

ATNF Spectral Analysis Package

User Guide v2.0beta

Chris Phillips

April 6, 2006

1 Introduction

ASAP is a single dish spectral line processing package currently being developed by the ATNF. It is intended to process data from all ATNF antennas, and can probably be used for other antennas if they can produce “Single Dish FITS” format. It is based on the AIPS++ package.

This userguide is currently being updated for the ASAP 2.0 release which has a number of significant changes which affect the user interface. Please report any mistakes you find in this userguide.

2 Installation and Running

Currently there are installations running on Linux machines at

- Epping - use hosts **draco** or **hydra**
- Narrabri - use host **kaputar**
- Parkes - use host **bourbon**
- Mopra - use host **minos**

To start asap log onto one of these Linux hosts and enter

```
> cd /my/data/directory
> asap
```

This starts the ASAP. To quit, you need to type `^d` (control-d) or type `%Exit`.

3 Interface

ASAP is written in C++ and python. The user interface uses the “ipython” interactive shell, which is a simple interactive interface to python. The user does not need to understand python to use this, but certain aspects python affect what the user can do. The current interface is object oriented. In the future, we will build a functional (non object oriented) shell on top of this to ease interactive use.

3.1 Integer Indices are 0-relative

Please note, all integer indices in ASAP and iPython are **0-relative**.

3.2 Objects

The ASAP interface is based around a number of “objects” which the user deals with. Objects range from the data which have been read from disk, to tools used for fitting functions to the data. The following main objects are used :

- scantable** The data container (actual spectra and header information)
- selector** Allows the user to select a subsection of the data, such as a specified or range of beam numbers, IFs, etc.
- plotter** A tool used to plot the spectral line data
- fitter** A tool used to fit functions to the spectral data
- reader** A tool which can be used to read data from disks into a scantable object (advanced use).

There can be many objects of the same type. Each object is referred to by a variable name made by the user. The name of this variable is not important and can be set to whatever the user prefers (i.e. “s” and “ParkesHOH-20052002” are equivalent). However, having a simple and consistent naming convention will help you a lot.

3.3 Member Functions (functions)

Following the object oriented approach, objects have associated “member functions” which can either be used to modify the data in some way or change global properties of the object. In this document member functions will be referred to simply as functions. From the command line, the user can execute these functions using the syntax:

```
ASAP> out = object.function(arguments)
```

Where **out** is the name of the returned variable (could be a new scantable object, or a vector of data, or a status return), **object** is the object variable name (set by the user), **function** is the name of the member function and **arguments** is a list of arguments to the

function. The arguments can be provided either through position or `name=`. A mix of the two can be used. E.g.

```
ASAP> av = scans.average_time(msk,weight='tsys')
ASAP> av = scans.average_time(mask=msk,weight='tsys')
ASAP> av = scans.average_time(msk,tsys)
ASAP> scans.poly_baseline(mask=msk, order=0, insitu=True)
ASAP> scans.poly_baseline(msk,0,True)
ASAP> scans.poly_baseline(mask, insitu=True)
```

3.4 Global Functions

It does not make sense to implement some functions as member functions, typically functions which operate on more than one scantable (e.g. time averaging of many scans). These functions will always be referred to as global functions.

3.5 Interactive environment

ipython has a number of useful interactive features and a few things to be aware of for the new user.

3.5.1 String completion

Tab completion is enabled for all function names. If you type the first few letters of a function name, then type `<TAB>` the function name will be auto completed if it is unambiguous, or a list of possibilities will be given. Auto-completion works for the user object names as well as function names. It does not work for filenames, nor for function arguments.

Example

```
ASAP> scans = scantable('MyData.rpf')
ASAP> scans.se<TAB>
ASAP> scans.set_in<TAB>
scans.set_cursor      scans.set_freqframe  scans.set_selection
scans.set_doppler     scans.set_instrument scans.set_unit
scans.set_fluxunit    scans.set_restfreqs

ASAP> scans.set_instrument()
```

3.5.2 Leading Spaces

Python uses leading space to mark blocks of code. This means that if you start a command line with a space, the command generally will fail with a syntax error.

3.5.3 Variable Names

During normal data processing, the user will have to create named variables to hold spectra etc. These must conform to the normal python syntax, specifically they cannot contain “special” characters such as \$ etc and cannot start with a number (but can contain numbers). Variable (and function) names are case sensitive.

3.5.4 Unix Interaction

Basic unix shell commands (`pwd`, `ls`, `cd` etc) can be issued from within ASAP. This allows the user to do things like look at files in the current directory. The shell command “`cd`” works within ASAP, allowing the user to change between data directories. Unix programs cannot be run this way, but the shell escape “`!`” can be used to run arbitrary programs. E.g.

```
ASAP> pwd
ASAP> ls
ASAP> cd /my/data/directory
ASAP> ! mozilla&
```

3.6 Help

ASAP has built in help for all functions. To get a list of functions type:

```
ASAP> commands()
```

To get help on specific functions, the built in help needs to be given the object and function name. E.g.

```
ASAP> help scantable.get_scan # or help(scantable.get_scan)
ASAP> help scantable.stats
ASAP> help plotter.plot
ASAP> help fitter.plot

ASAP> scans = scantable('mydata.asap')
ASAP> help scans.get_scan # Same as above
```

Global functions just need their name

```
ASAP> help average_time
```

Note that if you just type `help` the internal ipython help is invoked, which is probably *not* what you want. Type `~d` (control-d) to escape from this.

3.7 Customisation - .asaprc

ASAP use an `.asaprc` file to control the user's preference of default values for various functions arguments. This includes the defaults for arguments such as `insitu`, `scantable` `freqframe` and the plotters `set_mode` values. The help on individual functions says which arguments can be set default values from the `.asaprc` file. To get a sample contents for the `.asaprc` file use then command `list_rcparameters`.

Common values include:

```
# apply operations on the input scantable or return new one
insitu                                : False

# default output format when saving scantable
scantable.save                        : ASAP

# default frequency frame to set when function
# scantable.set_freqframe is called
scantable.freqframe                  : LSRK

# auto averaging on read
scantable.autoaverage                : True
```

For a complete list of `.asaprc` values, see the Appendix.

4 Scantables

4.1 Description

4.1.1 Basic Structure

ASAP data handling works on objects called scantables. A scantable holds your data, and also provides functions to operate upon it.

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 polarisation. For example Parkes multibeam data would contain many beams, one IF and 2-4 polarisations, while the new Mopra 8-GHz filterbank will eventually produce one beam, many IFs, and 2-4 polarisations. All of the combinations of Beams/IFs and Polarisations are contained in separate rows. These rows are grouped in cycles (same time stamp).

A collection of cycles for one source is termed a scan (and each scan has a unique numeric identifier, the SCANNO). A scantable is then a collection of one or more scans. If you have scan-averaged your data in time, i.e. you have averaged all cycles within a scan, then each scan would hold just one (averaged) integration.

Many of the functions which work on scantables can either return a new scantable with modified data or change the scantable `insitu`. Which method is used depends on the users preference. The default can be changed via the `.asaprc` resource file.

For example a Mopra scan with a 4s intergration time, two IFs and dual polarisations has two (2s) cycles.

SCANNO	CYCLEN0	BEAMNO	IFNO	POLNO
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	1	1
0	1	0	0	0
0	1	0	0	1
0	1	0	1	0
0	1	0	1	1

4.1.2 Contents

A scantable has header information and data (a scantable is actually an AIPS++ Table and it is generally stored in memory when you are manipulating it with ASAP. You can save it to disk and then browse it with the AIPS++ Table browser if you know how to do that !).

The data are stored in columns (the length of a column is the number of rows/spectra of course).

Two important columns are those that describe the frequency setup. We mention them explicitly here because you need to be able to understand the presentation of the frequency information and possibly how to manipulate it.

These columns are called `FREQ_ID` and `MOLECULE_ID`. They contain indices, for each IF, pointing into tables with all of the frequency and rest-frequency information for that integration. More on these below when we discuss the `summary` function in the next subsection.

There are of course many other columns which contain the actual spectra, the flags, the Tsys, the source names and so on.

There is also a function `summary` to list a summary of the scantable. You will find this very useful.

Example:

```
ASAP> scans = scantable('MyData.rpf')
ASAP> scans.summary()           # Brief listing

# Equivalent to brief summary function call
ASAP> print scan
```

The summary function gives you a scan-based summary, presenting the scantable as a cascading view of Beams and IFs.

4.2 Data Selection

ASAP contains flexible data selection. Data can be selected based on IF, beam, polarisation, scan number as well as values such as Tsys. Advanced users can also make use of the AIPS++ TAQL language to create selections based on almost any of the values recorded.

Selection is based on a **selector** object. This object is created and various selection functions applied to it (**set_ifs**, **set_beams** etc). The selection object then must be applied to a scantable using the **set_selection** function. A single selection object can be created and setup then applied to multiple scantables.

Once a selection has been applied, all following functions will only “see” the selected spectra (including functions such as **summary**). The selection can then be reset and all spectra are visible. Note that if functions such as **copy** are run on a scantable with active selection, only the selected spectra are copied.

The common selection functions are:

set_beams Select beams by index number

set_ifs Select ifs by index number

set_name Select by source name. Can contain “*” as a wildcard, e.g. “Orion*_R”.

set_ifs Select ifs by index number

set_polarisation Select but polarisation index or name. If polarisation names are given, the data will be on-the-fly converted (for example from linears to Stokes).

set_query Set query directly. For power users only!

set_tsys Select data based on Tsys. Also example of user definable query.

reset Reset the selection to include all spectra.

Note that all indices are zero based.

Examples:

```
ASAP> selection = selector()           # Create selection object
ASAP> selection.set_ifs(0)              # Just select the first IF
ASAP> scans.set_selection(selection)    # Apply the selection
ASAP> print scans                      # Will just show the first IF

ASAP> selection.set_ifs([0,1])          # Select the first two IFs
ASAP> selection.set_beams([1,3,5])      # Also select three of the beams
ASAP> scans.set_selection(selection)    # Apply the selection

ASAP> selection.set_name('G308*')      # Select by source name

ASAP> selection.reset()                 # Turn off selection
ASAP> scans.set_selection(selection)    # Apply the reset selection
```

4.3 State

Each scantable contains "state"; these are properties applying to all of the data in the scantable.

Examples are the selection of beam, IF and polarisation, spectral unit (e.g. km/s), frequency reference frame (e.g. BARY) and velocity Doppler type (e.g. RADIO).

4.3.1 Units, Doppler and Frequency Reference Frame

The information describing the frequency setup for each integration is stored fundamentally in frequency in the reference frame of observation (E.g. TOPO).

When required, this is converted to the desired reference frame (e.g. LSRK), Doppler (e.g. OPTICAL) and unit (e.g. km/s) on-the-fly. This is important, for example, when you are displaying the data or fitting to it. The reference frame is set on file read to the value set in the user `.asaprc` file.

For units, the user has the choice of frequency, velocity or channel. The `set_unit` function is used to set the current unit for a scantable. All functions will (where relevant) work with the selected unit until this changes. This is mainly important for fitting (the fits can be computed in any of these units), plotting and mask creation.

The velocity definition can be changed with the `set_doppler` function, and the frequency reference frame can be changed with the `set_freqframe` function.

Example usage:

```
ASAP> scans = scantable('2004-11-23_1841-P484.rpf') # Read in the data
ASAP> scans.set_freqframe('LSRK') # Use the LSR velocity frame
ASAP> scans.set_unit('km/s') # Use velocity for plots etc from now on
ASAP> scans.set_doppler('OPTICAL') # Use the optical velocity convention
ASAP> scans.set_unit('MHz') # Use frequency in MHz from now on
```

4.3.2 Rest Frequency

ASAP reads the line rest frequency from the RPFITS file when reading the data. The values stored in the RPFITS file are not always correct and so there is a function `set_restfreq` to set the rest frequencies for the currently selected data.

For each integration, there is a rest-frequency per IF (the rest frequencies are just stored as a list with an index into them). There are a few ways to set the rest frequencies with this function.

If you specify just one rest frequency, then it is set for all IF.

```
# Set all IFs
ASAP> scans.set_restfreqs(freqs=1.667359e9)
```

If set a rest frequency for each IF, specify a list of frequencies (of length the number of IFs).

Regardless of the source, the rest frequency will be set for each IF to the corresponding value in the provided list.

```
# Set rest frequency for all IFs
ASAP> scans.set_restfreqs(freqs=[1.6654018e9,1.667359e9,])
```

Currently this is not implemented

In both of the above modes, you can also specify the rest frequencies via names in a known list rather than by their values.

Examples:

```
ASAP> scans.set_restfreqs(freqs=['OH1665','OH1667'])
```

4.3.3 Masks

Many tasks (fitting, baseline subtraction, statistics etc) should only be run on range of channels. Depending on the current “unit” setting this range is set directly as channels, velocity or frequency ranges. Internally these are converted into a simple boolean mask for each channel of the abscissa. This means that if the unit setting is later changed, previously created mask are still valid. (This is not true for functions which change the shape or shift the frequency axis). You create masks with the function `create_mask` and this specified the channels to be included in the selection. When setting the mask in velocity, the conversion from velocity to channels is based on the current cursor setting, selected row and selected frequency reference frame.

Note that for multi IF data with different number of channels per IF a single mask cannot be applied to different IFs. To use a mask on such data the selector should be applied to select all IFs with the same number of channels.

Example :

```
# Select channel range for baselining
ASAP> scans.set_unit('channels')
ASAP> msk = scans.create_mask([100,400],[600,800])

# Select velocity range for fitting
ASAP> scans.set_unit('km/s')
ASAP> msk = scans.create_mask([-30,-10])
```

Sometimes it is more convenient to specify the channels to be excluded, rather included. You can do this with the “invert” argument.

Example :

```
ASAP> scans.set_unit('channels')
ASAP> msk = scans.create_mask([0,100],[900-1023], invert=True)
```

By default `create_mask` uses the frequency setup of the first row to convert velocities into a channel mask. If the rows in the data cover different velocity ranges, the scantable row to use should be specified:

```
ASAP> scans.set_unit('km/s')
ASAP> msk = q.create_mask([-30,-10], row=5)
```

Because the mask is stored in a simple python variable, the users is able to combine masks using simple arithmetic. To create a mask excluding the edge channels, a strong maser feature and a birdie in the middle of the band:

```
ASAP> scans.set_unit('channels')
ASAP> msk1 = q.create_mask([0,100],[511,511],[900,1023],invert=True)
ASAP> scans.set_unit('km/s')
ASAP> msk2 = q.create_mask([-20,-10],invert=True)

ASAP> mask = msk1 and msk2
```

4.4 Management

During processing it is possible to create a large number of scan tables. These all consume memory, so it is best to periodically remove unneeded scan tables. Use `list_scans` to print a list of all scantables and `del` to remove unneeded ones.

Example:

```
ASAP> list_scans()
The user created scantables are:
['s', 'scans', 'av', 's2', 'ss']

ASAP> del s2
ASAP> del ss
```

5 Data Input

Data can be loaded in one of two ways; using the reader object or via the scantable constructor. The scantable method is simpler but the reader allows the user more control on what is read.

5.1 Scantable constructor

This loads all of the data from filename into the scantable object scans and averages all the data within a scan (i.e. the resulting scantable will have one row per scan). The recognised input file formats are RPFITS, SDFITS (singledish fits), ASAP's scantable format and aips++ MeasurementSet2 format.

Example usage:

```

ASAP> scan = scantable('2004-11-23_1841-P484.rpf')

# Don't scan average the data
ASAP> scan = scantable('2004-11-23_1841-P484.rpf', average=False)

```

5.2 Reader object

For more control when reading data into ASAP, the reader object should be used. This has the option of only reading in a range of integrations, only a specified beam or IF and does not perform any scan averaging of the data, allowing analysis of the individual integrations. Note that due to limitation of the RPFITS library, only one reader object can be open at one time reading RPFITS files. To read multiple RPFITS files, the old reader must be destroyed before the new file is opened. However, multiple readers can be created and attached to SDFITS files.

Example usage:

```

ASAP> r = reader('2003-03-16_082048_t0002.rpf')
ASAP> r.summary()
ASAP> scan = r.read()
ASAP> del r

```

6 Basic Processing

In the following section, a simple data reduction to form a quotient spectrum of a single source is followed. It has been assume that the `.asaprc` file has *not* been used to change the `insitu` default value from `True`.

6.1 Auto quotient

Quotients can be computed “automatically”. This requires the data to have matching source/reference pairs or one reference for multiple sources. Auto quotient assumes reference scans have a trailing “R” in the source name for data from Parkes and Mopra, and a trailing “e” or “w” for data fro, Tidbinbilla.

```

ASAP> q = s.auto_quotient()

```

By default the quotient spectra is calculated to preserve continuum emission. If you wish to remove the continuum contribution, use the `preserve` argument:

```

ASAP> q = s.auto_quotient(preserve=True)

```

6.2 Time average separate scans

If you have observed the source with multiple source/reference cycles you will want to scan-average the quotient spectra together.

```
ASAP> av = q.average_time()
```

If for some you want to average multiple sets of scantables together you can:

```
ASAP> av = average_time(q1, q2, q3)
```

The default is to use integration time weighting. The alternative is to use none, variance, Tsys weighting or Tsys & integration time.

```
ASAP> av = average_time(q, weight='tintsys')
```

To use variance based weighting, you need to supply a mask saying which channel range you want it to calculate the variance from.

```
ASAP> msk = scans.create_mask([200,400],[600,800])
ASAP> av = average_time(scans, mask=msk, weight='var')
```

If you have not observed your data with Doppler tracking (or run `freq_align` explicitly) you should align the data in frequency before averaging.

```
ASAP> av = scans.average_time(align=True)
```

Note that, if needed, you should run `gain_el` and `opacity` before you average the data in time (§6.5.4 & 6.6).

6.3 Baseline fitting

To make a baseline fit, you must first create a mask of channels to use in the baseline fit.

```
ASAP> msk = scans.create_mask([100,400],[600,900])
ASAP> scans.poly_baseline(msk, order=1)
```

This will fit a first order polynomial to the selected channels and subtract this polynomial from the full spectra.

6.3.1 Auto-baselining

The function `auto_poly_baseline` 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.

Simple example

```
ASAP> scans.auto_poly_baseline(order=2,threshold=5)
```

`order` is the polynomial order for the fit. `threshold` is the SNR threshold to use to delimitate line emission from signal. Generally the value of threshold is not too critical, however making this too large will compromise the fit (as it will include strong line features) and making it too small will mean it cannot find enough line free channels.

Other examples:

```
# Don't try and fit the edge of the bandpass which is noisier
ASAP> scans.auto_poly_baseline(edge=(500,450),order=3,threshold=3)

# Only fit a given region around the line
ASAP> scans.set_unit('km/s')
ASAP> msk = scans.create_mask([-60,-20])
ASAP> scans.auto_poly_baseline(mask=msk,order=3,threshold=3)
```

6.4 Average the polarisations

If you are just interested in the highest SNR for total intensity you will want to average the parallel polarisations together.

```
ASAP> scans.average_pol()
```

6.5 Calibration

For most uses, calibration happens transparently as the input data contains the Tsys measurements taken during observations. The nominal “Tsys” values may be in Kelvin or Jansky. The user may wish to supply a Tsys correction or apply gain-elevation and opacity corrections.

6.5.1 Brightness Units

RPFITS files do not contain any information as to whether the telescope calibration was in units of Kelvin or Janskys. On reading the data a default value is set depending on the telescope and frequency of observation. If this default is incorrect (you can see it in the listing from the `summary` function) the user can either override this value on reading the data or later. E.g:

```
ASAP> scans = scantable('2004-11-23_1841-P484.rpf', unit='Jy')
# Or in two steps
ASAP> scans = scantable('2004-11-23_1841-P484.rpf')
ASAP> scans.set_fluxunit('Jy')
```

6.5.2 Tsys scaling

Sometime the nominal Tsys measurement at the telescope is wrong due to an incorrect noise diode calibration. This can easily be corrected for with the `scale` function. By default,

`scale` only scans the spectra and not the corresponding `Tsys`.

```
ASAP> scans.scale(1.05, tsys=True)
```

6.5.3 Unit Conversion

To convert measurements in Kelvin to Jy (and vice versa) the global function `convert_flux` is needed. This converts and scales the data from K to Jy or vice-versa depending on what the current brightness unit is set to. The function knows the basic parameters for some frequencies and telescopes, but the user may need to supply the aperture efficiency, telescope diameter or the Jy/K factor.

```
ASAP> scans.convert_flux()           # If efficiency known
ASAP> scans.convert_flux(eta=0.48)   # If telescope diameter known
ASAP> scans.convert_flux(eta=0.48,d=35) # Unknown telescope
ASAP> scans.convert_flux(jypk=15)    # Alternative
```

6.5.4 Gain-Elevation and Opacity Corrections

As higher frequencies (particularly >20 GHz) it is important to make corrections for atmospheric opacity and gain-elevation effects.

Note that currently the elevation is not written correctly into Tidbinbilla rpfits files. This means that gain-elevation and opacity corrections will not work unless these get recalculated.

```
ASAP> scans.recalc_azel()           # recalculate az/el based on pointing
```

Gain-elevation curves for some telescopes and frequencies are known to ASAP (currently only for Tidbinbilla at 20 GHz). In these cases making gain-corrections is simple. If the gain curve for your data is not known, the user can supply either a gain polynomial or text file tabulating gain factors at a range of elevations (see `help scantable.gain_el`).

Examples:

```
ASAP> scans.gain_el()    # If gain table known
ASAP> scans.gain_el(poly=[3.58788e-1,2.87243e-2,-3.219093e-4])
```

Opacity corrections can be made with the global function `opacity`. This should work on all telescopes as long as a measurement of the opacity factor was made during the observation.

```
ASAP> scans.opacity(0.083)
```

Note that at 3 mm Mopra uses a paddle wheel for `Tsys` calibration, which takes opacity effects into account (to first order). ASAP opacity corrections should not be used for Mopra 3-mm data.

6.6 Frequency Frame Alignment

When time averaging a series of scans together, it is possible that the velocity scales are not exactly aligned. This may be for many reasons such as not Doppler tracking the observations, errors in the Doppler tracking etc. This mostly affects very long integrations or integrations averaged together from different days. Before averaging such data together, they should be frequency aligned using `freq_align`.

E.g.:

```
ASAP> scans.freq_align()
ASAP> av = average_time(scans)
```

A Global freq_align command will be made eventually

To average together data taken on different days, which are in different scantables, each scantable must be aligned to a common reference time then the scantables averaged. The simplest way of doing this is to allow ASAP to choose the reference time for the first scantable then using this time for the subsequent scantables.

```
ASAP> scans1.freq_align() # Copy the reference Epoch from the output
ASAP> scans2.freq_align(reftime='2004/11/23/18:43:35')
ASAP> scans3.freq_align(reftime='2004/11/23/18:43:35')
ASAP> av = average_time(scans1, scans2, scans3)
```

7 Scantable manipulation

While it is very useful to have many independent sources within one scantable, it is often inconvenient for data processing. The `get_scan` function can be used to create a new scantable with a selection of scans from a scantable. The selection can either be on the source name, with simple wildcard matching or set of scan ids. Internally this uses the selector object, so for more complicated selection the selector should be used directly instead.

For example:

```
ASAP> ss = scans.get_scan(10) # Get the 11th scan (zero based)
ASAP> ss = scans.get_scan(range(10)) # Get the first 10 scans
ASAP> ss = scans.get_scan(range(10,20)) # Get the next 10 scans
ASAP> ss = scans.get_scan([2,4,6,8,10]) # Get a selection of scans

ASAP> ss = scans.get_scan('345p407') # Get a specific source
ASAP> ss = scans.get_scan('345*') # Get a few sources

ASAP> r = scans.get_scan('*_R') # Get all reference sources (Parkes/Mopra)
ASAP> s = scans.get_scan('*_S') # Get all program sources (Parkes/Mopra)
ASAP> r = scans.get_scan('*[ew]') # Get all reference sources (Tid)
ASAP> s = scans.get_scan('*[^ew]') # Get all program sources (Tid)
```

To copy a scantable the following does not work:

```
ASAP> ss = scans
```

as this just creates a reference to the original scantable. Any changes made to `ss` are also seen in `scans`. To duplicate a scantable, use the `copy` function.

```
ASAP> ss = scans.copy()
```

8 Data Output

ASAP can save scantables in a variety of formats, suitable for reading into other packages. The formats are:

ASAP This is the internal format used for ASAP. It is the only format that allows the user to restore the data, fits etc. without losing any information. As mentioned before, the ASAP scantable is an AIPS++ Table (a memory-based table). This function just converts it to a disk-based Table. You can access that Table with the AIPS++ Table browser or any other AIPS++ tool.

SDFITS The Single Dish FITS format. This format was designed to for interchange between packages, but few packages actually can read it.

FITS This uses simple “image” fits to save the data, each row being written to a separate fits file. This format is suitable for importing the data into CLASS.

ASCII A simple text based format suitable for the user to processing using Perl or, Python, gnuplot etc.

MS2 Saves the data in an aips++ MeasurementSet V2 format. You can also access this with the Table browser and other AIPS++ tools.

The default output format can be set in the users `.asaprc` file. Typical usages are:

```
ASAP> scans.save('myscans') # Save in default format
ASAP> scans.save('myscans', 'FITS') # Save as FITS for exporting into CLASS
ASAP> scans.save('myscans', overwrite=True) # Overwrite an existing file
```

9 Plotter

Scantable spectra can be plotted at any time. An `asapplotter` object is used for plotting, meaning multiple plot windows can be active at the same time. On start up a default `asapplotter` object is created called “plotter”. This would normally be used for standard plotting.

The plotter, optionally, will run in a multipanel mode and contain multiple plots per panel. The user must tell the plotter how they want the data distributed. This is done using the `set_mode` function. The default can be set in the users `.asaprc` file. The units (and frame etc) of the abscissa will be whatever has previously been set by `set_unit`, `set_freqframe` etc.

Typical plotter usage would be:

```
ASAP> scans.set_unit('km/s')
ASAP> plotter.set_mode(stacking='p',panelling='t')
ASAP> plotter.plot(scans)
```

This will plot multiple polarisation within each plot panel and each scan row in a separate panel.

Other possibilities include:

```
# Plot multiple IFs per panel
ASAP> plotter.set_mode(stacking='i',panelling='t')

# Plot multiple beams per panel
ASAP> plotter.set_mode(stacking='b',panelling='t')

# Plot one IF per panel, time stacked
ASAP> plotter.set_mode('t', 'i')

# Plot each scan in a seperate panel
ASAP> plotter.set_mode('t', 's')
```

9.1 Plot Selection

The plotter can plot up to 25 panels and stacked spectra per panel. If you have data larger than this (or for your own sanity) you need to select a subset of this data. This is particularly true for multibeam or multi IF data. The selector object should be used for this purpose. Selection can either be applied to the scantable or directly to the plotter, the end result is the same. You don't have to reset the scantable selection though, if you set the selection on the plotter.

Examples:

```
ASAP> selection = selector()
# Select second IF
ASAP> selection.set_ifs(1)
ASAP> plotter.set_selection(selection)

# Select first 4 beams
ASAP> selection.set_beams([0,1,2,3])
```

```

ASAP> plotter.set_selection(selection)

# Select a few scans
ASAP> selection.set_scans([2,4,6,10])
ASAP> plotter.set_selection(selection)

# Multiple selection
ASAP> selection.set_ifs(1)
ASAP> selection.set_scans([2,4,6,10])
ASAP> plotter.set_selection(selection)

```

9.2 Plot Control

The plotter window has a row of buttons on the lower left. These can be used to control the plotter (mostly for zooming the individual plots). From left to right:

Home This will unzoom the plots to the original zoom factor

Plot history (left and right arrow). The plotter keeps a history of zoom settings. The left arrow sets the plot zoom to the previous value. The right arrow returns back again. This allows you, for example, to zoom in on one feature then return the plot to how it was previously.

Pan (The Cross) This sets the cursor to pan, or scroll mode allowing you to shift the plot within the window. Useful when zoomed in on a feature.

Zoom (the letter with the magnifying glass) lets you draw a rectangle around a region of interest then zooms in on that region. Use the plot history to unzoom again.

Adjust (rectangle with 4 arrows) adjust subplot parameters (space at edge of plots)

Save (floppy disk). Save the plot as a postscript or .png file

You can also type “g” in the plot window to toggle on and off grid lines. Typing ‘l’ turns on and off logarithmic Y-axis.

9.3 Other control

The plotter has a number of functions to describe the layout of the plot. These include `set_legend`, `set_layout` and `set_title`.

To set the exact velocity or channel range to be plotted use the `set_range` function. To reset to the default value, call `set_range` with no arguments. E.g.

```

ASAP> scans.set_unit('km/s')
ASAP> plotter.plot(scans)
ASAP> plotter.set_range(-150,-50)
ASAP> plotter.set_range() # To reset

```

Both the range of the “x” and “y” axis can be set at once, if desired:

```
ASAP> plotter.set_range(-10,30,-1,6.6)
```

To save a hardcopy of the current plot, use the save function, e.g.

```
ASAP> plotter.save('myplot.ps')
ASAP> plotter.save('myplot.png', dpi=80)
```

10 Fitting

Currently multicomponent Gaussian function is available. This is done by creating a fitting object, setting up the fit and actually fitting the data. Fitting can either be done on a single scantable selection or on an entire scantable using the `auto_fit` function. If single value fitting is used, and the current selection includes multiple spectra (beams, IFs, scans etc) then the first spectrum in the scantable will be used for fitting.

```
ASAP> f = fitter()
ASAP> f.set_function(gauss=2) # Fit two Gaussians
ASAP> f.set_scan(scans)
ASAP> selection = selector()
ASAP> selection.set_polarisations(1) # Fit the second polarisation
ASAP> scans.set_selection(selection)
ASAP> scans.set_unit('km/s') # Make fit in velocity units
ASAP> f.fit(1) # Run the fit on the second row in the table
ASAP> f.plot() # Show fit in a plot window
ASAP> f.get_parameters() # Return the fit parameters
```

This auto-guesses the initial values of the fit and works well for data without extra confusing features. Note that the fit is performed in whatever unit the abscissa is set to.

If you want to confine the fitting to a smaller range (e.g. to avoid band edge effects or RFI) you must set a mask.

```
ASAP> f = fitter()
ASAP> f.set_function(gauss=2)
ASAP> scans.set_unit('km/s') # Set the mask in channel units
ASAP> msk = s.create_mask([1800,2200])
ASAP> scans.set_unit('km/s') # Make fit in velocity units
ASAP> f.set_scan(s,msk)
ASAP> f.fit()
ASAP> f.plot()
ASAP> f.get_parameters()
```

If you wish, the initial parameter guesses can be specified and specific parameters can be fixed:

```

ASAP> f = fitter()
ASAP> f.set_function(gauss=2)
ASAP> f.set_scan(s,msk)
ASAP> f.fit() # Fit using auto-estimates
# Set Peak, centre and fwhm for the second gaussian.
# Force the centre to be fixed
ASAP> f.set_gauss_parameters(0.4,450,150,0,1,0,component=1)
ASAP> f.fit() # Re-run the fit

```

The `fitter` `plot` function has a number of options to either view the fit residuals or the individual components (by default it plots the sum of the model components).

Examples:

```

# Plot the residual
ASAP> f.plot(residual=True)

# Plot the first 2 components
ASAP> f.plot(components=[0,1])

# Plot the first and third component plus the model sum
ASAP> f.plot(components=[-1,0,2]) # -1 means the component sum

```

10.1 Fit saving

Once you are happy with your fit, it is possible to store it as part of the scantable.

```

ASAP> f.storefit()

```

This will be saved to disk with the data, if the “ASAP” file format is selected. Multiple fits to the same data can be stored in the scantable.

The scantable function `get_fit` can be used to retrieve the stored fits. Currently the fit parameters are just printed to the screen.

```

ASAP> scans.get_fit(4) # Print fits for row 4

```

11 Polarisation

Currently ASAP only supports polarimetric analysis on linearly polarised feeds and the cross polarisation products measured. Other cases will be added on an as needed basis.

Conversions of linears to Stokes or Circular polarisations are done “on-the-fly”. Leakage cannot be corrected for nor are there routines to calibrate position angle offsets.

11.1 Simple Calibration

It is possible that there is a phase offset between polarisation which will effect the phase of the cross polarisation correlation, and so give rise to spurious polarisation. `rotate_xyphase` can be used to correct for this error. At this point, the user must know how to determine the size of the phase offset themselves.

```
ASAP> scans.rotate_xyphase(10.5)           # Degrees
```

Note that if this function is run twice, the sum of the two values is applied because it is done in-situ.

A correction for the receiver parallactic angle may need to be made, generally because of how it is mounted. Use `rotate_linpolphase` to correct the position angle. Running this function twice results in the sum of the corrections being applied because it is applied in-situ.

```
ASAP> scans.rotate_linpolphase(-45) # Degrees; correct for receiver mounting
```

If the sign of the complex correlation is wrong (this can happen depending on the correlator configuration), use `invert_phase` to change take the complex conjugate of the complex correlation term. This is always performed in-situ.

```
ASAP> scans.invert_phase()
```

Depending on how the correlator is configured, “BA” may be correlated instead of “AB”. Use `swap_linears` to correct for this problem:

```
ASAP> scans.swap_linears()
```

11.2 Conversion

Data can be converted between linear and circular polarisations and stokes.

```
ASAP> stokes_scans = linearscans.convert_pol("stokes")
```

11.3 Plotting

To plot Stokes values, a selector object must be created and the `set_polarisation` function used to select the desired polarisation products.

The values which can be plotted include a selection of [I,Q,U,V], [I, Plinear, Pangle, V], [RR, LL] or [XX, YY, Real(XY), Imaginary(XY)]. (Plinear and Pangle are the percentage and position angle of linear polarisation).

Example:

```

ASAP> selection = selector()

ASAP> selection.set_polarisations('I Q U V')
ASAP  plotter.set_selection(selection);           # Select I, Q, U \& V

ASAP> selection.set_polarisations('I Q')
ASAP  plotter.set_selection(selection);           # Select just I \& Q

ASAP> selection.set_polarisations('RR LL')
ASAP  plotter.set_selection(selection);           # Select just RR \& LL

ASAP> selection.set_polarisations('XX YY')
ASAP  plotter.set_selection(selection);           # Select linears

ASAP> selection.set_polarisations('I Plinear')
ASAP  plotter.set_selection(selection);           # Fractional linear

ASAP> selection.set_polarisations('Pangle')
ASAP  plotter.set_selection(selection);           # Position angle

```

Scan, beam and IF selection are also available in the selector object as describe in section 4.2.

11.4 Saving

When saving data using the `save` function, the `stokes` argument can be used to save the data as Stoke values when saving in FITS format.

Example:

```

ASAP> scans.save('myscan.sdfits', 'SDFITS', stokes=True)

```

12 Scantable Mathematics

It is possible to to simple mathematics directly on scantables from the command line using the `+`, `-`, `*`, `/` operators as well as their cousins `+=`, `-=`, `*=`, `/=`. This works between a scantable and a float. (Note that it does not work for integers).

Currently mathematics between two scantables is not available

```

ASAP> scan2 = scan1+2.0
ASAP> scan *= 1.05

```

13 Scripting

Because asap is based on python, it is easy for the user to write their own scripts and functions to process data. This is highly recommended as most processing of user data could then be done in a couple of steps using a few simple user defined functions. A Python primer is beyond the scope of this userguide. See the asap home pages for a scripting tutorial or the main python website for comprehensive documentation.

<http://www.atnf.csiro.au/computing/software/asap/tutorials>

<http://www.python.org/doc/Introduction.html>

13.1 Running scripts

The asap global function `execfile` reads the named text file and executes the contained python code. This file can either contain function definitions which will be used in subsequent processing or just a set of commands to process a specific dataset.

13.2 asapuserfuncs.py

The file `~/asap/asapuserfuncs.py` is automatically read in when asap is started. The user can use this to define a set of user functions which are automatically available each time asap is used. The `execfile` function can be called from within this file.

14 Worked examples

In the following section a few examples of end-to-end processing of some data in asap are given.

14.1 Mopra

The following example is of some dual polarisation, position switched data from Mopra. The source has been observed multiple times split into a number of separate rf fits files. To make the processing easier, the first step is to `cat` the separate rf fits files together and load as a whole (future versions of asap will make this unnecessary).

```
# get a list of the individual rf fits files in the current directory
myfiles = list_files()

# Load the data into a scantable
data = scantable(myfiles)
print data

# Form the quotient spectra
q = data.auto_quotient()
print q
```

```

# Look at the spectra
plotter.plot(q)

# Set unit and reference frame
q.set_unit('km/s')
q.set_freqframe('LSRK')

# Average all scans in time, aligning in velocity
av = q.average_time(align=True)
plotter.plot(av)

# Remove the baseline
msk = av.create_mask([100,130],[160,200])
av.poly_baseline(msk,2)

# Average the two polarisations together
iav = av.average_pol()
print iav
plotter.plot(iav)

# Set a sensible velocity range on the plot
plotter.set_range(85,200)

# Smooth the data a little
av.smooth('gauss',4)
plotter.plot()

# Fit a gaussian to the emission
f = fitter()
f.set_function(gauss=1)
f.set_scan(av)
f.fit()

# View the fit
f.plot()

# Get the fit parameters
f.get_parameters()

```

14.2 Parkes Polarimetry

The following example is processing of some Parkes polarimetric observations of OH masers at 1.6 GHz. Because digital filters were used in the backend, the baselines are stable enough not to require a quotient spectra. The 4 MHz bandwidth is wide enough to observe

both the 1665 and 1667 MHz OH maser transitions. Each source was observed once for about 10 minutes. Tsys information was not written to the rpfits file (a nominal 25K values was used), so the amplitudes need to be adjusted based on a separate log file. A simple user function is used to simplify this, contained in a file called mypol.py:

```
def xyscale(data,xtsys=1.0,ytsys=1.0,nomtsys=25.0) :  
  
    selection = selector()  
    selection.set_polarisations(0)  
    data.set_selection(selection)  
    data.scale(xtsys/nomtsys)  
  
    selection.set_polarisations(1)  
    data.set_selection(selection)  
    data.scale(ytsys/nomtsys)  
  
    selection.set_polarisations(0)  
    data.set_selection(selection)  
    data.scale((xtsys+ytsys)/(2*nomtsys))  
  
    selection.set_polarisations(0)  
    data.set_selection(selection)  
    data.scale((xtsys+ytsys)/(2*nomtsys))
```

The typical asap session would be

```
# Remind ourself the name of the rpfits files  
ls  
  
# Load data from an rpfits file  
d1665 = scantable('2005-10-27_0154-P484.rpf')  
  
# Check what we have just loaded  
d1665.summary()  
  
# View the data in velocity  
d1665.set_unit('km/s')  
d1665.set_freqframe('LSRK')  
  
# Correct for the known phase offset in the crosspol data  
d1665.rotate_xyphase(-4)  
  
# Create a copy of the data and set the rest frequency to the 1667 MHz  
# transition  
d1667 = d1665.copy()
```

```

d1667.set_restfreqs([1667.3590], 'MHz')
d1667.summary()

# Copy out the scan we wish to process
g351_5 = d1665.get_scan('351p160')
g351_7 = d1667.get_scan('351p160')

# Baseline both
msk = g351_5.create_mask([-30,-25],[-5,0])
g351_5.poly_baseline(msk,order=1)
msk = g351_7.create_mask([-30,-25],[-5,0])
g351_7.poly_baseline(msk,order=1)

# Plot the data. The plotter can only plot a single scantable
# So we must merge the two tables first

plotskans = merge(g351_5, g351_7)

plotter.plot(plotskans) # Only shows one panel

# Tell the plotter to stack polarisation and panel scans
plotter.set_mode('p','s')

# Correct for the Tsys using our predefined function
execfile('mypol.py') # Read in the function xyscale
xyscale(g351_5,23.2,22.7) # Execute it on the data
xyscale(g351_7,23.2,22.7)

# Only plot the velocity range of interest
plotter.set_range(-30,10)

# Update the plot with the baselined data
plotter.plot()

# Look at the various polarisation products
selection = selector()
selection.set_polarisations('RR LL')
plotter.set_selection(selection)
selection.set_polarisations('I Plinear')
plotter.set_selection(selection)
selection.set_polarisations('I Q U V')
plotter.set_selection(selection)

# Save the plot as postscript
plotter.save('g351_stokes.ps')

```

```
# Save the process spectra
plotskans.save('g351.asap')
```

14.3 Tidbinbilla

The following example is processing of some Tidbinbilla observations of NH_3 at 12 mm. Tidbinbilla has (at the time of observations) a single polarisation, but can process two IFs simultaneously. In the example, the first half of the observation was observing the (1,1) and (2,2) transitions simultaneously). The second half observed only the (4,4) transition due to bandwidth limitations. The data is position switched, observing first an reference to the west, then the source twice and finally reference to the east.

```
# Load the rpfits file and inspect
d = scantable('2003-03-16_082048_t0002.rpf')
print d

# Make the quotient spectra
q = d.auto_quotient()
print q

del d

# Plot/select in velocity
q.set_freqframe('LSRK')
q.set_unit('km/s')

# Correct for gain/el effects

q.recalc_azel() # Tid does not write the elevation
q.gain_el()
q.opacity(0.05)

# Seperate data from the (1,1)&(2,2) and (4,4) transitions
g1 = q.get_scan(range(6))      # scans 0..5
g2 = q.get_scan(range(6,12))  # scans 6..11

# Align data in velocity
g1.freq_align()
g2.freq_align()

# Average individual scans
a1 = g1.average_time()
a2 = g2.average_time()
```

```
# Rpfits file only contains a single rest frequency. Set both
a1.set_restfreqs([23694.4700e6,23722.6336e6])

plotter.plot(a1)
plotter.set_mode('i','t')

a1.auto_poly_baseline()

plotter.plot()

a1.smooth('gauss',5)
plotter.plot()
```

15 Appendix

15.1 Function Summary

ASAP> commands()

[The scan container]

scantable	- a container for integrations/scans (can open asap/rpfits/sdfits and ms files)
copy	- returns a copy of a scan
get_scan	- gets a specific scan out of a scantable (by name or number)
set_selection	- set a new subselection of the data
get_selection	- get the current selection object
summary	- print info about the scantable contents
stats	- get specified statistic of the spectra in the scantable
stddev	- get the standard deviation of the spectra in the scantable
get_tsys	- get the TSys
get_time	- get the timestamps of the integrations
get_sourcename	- get the source names of the scans
get_azimuth	- get the azimuth of the scans
get_elevation	- get the elevation of the scans
get_parangle	- get the parallactic angle of the scans
get_unit	- get the current unit
set_unit	- set the abscissa unit to be used from this point on
get_abscissa	- get the abscissa values and name for a given row (time)
set_freqframe	- set the frame info for the Spectral Axis (e.g. 'LSRK')
set_doppler	- set the doppler to be used from this point on
set_dirframe	- set the frame for the direction on the sky
set_instrument	- set the instrument name
get_fluxunit	- get the brightness flux unit
set_fluxunit	- set the brightness flux unit
create_mask	- return an mask in the current unit for the given region. The specified regions are NOT masked
get_restfreqs	- get the current list of rest frequencies
set_restfreqs	- set a list of rest frequencies
flag_spectrum	- flag data
save	- save the scantable to disk as either 'ASAP' or 'SDFITS'
nbeam,nif,nchan,npol	- the number of beams/IFs/Pols/Chans
nscan	- the number of scans in the scantable

nrow	- the number of spectra in the scantable
history	- print the history of the scantable
get_fit	- get a fit which has been stored with the data
average_time	- return the (weighted) time average of a scan or a list of scans
average_pol	- average the polarisations together. The dimension won't be reduced and all polarisations will contain the averaged spectrum.
convert_pol	- convert to a different polarisation type
auto_quotient	- return the on/off quotient with automatic detection of the on/off scans (matched pairs and 1 off - n on)
scale	- return a scan scaled by a given factor
add	- return a scan with given value added
bin	- return a scan with binned channels
resample	- return a scan with resampled channels
smooth	- return the spectrally smoothed scan
poly_baseline	- fit a polynomial baseline to all Beams/IFs/Pols
auto_poly_baseline	- automatically fit a polynomial baseline
recalc_azel	- recalculate azimuth and elevation based on the pointing
gain_el	- apply gain-elevation correction
opacity	- apply opacity correction
convert_flux	- convert to and from Jy and Kelvin brightness units
freq_align	- align spectra in frequency frame
rotate_xyphase	- rotate XY phase of cross correlation
rotate_linpolfphase	- rotate the phase of the complex polarization $O=Q+iU$ correlation
freq_switch	- perform frequency switching on the data
stats	- Determine the specified statistic, e.g. 'min' 'max', 'rms' etc.
stddev	- Determine the standard deviation of the current beam/if/pol

[Math] Mainly functions which operate on more than one scantable

average_time	- return the (weighted) time average of a list of scans
quotient	- return the on/off quotient
simple_math	- simple mathematical operations on two scantables, 'add', 'sub', 'mul', 'div'

[Fitting]

fitter	
auto_fit	- return a scan where the function is applied to all Beams/IFs/Pols.

commit	- return a new scan where the fits have been committed.
fit	- execute the actual fitting process
store_fit	- store the fit parameters in the data (scantable)
get_chi2	- get the Chi ²
set_scan	- set the scantable to be fit
set_function	- set the fitting function
set_parameters	- set the parameters for the function(s), and set if they should be held fixed during fitting
set_gauss_parameters	- same as above but specialised for individual gaussian components
get_parameters	- get the fitted parameters
plot	- plot the resulting fit and/or components and residual

[Plotter]

asapplotter	- a plotter for asap, default plotter is called 'plotter'
plot	- plot a scantable
save	- save the plot to a file ('png' , 'ps' or 'eps')
set_mode	- set the state of the plotter, i.e. what is to be plotted 'colour stacked' and what 'panelled'
set_selection	- only plot a selected part of the data
set_range	- set a 'zoom' window [xmin,xmax,ymin,ymax]
set_legend	- specify user labels for the legend indeces
set_title	- specify user labels for the panel indeces
set_abscissa	- specify a user label for the abscissa
set_ordinate	- specify a user label for the ordinate
set_layout	- specify the multi-panel layout (rows,cols)
set_colors	- specify a set of colours to use
set_linestyles	- specify a set of linestyles to use if only using one color
set_mask	- set a plotting mask for a specific polarization

[Reading files]

reader	- access rpfits/sdfits files
open	- attach reader to a file
close	- detach reader from file
read	- read in integrations
summary	- list info about all integrations

[General]

commands	- this command
print	- print details about a variable
list_scans	- list all scantables created bt the user
list_files	- list all files readable by asap (default rpf)
del	- delete the given variable from memory

range	- create a list of values, e.g. range(3) = [0,1,2], range(2,5) = [2,3,4]
help	- print help for one of the listed functions
execfile	- execute an asap script, e.g. execfile('myscript')
list_rcparameters	- print out a list of possible values to be put into \$HOME/.asaprc
mask_and,mask_or, mask_not	- boolean operations on masks created with scantable.create_mask

Note:

How to use this with help:

function 'summary'

[xxx] is just a category

Every 'sub-level' in this list should be replaced by a '.' Period when using help

Example:

ASAP> help scantable # to get info on this scantable

ASAP> help scantable.summary # to get help on the scantable's

ASAP> help average_time

15.2 ASCII output format

15.3 .asaprc settings

verbose	True /False	Print verbose output, good to disable in scripts
insitu	True /False	Apply operations on the input scantable or return new one
useplotter	True /False	Preload a default plotter
plotter.gui	True /False	Do we want a GUI or plot to a file
plotter.stacking	Pol Beam IF Scan Time	Default mode for colour stacking
plotter.panelling	Pol Beam IF Scan Time	Default mode for panelling
plotter.ganged	True /False	Push panels together, to share axislabels
plotter.decimate	True/ False	Decimate the number of points plotted by a factor of nchan/1024

scantable.save	ASAP SDFITS FITS ASCII MS2	Default output format when saving
scantable.autoaverage	True /False	Auto averaging on read
scantable.freqframe	LSRK TOPO BARY etc	default frequency frame to set when function scantable.set_freqframe is called or the data is imported

15.4 Installation

ASAP depends on a number of third-party libraries which you must have installed before attempting to build ASAP. These are:

- AIPS++
- Boost
- Matplotlib
- python/ipython

Debian Linux is currently supported and we intend also to support other popular Linux flavours, Solaris and Mac.

Of the dependencies, AIPS++ is the most complex to install.