CASA Roadmap

Renkun Kuang

November 13, 2019

Contents

CASA, the Common Astronomy Software Applications, is the primary data processing software for the Atacama Large Millimeter/submillimeter Array (ALMA) and Karl G. Jansky Very Large Array (VLA), and is often used also for other radio telescopes.

1 Related links

List of all tutorials
CASA 5.6 documentation
CASA guides
Karl G. Jansky VLA Tutorials
VLBI in CASA: a tutorial with EVN data
CASA VLBI tutorial at the EVN symposium 2018
2nd-school-on-multiwavelength-astronomy, and all the materials in this school!

2 Start Playing CASA Directly

2.1 importfits

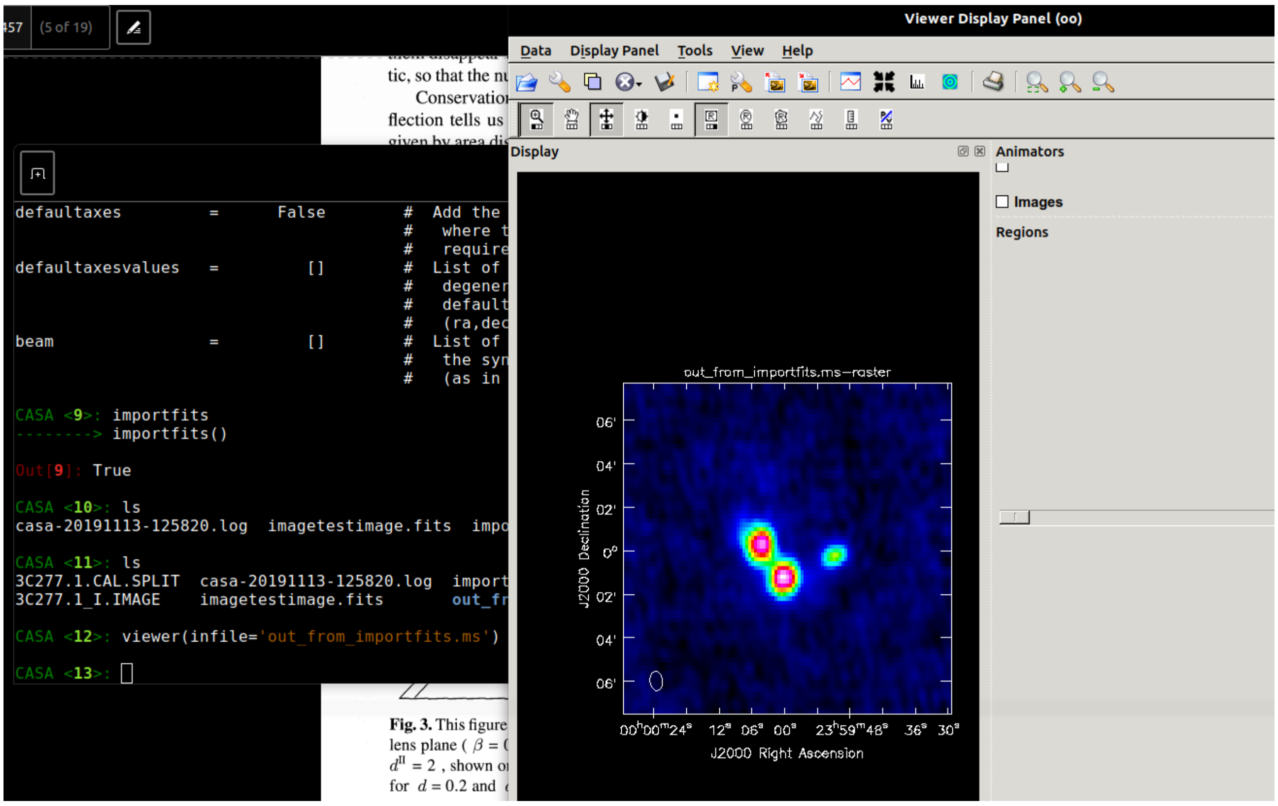
Tip: first look at the input parameters of **importfits** using:
inp(improtfits)
then we know there are two parameters:
fitsimage and imagename should be assigned (otherwise importfits function will not be executed correctly.)

after that we can safely type:
importfits

and the result is like this:

```
CASA < 9 >: importfits
———> importfits()
Out[9]: True
```

Using *ls* you can see there shows up a new directory in current path.
Now try viewing it using:
viewer(infile='out_from_importfits.ms')
which should give we something like:



tic, so that the nu
Conservatio
flection tells us
given by area di

```
defaultaxes          =      False        #  Add the
                                         #   where t
                                         #   require
defaultaxesvalues    =         []        #  List of
                                         #   degener
                                         #   default
                                         #   (ra,dec
beam                 =         []        #  List of
                                         #   the syn
                                         #   (as in

CASA <9>: importfits
--------> importfits()

Out[9]: True

CASA <10>: ls
casa-20191113-125820.log  imagetestimage.fits  impo

CASA <11>: ls
3C277.1.CAL.SPLIT  casa-20191113-125820.log  import
3C277.1_I.IMAGE    imagetestimage.fits       out_fr

CASA <12>: viewer(infile='out_from_importfits.ms')

CASA <13>:
```

Fig. 3. This figure
lens plane (β = (
d^{ll} = 2 , shown o
for d = 0.2 and

3 CASA Fundamentals

Fundamentals of CASA: Measurement Equation, Science Data Model, and MeasurementSet

3.1 Measurement Equation - The synthesis calibration model

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.

3.1.1 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]

$$\vec{V}_{ij} = J_{ij}\vec{V}_{ij}^{IDEAL}$$

where $\vec{V}_{ij}$ represents the observed visibility, a complex number representing the amplitude and phase of the correlated data from a pair of antennas in each sample time, per spectral channel. $\vec{V}_{ij}^{IDEAL}$ 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 $J_{ij} = J_i \otimes J_j^*$. 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:

$$\vec{V}_{ij} = M_{ij}B_{ij}G_{ij}D_{ij}E_{ij}P_{ij}T_{ij}\vec{V}_{ij}^{IDEAL}$$

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.

[1] Hamaker, Bregman, & Sault (1996),Understanding radio polarimetry. I. Mathematical foundations and Sault, Hamaker, Bregman (1996),Understanding radio polarimetry. II. Instrumental calibration of an interferometer array.

- 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., 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 JVLA 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 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.

3.1.2 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 3, 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).

3.2 Science Data Model - Introduction to the Science Data Model

It was decided realtively early in the preparatory phase of ALMA and EVLA that the two projects would

1. use the same data analysis software (CASA) and
2. use essentially the same archive data format, the (ALMA) Science Data Model, (A)SDM.

The SDM was developed to a first prototype by Francois Viallefond (Observatoire de Paris) as an extension and full generalisation of the MeasurementSet. The SDM is superior to the MS w.r.t. the storage of observatory raw data in that it is capable of capturing the metadata of an interferometic or total-power dataset completely without any compromise including all data relevant for calibration and observatory administration.

Just like for the MS, one can think of the SDM as a relational database. And both databases have in principle a very similar layout. However, while the MS has only 12 required Subtables, the SDM uses typically 40 Subtables, and there are more optional ones.

SDMs, however, are for data storage and data reduction should be done on the MeasurementSet (although when importing data through importasdm with option lazy=True the ASDM is restructured to resemble an MS).

For the implementation of the SDM, (then) novel source code generation techniques were applied which permitted simultaneous implementation in Java and C++. As the actual representation of the data on disk, a hybrid format was chosen: all low-volume metadata is stored as XML files (one per table) while the bulk data is stored in a binary format (MIME) in so-called Binary Large Objects (BLOBs). In particular the Main table is stored as a series of BLOBs of a few GB each with lossless compression. This makes the SDM more efficient as a bulk data format than the MS which stores the the DATA column of the Main table as one single monolithic file.

An up to date description of the tables of the SDM is given in this pdf.
This tar file contains the XML Schema Definition (xds) files for all of the tables described in the associated SDM Short Table Description.
This tar file contains the XML Schema Definition (xds) files for all of the enumerations used by the SDM tables.
The binary data format is given in this pdf.

3.3 Basics MeasurementSet - Description of the CASA UV Data Format

Data is handled in CASA via the table system. In particular, visibility data are stored in a CASA table known as a MeasurementSet (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 (as an auto-correlation-only data set), see

"Single-dish data calibration and reduction".
A full description of the MeasurementSet can be found here, and a description of the MS model column can be found in the Synthesis Calibration section.

Once in MeasurementSet 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, which allows you to specify the subset of the data on which the tasks and tools will operate.

3.4 MeasurementSet v2 - Definition of the MeasurementSet v2

The MeasurementSet version 2 [2] , is a database designed to hold radioastronomical data to be calibrated following the MeasurementEquation approach by Hamaker, Bregman, and Sault (1996).

Since its publication, the MeasurementSet (MS) design has been implemented by several software development groups, among them the CASA team and, e.g., the European VLBI Network team. CASA has also adopted the MeasurementEquation as its fundamental calibration scheme and has thus embraced the MS as its native way to store radio observations. With CASA becoming the designated analysis package for ALMA and the VLA, this means that the MS is now the default way of storing ALMA and VLA data during the actual analysis.

The ALMA and VLA raw data format, however, is not the MS but the so-called ALMA Science Data Model (ASDM) for ALMA, and the Science Data Model (SDM) for the VLA. Both of which are closely related and are discussed in a separate section. The ALMA and VLA archives hence do not store data in MS format but in (A)SDM format, and when a CASA user starts to work with this data, the first step has to be the import of the (A)SDM into the CASA MS format.

The MS is effectively a relational database which on the one hand tries to permit the storage of all imaginable radio (interferometric, single-dish) data with corresponding metadata, and on the other hand ventures to be storage-space and data-maintenance efficient by avoiding data redundancy.

3.5 Definition Synthesized Beam - Definition of the Gaussian function that CASA uses for the synthesized beam

CASA uses the following zero-centered two dimensional elliptical Gaussian function or Gaussian beam:

$$f(x,y) = Aexp\Big\{-\Big(\frac{4ln(2)}{d_1^2}(\cos(\theta)x+\sin(\theta)y)^2+\frac{4ln(2)}{d_2^2}(-\sin(\theta)x+\cos(\theta)y)^2\Big)\Big\}$$

where A is the amplitude (usually set to unity) and θ is the anti-clockwise angle from the x axis to the line that lies along the greatest width of $f(x,y)$ (the line and the x axis must be

[2] MeasurementSet definition v2

coplanar). The factors d_1 and d_2 are respectively the semi-major and semi-minor axis of the ellipse, which is formed by the cross-section that lies parallel to the x, y plane, at a height so that d_1 is equal to the FWHM (full width at half maximum) distance of the one dimensional Gaussian which lies on the plane formed by the z axis and d_1. Note that $d_1 \geqslant d_2 > 0$, since d_1 is the semi-major axis.

For calculating the Fourier transform of the two dimensional elliptical Gaussian, the above Equation can be re-written by grouping the x and y terms: ref

4 Using CASA - Description of how CASA interacts with the python environment

4.1 Python Basics - A few Python details every CASA user should know

More details concerning the use of Python in CASA

4.2 Executing Python scripts

execfile('myscript.py')
or 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 "CASA Tasks").
A script can also be executed from the command line without starting CASA first. To do so, use the -c CASA startup option:
casa -c myscript.py

4.3 CASA Data - The CASA data format and how to work with it

4.3.1 Visibility Data

4.3.2 How do I get rid of my data in CASA?

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.

In scripts, the ! escape to the OS will not work. Instead, use the os.system() function (see page "Python and CASA") to do the same thing:
os.system('rm -r AM675*')

If you are within CASA, then the CASA system is keeping a cache of tables that you have been using and using the OS to delete them will confuse things. For example, running a script that contains rm commands multiple times will often not run or crash the second time as the cache gets confused. The clean way of removing CASA tables (MS, caltables, images) inside CASA is to use the rmtables task:
rmtables('AM675.ms')
and this can also be wildcarded (though you may get warnings if it tries to delete files or directories that fit the name wildcard that are not CASA tables).

4.3.3 What's in my data?

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

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 and an optional **MODEL_DATA** column, which may hold the Fourier inversion of a particular model image. 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.

4.4 Starting CASA - Important information about starting CASA

4.4.1 Before Starting CASA

First, you will most likely be starting CASA running from a working directory that has your data in it, or at least where you want your output to go. It is easiest to start from there rather than changing directories inside CASA.

4.4.2 Environment Variables

4.4.3 Where is CASA?

4.4.4 Starting CASA

For example, you can execute a CASA script script.py directly with the command
casa -c script.py

You can also launch the plotms, viewer, casafeather, and browsetable GUIs separately in a terminal without starting CASA itself. To do so, type:
casaplotms
casaviewer
casafeather
casabrowser
respectively. (Not working on my ubuntu.)

4.4.5 Ending CASA

quit

4.4.6 What happens if something goes wrong?

ALERT: Please check the CASA Home Page for Release Notes and FAQ information including a list of known problems. If you think you have encountered an unknown problem, please consult the CASA HelpDesk help.nrao.edu for general CASA questions or help.almascience.org for ALMA related questions.
First, always check that your inputs are correct; use
help <taskname>
to review the inputs/output. Check the error message carefully to see if there are hints on what went wrong. Typical mistakes are missing quotes around strings, wrong format of the inputs (e.g. a list where a string is expected), letter "O" instead of a zero, wrong file names, or python indentation.

4.4.7 Aborting CASA execution

If something has gone wrong and you want to stop what is executing, then typing CTRL-C (Control and C keys simultaneously) will usually cleanly abort the application. If this does not work on your system then try CTRL-Z to put the task or shell in the background, and then follow up with a kill -9 ¡PID¿ where you have found the relevant casa process ID (PID) using ps aux— grep casa.

4.4.8 What happens if CASA crashes?

Usually, restarting CASA 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.

4.5 Getting Help in CASA - How to get help while running CASA

4.5.1 The doc() command

The new CASA documentation system introduced with CASA 5.0 is packaged with CASA and is accessible from the CASA command line by means of a new command: doc(). Typing doc() will open the packaged documentation in your browser. The doc() command can also invoke documentation on select CASA tasks when given the task name as a string parameter. For example:
doc('applycal')
doc('plotms')
doc('tclean')
The doc command will only invoke documentation for tasks that have been included in the new CASA documentation system. To see a list of included tasks, open the Global Task List chapter. For help with tasks not yet included, see below.

4.5.2 Task and Tool Listing Commands

There are several commands that can be used at the CASA command line to list available tasks and tools. taskhelp will print a list of every CASA task along with a one-line description of each. tasklist will print every CASA task categorized by task purpose (e.g. imaging, calibration, visualization). Both taskhelp and tasklist are described on the CASA Tasks page. The command toolhelp will print all CASA tools with a one-line description and is documented on the CASA Tools page.

4.5.3 Terminal-based Task and Tool Help

4.5.4 Task help

Task help can be obtained by typing one of the following
pdoc TASK
help TASK
help 'TASK'
TASK?
where TASK is replaced with the name of a CASA task. The pdoc command currently gives the cleanest documentation format with the smallest amount of object-oriented (programmer) output.

4.5.5 Tool help

You can also get the short help for a CASA tool method by typing 'help tool.method'.

4.5.6 The PAGER Environment Variable

4.5.7 The TAB key

4.5.8 Python help

4.6 CASA Logger - Detailed description of the CASA logger

4.7 CASA Tasks & Tools - Summary of CASA tasks and tools

Originally, CASA consisted of a collection of tools, combined in the so-called toolkit. The current toolkit still contains the full functionality of CASA, though most data reduction steps bundle a number of tools together to create the more familiar CASA tasks. That is to say, a task makes calls to a number of tools.
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.

4.7.1 CASA Tasks - Summary of CASA tasks

What Tasks are Available?

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.
The parameters used and their defaults can be obtained by typing help <taskname> or <taskname>? at the CASA prompt.
The tasks with name in parentheses () are experimental, those in curly brackets {} are deprecated and will be removed in future releases. The functionality of deprecated tasks is usually available in some other task (e.g., instead of mosaic one should use clean).

Running Tasks and Tools

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

The go() Command

You can execute a task using the go() command

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 $< 5 >$: taskname = 'clean'
CASA $< 6 >$: inp()
———-> inp()

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')

The saveinputs Command

The saveinputs command will save the current values of a given task parameters to a Python (plain ascii) file. It can take up to two arguments, e.g.
saveinputs(taskname, outfile)

To read these back in, use the Python execfile command. For example,
CASA $< 5 >$: execfile('listobs.saved') and we are back.
An example save to a custom named file:
saveinputs('listobs','ngc5921_listobs.par')
You can also use the CASA tget command (see next section) instead of the Python execfile to restore your inputs.

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

The tput Command

The tput command will save the current parameter values of a task to its $<$ *taskname* $>$.last file. This is a shorthand to saveinputs and is a counterpart to tget.

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.

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.

4.8 CASA Tools - Summary of CASA tools

Tools in CASA

The CASA toolkit is the foundation of the functionality in the package, and consists of a suite of functions that are callable from Python. The tools are used by the tasks, and can be used by advanced users to perform operations that are not available through the tasks.
It is beyond the scope of this reference to describe the toolkit in detail. Occasionally, examples will be given that utilize the tools. In short, tools are always called as functions, with any parameters that are not to be defaulted given as arguments. For example:
ia.open('ngc5921.chan21.clean.cleanbox.mask')
ia.calcmask('"ngc5921.chan21.clean.cleanbox.mask">0.5','mymask')
ia.summary()
ia.close()
uses the image tool (ia) to turn a clean mask image into an image mask. Tools never use the CASA global parameters.
To find what tools are available, use the toolhelp command:
CASA $< 16 >$: toolhelp

4.8.1 Writing Tasks in CASA - How to create your own CASA tasks

The Basics

It is possible to write your own task and have it appear in CASA. For example, if you want to create a task named yourtask, then must create two files, yourtask.xml and a task_yourtask.py. The .xml file is use to describe the interface to the task and the task_yourtask.py does the actual work. The argument names must be the same in both the yourtask.xml and task_yourtask.py file. The yourtask.xml file is used to generate all the interface files so yourtask will appear in the CASA system. It is easiest to start from one of the existing tasks when constructing these. You would make the name of the function in the yourtask.py be yourtask in this example.

We have provided the buildmytasks command in order to assemble your Python and XML into a loadable Python file. Thus, the steps you need to execute (again for an example task named "yourtask"):

1. Create python code for task as task_yourtask.py
2. Create xml for task as yourtask.xml
3. Execute buildmytasks from the CASA prompt: !buildmytasks
4. Initialize your new task inside CASA: execfile 'mytasks.py'

After this, you should see the help and inputs inside CASA, e.g. inp yourtask should work. Note that for the final step you invoke the file called mytasks.py, regardless of what you named the actual task. You now have a shiny new task yourtask that you can run and use in the same way as all other CASA tasks.
Note that if multiple custom tasks are stored in the same directory, they will all be built by !buildmytasks and will all be initialized by executing mytasks.py. To build and initialize only a single task, instead use !buildmytasks taskname; you are then free to rename mytasks.py (e.g. load_taskname.py) and repeat this procedure for your other tasks. Our recommendation, for those of you who are managing multiple custom tasks, is to have each task live in its own directory. The mytasks.py file need not be in the current working directory to initialize your task, since you can provide the full path upon initialization
#Example
execfile('/full_path_to_my_task/mytasks.py')

The XML file

The task yourtask.py file

Example: The clean task

4.9 Loading Data to Images - Steps involved in the CASA workflow

4.10 Crash Reporter - Understanding the crash reporter and how to configure and test it

4.11 Telemetry - Understanding telemetry data collection and how to disable it

Casa sends telemetry data to NRAO by default. This data is anonymous and is used to measure Casa task usage.
You can disable telemetry by adding the following line in /.casarc
EnableTelemetry: False

4.12 Amazon Web Services - Overview of how to use CASA on Amazon Web Services

4.13 CASA Reference - Citation for use in publications

5 Calibration & Visibilities

5.1 Visibilities Import Export - Description of how to import and export Visibility Data to CASA

5.1.1 Overview - Overview of Visibility Data Import Export chapter

5.1.2 UV Data Import - Converting Telescope UV Data to a MeasurementSet

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

- ALMA and VLA Science Data Model format via **importasdm** and **importevla**
- historic VLA Archive format data via **importvla**
- ATCA Data via **importatca**
- MIRIAD Data from various telescopes via **importmiriad**
- GMRT Data via **importgmrt**
- UVFITS format can be imported into and exported from CASA (**importuvfits**, **importfitsidi**, and **exportuvfits**)

ALMA and VLA Filling of Science Data Model (ASDM) data

Specifics on importing Janksy VLA data with importasdm

Import of ASDM data with option lazy=True

VLA: Filling data from archive format (importvla)

Import ATCA and CARMA data

Import MIRIAD visibilities (importmiriad)

Import ATCA RPFITS data (importatca)

UVFITS Import

5.1.3 MeasurementSet Export - Convert a MeasurementSet to UVFITS

5.2 Visibility Data Selection - How to select visibility data