Volume 2 - Generic Processing

AIPS++ Group, Editor: Neil Killeen

October 8, 2002

Contents

Chapter 1

Synthesis Calibration

Athol Kemball & George Moellenbrock

1.1 Introduction

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

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

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

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

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

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

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

1.2 Calibration philosophy

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

1.2.1 The Measurement Equation

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

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

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

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

$$\vec{V}_{\rm ij} = \left(\begin{array}{c} V_{\rm pp} \\ V_{\rm pq} \\ V_{\rm qp} \\ V_{\rm qq} \end{array} \right)_{ij} \tag{1.1}$$

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

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

$$\vec{\mathcal{I}} = \begin{pmatrix} I \\ Q \\ U \\ V \end{pmatrix} \tag{1.2}$$

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

$$J_{ij} = \left[J_i \otimes J_j^*\right] \tag{1.3}$$

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

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \otimes \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00} & a_{00}b_{01} & a_{01}b_{00} & a_{01}b_{01} \\ a_{00}b_{10} & a_{00}b_{11} & a_{01}b_{10} & a_{01}b_{11} \\ a_{10}b_{00} & a_{10}b_{01} & a_{11}b_{00} & a_{11}b_{01} \\ a_{10}b_{10} & a_{10}b_{11} & a_{11}b_{10} & a_{11}b_{11} \end{pmatrix} \tag{1.4}$$

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

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

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

$$\begin{aligned} \vec{V}_{ij} = \quad & M_{ij} \prod \left[{J^{vis}}_i \otimes {J^{vis}}^*_j\right] \times \\ & \sum_k \prod \left[{J^{sky}}_i\left(\underline{\rho}_k\right) \otimes {J^{sky}}^*_j\left(\underline{\rho}_k\right)\right] \times \\ & S\, \vec{\mathcal{I}}_k \; e^{-2\pi i\left(\underline{r}_i - \underline{r}_j\right)\underline{\rho}_k} + A_{ij} \end{aligned} \tag{1.5}$$

where:

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

- $\underline{\rho}_k$ denotes direction, subscript k indexes directions, and $(\underline{r}_i - \underline{r}_j)$ denotes the interferometer baseline vector.

- The four-by-four matrix S converts the sky brightness Stokes polarization basis to the polarization basis used in the interferometer itself.

- M_{ij} and A_{ij} denote multiplicative and additive baseline-based instrumental effects respectively.

- $\prod$ and $\sum$ denote product and summation, respectively.

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

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

$$\vec{V}_{ij} = \prod \left[{J^{vis}}_i \otimes {J^{vis}}^*_j \right] \vec{V}^{ideal}_{ij} \tag{1.6}$$

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

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

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

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

F_i - propagation effects due to Faraday rotation.

D_i - instrumental polarization response.

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

B_i - bandpass response, possibly normalized relative to G_i.

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

P_i - parallactic angle. This is known analytically.

Since these matrix components may not commute, the order in which they appear in the measurement equation is important. In general, the correct order is given by the order in which the terms affect the incoming wavefront. Thus, a spectral line polarization observation would be described (ignoring primary beam and direction-dependent Faraday rotation) by the measurement equation as (recalling Equation 1.3):

$$\vec{V}_{ij} = B_{ij} G_{ij} D_{ij} P_{ij} T_{ij} \vec{V}_{ij}^{ideal} \tag{1.7}$$

The measurement equation is implemented in AIPS++ by sub-dividing the machinery dealing with image-plane and uv-plane calibration between imager and calibrater respectively. The remainder of this chapter treats the case illustrated in equation 1.7, i.e., uv-plane effects which are handled by the calibrater tool.

1.2.2 Data representation and the calibration mechanism

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

The MODEL_DATA column, in contrast, is formed by propagating initial source models (perhaps for trial imaging) from right to left through the ME, applying the supported image-plane corrections, and Fourier transforming to the uv-plane. This realizes a best guess for the $\vec{V}_{ij}^{ideal}$ term in equation 1.7. The point at which these two columns meet in the ME allows a residual to be computed and thus an estimate of χ^2. The calibration component(s) at the intersection point in the ME are solved for by computing gradients of χ^2 with respect to the Jones matrices, and performing a standard non-linear least-squares minimization[1].

As an example of the calibration mechanism, we recast equation 1.7 to the implicit form used when solving for D, given an existing solution for G and knowledge of P. We ignore the B calibration component, either because the data is from a continuum observation (only one channel), or because we assume that the bandpass shape is unimportant, at least for this stage of the calibration. Since G and P appear on either side of D in the ME, they are applied to the observed data ($\vec{V}_{ij}$) and model data ($\vec{V}_{ij}^{ideal}$) terms, respectively:

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

$$G_{ij}^{-1}\vec{V}_{ij} = \left(G_{ij}^{-1}G_{ij}\right) D_{ij} \left(P_{ij}\vec{V}_{ij}^{ideal}\right) \tag{1.8}$$

$$D_{ij} = G_{ij}^{-1}\vec{V}_{ij} \left(P_{ij}\vec{V}_{ij}^{ideal}\right)^{-1} \tag{1.9}$$

$$\left[{D^{vis}}_i \otimes {D^{vis*}}_j\right] = G_{ij}^{-1}\vec{V}_{ij} \left(P_{ij}\vec{V}_{ij}^{ideal}\right)^{-1} \tag{1.10}$$

In equation 1.10, the D calibration is realized as the feed-based (two-by-two) Jones matrices which are solved for in the χ^2 minimization. It is straightforward to see how this algebraic manipulation can be extended to any of the calibration components, and the flexible implementation of the calibrater tool is clearly evident.

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

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

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

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

1.3 Practical use of the calibrater tool

1.3.1 Setting up the calibrater tool

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

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

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

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

1.3.2 Initial calibrator models

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

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

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

1.3.3 Uv-data selection

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

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

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

1.3.4 Calibration table conventions

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

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

1.3.5 Solving for visibility-plane effects

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

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

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

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

1.3.6 Establishing the flux density scale

Once an electronic gain solution (G_i) has been performed for all calibrator sources, those secondary sources which were assumed to be of unit flux density can be scaled to a primary flux density scale set by known calibrators included in the run. This is achieved using a utility function Calibrater.fluxscale, which takes as input the electronic gain calibration table, a list of primary and secondary sources, and the name of an optional output electronic gain calibration table. The gain factors in the output table will be scaled correctly once the function has completed. It will also report the estimated point source flux density for the secondary sources, based on the ratio of the mean gain modulus between the primary and secondary calibrators, as is standard practice in cases when this method can be used.

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

1.3.7 Correcting the observed data

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

To apply the scaled 'G' calibration obtained above:

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

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

1.3.8 Extending calibration to bandpass, polarization, etc.

In general, it is a fairly simple matter to extend calibration to additional calibration components like bandpass and polarization. In the following examples, we assume that the corrupting effects of G dominate the observed data, and that this calibration component has been solved for first, as described above. This is typical of an observation at the VLA using feeds sensitive to RCP and LCP. For arrays using linear feeds (e.g. ATCA), this assumption may be inappropriate for strongly polarized calibrators (see the chapter for ATCA processing). In any case, the general strategy for treatment of several calibration components is to solve for the dominating effects first, and iterate solutions for components which interact significantly.

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

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

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

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

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

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

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

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

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

1.3.9 Self-calibration

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

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

1.4 References

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

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

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

Chapter 2

Imaging, Deconvolution and Self-calibration

Tim Cornwell and Kumar Golap

2.1 Overview

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

2.1.1 Required Tools

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

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

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

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

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

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

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

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

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

2.1.2 Data Requirements

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

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

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

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

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

2.2 Setting up Imager

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

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

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

2.2.1 Construction

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

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

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

2.2.2 Selecting Data

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

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

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

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

2.2.3 Setting the Image Parameters

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

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

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

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

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

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

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

2.2.4 Channel Selection and Combination

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

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

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

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

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

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

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

OR

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

Here we have selected 32 input channels, starting with channel 10, and gridded them together to form one output image plane. Using 'mfs' in setimage grids all data selected in setdata in one plane (nchan, start or step are ignored). One can achieve a similar thing in using 'channel' mode in setimage as done in the second case.

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

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

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

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

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

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

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

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

2.2.5 Weighting

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

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

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

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

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

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

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

- plotweights plots the imaging weights either as function of uv distance or on a gridded plane.

- sensitivity calculates and returns the sensitivity of the resulting weighting both absolutely and relative to natural weighting.

- fitpsf calculates the dirty point spread function and returns the best fitting Gaussian.

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

```
include 'table.g'
t:=table('3C273XC1.ms', readonly=F)
wt:=t.getcol('IMAGING_WEIGHT')
wt[wt>0.0] := sqrt(wt[wt>0.0])
t.putcol('IMAGING_WEIGHT')
t.flush();
t.close()
```

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

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

2.3 Creating and Deconvolving Images

2.3.1 Creating Images

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

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

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

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

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

   You might also like to make the point spread function.

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

2.3.2 Deconvolution

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

The options for deconvolving are :

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

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

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

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

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

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

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

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

The deconvolution functions can return the residual and restored images, as well as the updated model image. In addition, there are Imager.residual and Imager.restore functions that also compute residual and restored images. The distinction is that in the former, the residual is that calculated by the deconvolution method at the end of iteration, whereas the latter returns the residual calculated by going back to the original visibility data (except for the cases of multi-field and wide-field algorithms).

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

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

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

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

2.3.3 Specifying the deconvolution region

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

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

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

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

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

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

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

NNLS requires two such masks images : one for the region where flux is to be allowed and one for the region to be used as constraints. Typically the latter region will be bigger than the former. The product of the pixels in the two regions should not exceed about 5-10 million.

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

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

2.4 Wizard interface to Imager

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

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

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

2.5 Component Models and Primary Beams

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

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

When you are imaging only single fields (i.e. not mosaicing), as is relevant to this chapter, the Componentlist is *not* a true representation of the sky brightness. The values it holds must include the attenuation of the primary beam.

This is because in single field mode, Imager makes no primary beam corrections of any form whatsoever.

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

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

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

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

2.6 Multi-scale CLEAN

2.6.1 General

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

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

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

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

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

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

2.6.2 The `imagermultiscale()` function

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

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

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

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

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

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

2.7 Wide-field imaging and Mosaicing

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

2.8 Combining single dish and interferometer images

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

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

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

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

2.9 Self-calibration

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

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

The steps are:

1. Set up a Calibrater tool for the MeasurementSet. Set the data selection to be consistent with that in the Imager `tool`. The type of solution is set in the Calibrater.setapply and Calibrater.setsolve functions. More on this below. Be sure to set the solution time scales appropriately.

2. Make a deconvolution using your favored method.

3. You might wish to clip the low unreliable levels of the model image using Imager.clipimage.

4. Use the Imager.selfcal function to actually perform the gain solution and re-calibration.

5. Remake the deconvolved image.

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

The types of solution allowed are:

T Amplitude and phase per antenna

G Amplitude and phase per feed

D Leakage between polarizations

B Bandpass per antenna

More details may be found in the Calibrater documentation.

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

2.10 Summary of Imager functions

2.10.1 Setting/seeing the basic state

- Imager.advise - give some advice on the image parameters.

- Imager.setdata - select a subset of the data from the MeasurementSet

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

2.10.2 uv selection and filtering

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

2.10.3 Masking

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

2.10.4 Create images

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

2.10.5 Deconvolution

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

2.10.6 Image Combination

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

2.10.7 Utility

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

2.10.8 Calibration and self-calibration

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

2.11 Examples of Imaging Scripts

2.11.1 Multi-frequency synthesis image of one field

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

2.11.2 Multiple regions on the sky simultaneously

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

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

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

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

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

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

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

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

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

2.11.3 Spectral-line imaging

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

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

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

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

2.11.4 Using Clean and MEM

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

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

2.11.5 Deconvolver and multi-scale CLEAN example

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

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

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

2.11.6 Imaging and Self-Calibrating

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

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

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

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

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

Chapter 3

Wide field imaging

Kumar Golap and Tim Cornwell

3.1 Background

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

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

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

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

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

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

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

3.1.1 Possible solution to the wide-field imaging problem

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

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

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

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

  1. By phasing the visibility data appropriately, the dirty image and PSF are made for each facet and the above process is applied to the facets. At the end of processing, the final CLEANed images have to be transformed to a single common plane.

  2. By phasing the visibility data and reprojecting the uv coordinates appropriately, the dirty image and PSF are made for each facet onto a common tangent plane, and the above process is applied to the facets. No final reprojection is required.

3.2 Available Tools

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

3.2.1 Basic capabilities

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

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

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

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

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

3.2.2 Faceting size

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

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

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

Further Advice

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

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

3.2.3 Deconvolution

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

3.2.4 Self-calibration

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

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

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

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

3.2.5 Masking

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

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

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

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

3.2.6 Outlier fields

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

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

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

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

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

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

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

3.2.7 Models

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

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

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

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

3.3 A worked example: VLA 4m imaging of Coma

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

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

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

3.4 Special cases

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

3.5 Troubleshooting

3.5.1 Insufficient faceting

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

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

3.5.2 Non-isoplanatism

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

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

3.5.3 Asymmetric primary beams

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

3.5.4 Clean diverging on the edges of the facets

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

3.6 Bibliography

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

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

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

Chapter 4

Mosaicing (Multi-field imaging)

Mark Holdaway

4.1 Mosaicing Background

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

$$V(u) = \int A(x)I(x)e^{-i2\pi ux}dx \tag{4.1}$$

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

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

$$I(x) = \frac{\sum_f I_f(x)A_f(x)}{\sum_f A_f^2(x)}. \tag{4.2}$$

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

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

4.1.1 The AIPS++ Mosaicing Solution

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

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

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

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

4.1.2 Advantages of Incremental Deconvolution with an Approximate PSF

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

4.2 Mosaicwizard for Quick, Simple Mosaicing

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

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

4.3 Fundamental and Necessary Details

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

4.3.1 Set the Data Fields

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

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

4.3.2 Set the Image

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

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

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

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

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

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

4.3.3 Setting the Voltage Pattern (primary beam)

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

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

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

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

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

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

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

4.3.4 Weighting

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

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

4.3.5 Deconvolving

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

4.4 Advanced Details

4.4.1 Controlling the Major Cycles

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

One can see fairly easily if one is cleaning too deeply or not cleaning deeply enough in a given major cycle by looking at the plots given when the argument `displayprogress=T` (in `tool function clean`). When trying to clean too deeply, the peak residual level towards the end of a major cycle will flatten out (or may even increase and diverge) for many iterations. At the start of the next major cycle, the peak residual level will begin at a higher level than at the end of the previous major cycle. (This can be especially true for mosaicing with multi-scale clean, where the approximate PSF may actually be a pretty poor match for the true PSF's on the larger size-scales). If all is well, this major cycle will still get to a lower peak residual than the previous one. However, stopping the major cycles sooner will prevent us from spinning our wheels, resulting in a comparable deconvolved image in fewer iterations.

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

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

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

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

4.4.2 Details with Multi-Scale CLEAN

See the basic Imaging for multi-scale basics.

Sometimes in the first few iterations of Multi-scale CLEAN in the mosaicing context, the largest scale will be dominated by large negative residuals (i.e. this is just the negative bowl, integrated over the area of the largest clean-component scale). One way to fix this is to make the largest scale smaller. Another way is to use a tighter mask which excludes finding large scale components in the bowl region. And a third, ad hoc way is to stop the major cycle when a negative component is found on the largest scale. This allows the exact subtraction to proceed, often resulting in a reduced bowl level. Stopping the major cycle upon encountering a negative component on the largest scale should only be performed on the first few cycles, as during subsequent cycles small amplitude large scale components may be required to make adjustments in the image. The number of cycles for which stopping when a negative component is found on the largest scale can be controlled by the parameter `stoplargenegatives` in `imager.setmfcontrol`. As smaller scales may require negative components to correct for errors made by "over-cleaning" in the larger cycles, no restriction should be placed on negative components from smaller size scales.

4.4.3 The `imagermultiscale()` Function

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

4.4.4 Details with MEM

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

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

4.4.5 Using Convolutions Instead of Visibility Subtraction

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

4.4.6 Outlier Fields

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

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

Here is an example:

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

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


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

4.4.7 Component Models

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

Component Models are handled by the Componentlist `tool`.

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

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

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

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

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

4.4.8 Flux Scale Images

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

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

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

4.4.9 Masks

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

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

4.5 Self-calibration

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

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

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

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

4.6 An Example Mosaicing Script

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

```
include 'imager.g'

# Generate MS and construct Imager tool

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

# Use the first field as the image center

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

# Weight each field individually

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

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

# Use the voltage pattern (primary beam)

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

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

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

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

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

# Make a multi-scale CLEAN image  for comparison

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

# Destroy Imager tool

imgr.done();
```

4.7 Bibliography

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

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

Chapter 5

GBT Continuum Single dish Imaging

Tim Cornwell/Joe McMullin

5.1 Introduction

This chapter describes how to make continuum OTF single dish images. It should be read in conjunction with several other chapters in *Getting Results*, including the chapters which cover Display of data and Image analysis.

The examples here are worked from GBT data but the principles apply to other telescopes. In particular, this chapter provides a facile means of reducing some standard mode continuum images. Different modes/analysis can still be obtained through the direct use of the basic AIPS++ tools.

5.2 Filling single dish data into AIPS++

5.2.1 GBT

GBT data are read into an AIPS++ MeasurementSet using the gbtmsfiller `tool`.

```
filler := gbtmsfiller()
filler.setproject (/home/gbtdata/pnt_prime_13')
filler.fillall()
filler.done()
```

This will create a MeasurementSet called *pnt_prime_13_DCR* in the current directory.

This can also be done from the command line before entering the AIPS++ shell.

```
> gbtmsfiller project=/home/gbtdata/zeroSpacing01 minscan=101 maxscan=301
msrootname='orion_' fillrawpointing=True
```

This will create a MeasurementSet called orion_DCR in the current directory (it appends the name of the backend, in this case the GBT DCR).

5.2.2 Loading the continuum calibration utilities

The continuum calibration imaging utilities are currently packaged under a single script which can be optionally added to an aips++ session. This script is called 'gbtcalutils.g'. It can be loaded upon startup as:

```
> aips++ -l gbtcalutils.g
```

For a minimal startup (only opens the necessary AIPS++ utilities required for continuum processing), one can type:

```
> glish -l gbtcalutils.g
```

Or it can be loaded to an existing session of AIPS++:

```
- include 'gbtcalutils.g'
```

This script makes a tool, gc (Gbt Continuum imaging), which has several functions appropriate for calibrating and imaging GBT continuum data; the tool assumes that you already have available a MeasurementSet (MS) of the observed map.

5.3 Data examination and inspection

Interactive editing is possible on almost all graphical views of the data which are possible in msplot. To enable the editing option, it is necessary to set parameter `edit=T` when the msplot `tool` is created.

```
include 'msplot.g'
msp := msplot('pnt_prime_13_DCR', edit=T)
```

The GUI panel controlling flagging in msplot allows global flag masks to be set, such as that for all polarizations if any one is bad, as well as access to the list of accumulated flags, reversal of flags already applied, and other related utility operations. Raster displays of data are possible also, and are recommended for spectral-line data editing.

You may also view and edit the data at a lower level: that of the tables themselves. To browse the MeasurementSet table using a graphical browser, do:

```
include 'table.g'
t := table('pnt_prime_13_DCR')
t.browse()
```

Note that editing of individual values using the browser is possible (but unlikely to be needed unless there is something structurally wrong with the tables).

5.4 Basic gain calibration

As noted above, AIPS++ data are stored in a MeasurementSet (MS). The observed data are stored in the DATA (complex) or FLOAT_DATA (real) column in the MAIN table of the MS; these are the raw data as loaded from the telescope archive or from a UVFITS file. Associated with the DATA column are related columns to hold the most recent version of the calibrated data (CORRECTED_DATA), and the latest data plane representation of the source or field model (MODEL_DATA). The latter two columns are populated at the filling stage (for GBT data these are initially just copies of the raw FLOAT_DATA column) or upon invocation of the imager tool on the MS. The observed data do not change during reduction, but the related columns do, and should be viewed in this context. When plotting the data using a `tool` such as msplot, the data sub-types (`observed, corrected and model`) can be selected individually.

An example data set checked into the AIPS++ system can be converted into a MS by using the following from the Unix prompt:

```
> gbtmsfiller project=/home/aips++/data/nrao/GBT/pnt_prime_13 minscan=2350 \
      maxscan=2424 msrootname='cygnus_' fillrawpointing=True
```

This creates a MS, cygnus_DCR. Another data set which has already been converted into a data set is also available: /aips++/data/demo/dishdemo/pnt_lowgreg_11_DCR. Either of these can be used in the following examples.

5.4.1 gc.setdata(msname)

setdata takes a single argument, the MeasurementSet name, and examines it for suitability (e.g., is it a GBT continuum map) and content, i.e., does it contain all of the requisite information to reduce automatically. In particular, it explores the data set to confirm that information on the polarization, bandwidth, and frequency are all present. In addition, it also obtains the extent of the map and calculates the appropriate cell size for gridding (hardwired to $\lambda/2D$).

5.4.2 gc.contcal(tcal=1.0,average=T,baseline=F,nfit=0,range=20)

contcal performs the basic calibration on the data. It has arguments for tcal (to over-ride the system determined value). The default is to average all polarizations (as available); this can be toggled off by setting average=F. Optionally, a baseline can be subtracted from each row of the data. The range argument determines the beginning and end percentage of each scan row that will be used for the baseline determination; the default is 20%.

The initial continuum maps obtained at the GBT are four phase data representing the states of noise-diode off and on for two separate receptors. *contcal* performs the following calculation for each OTF strip of the map:

$$T_{ANT} = T_{CAL} \times \frac{V_{ON} + V_{OFF}}{2\times < V_{ON} - V_{OFF} >} - < T_{CAL}/2. > \qquad (5.1)$$

```
V_ON are the counts for the CAL ON phase
V_OFF are the counts for the CAL OFF phase
T_CAL is the K/counts conversion
```

5.5 Imaging

Imaging is done with the Imager tool, which supports a broad range of imaging modes and deconvolution algorithms, as described in the Imaging chapter of `Getting Results` and in the `User Reference Manual`.

The Imager `tool` requires initial selection of the data and initialization of the imaging coordinates.

A single-field, continuum GBT *unnormalized* image can be made using:

5.5.1 gc.makeimage(imname='scanimage',gridfn='SF',receptor=1)

makeimage performs the steps to select, regrid and construct the image. It accepts arguments to specify the output image, the gridding function to be used and which receptor's data to image. In addition, an image of the coverage (sampling) is made which is simply the image name with a "_weight" appended to it. The coverage image is simply the gridded weights. The receptor is a numeric designation for the polarizations to which each receiver is sensitive (e.g., XX, YY, etc). If contcal was executed with average=T, then this will be an average of these polarizations; otherwise, each can be imaged separately.

The process of making a single dish image is very straightforward. The data are converted into the coordinate system of the image (set via Imager.setimage), and then added to the image using a gridding function. A variety of convolution functions can be used: BOX, SF, or PB. The first is a simple nearest neighbor gridding, the second uses a prolate spheroidal wavefunction, and the last uses a primary beam model appropriate to the telescope being used. The primary beam is optimal in the least squares sense but degrades the resolution, whereas the prolate spheroidal function can avoid the resolution degradation but the noise level will be higher.

5.5.2 gc.covercorr(imname=F,wname=F)

The image from *makeimage* is not normalized by the sampling density. *covercorr* corrects for this by correcting for the coverage image to avoid nonsampled points (hardwired to the 10% level of the coverage map). If makeimage was previously executed, the values of "imname" (the source image) and "wname" (the weights image) are set. They may also be specified explicitly. The output image will be the name from "imname" with a "_corr" concatenated onto the end.

5.5.3 gc.plotsource(imname,nsource=1)

At this point, an image has been made which can be viewed simply by:

```
- im:=image('scanimage_corr')
- im.view();
```

plotsource also enables viewing of the contstructed images. It takes as an argument the name of the image to view and an optional parameter for the expected number of sources. The image module supports a findsources utility which identifies the positions of strong point sources in an image; it labels the position with 'Source x', where x is the number of the source found.

At this stage, use the viewer to manipulate the image in any way desired. The 'Adjust' button on the lower left provides the controls to the raster, contour and skycatalog information.

5.6 Example

The following is an example of how to calibrate and image a GBT continuum data set called pnt_lowgreg_11_DCR (contained in the standard AIPS++ data repository); the data is an OTF image obtained with the GBT DCR (digital continuum receiver). These functions are illustrated in the following example; note, for this example the logger is closed so all output is sent to the screen.

```
> cp -r /home/aips++/data/demo/dishdemo/pnt_lowgreg_11_DCR .
> glish -l gbtcalutils.g
NORMAL: gbtcalutils (gc) is ready for use
- field_names(gc)              # look at the available tools in gc
                               # gc = GBT continuum imaging
checkms setdata contcal makeimage covercorr plotsource debug
- gc.setdata('cassaDCR');
successful readonly open of default-locked table cassaDCR/NRAO_GBT_GLISH: 21 col
umns, 120 rows
successful readonly open of default-locked table cassaDCR/NRAO_GBT_IF: 22 column
s, 240 rows
successful read/write open of default-locked table cassaDCR/FEED: 12 columns, 2
rows
successful read/write open of default-locked table cassaDCR/POLARIZATION: 4 colu
mns, 2 rows
successful read/write open of default-locked table cassaDCR/SPECTRAL_WINDOW: 14
columns, 1 rows
successful readonly open of default-locked table cassaDCR/POINTING: 15 columns,
48000 rows
```

```
T
- gc.contcal()
successful read/write open of default-locked table cassaDCR: 28 columns, 192000
rows
T
- gc.makeimage()
successful readonly open of default-locked table cassaDCR: 28 columns, 192000 ro
ws
Starting server imager
Server started: /export/home/guthwine/aips++/linux_gnu/bin/imager (AIPS++ versio
n: 1.7 (build #405))
Opening MeasurementSet cassaDCR
defaultcatalog (dc) ready
Starting imager::setdata
Selecting data
Performing selection on MeasurementSet
Selecting on field and spectral window ids
By selection 192000 rows are reduced to 48000
Finished imager::setdata
       0.96 real        0.69 user        0.19 system
successful readonly open of default-locked table cassaDCR/NRAO_GBT_GLISH: 21 col
umns, 120 rows
Cell size is  0.289495253arcmin
Map size is  117.705376  by  59.9669179
Grid sizes are  406   208
Defining image properties
nx = 406 is not composite; nx = 432 or 400 will be more efficient
ny = 208 is not composite; ny = 216 or 200 will be more efficient
Setting processing options
Starting imager::weight
Weighting MS: IMAGING_WEIGHT column will be changed
Natural weighting
Sum of weights = 48000
Finished imager::weight
       32.6 real       23.58 user        6.11 system
Starting imager::makeimage
Calculating image (without full skyequation)
Making single dish image from corrected data
Image is : scanimage
Frequency = 8.9 GHz, synthesized continuum bandwidth = 0.08 GHz
Image polarization = Stokes I
Frequency = 8.9 GHz, synthesized continuum bandwidth = 0.08 GHz
Image polarization = Stokes I
Performing Single Dish gridding with convolution function SF
Gridding will use specified common tangent point:
     23:21:00.00         +58.32.00.00             B1950
```

```
Using prolate spheriodal wave function as the convolution function
Finished imager::makeimage
      10.25 real        6.77 user        0.49 system
Starting imager::makeimage
Calculating image (without full skyequation)
Making single dish coverage function
Image is : scanimage_weight
Frequency = 8.9 GHz, synthesized continuum bandwidth = 0.08 GHz
Image polarization = Stokes I
Frequency = 8.9 GHz, synthesized continuum bandwidth = 0.08 GHz
Image polarization = Stokes I
Using prolate spheriodal wave function as the convolution function
Finished imager::makeimage
       8.12 real        6.58 user        0.52 system
Successfully closed empty server: imager
T
- gc.covercorr()
Starting server app_image
Server started: /export/home/guthwine/aips++/linux_gnu/bin/app_image (AIPS++ ver
sion: 1.7 (build #405))
Selected bounding box :
    [1, 1, 1, 1] to [406, 208, 1, 1]  (23:28:23.897, +58.01.04.977, I, 8.900000e
+09Hz to 23:13:25.534, +59.00.59.643, I, 8.900000e+09Hz)
Creating new statistics storage lattice of shape [9]

Number points =   8.444800e+04       Sum      =   8.444801e+04
Mean          =   1.000000e+00
Variance      =   1.026810e+00       Std dev   =   1.013316e+00
Rms           =   1.423657e+00

Minimum value   0.000000e+00 at [1, 1, 1, 1] (23:28:23.897, +58.01.04.977, I, 8.
900000e+09Hz)
Maximum value   4.499732e+00 at [105, 182, 1, 1] (23:24:41.961, +58.54.05.639, I
, 8.900000e+09Hz)

Creating image `scanimage_corr' of shape [406, 208, 1, 1]
Created mask `mask0'
T
- #Scale images to Jy (e.g., 1.76 K/Jy)
- im:=imagecalc(outfile='scanimage_tmb',pixels='(scanimage_corr)/1.76');
- im.view(); #note the higher amplitudes
- #You can also modify/add header information to the image
- #See the images module for more details; below is an example of adding
- #information on the restoring beam
- rb:=[=];
- rb.major.value:=87.6;
```

```
- rb.major.unit :='arcsec';
- rb.minor.value:=87.6;
- rb.minor.unit :='arcsec';
- rb.positionangle.value:=0;
- rb.positionangle.unit :='deg'
- im.setrestoringbeam(beam=rb);
- im.restoringbeam(); #see restoring beam values
- im.setbrightnessunit('Jy/beam');
- im.brightnessunit(); #see brightness unit value
```

Chapter 6

Single Dish Analysis

R.W. Garwood & J.P. McMullin

6.1 Introduction

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

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

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

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

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

DISH may be accessed in several ways:

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

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

   execute:

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

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

    (a) Select on Packages, dish.

    (b) Select on Modules, dish.

    (c) Select on Tools, dish.

    (d) Press create.

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

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

    (a) include 'dish.g'

    (b) d:=dish();

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

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

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

6.2 Getting Data Into Dish

There are several ways to get data into DISH:

6.2.1 MeasurementSets

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

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

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

6.2.2 Facility filler (e.g., GBT)

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

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

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

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

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

1. Convert to MS

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

   ```
   sdfits2ms sdfits_name ms_name, for example:

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

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

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

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

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

6.2.4 UniPOPS SDD data file

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

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

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

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

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

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

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

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

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

Previewing the UniPOPS file.

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

Converting the data to FITS.

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

```
FITS writing complete.
```

2. Convert UniPOPS FITS to standard SDFITS.

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

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

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

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

6.2.5 Dish demo data

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

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

6.3 The Dish Graphical User Interface

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

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

The following sections detail the major graphical components of DISH.

6.3.1 Results Manager

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

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

Working sets, SDRECORDs, and everything else

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

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

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

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

6.3.2 Menubar and Message Line

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

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

6.3.3 Browsing

A working set within the results manager can be 'browsed' or examined by hitting the 'Browse' button. This brings up a frame which lists all of the scan numbers and objects, record by record, within the working set. Each SDRECORD can also be browsed to examine all of the contents (e.g., all of the header information and data).

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

6.3.4 Inspecting

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

6.3.5 Operations

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

Averaging

The Averaging operation offers several options for performing the averaging (Figure 6.1). Alignment which may be either None, By Velocity, or By X-Axis, 2) Rest Frequency, which allows shifting to match the first in the average (for example, for averaging spectra from transients), 3) Weighting which may be None, Tsys and Time or by RMS, and 4) which will use any selections chosen from the Selection Operation prior to averaging.

Baselines

The Baseline operation allows either a polynomial fit or a sinusoid fit. The baseline regions may be typed into the regions entry box at the bottom of the frame, or selected via cursor (the Plot Ranges and Cursor Active buttons should both be checked). The baseline operation frame can be seen in Figure 6.2.

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

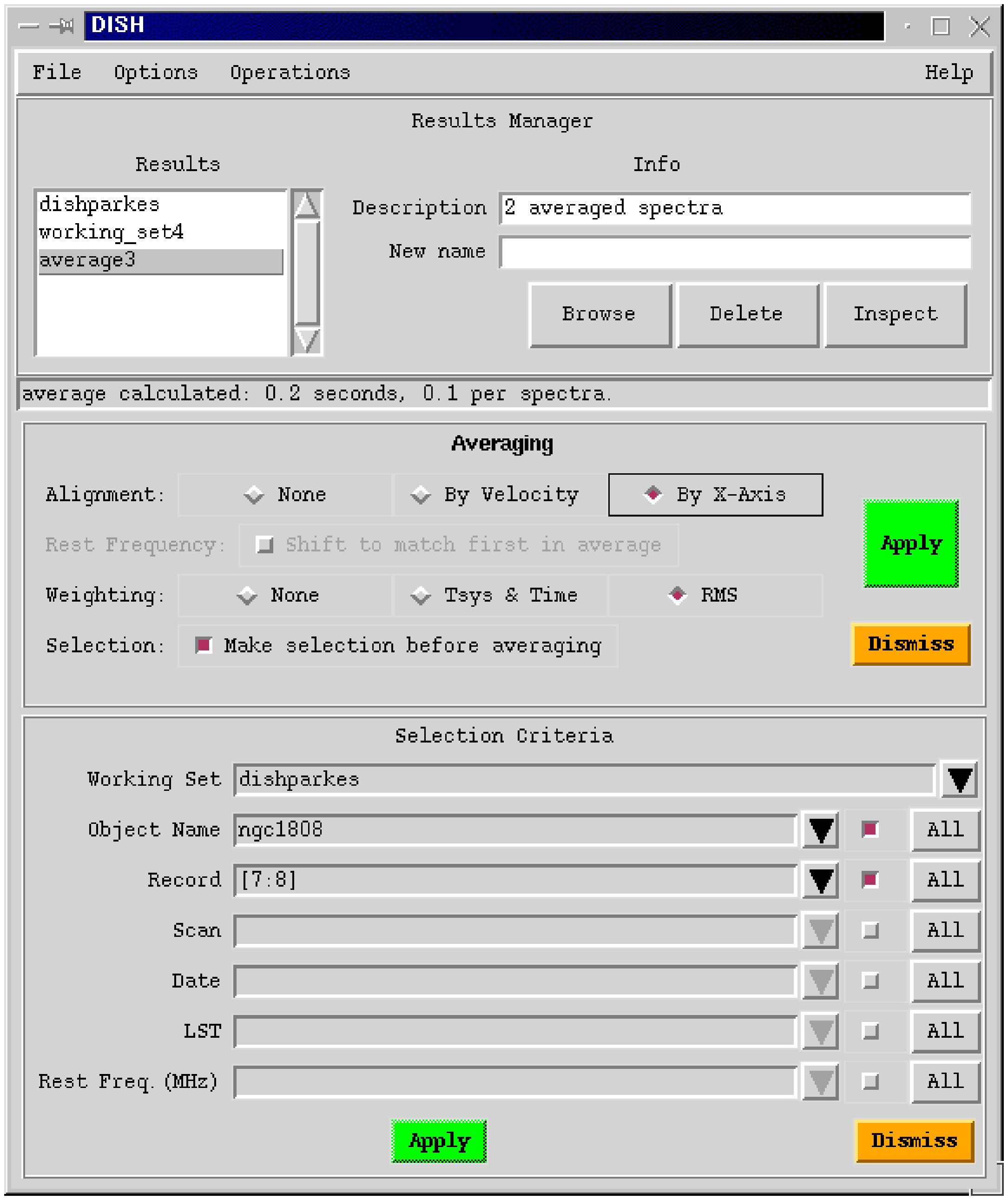

Figure 6.1: Example window showing basic Dish frame with the Averaging and Selection Operation frames open.

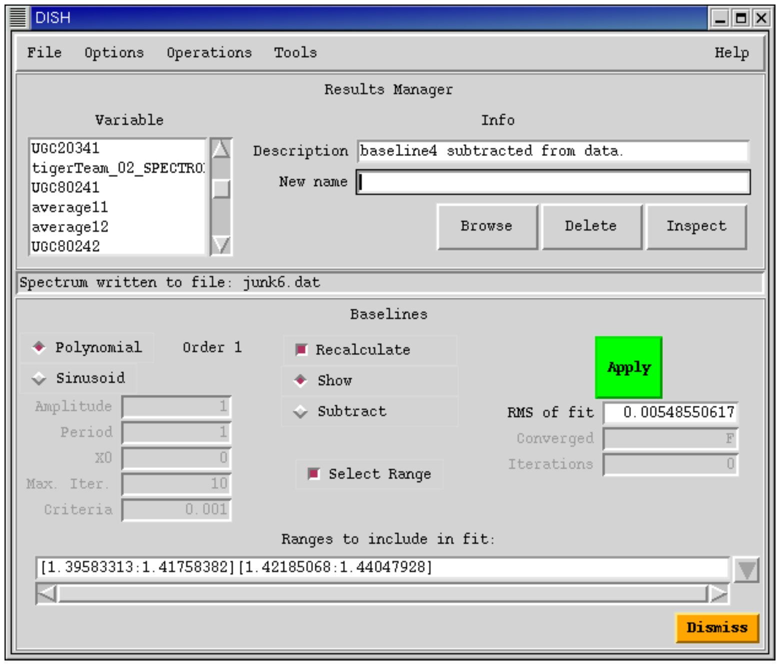

Figure 6.2: Example window showing basic Dish frame with the Baseline Operation frame open. In this example, one baseline region is selected and a 5th order polynomial will be shown overlayed on top of the spectrum.

Calculator

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

The Copy/Paste facility is activated by the right mouse button. A popup menu appears with a two copy options and a paste option. "Copy to clibpoard" will copy whatever is selected in the Calculator stack listbox to the Clipboard; the Clipboard is simply a convenient virtual storage area. "Copy to results manager" copy whatever is selected in the Calculator stack listbox to the results manager. "Paste from clipboard" retrieves whatever was most recently sent to the clipboard. For example, in Figure 6.3, the Results Manager spectrum "average3" was copied into the Calculator stack. An "_1" designation was added to the label to distinguish it from the same value in the Results Manager. The spectrum was then multiplied by a factor, producing "average3_1_2". The additional id tag is an underscore and the stack number.

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

Function on Data

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

DATA sdrecname.data, the data record of sdrecname.

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

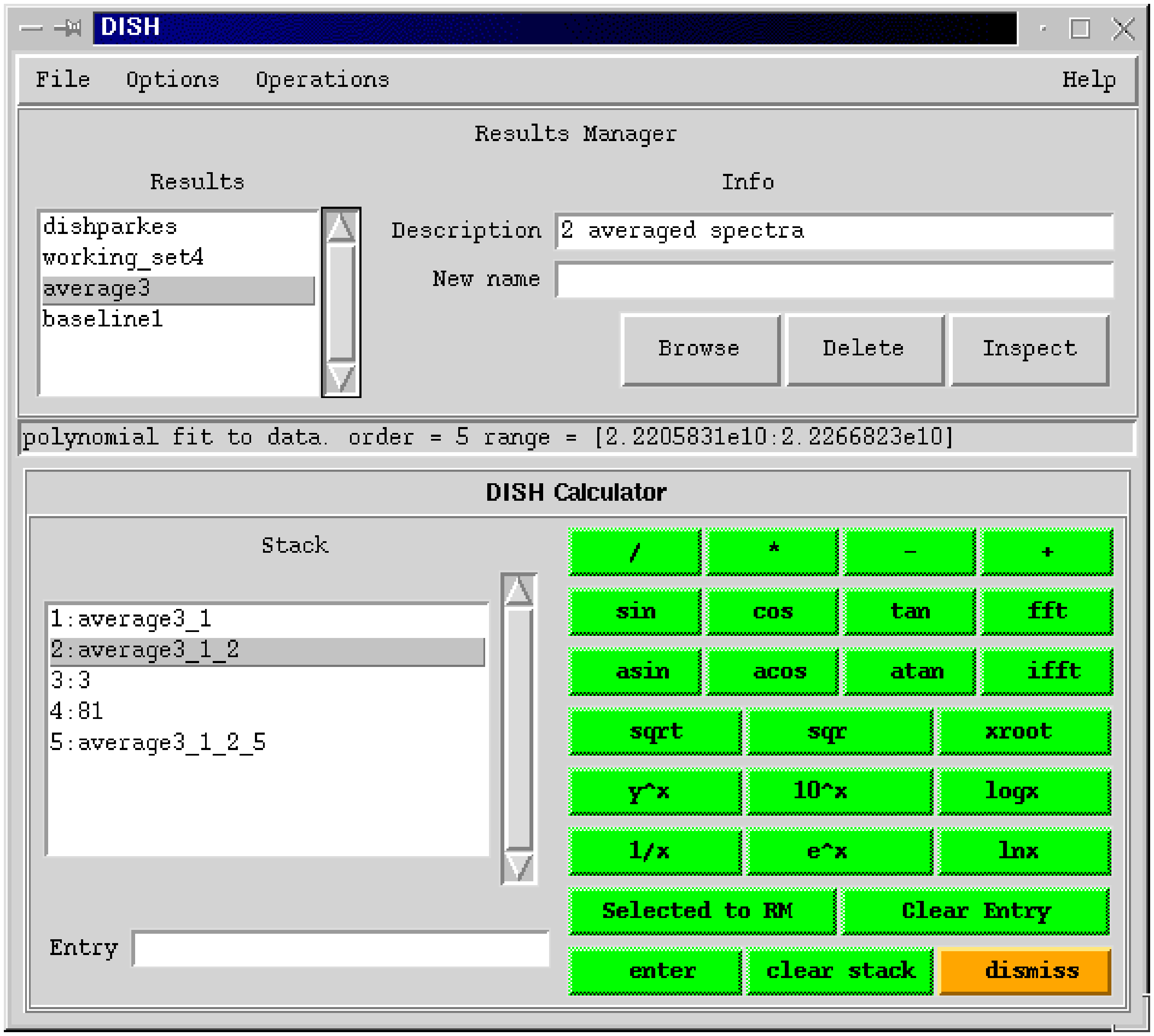

Figure 6.3: Example window showing basic Dish frame with the Calculator Operation frame open

ARR sdrecname.data.arr, the data array.

HEADER sdrecname.header, the header record.

NS_HEADER sdrecname.ns_header, the ns_header record.

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

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

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

G Fit

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

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

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

Re-gridding

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

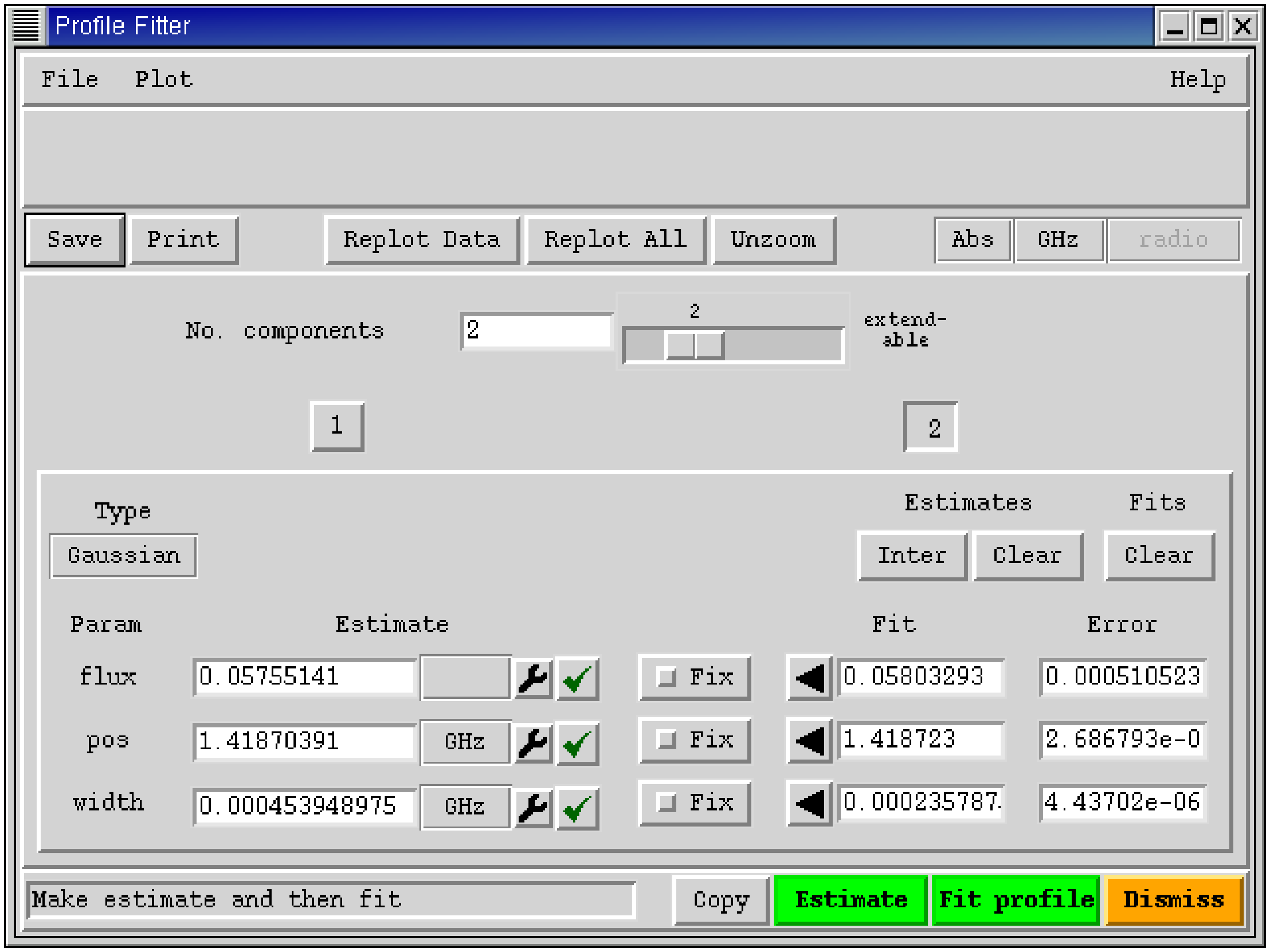

Figure 6.4: Example window showing the GUI gaussian fitting interface

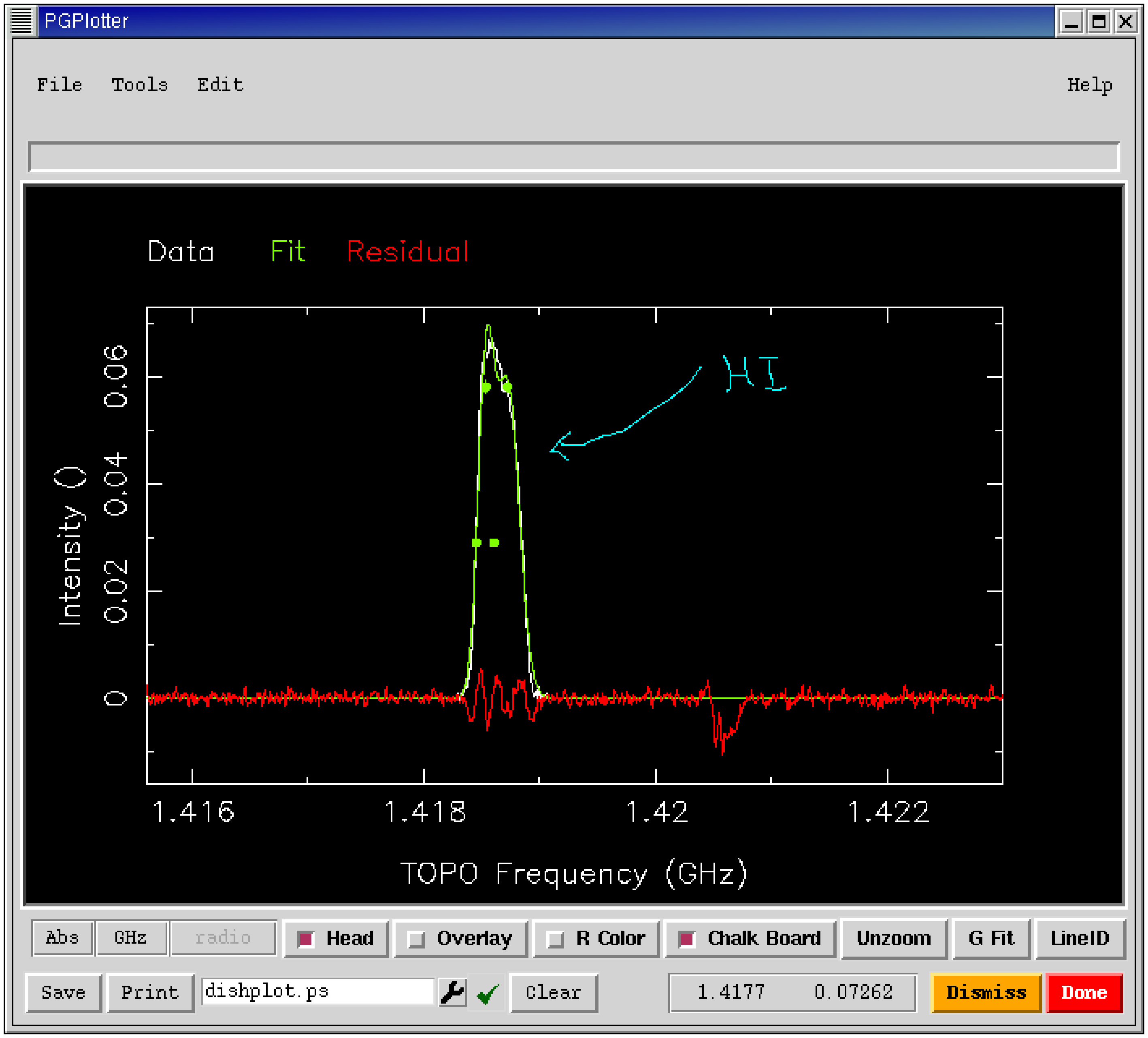

Figure 6.5: Plot showing the results of a 2-component gaussian fit to line profile. The fit and residuals only are displayed. The chalkboard feature has been used to draw a label indicating the line ("HI"). fitting interface

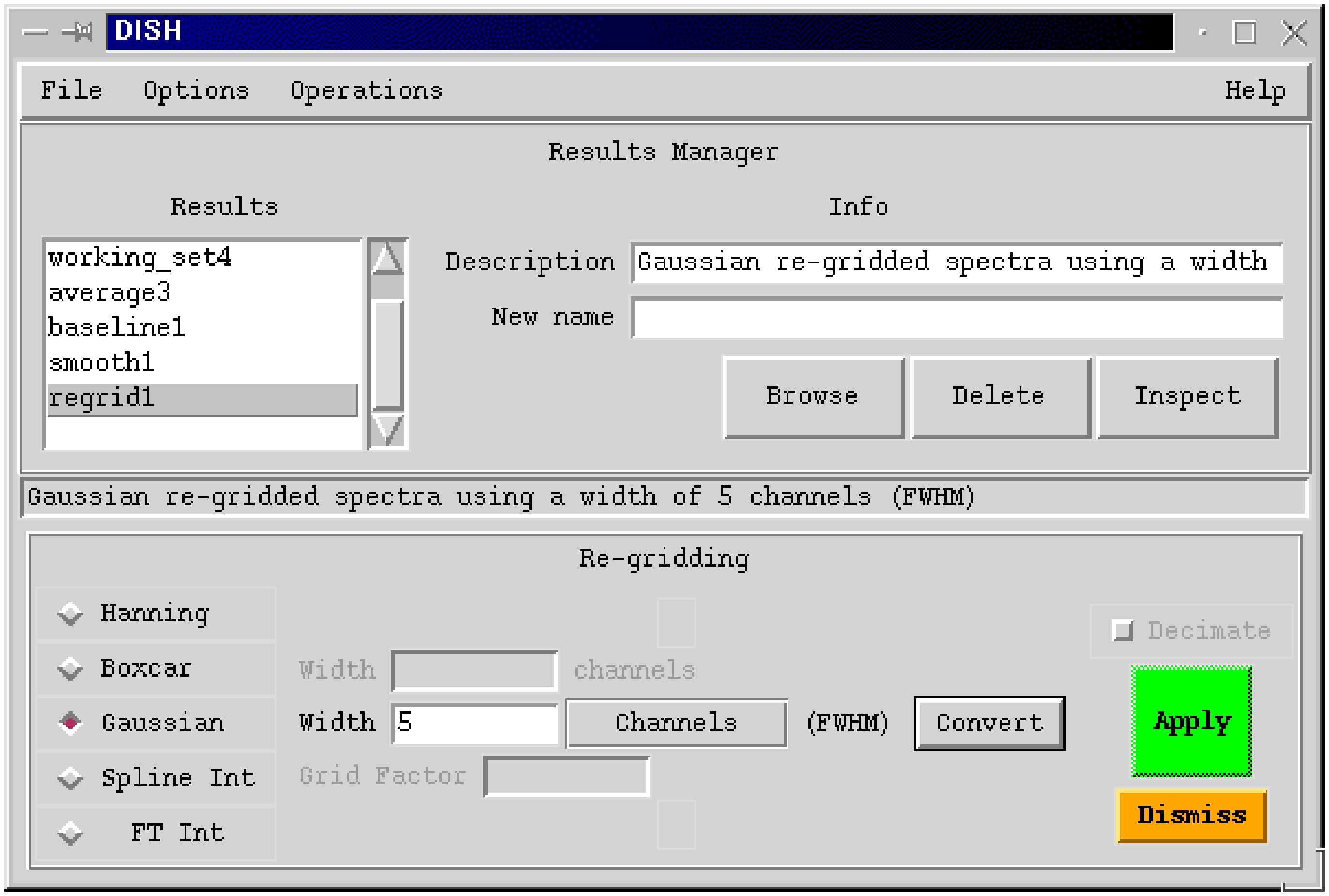

Figure 6.6: Example window showing basic Dish frame with the Regrid Operation frame open

Saving

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

Selection

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

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

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

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

Smoothing

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

Statistics

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

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

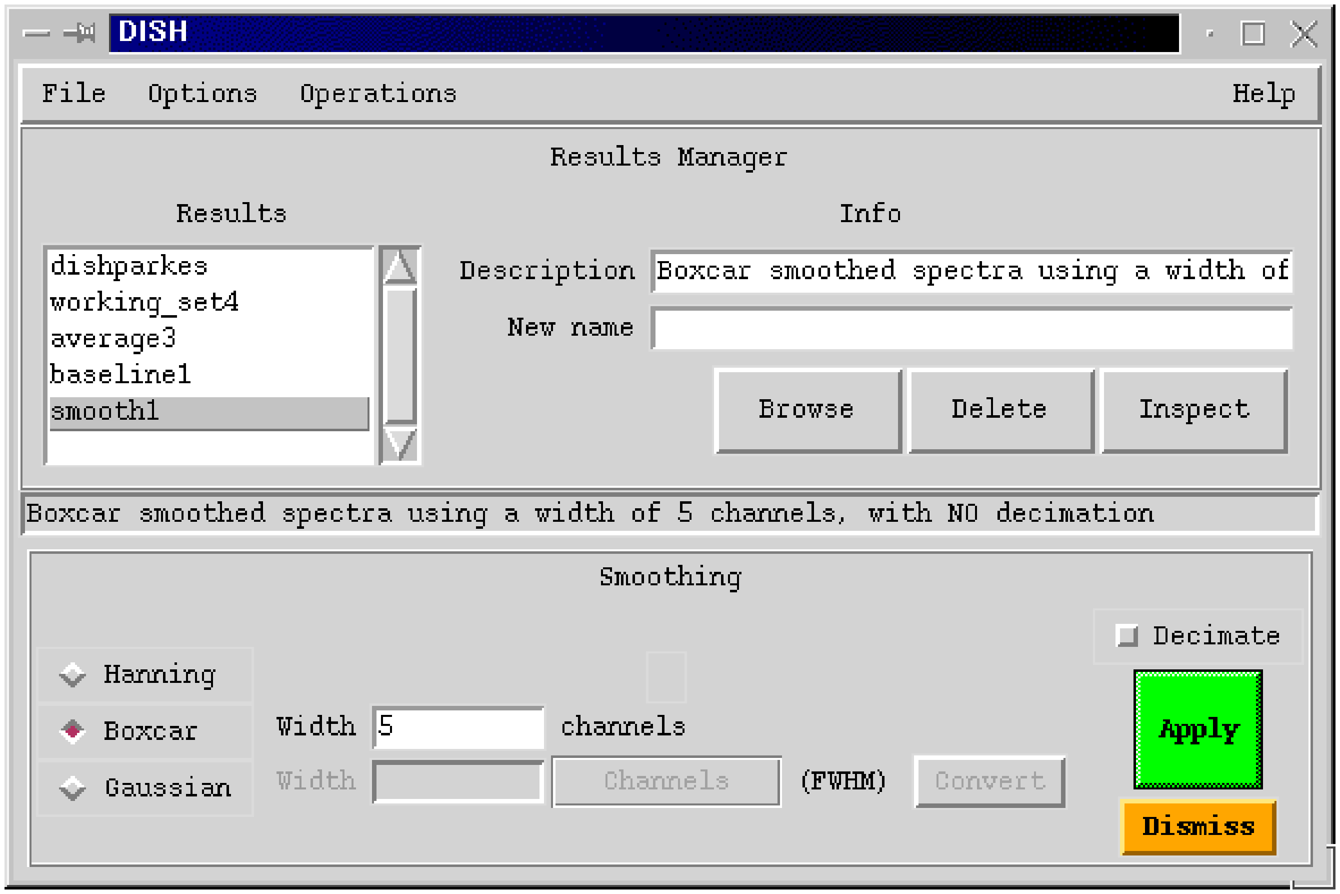

Figure 6.7: Example window showing basic Dish frame with the Smoothing Operation frame open

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

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

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

Write to File

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

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

6.4 Saving/Restoring State

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

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

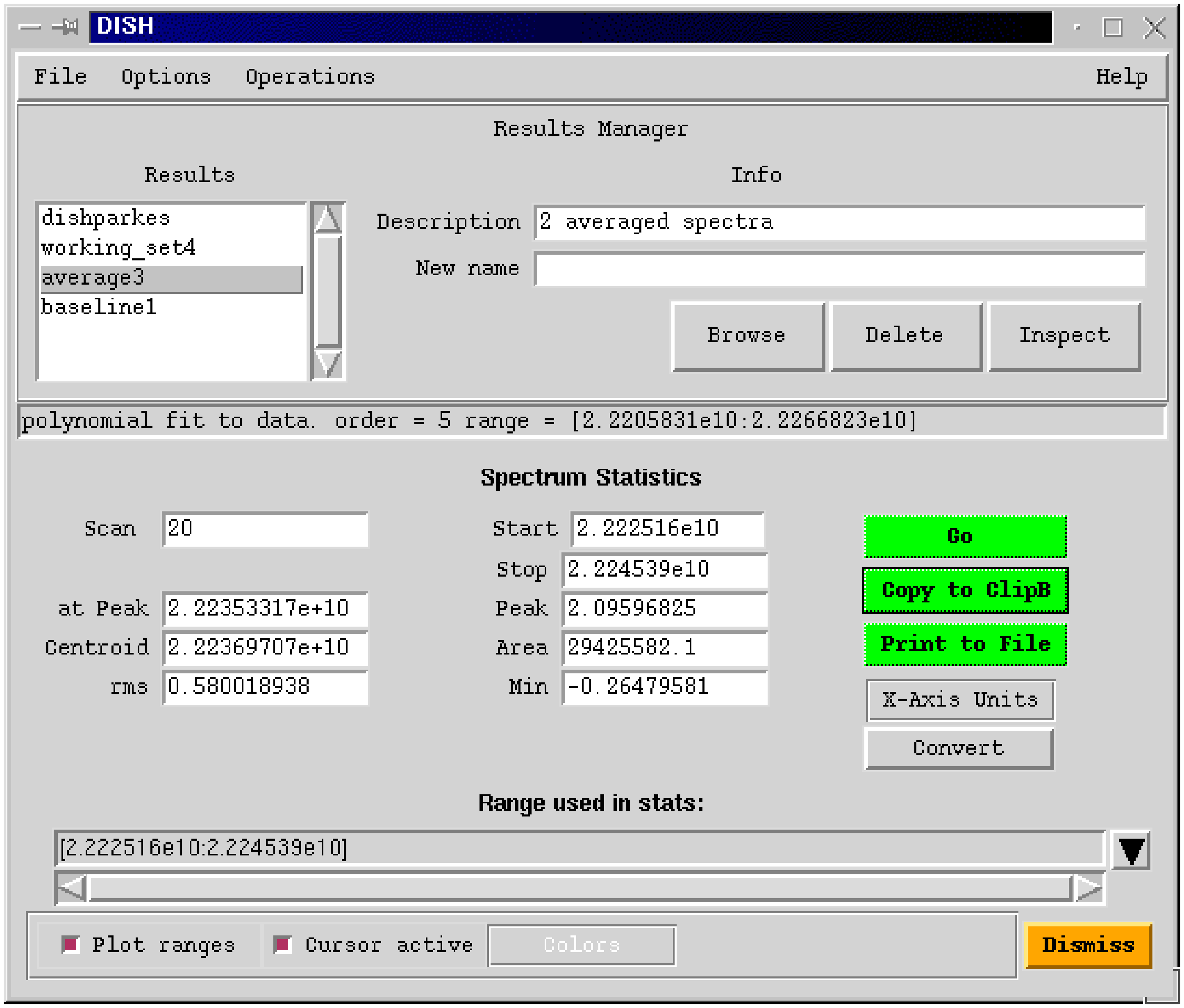

Figure 6.8: Example window showing basic Dish frame with the interval Statistics Operation frame open

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

6.5 The Dish Plotter

The Dish Plotter is the familiar AIPS++ pgplotter tool, modified for additional utilities and capabilities. Any activity or task within Dish which produces a plottable result, will automatically have the result displayed in the Dish Plotter; for example, selecting entries within the Results Manger, the SDRecord Browser, or Calculator Operation stack will plot the selection chosen. The plotter features tools for 1) changing the X-axis units, 2) changing the reference frame for the X-axis, 3) toggling header information display, 4) toggling the overlay of subsequent plots, 5) options for a) altering the colors of the plot (reverse color) and b) allowing a draw facility (chalkboard), 6) toggling the X-axis autoscale, 7) unzoom (incrementally unzooms any zooms), 7) activate the gaussian fit tool, and 8) line identification (marks any molecular lines within the frequency ranges of the displayed plot. The cursor position is continuously displayed in the grooved box at the bottom of the plot whenever the cursor is within the DISH plotter frame. The plotter is arrayed as shown in Figure 6.9.

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

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

Edit Menu This brings up a frame which allows interactive editing (changing of existing commands) or addition of new commands. For example, additional labeling or fiducial lines may be easily added. The frame lists all of the available PGPLOT commands. Selecting one brings up a brief explanation along with the various arguments required. The frame also lists the PGPLOT commands which produced the displayed frame. Again selecting a command brings up a small explanation along with the arguments used in its execution. Cut, Copy, and Paste features are available to duplicate and add additional commands. An example frame is shown in Figure 6.10.

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

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

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

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

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

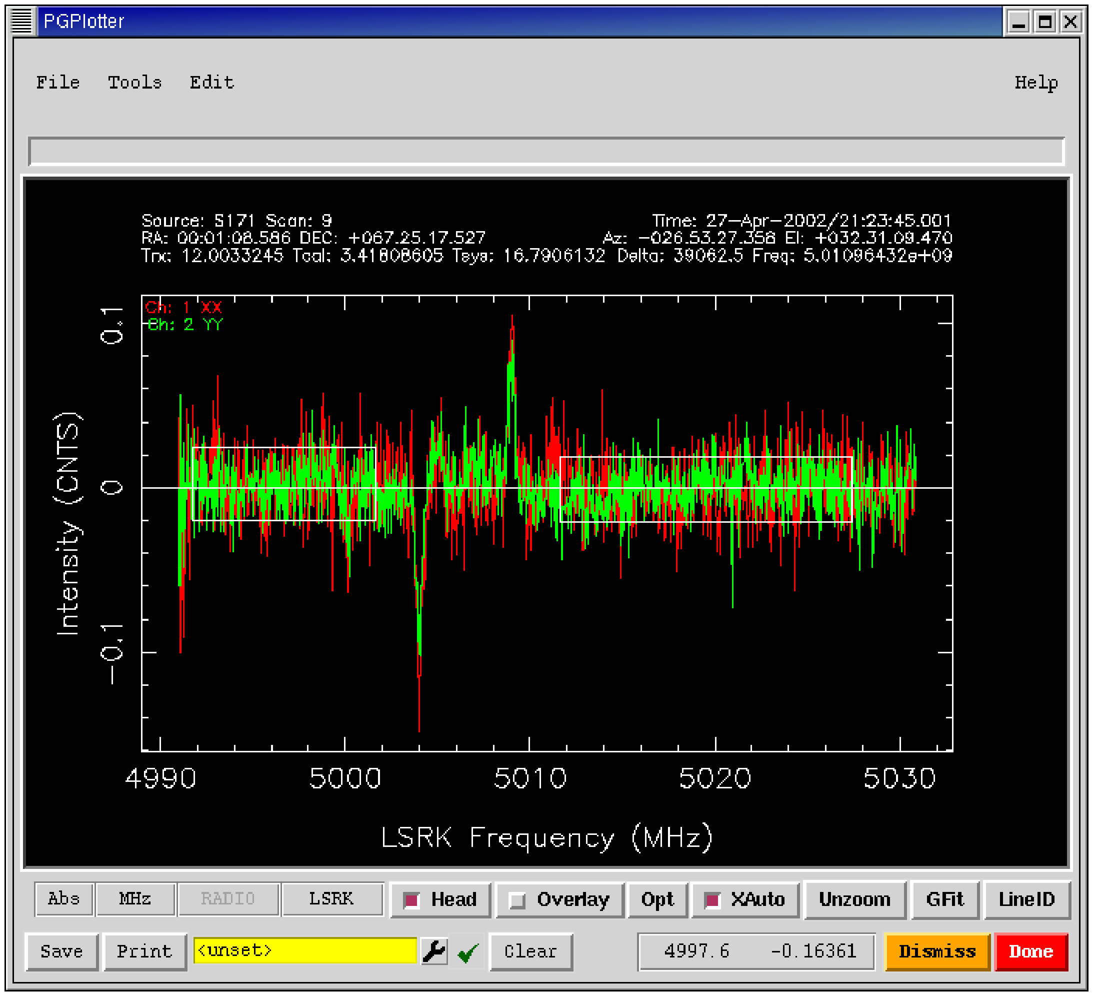

Figure 6.9: Dish Plotter Appearance. Displayed is a baselined GBT frequency switched spectrum. The baseline regions are marked by the white boxes whose height indicates the rms in that region. The different lines are different polarizations as labelled in the upper left.

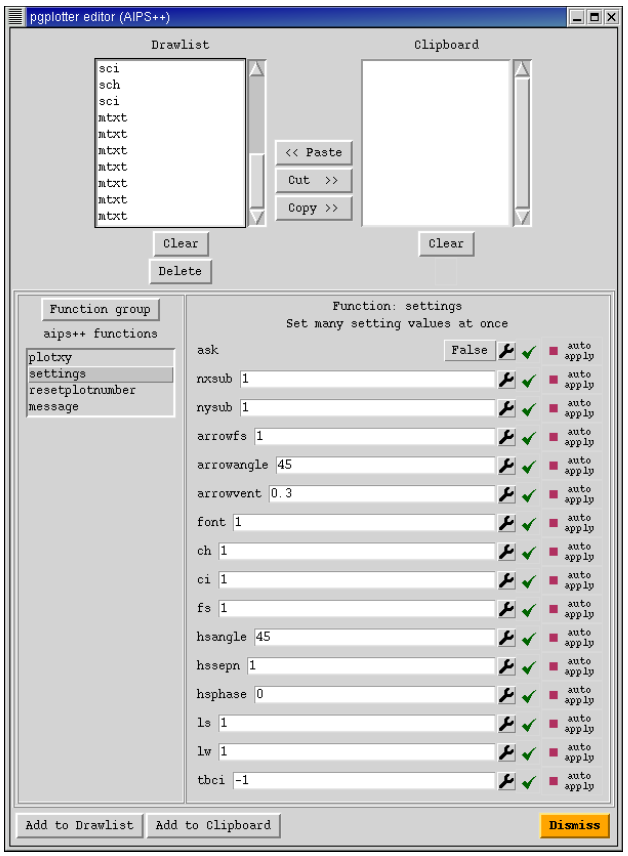

Figure 6.10: Add commands frame for Dish Plotter.

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

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

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

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

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

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

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

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

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

6.6 The Dish Command Line Interface

The DISH CLI is detailed in the User Reference Manual.

6.7 Recipes

6.7.1 Recipe 1: Change syntax to suit

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

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

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

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

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

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

```
return T;
}
```

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

Running this operation within dish looks like the following:

```
- field_names(d)
done dismiss gui rm ops addop normalcursor busycursor savestate restorestate debug open showscript message plotter gaufit myfunction
- d.square(4)
the square of 4 is: 16
T
```

6.8 sditerator

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

6.9 SDFITS

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

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

6.9.1 EXTNAME keyword

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

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

6.9.2 Virtual columns

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

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

DATAMAX and DATAMIN are special keywords. When used as keywords it is most likely that the writer of that table meant them to reflect the range of all of the data found in that table and not to be used on a row-by-row sense. If the writer feels a need to supply row-by-row values for these two keywords, they should explicitly make them true columns.

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

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

6.9.3 The DATA column and the DATA axes

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

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

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

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

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

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

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

CTYPEn The type of physical coordinate on axis n.

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

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

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

CROTAn The axis rotation

CUNITn The unit of coordinate values along this axis.

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

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

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

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

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

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

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

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

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

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

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

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

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

'STOKES' The Stokes parameter of the data.

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

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

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

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

6.9.4 CORE keywords and columns

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

OBJECT A string value giving an object name.

TELESCOP A string value giving the telescope name.

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

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

EXPOSURE The effective integration time in seconds.

TSYS The system temperature in Kelvin.

6.9.5 SHARED keywords and columns

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

OBSERVER A string giving the observer's name.

OBSID A string describing the observation.

PROJID A string describing the project.

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

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

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

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

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

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

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

TCAL The calibration temp (K).

THOT The hot load temp (K).

TCOLD The cold load temp (K).

TRX The receiver temp (K).

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

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

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

RADI radio

OPTI optical

RELA relativistic

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

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

RVSYS The radial velocity, $V_source - V_telescope$.

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

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

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

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

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

TAU The opacity at OBSFREQ.

TAUIMAGE The opacity at IMAGFREQ.

TAUZENIT The opacity per unit air mass.

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

TAMBIENT The ambient temp (K).

PRESSURE The atmospheric pressure (mm Hg).

DEWPOINT The dew point (K).

WINDSPEE The wind speed (m/s).

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

BEAMEFF The main-beam efficiency.

APEREFF The antenna aperture efficiency.

ETAL The rear spillover and scattering efficiency.

ETAFSS The forward spillover and scattering efficiency.

ANTGAIN K per Jy.

BMAJ The major main-beam FWHM (deg).

BMIN The minor main-beam FWHM (deg).

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

SITELONG The site longitude (deg).

SITELAT The site latitude (deg).

SITEELEV The site elevation (m).

RESTFREQ The rest frequency (Hz).

6.9.6 Other columns

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

6.9.7 Multiple SDFITS tables in a single file

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

Copyright

The AIPS++ consortium may be reached by email at aips2-request@nrao.edu. The postal address is: AIPS++ Consortium, c/o NRAO, 520 Edgemont Rd., Charlottesville, Va. 22903-2475 USA.