.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.

#### A.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 loosing 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.
- Measurement Set (V2: CASA format) – Saves the data in a Measurement Set. 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')
```

### A.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.

#### A.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
```

#### A.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 telecope (alternative)
scans.convert_flux() # known telescope (mostly AT telescopes)
scans.convert_flux(eta=0.48) # if telescope diameter known
```

#### A.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)
```

#### A.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:

$T_{sys} = T_{cal}$ x $\frac{<ref_{caloff}>}{<ref_{calon} - ref_{caloff}>} + \frac{T_{cal}}{2}$

`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:

$T_a(\nu) = T_{sys}$ x $\frac{sig(\nu) - ref(\nu)}{ref(\nu)}$

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 minimal checking.

### A.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)
```

### A.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
```

### A.3.7 Baseline Fitting

The function `sd.scantable.poly_baseline` carries out a baseline fit, given an mask of channels (if desired):

```
msk=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 retreived 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 consequtive 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
                indivual 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)
```

### A.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)
```

### A.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.

### A.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 HNCO
#  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 HNC18O
#  15       8192 LSRK  43397.8198  6.10423356  50005.8813  43422.8227  RR  LL SiO

# Scans: 21-24  Setup 1 HC3N et al
# Scans: 25-28  Setup 2 SiO et al

casapath=os.environ['AIPSPATH']

#ASAP script                              # COMMENTS
#------------------------------------- ----------------------------------------------
import asap as sd                         #import ASAP package into CASA
                                          #Orion-S (SiO line reduction only)
                                          #Notes:
                                          #scan numbers (zero-based) as compared to GBTIDL

                                          #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
```

## A.4 Single Dish Imaging

Single dish imaging is supported within CASA using standard tasks and tools. The data must be in the Measurement Set 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=1000000000)      # 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')
```

### A.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.setsdoptions(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
```

## A.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 `'TOPO'`) 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.

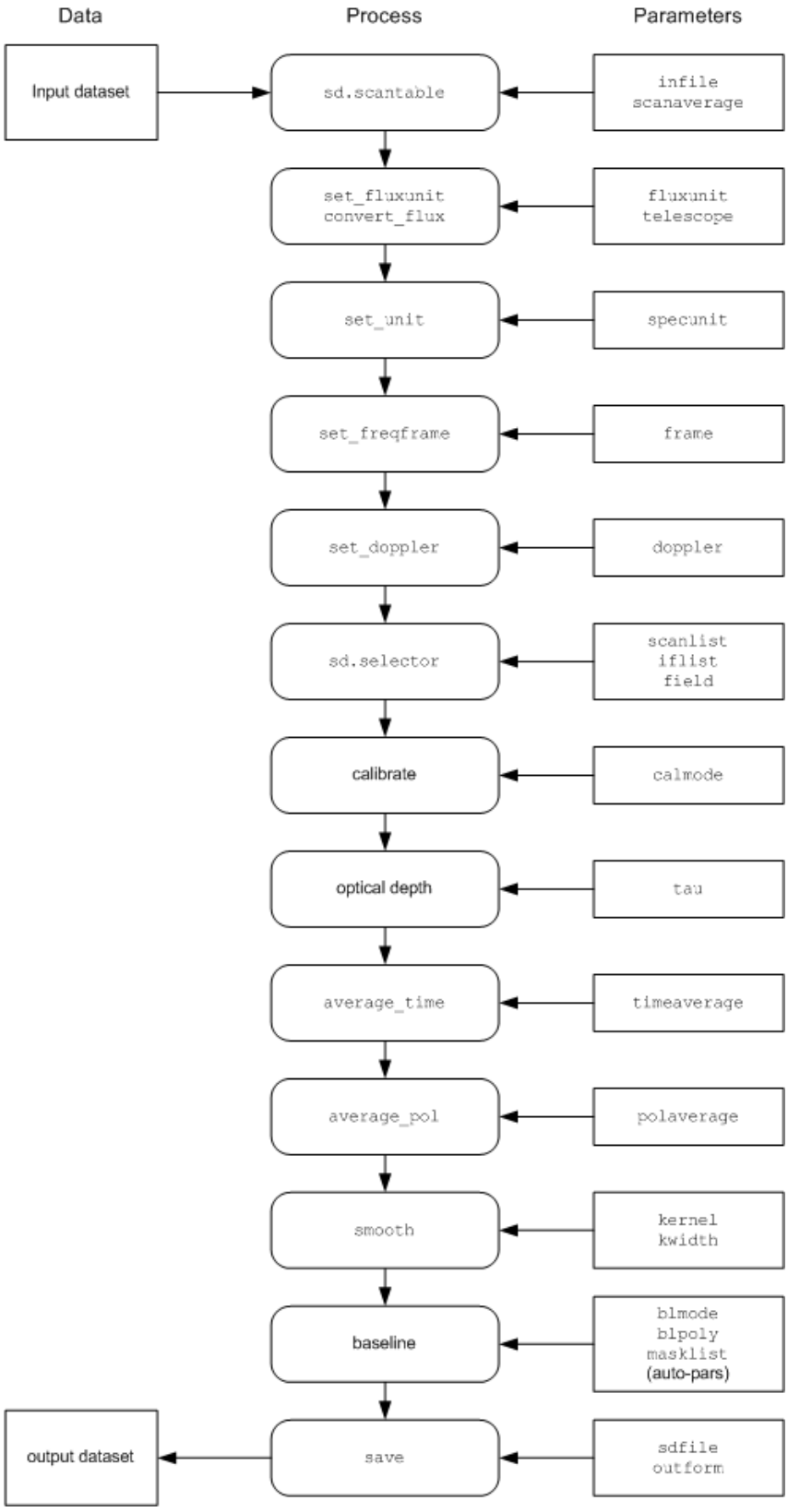



Figure A.1: Wiring diagram for the SDtask `sdcal`. The stages of processing within the task are shown, along with the parameters that control them.

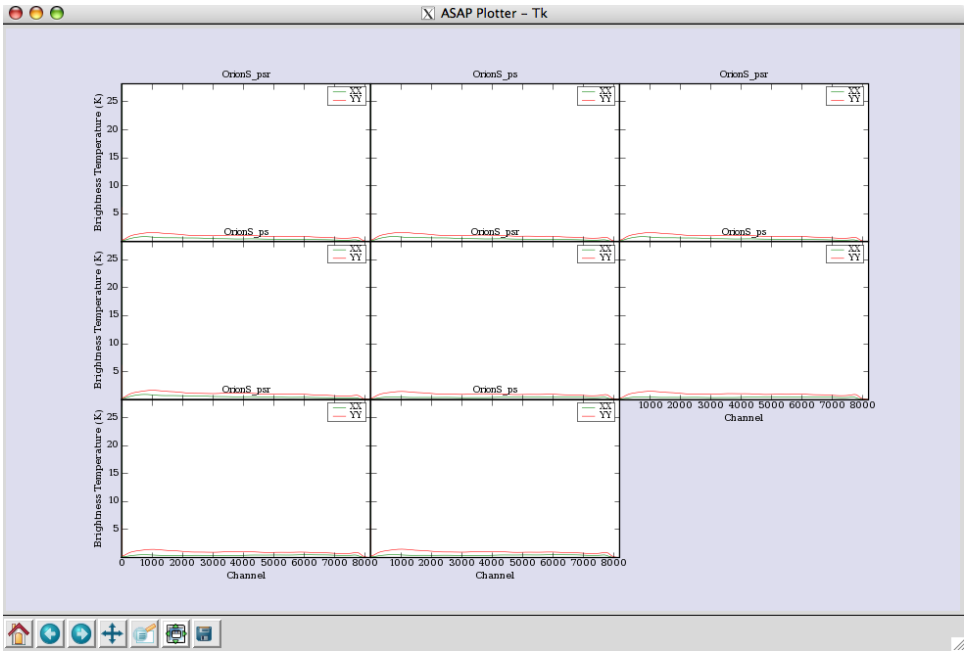



Figure A.2: Multi-panel display of the scantable. There are two plots per scan indicating the _psr (reference position data) and the _ps (source data).

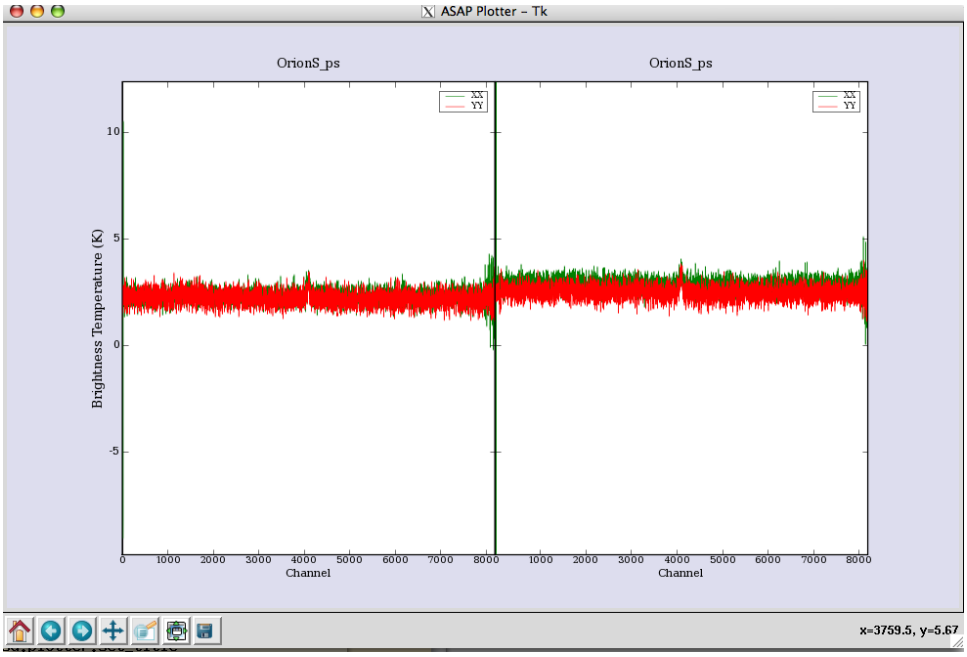



Figure A.3: Two panel plot of the calibrated spectra. The GBT data has a separate scan for the SOURCE and REFERENCE positions so scans 20,21,22 and 23 result in these two spectra.

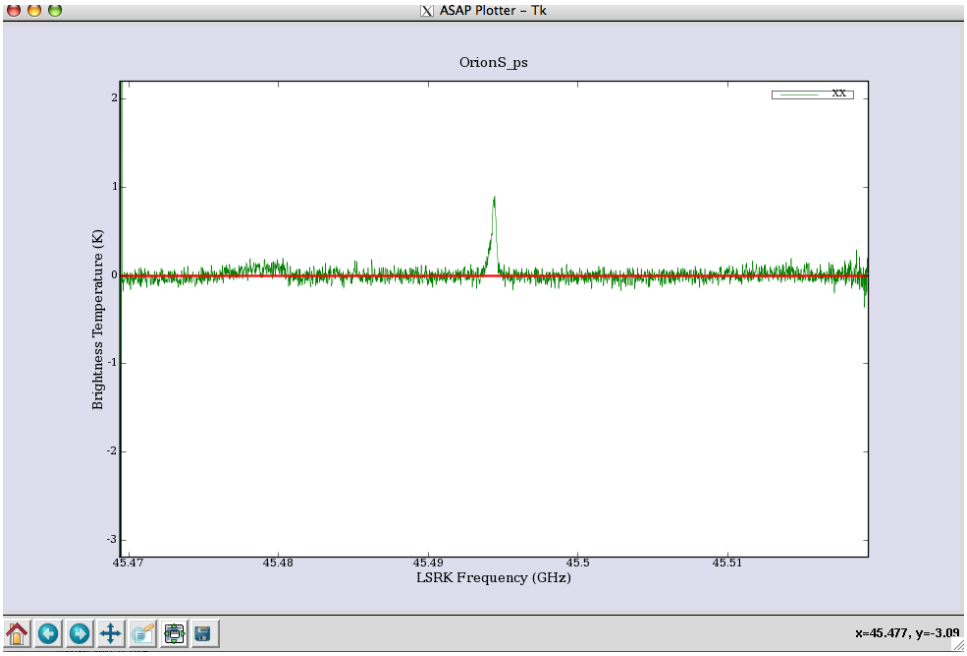



Figure A.4: Calibrated spectrum with a line at zero (using histograms).

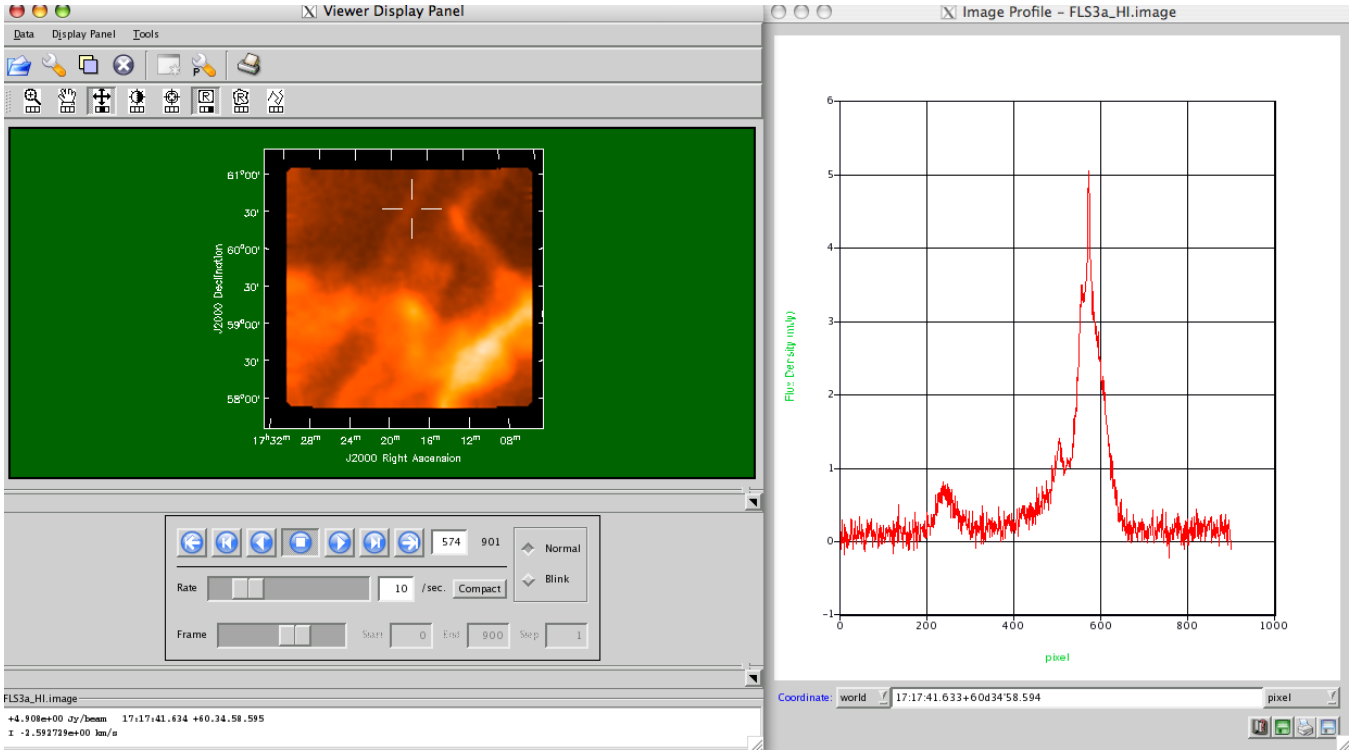



Figure A.5: FLS3a HI emission. The display illustrates the visualization of the data cube (left) and the profile display of the cube at the cursor location (right); the Tools menu of the Viewer Display Panel has a Spectral Profile button which brings up this display. By default, it grabs the left-mouse button. Pressing down the button and moving in the display will show the profile variations.

# Appendix B

# Simulation

**BETA ALERT:** The simulation capabilities are currently under development. What we do have is mostly at the Toolkit level. We have only a single task `almasimmos` at the present time. Stay tuned. For the Beta Release, we include this chapter in the Appendix for the use of telescope commissioners and software developers.

The capability for simulating observations and datasets from the EVLA and ALMA are an important use-case for CASA. This not only allows one to get an idea of the capabilities of these instruments for doing science, but also provides benchmarks for the performance and utility of the software for processing "realistic" datasets. To that end, we are developing the `simulator` (`sm`) tool, as well as a series of simulation tasks.

**Inside the Toolkit:**
The simulator methods are in the `sm` tool. Many of the other tools are also helpful when constructing and analyzing simulations.

## B.1 Simulating ALMA with `almasimmos`

**BETA ALERT:** This is an experimental task that is under development. Its functionality and parameters will be changing, so check the on-line documentation for the latest updates.

The inputs are:

```
#  almasimmos :: ALMA Mosaic simulation task

                Please see the on-line documentation for this task.

project         =    'mysim'   #   name of simulated project
modelimage      =         ''   #   image name to derive simulate visibilities
complist        =         ''   #   componentlist table to derive simulated visibilities
antennalist     =         ''   #   antenna position ascii file
direction       = 'J2000 19h00m00 -40d00m00'  #   mosaic center direction
nmosx           =          1   #   number of pointings along x
```

```
nmosy           =          1   #   number of pointings along y
pointingspacing =  '5arcmin'   #   spacing in between beams
refdate         = '2012/05/21/22:05:00'     #   center time/date of simulated observation
totaltime       =    '7200s'   #   total time of observation
integration     =      '10s'   #   integration (sampling) time
mode            =  'channel'   #   type of selection: channel, continuum
alg             =    'clark'   #   deconvolution algorithm: clark,hogbom,multiscale
niter           =        500   #   number iterations
nchan           =          1   #   number of channels to select
startfreq       =    '89GHz'   #   nrequency of first channel
chanwidth       =    '10MHz'   #   channel width
imsize          = [250, 250]   #   Image pixel size (x,y)
cell            = '10arcsec'   #   Cell size e.g., 10arcsec
stokes          =        'I'   #   Stokes parameters to image
weighting       =  'natural'   #   Weighting of visibilities
display         =       True   #   Plot simulation result images,figures
```

This task takes an input model image or list of components, plus a list of antennas (locations and sizes), and simulates a particular observation (specifies by mosaic setup and observing cycles and times). This is currently very simplistic. For example, it does not include noise by default, or gain errors (but see the on-line wiki documentation for how to do these). The output is a MS suitable for further processing in CASA.

Its name implies that it is for ALMA, but it is mostly general as you can give it any antenna setup — it does have the ALMA observatory location hardwired in and sets the telescope name to `'ALMA''` but thats about it. The task could be easily modified for other instruments.

**BETA ALERT:** Because of the experimental nature of this task, we do not provide extensive documentation in this cookbook. For this purpose, there is an on-line "wiki" devoted to this task:

`https://wikio.nrao.edu/bin/view/ALMA/SimulatorCookbook`

Here you can find what documentation we do have, along with example files that are needed to specify antenna locations, and a FAQ.

# Appendix C

# Obtaining and Installing CASA

## C.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/administrater to install CASA in this manner.

See the following for more details:

https://wikio.nrao.edu/bin/view/Software/ObtainingCASA

## C.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 D

# 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.

## D.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.

## D.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.

## D.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]
```

## D.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'
```

## D.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:

## D.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.

## D.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]:
```

## D.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
```

## D.9 Executing Python scripts

Python scripts are simple text files containing lists of commands as if typed at the keyboard. Note: the auto-parentheses 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')
```

## D.10 How do I exit from CASA?

Hit <Cntl-d> or type at the CASA command line prompt:

```
CASA>%Exit
```

and press return.

# Appendix E

# The Measurement Equation and Calibration

The visibilities measured by an interferometer must be calibrated before formation of an image. This is because the wavefronts received and processed by the observational hardware have been corrupted by a variety of effects. These include (but are not exclusive to): the effects of transmission through the atmosphere, the imperfect details amplified electronic (digital) signal and transmission through the signal processing system, and the effects of formation of the cross-power spectra by a correlator. Calibration is the process of reversing these effects to arrive at corrected visibilities which resemble as closely as possible the visibilities that would have been measured in vacuum by a perfect system. The subject of this chapter is the determination of these effects by using the visibility data itself.

## E.1 The HBS Measurement Equation

The relationship between the observed and ideal (desired) visibilities on the baseline between antennas i and j may be expressed by the Hamaker-Bregman-Sault *Measurement Equation*[1]:

$$\vec{V}_{ij} \;=\; J_{ij}\;\vec{V}_{ij}^{\,\mathrm{IDEAL}}$$

where $\vec{V}_{ij}$ represents the observed visibility, $\vec{V}_{ij}^{\,\mathrm{IDEAL}}$ represents the corresponding ideal visibilities, and $J_{ij}$ represents the accumulation of all corruptions affecting baseline $ij$. The visibilities are indicated as vectors spanning the four correlation combinations which can be formed from dual-polarization signals. These four correlations are related directly to the Stokes parameters which fully describe the radiation. The $J_{ij}$ term is therefore a 4×4 matrix.

Most of the effects contained in $J_{ij}$ (indeed, the most important of them) are antenna-based, i.e., they arise from measurable physical properties of (or above) individual antenna elements in a synthesis array. Thus, adequate calibration of an array of $N_{ant}$ antennas forming $N_{ant}(N_{ant}-1)/2$ baseline visibilities is usually achieved through the determination of only $N_{ant}$ factors, such that

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

$J_{ij} = J_i \otimes J_j^*$. For the rest of this chapter, we will usually assume that $J_{ij}$ is factorable in this way, unless otherwise noted.

As implied above, $J_{ij}$ may also be factored into the sequence of specific corrupting effects, each having their own particular (relative) importance and physical origin, which determines their unique algebra. Including the most commonly considered effects, the Measurement Equation can be written:

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

where:

- $T_{ij}$ = Polarization-independent multiplicative effects introduced by the troposphere, such as opacity and path-length variation.
- $P_{ij}$ = Parallactic angle, which describes the orientation of the polarization coordinates on the plane of the sky. This term varies according to the type of the antenna mount.
- $E_{ij}$ = Effects introduced by properties of the optical components of the telescopes, such as the collecting area's dependence on elevation.
- $D_{ij}$ = Instrumental polarization response. "D-terms" describe the polarization leakage between feeds (e.g. how much the R-polarized feed picked up L-polarized emission, and vice versa).
- $G_{ij}$ = Electronic gain response due to components in the signal path between the feed and the correlator. This complex gain term $G_{ij}$ includes the scale factor for absolute flux density calibration, and may include phase and amplitude corrections due to changes in the atmosphere (in lieu of $T_{ij}$). These gains are polarization-dependent.
- $B_{ij}$ = Bandpass (frequency-dependent) response, such as that introduced by spectral filters in the electronic transmission system
- $M_{ij}$ = Baseline-based correlator (non-closing) errors. By definition, these are not factorable into antenna-based parts.

Note that the terms are listed in the order in which they affect the incoming wavefront ($G$ and $B$ represent an arbitrary sequence of such terms depending upon the details of the particular electronic system). Note that $M$ differs from all of the rest in that it is not antenna-based, and thus not factorable into terms for each antenna.

As written above, the measurement equation is very general; not all observations will require treatment of all effects, depending upon the desired dynamic range. E.g., bandpass need only be considered for continuum observations if observed in a channelized mode and very high dynamic range is desired. Similarly, instrumental polarization calibration can usually be omitted when observing (only) total intensity using circular feeds. Ultimately, however, each of these effects occurs at some level, and a complete treatment will yield the most accurate calibration. Modern high-sensitivity instruments such as ALMA and EVLA will likely require a more general calibration

treatment for similar observations with older arrays in order to reach the advertised dynamic ranges on strong sources.

In practice, it is usually far too difficult to adequately measure most calibration effects absolutely (as if in the laboratory) for use in calibration. The effects are usually far too changeable. Instead, the calibration is achieved by making observations of calibrator sources on the appropriate timescales for the relevant effects, and solving the measurement equation for them using the fact that we have $N_{ant}(N_{ant}-1)/2$ measurements and only $N_{ant}$ factors to determine (except for $M$ which is only sparingly used). (*Note: By partitioning the calibration factors into a series of consecutive effects, it might appear that the number of free parameters is some multiple of $N_{ant}$, but the relative algebra and timescales of the different effects, as well as the the multiplicity of observed polarizations and channels compensate, and it can be shown that the problem remains well-determined until, perhaps, the effects are direction-dependent within the field of view. Limited solvers for such effects are under study; the* `calibrater` *tool currently only handles effects which may be assumed constant within the field of view. Corrections for the primary beam are handled in the* `imager` *tool.*) Once determined, these terms are used to correct the visibilities measured for the scientific target. This procedure is known as cross-calibration (when only phase is considered, it is called phase-referencing).

The best calibrators are point sources at the phase center (constant visibility amplitude, zero phase), with sufficient flux density to determine the calibration factors with adequate SNR on the relevant timescale. The primary gain calibrator must be sufficiently close to the target on the sky so that its observations sample the same atmospheric effects. A bandpass calibrator usually must be sufficiently strong (or observed with sufficient duration) to provide adequate per-channel sensitivity for a useful calibration. In practice, several calibrators are usually observed, each with properties suitable for one or more of the required calibrations.

Synthesis calibration is inherently a bootstrapping process. First, the dominant calibration term is determined, and then, using this result, more subtle effects are solved for, until the full set of required calibration terms is available for application to the target field. The solutions for each successive term are relative to the previous terms. Occasionally, when the several calibration terms are not sufficiently orthogonal, it is useful to re-solve for earlier types using the results for later types, in effect, reducing the effect of the later terms on the solution for earlier ones, and thus better isolating them. This idea is a generalization of the traditional concept of self-calibration, where initial imaging of the target source supplies the visibility model for a re-solve of the gain calibration ($G$ or $T$). Iteration tends toward convergence to a statistically optimal image. In general, the quality of each calibration and of the source model are mutually dependent. In principle, as long as the solution for any calibration component (or the source model itself) is likely to improve substantially through the use of new information (provided by other improved solutions), it is worthwhile to continue this process.

In practice, these concepts motivate certain patterns of calibration for different types of observation, and the `calibrater` tool in CASA is designed to accommodate these patterns in a general and flexible manner. For a spectral line total intensity observation, the pattern is usually:

1. Solve for $G$ on the bandpass calibrator
2. Solve for $B$ on the bandpass calibrator, using $G$

3. Solve for $G$ on the primary gain (near-target) and flux density calibrators, using $B$ solutions just obtained
4. Scale $G$ solutions for the primary gain calibrator according to the flux density calibrator solutions
5. Apply $G$ and $B$ solutions to the target data
6. Image the calibrated target data

If opacity and gain curve information are relevant and available, these types are incorporated in each of the steps (in future, an actual solve for opacity from appropriate data may be folded into this process):

1. Solve for $G$ on the bandpass calibrator, using $T$ (opacity) and $E$ (gain curve) solutions already derived.
2. Solve for $B$ on the bandpass calibrator, using $G$, $T$ (opacity), and $E$ (gain curve) solutions.
3. Solve for $G$ on primary gain (near-target) and flux density calibrators, using $B$, $T$ (opacity), and $E$ (gain curve) solutions.
4. Scale $G$ solutions for the primary gain calibrator according to the flux density calibrator solutions
5. Apply $T$ (opacity), $E$ (gain curve), $G$, and $B$ solutions to the target data
6. Image the calibrated target data

For continuum polarimetry, the typical pattern is:

1. Solve for $G$ on the polarization calibrator, using (analytical) $P$ solutions.
2. Solve for $D$ on the polarization calibrator, using $P$ and $G$ solutions.
3. Solve for $G$ on primary gain and flux density calibrators, using $P$ and $D$ solutions.
4. Scale $G$ solutions for the primary gain calibrator according to the flux density calibrator solutions.
5. Apply $P$, $D$, and $G$ solutions to target data.
6. Image the calibrated target data.

For a spectro-polarimetry observation, these two examples would be folded together.

In all cases the calibrator model must be adequate at each solve step. At high dynamic range and/or high resolution, many calibrators which are nominally assumed to be point sources become slightly resolved. If this has biased the calibration solutions, the offending calibrator may be imaged at any point in the process and the resulting model used to improve the calibration. Finally, if sufficiently strong, the target may be self-calibrated as well.

## E.2 General Calibrater Mechanics

The calibrater tasks/tool are designed to solve and apply solutions for all of the solution types listed above (and more are in the works). This leads to a single basic sequence of execution for all solves, regardless of type:

1. Set the calibrator model visibilities
2. Select the visibility data which will be used to solve for a calibration type
3. Arrange to apply any already-known calibration types (the first time through, none may yet be available)
4. Arrange to solve for a specific calibration type, including specification of the solution timescale and other specifics
5. Execute the solve process
6. Repeat 1-4 for all required types, using each result, as it becomes available, in step 2, and perhaps repeating for some types to improve the solutions

By itself, this sequence doesn't guarantee success; the data provided for the solve must have sufficient SNR on the appropriate timescale, and must provide sufficient leverage for the solution (e.g., D solutions require data taken over a sufficient range of parallactic angle in order to separate the source polarization contribution from the instrumental polarization).

# Appendix F

# 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.

## F.1 NGC 5921 — VLA red-shifted HI emission

Note: This script does not include any self-calibration steps.

```
##########################################################################
#                                                                        #
# Use Case Script for NGC 5921                                           #
#                                                                        #
# Updated      STM 2007-11-08 (Beta Patch 0.5) add RRusk stuff           #
#                                                                        #
# Features Tested:                                                       #
#    The script illustrates end-to-end processing with CASA              #
#    as depicted in the following flow-chart.                            #
#                                                                        #
#    Filenames will have the <prefix> = 'ngc5921.usecase'                #
#                                                                        #
#    Input Data           Process          Output Data                   #
#                                                                        #
#   NGC5921.fits --> importuvfits  -->  <prefix>.ms   +                  #
#   (1.4GHz,               |            <prefix>.ms.flagversions         #
#    63 sp chan,           v                                            #
#    D-array)           listobs    -->  casapy.log                      #
#                          |                                             #
#                          v                                             #
#                     flagautocorr                                       #
#                          |                                             #
#                          v                                             #
#                        setjy                                           #
```

```
#                          |                                                #
#                          v                                                #
#                       bandpass   -->  <prefix>.bcal                       #
#                          |                                                #
#                          v                                                #
#                       gaincal    -->  <prefix>.gcal                       #
#                          |                                                #
#                          v                                                #
#                      fluxscale   -->  <prefix>.fluxscale                  #
#                          |                                                #
#                          v                                                #
#                      applycal    -->  <prefix>.ms                         #
#                          |                                                #
#                          v                                                #
#                        split     -->  <prefix>.cal.split.ms               #
#                          |                                                #
#                          v                                                #
#                        split     -->  <prefix>.src.split.ms               #
#                          |                                                #
#                          v                                                #
#                    exportuvfits  -->  <prefix>.split.uvfits               #
#                          |                                                #
#                          v                                                #
#                      uvcontsub   -->  <prefix>.ms.cont +                  #
#                          |            <prefix>.ms.contsub                 #
#                          v                                                #
#                        clean     -->  <prefix>.clean.image +              #
#                          |            <prefix>.clean.model +              #
#                          |            <prefix>.clean.residual             #
#                          v                                                #
#                     exportfits   -->  <prefix>.clean.fits                 #
#                          |                                                #
#                          v                                                #
#                       imhead     -->  casapy.log                          #
#                          |                                                #
#                          v                                                #
#                     immoments   -->  <prefix>.moments.integrated +       #
#                          |            <prefix>.moments.weighted_coord     #
#                          v                                                #
##########################################################################

import time
import os

#
# Set up some useful variables
#
# Get to path to the CASA home and stip 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 stablility 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 = ''
gainfield = ''
interp = ''

# 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')
#caltable = btable
#yaxis = 'amp'
#field = '0'
#iteration = 'antenna'
#subplot = 221
#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
gaintable = btable
gainfield = ''

# Use nearest (there is only one bandpass entry)
interp = 'nearest'

# 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'

# minimum SNR allowed
minsnr = 1.0

# reference antenna 15 (15=VLA:N2)
refant = '15'

gaincal()

# You can use plotcal to examine the gain solutions
#default('plotcal')
#caltable = gtable
#yaxis = 'amp'
#field = '0,1'
#iteration = 'antenna'
#subplot = 211
#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
gaintable = [ftable,btable]

# pick the 1445+099 out of the gain table for transfer
# use all of the bandpass table
gainfield = ['1','*']

# interpolation using linear for gain, nearest for bandpass
interp = ['linear','nearest']

# only one spw, do not need mapping
spwmap = []

# all channels
spw = ''
selectdata = False

# as before
gaincurve = False
opacity = 0.0

# select the fields for 1445+099 and N5921
field = '1,2'

applycal()

# Now for completeness apply 1331+305 to itself

field = '0'
gainfield = ['0','*']

# The CORRECTED_DATA column now contains the calibrated visibilities

applycal()

#=====================================================================
#
# Split the gain calibrater data, then the target
#
print '--Split 1445+099 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()

#
# Now split NGC5921 data (before continuum subtraction)
#
print '--Split NGC5921 Data--'

splitms = prefix + '.src.split.ms'
outputvis = splitms

# Pick off N5921
field = 'N5921*'

split()

#=============================================================================
#
# Export the NGC5921 data as UVFITS
# Start with the split file.
#
print '--Export UVFITS--'
default('exportuvfits')

srcuvfits = prefix + '.split.uvfits'

vis = splitms
fitsfile = srcuvfits

# Since this is a split dataset, the calibrated data is
# in the DATA column already.
datacolumn = 'data'

# Write as a multisource UVFITS (with SU table)
# even though it will have only one field in it
multisource = True

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importuvfits)
async = True

exportuvfits()

#=============================================================================
#
# 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'
# BETA 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.cont
# 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'
imagename = 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 = ''

# Standard gain factor 0.1
gain = 0.1

# Set the output image size and cell size (arcsec)
#imsize = [256,256]

# Do a simple Clark clean
alg = 'clark'

# If desired, you can do a Cotton-Schwab clean
# but will have only marginal improvement for this data
#alg = 'csclean'
# Twice as big for Cotton-Schwab (cleans inner quarter)
#imsize = [512,512]

# Pixel size 15 arcsec for this data (1/3 of 45" beam)
# VLA D-config L-band
cell = [15.,15.]

# 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')

#============================================================================
#
# Export the Final CLEAN Image as FITS
#
print '--Final Export CLEAN FITS--'
default('exportfits')

clnfits = prefix + '.clean.fits'

imagename = clnimage
fitsimage = clnfits

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importfits)
async = True

exportfits()

#============================================================================
```

```
#
# Get some image statistics
#
print '--Imhead--'
default('imhead')

imagename = clnimage

mode = 'stats'

cubestats = imhead()

# A summary of the cube will be seen in the logger
# cubestats will contain a dictionary of the statistics

#=============================================================================
#
# Get some image moments
#
print '--ImMoments--'
default('immoments')

imagename = clnimage

# Do first and second moments
moments = [0,1]

# Need to mask out noisy pixels, currently done
# using hard global limits
excludepix = [-100,0.009]

# Include all planes
planes = ''

# Output root name
momfile = prefix + '.moments'
outfile = momfile

immoments()

momzeroimage = momfile + '.integrated'
momoneimage = momfile + '.weighted_coord'

# It will have made the images:
# -------------------------------------
# ngc5921.usecase.moments.integrated
# ngc5921.usecase.moments.weighted_coord

#=============================================================================
#
# Get some statistics of the moment images
```

```
#
print '--Imhead--'
default('imhead')

mode = 'stats'
imagename = momzeroimage

momzerostats = imhead()

imagename = momoneimage

momonestats = imhead()

#============================================================================
#
# Can do some image statistics if you wish
# Treat this like a regression script
# WARNING: currently requires toolkit
#
print ' NGC5921 results '
print ' =============== '

#
# Use the ms tool to get max of the MSs
# Eventually should be available from a task
#
# 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

#
# Now use the stats produced by imhead above
#
```

```
print ''
# Pull the max from the cubestats dictionary
# created above using imhead
thistest_immax=cubestats['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 ''
# Pull the rms from the cubestats dictionary
thistest_imrms=cubestats['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

print ''
# Pull the max from the momzerostats dictionary
thistest_momzeromax=momzerostats['max'][0]
oldtest_momzeromax = 1.40223777294
print ' Moment 0 image max should be ',oldtest_momzeromax
print ' Found : Moment 0 Max = ',thistest_momzeromax
diff_momzeromax = abs((oldtest_momzeromax-thistest_momzeromax)/oldtest_momzeromax)
print ' Difference (fractional) = ',diff_momzeromax

print ''
# Pull the mean from the momonestats dictionary
thistest_momoneavg=momonestats['mean'][0]
oldtest_momoneavg = 1479.77119646
print ' Moment 1 image mean should be ',oldtest_momoneavg
print ' Found : Moment 1 Mean = ',thistest_momoneavg
diff_momoneavg = abs((oldtest_momoneavg-thistest_momoneavg)/oldtest_momoneavg)
print ' Difference (fractional) = ',diff_momoneavg

print ''
print '--- Done ---'

#
#============================================================================
```

### F.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_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

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>
```

## F.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 2007-10-10 (Beta)                                      #
#                                                                         #
###########################################################################

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'

clnimsize = [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 stip 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/sandrock2/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 calibrater 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 = 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=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()

#=============================================================================
#
# Export the Jupiter data as UVFITS
# Start with the split file.
#
print '--Export UVFITS--'
default('exportuvfits')

srcuvfits = prefix + '.split.uvfits'

vis = srcsplitms
fitsfile = srcuvfits

# Since this is a split dataset, the calibrated data is
# in the DATA column already.
datacolumn = 'data'

# Write as a multisource UVFITS (with SU table)
# even though it will have only one field in it
multisource = True

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importuvfits)
async = True
```

```
exportuvfits()

#
#=============================================================================
# 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'
imagename = 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'
caltable = selfcaltab1
```

```
# Don't need a-priori cals
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

imname2 = prefix + '.clean2'
imagename = 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.
#
# Pause script if you are running in scriptmode
if scriptmode:
    user_check=raw_input('Return to continue script\n')

# 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'
imagename = 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

#====================================================================
#
# Export the Final CLEAN Image as FITS
#
print '--Final Export CLEAN FITS--'
default('exportfits')
```

```
clnfits = prefix + '.clean.fits'

imagename = clnimage
fitsimage = clnfits

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importfits)
async = True

exportfits()

#============================================================================
#
# Export the Final Self-Calibrated Jupiter data as UVFITS
#
print '--Final Export UVFITS--'
default('exportuvfits')

caluvfits = prefix + '.selfcal.uvfits'

vis = srcsplitms
fitsfile = caluvfits

# The self-calibrated data is in the CORRECTED_DATA column
datacolumn = 'corrected'

# Write as a multisource UVFITS (with SU table)
# even though it will have only one field in it
multisource = True

# Run asynchronously so as not to interfere with other tasks
# (BETA: also avoids crash on next importuvfits)
async = True

exportuvfits()

#
#============================================================================
# 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 G

# CASA Dictionaries

BETA ALERT: These tend to become out of date as we add new tasks or change names.

## G.1 AIPS – CASA dictionary

Please see:

- `https://wikio.nrao.edu/bin/view/Software/CASA-AIPSDictionary`

BETA ALERT: This link is out-of-date and refers mostly to the Toolkit. We will update this with a task dictionary.

## G.2 MIRIAD – CASA dictionary

Table G.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.

## G.3 CLIC – CASA dictionary

Table G.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 G.1: MIRIAD – CASA dictionary

| **MIRIAD Task** | **Description** | **CASA task/tool** |
|---|---|---|
| atlod | load ATCA data | atcafiller tool |
| blflag | Interactive baseline based editor/flagger | mp raster displays |
| cgcurs | Interactive image analysis | viewer |
| cgdisp | Image display, overlays | viewer |
| clean | Clean an image | clean |
| fits | FITS image filler | importfits |
| gpboot | Set flux density scale | fluxscale |
| gpcal | Polarization leakage and gain calibration | cb with 'G' and 'D' |
| gpcopy | copy calibration tables | *not needed* |
| gpplt | Plot calibration solutions | plotcal |
| imcomb | Image combination | im tool |
| 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 | invert, im tool |
| linmos | linear mosaic combination of images | mosaic |
| maths | Calculations involving images | ia.imagecalc, ia.calc |
| mfcal | Bandpass and gain calibration | bandpass |
| prthd | Print header of image or uvdata | imhead, listobs |
| restor | Restore a clean component model | im tool |
| selfcal | selfcalibration of visibility data | clean, gaincal, etc. |
| tvclip | automated flagging based on clip levels | flagdata |
| tvdisp | Load image to TV display | viewer |
| tvflag | Interactive TB data editing | viewer |
| uvaver | Average/select data, apply calibration | applycal, split |
| uvfit | uv-plane component fitter | uvmodelfit |
| uvflag | Command-based flagging | flagdata |
| uvgen | Simulator | sm tool |
| uvlist | List uv-data | listvis (TBD) |
| uvmodel | Source model computation | ft |
| uvplt | uv-data plotting | plotxy |
| uvsplit | split uv file in sources and spectral windows | split |

Table G.2: CLIC–CASA dictionary

| **CLIC Function** | **Description** | **CASA task/tool** |
|---|---|---|
| load | Load data | almatifiller tool |
| print | Print text summary of data | listobs |
| flag | Flag data | plotxy, flagdata, viewer |
| phcor | Atmospheric phase correction | almatifiller |
| rf | Radio frequency bandpass | bandpass |
| phase | Phase calibration | gaincal |
| flux | Absolute flux calibration | setjy, fluxscale |
| ampl | Amplitude calibration | gaincal |
| table | Split out calibrated data (uv table) | split |