Single Dish Calibration and Imaging (SDCI)

R.W. Garwood

January 16, 1995

Contents

1 Introduction

[This is a snapshot of the SDCI effort as of the AIPS++ review. It represents the state of affairs as of about December 2, 1994. There has beem some clean-up of the document that was available at the time of the review. In addition, the SDCI demonstration that was part of the review is now summarized as part of this document.]

This document describes the initial implementation of the basic components of single dish calibration and imaging in AIPS++ . This initial implementation was guided by the immediate goal of calibrating and imaging a very specific data set (on-the-fly spectral-line data from the NRAO 12-m). The lessons learned during this phase are being applied toward the design of more general single dish calibration and imaging classes and applications. They will also strongly influence the UV calibration and imaging design (especially with regard to the `MeasurementSet` that all forms of data in AIPS++ will share). Future development is outlined in the final section of this document.

1.1 Background

The AIPS++ project has spent a long time struggling with the concepts of `MeasurementSet`, `TelescopeModel`, and `MeasurementModel` with little to show for our efforts. These abstract concepts were first described at a meeting in Green Bank, West Virginia in early 1992. They are described in more detail in the AIPS++ design overview (Glendenning).

This paragraph roughly summarizes the Green Bank model. The `MeasurementSet` is the class that is intended to contain astronomical data, it is little more than a simple container. The `TelescopeModel` consists of a number of `TelescopeComponents`. Each `TelescopeComponent` has an `apply`, `corrupt` and `solve` member functions. `apply` applies some calibration system to a `MeasurementSet`. `corrupt` is the opposite of `apply`. `solve` uses a model of the sky along with a `MeasurementSet` to solve for appropriate calibration parameters (which could then be "applied" to the data through `apply`). The job of the `TelescopeModel` is to coordinate the various `TelescopeComponents` that are present in the model. A `MeasurementModel` has an `invert` and a `predict` member function. `invert` "inverts" the data to produces a model of the sky. `predict` is the opposite of `invert`.

The lack of a concrete `MeasurementSet` has been a particularly large obstacle to forward progress. Without any place to put real astronomical data (the `MeasurementSet`) it has been difficult to make any real progress on a number of other fronts.

Faced with this observation, it was decided to adopt a well defined immediate goal in order to obtain concrete working examples of the `MeasurementSet` and `TelescopeModel`. The goal has been to calibrate and image a spectral-line on-the-fly data set from the NRAO 12-m telescope.

1.1.1 NRAO 12-m On-The-Fly (OTF) Observing.

OTF observing at the 12-m consists of moving the telescope while recording data as fast as the hardware and telescope and instrument control software will allow (currently this is 1/10 second). The instantaneous telescope position is recorded for each data sample. The speed of the telescope along the sky is such that the telescope beam is over-sampled in the direction of telescope motion (typically more than 10 samples per telescope beam width). A data set consists of several horizontal OTF rows with the spacing between rows being close to the Nyquist size appropriate for the telescope beam. Although the pattern scanned on the sky is horizontal rows, various effects (wind, variations in drive speed during the scan, etc) mean that the sky has not been regularly sampled. The motivation for the OTF technique is to account for these variations in tracking during an observation as well as to map the region as fast as possible to minimize receiver drifts so that the spectral baselines are more consistent. OTF observing can be done in continuum and spectral-line mode. For the rest of this document, OTF refers to spectral-line data.

The OTF data samples are total-power data. Calibration is accomplished by observing appropriate OFF (emission/absorption free) regions. This is usually done at the end of each OTF row. It may alternatively be possible to use the emission free regions in the area being mapped during the calibration phase. However, this requires that the calibration occur after the data has been imaged onto a regular grid.

The end result of a typical OTF spectral line observation is a data set with a number of rows consisting of several hundred individual spectra and the telescope position when each spectrum was taken. The rows may be interspersed with calibration (OFF) scans.

OTF Calibration Needs Calibration of the data within AIPS++ is done using a `TelescopeModel` consisting of any number of `TelescopeComponents` each having their own `apply()` method. The OTF observing mode at the 12-m consists of a number of samples from the region of interest (ON) and some samples from regions that the observer hopes are free of astronomical signal (OFF). One hopes that the non-astronomical signal does not vary between any of these regions and that the difference signal will contain the astronomical signal. Constructing this difference signal is a typical calibration at any single dish telescope and is not specific to OTF data. It is also necessary to correct the difference signal for the sensitivity or gain of the telescope. For position switched data this is generally done at the same time as the construction of the difference signal. At the 12-m, gains are recorded for each spectral channel while at the NRAO 140' telescope, a single value is assumed to apply to the entire spectra. This means that position switched calibration is a telescope dependent operation.

Although not yet implemented as an observing mode at the 12-m it should be possible to use the emission free regions in the OTF mapped area to construct appropriate OFFs when calibrating an OTF image (*i.e.*,it would not be necessary to make specific calibration observations, they would be constructed from the known emission-free regions of the OTF data). This calibration can occur after the data has been gridded onto a cube. This will reduce the observing time by eliminating the need for any specific off-source observations. It will also reduce the time required to produce a calibrated image by reducing the number of subtraction and multiplication operations. For large spectral-line data sets, that may be extremely important and may be required in order to keep up with the data rate.

OTF Imaging Needs The irregularly spaced, over sampled OTF data need to be placed onto a regular grid with a cell size appropriate to the telescopes beam width. This is the imaging process required for OTF data. It is quite similar to the gridding of UV data from a synthesis array telescope.

OTF Data Rates The calibration and imaging of single dish data must keep up with the data rate. Observers at single dish telescopes generally plan observations over short terms based on the most recent observations. It is vital to be able to asses the quality of the data as soon as possible. A small 0.25x0.25 degree field mapped using the OTF observing mode consists of data from roughly 14,000 points on the sky for each feed. Currently there is only one feed but there will shortly be an 8 feed receiver. There are at most 1536 spectral channels that can be split in any number of ways between polarizations and feeds. Given the current practice of 1/10 second per data point and allowing for the calibration scans (10 seconds every other row is typical) and time to start and stop the telescope motion at the end of each row. This small field takes about 40 minutes to map.

The test data set used in this initial single dish calibration and imaging effort consists of OTF data from an 0.25x0.25 degree field. There are 4 spectra at each location, 2 polarizations at each of two spectral resolutions (128 channels and 256 channels). Execution times reported in this document refer to this test data set and were done on a single processor of a Sparc-10.

1.1.2 UniPOPS

UniPOPS is the current NRAO single dish analysis package. It is primarily a vector calculator. For many single dish problems, this is quite adequate. It is especially suited to the needs of observers while they are at the telescope for most observing strategies. The command-line interpretor is a version of the POPS parser related to the AIPS POPS interpreter. Unlike AIPS, UniPOPS relies exclusively on verbs that manipulate several large internal arrays (there are 10 1-dimensional arrays and 4 2-dimensional arrays) as opposed to tasks that run independently of the interpreter. Users of UniPOPS, especially at the 12-m, rely heavily on POPS procedures to effectively create new verbs to do the specific tasks that their observations require.

UniPOPS is adept at analyzing 1-dimensional data. It is not adequate for serious analysis of multi-dimensional data. There is very little in the way of data-base management tools and no real ability to deal

with data cubes. With the OTF observing mode described above as well as the multi-feed receivers that will be increasingly common at single dish telescopes, UniPOPS will shortly be inadequate for many observers.

2 Initial Implementations

2.1 SDMeasurementSet

The SDMeasurementSet is our initial attempt at a MeasurementSet for single dish data. The MeasurementSet is any collection of data produced by a telescope or a model. It includes astronomical data, instrument data, and monitor data.

2.1.1 As a Table

The AIPS++ table system is clearly more than adequate as a container of data. Early designs of the MeasurementSet utilized the flexibility of the table system to it's fullest. These early designs consisted of multiple tables. Multiple tables are useful as an organizational aid: similar quantities can be grouped together. They may also be useful as a storage mechanism. Quantities that vary slowly or are constant can be grouped together to reduce the size of the data set. On the other hand, astronomical data has been traditionally viewed as coming in chunks with associated header words. This view is analogous to a row of a table, with each row being an atomic piece of the data.

The telescope-dependent nature of the MeasurementSet makes it difficult to define a general multiple table structure with enough shared parts to be useful from an object-oriented approach. Groupings of quantities are likely to be telescope dependent. Values that are constant or slowly varying at one telescope may vary rapidly at another telescope. Classes which use the MeasurementSet as well as application programs should be as telescope *independent* as possible. An application programmer should not need to know in which sub-table a quantity is found. Users should also not need to know in which sub-table a quantity is found.

The conflict between the desire to use the full flexibility of the table system for organizational needs and data storage efficiency on the one hand and the need to present the data in single "atomic" units or rows was a large factor in the slow development of a working MeasurementSet. This conflict was resolved in part with the realization that a suitable storage manager could reduce data set bloat. The current table system allows for each column to have it's own storage manager. Through careful choice of an appropriate storage manager, data set bloat can be avoided as needed for each column (there is no longer any need to group values into tables to accomplish this). Organizational needs can be dealt with through the use of reference tables. If a class or application finds it convenient to group columns X,Y, and Z together, that can be done without any need to impact the actual MeasurementSet. The MeasurementSet that we have implemented consists of a single table.

2.1.2 The SDMeasurementSet Table Layout

This section describes the specific layout of the table that is used in the initial SDMeasurementSet. All column names are unique (they only occur once in any given MeasurementSet) and have a fixed definition. The layout described here is more homogeneous that the ultimate MeasurementSet will likely require, primarily due to the lack of a range of suitable storage managers. The planned storage managers and use of virtual column engines will allow future MeasurementSets to be extremely heterogeneous.

Units All values in any column will have the same units. The specific unit for a given column is indicated by the value of an associated string keyword named "UNIT". Some columns are defined to have a specific unit (*e.g.*,integration time is always in seconds). For those columns, no "UNIT" keyword is required (and if present, is ignored). This convention will likely change when a complete units class is available.

Sky Coordinate System The coordinate system used for celestial coordinate columns is indicated by associated keywords as described below. All values in any given column are in the same coordinate system although coordinate systems may differ between columns. Units are degrees for all angular columns (this

is consistent with the FITS usage). This leads to a more homogeneous MeasurementSet than is probably necessary (one would certainly like to mix Galactic with Equatorial observations, etc). This decision was made for simplicity with the recognition that once a complete coordinate class was available this convention would likely change.

The following keywords, when associated with a column, indicate that that column is a celestial coordinate of the indicated type.

CTYPE	a String keyword containing the first 4 characters of a valid FITS World Coordinate System (*e.g.*,"RA–" or "DEC-" for equatorial coordinates, "GLON" or "GLAT" for galactic coordinates, "ELON" and "ELAT" for ecliptic coordinates, etc).
EQUINOX	Ecliptic and equatorial coordinates require that the equinox of the coordinate system be specified (*e.g.*,1950, 2000, etc). This is stored as double precision value.

Strings In all cases in this initial `MeasurementSet`, columns of Strings (source name, observing mode, integration state, instrument name) are likely to represent choices from a small set of possible values (for example, there are a limited number of observing modes available). These values may change frequently throughout an observing session (for example, an OTF data set consists of "OTF" observing modes - the mode where the actual on-the-fly data is taken - interspersed with "TPMF" observing modes - the total power mode used when taking the calibration scans at the "OFF" positions). For storage efficiency, these columns are stored as short unsigned integers with an associated keyword containing the array of possible string values. The integer actually stored in each row of the table points at the element of the associated string array where the actual string value is found. This will ultimately be implemented more transparently using virtual columns.

Keywords The following keywords apply to the entire `MeasurementSet`.

Keyword	**Data Type**	**Description**
ReferenceDate	Double	The reference date, used to construct the full date from the UTime column. The units of this value are modified Julian date: $(JD - 2400000.5)$. **Required.**
Telescope	String	The name of the telescope. **Optional.**
Observer	String	The name of the observer. **Optional.**
Project	String	A string identifying the project. **Optional.**

The optional keywords above are likely to be columns in the future so that a `MeasurementSet` can contain data from multiple telescopes, observers and projects.

Required Columns The following columns are required to exist for the table to be considered a valid `SDMeasurementSet`. They are the primary means of sorting and selecting data.

Column Name	Data Type	Description
TimeStamp	Int	A number guaranteed to be unique for each unique time in the `MeasurementSet`. Since time is a floating point value, by guaranteeing a unique integer for each time, comparisons are simpler. Traditionally, data from a specific time is assigned a number (the scan number). While this number is usually unique during a particular observing session, it is not unique over several sessions. This column is intended to take the place of atomic identification that scan number has traditionally provided during an observing session. Valid time stamps are positive.
ScanNumber	Int	As long as on-line control systems stamp the data with a scan number, observers will continue to use such numbers. These values are likely to be unique only over single days or possibly observing sessions.
SpectralWindow	Int	An integer specifying the spectral window of the data (previously this has been known as "IF"). The current working definition is to use the "sub-scan number" used at the 140' and 12-m, but those conventions are somewhat arbitrary and telescope dependent. SpectralWindow needs a more complete definition. At a minimum, the characteristics of each SpectralWindow need to be available (possibly as associated keywords).
PolarizeType	Int	The polarization type appropriate for this row. The value of this integer is from the `StokesTypes` enumeration found in the `Stokes` class. Available enumerations appropriate for single dish data are `RCircular`, `LCircular`, and `Linear`.
Feed	Int	The feed number for multi-feed systems. Currently this is always 1 as there are no NRAO multi-feed systems presently taking data.
IntegrationState	String	A description of the integration state (known at NRAO single dish telescopes as "phase"). For example, for a simple On/Off sequence, this might be "ON-POS" or "OFF-POS". This is the last indexing value to be used when all of the above do not yield a unique row. Most on-line systems (or fillers of an `SDMeasurementSet`) will average all IntegrationStates appropriately and make this column unnecessary. This will be used under those circumstances where it is desirable to edit individual parts of an integration (known to UniPOPS users as "record editing").
Flag	Bool	A flag. "True" indicates that the data is "ok". "False" indicates that the data is "bad".
Data	Double	This is a vector containing the actual data. All Data vectors must be the same length in a single `MeasurementSet`. Data of different lengths (even from the same observing session) must reside in different `MeasurementSets`. This restriction will be lifted in the future, although specific fillers may wish follow this rule.

Optional Columns Most of the information in an `SDMeasurementSet` is considered optional. The specific `SDMeasurementSet` filler (which will be telescope dependent) will determine which additional columns are present. It is therefore extremely important that column names be registered so that applications can rely on standard definitions. The following have been tentatively identified as useful optional columns.

Column Name	Data Type	Description
UTime	Double	The universal time in number of seconds since the ReferenceDate keyword. This time corresponds to the mid-point of the integration.
Object	String	A descriptive name.
ObservingMode	String	A string describing the observing mode. This will be telescope dependent although some attempt should be made to identify common observing modes (*e.g.*,frequency switched, position switched, etc).
PolarizeAngle	Float	The polarization angle, meaningful for `Linear` PolarizeType only.
UserTag	Int	A tag set by the user to identify groups of rows.
IntegrationTime	Float	The total integration time (seconds). This is the real time spent actually collecting all of the data in this row.
EffectiveIntTime	Float	The effective integration time (seconds). This is the time that appears in the radiometer equation (*i.e.*,use this time when calculating expected noise levels). The exact relationship of this quantity to IntegrationTime depends on the ObservingMode and is likely to be telescope dependent.
HSource	Double	
VSource	Double	"Horizontal" and "Vertical" source position in the coordinate system described by the associated keywords defined above. This is where the telescope was pointed (for position switched data, this is the "Source" or "On" position).
HReference	Double	
VReference	Double	"Horizontal" and "Vertical" reference position in the coordinate system described by the associated keywords defined above. This is a reference positions (for position switched data, this is the "Off" position).
Instrument	String	A descriptive string identifying the instrument. This will likely be split into constituent parts (*e.g.*,receiver, feed, correlator, etc).
ReferenceChannel	Int	The reference channel for the Data array. This may be renamed to ReferencePixel to be more generic.
ReferenceValue	Double	The x-axis value at the ReferenceChannel.
DeltaValue	Double	The increment along the x-axis. Positive values increase with increasing channel (pixel) number.
RestFrequency	Double	The rest frequency corresponding to ReferenceChannel.
VelocityDef	String	The velocity definition ("Radio", "Optical", "Relativistic"). In the future, this may be more appropriately an Int with the values corresponding to an enumeration in much the same way that PolarizeType corresponds to enumerations found in the Stokes class.
VelocitySystem	String	The velocity reference frame ("LSR", "HELO", "EART", "BARI"). As with VelocityDef, in the future this may be an Int corresponding to an enumeration.
VelWRTReference	Double	The velocity (in the VelocityDef for that row) of the ReferenceChannel with respect to VelocitySystem.
SignalFreqOffset	Double	The frequency offset for frequency switched data for the "signal".
ReferenceFreqOffset	Double	The frequency offset for frequency switched data for the "reference". Note that this and SignalFreqOffset will not be adequate to describe more complex frequency switching observing modes.
SourceTSystem	Float	The system temperature (K) at the source position.
ReferenceTSystem	Float	The system temperature (K) at the reference position.
TReceiver	Float	The receiver temperature (K).
TCalibration	Float	The value of the noise tube diode or other calibration temperature (K) used to calibrate the data.
BeamFWHM	Float	The full telescope main beam width at half maximum at the observing frequency.
OffTimeStamp	Int	The TimeStamp corresponding to the designated "Off" or calibration data. If OffTimeStamp is equal to TimeStamp for that row, then that row is "Off" data. If OffTimeStamp is zero, there is no designated "Off" data for that row.
GainTimeStamp	Int	The TimeStamp corresponding to the designated per channel gain values (currently 12-m only). If GainTimeStamp is equal to TimeStamp for that row, then that row is Gain data. If GainTimeStamp is zero, there is no designated "Gain" data for that row.

Often a region of the sky is scanned in some regular pattern (a "map" is made) through one command of the operator or observer. This will generate several rows in the `MeasurementSet`. The original map description may be available from the on-line data. The following columns are intended to hold such information when available. These can assist in imaging the data or they can be ignored. Maps always have two axes (in these names, then, n=1 or 2).

MapNAXISn	Int	The number of map "pixels" on the nth axis.
MapCRVALn	Double	Coordinate value at the reference pixel on the nth axis.
MapCRPIXn	Double	The reference pixel on the nth axis.
MapCDELTn	Double	The coordinate increment on the nth axis.
MapCTYPEn	String	The WCS coordinate type for the map (applies to any coordinate expressed in these columns) for this particular row.
MapEQUINOX	Double	The equinox of the coordinate system.

The following describe various flags and weight options (in addition to Flag).

Sigma	Float	The theoretical RMS in the same units as the data. This value is usually provided by the operating system. It is used with weight when averaging or combining data.
Weight	Float	This is a unit-less number that is initially 1. The total effective sigma is weight*sigma. This allows the user to modify the effective sigma used when combining data without discarding the value provided by the operating system.
ChannelSigma	Float	Per-channel version of Sigma
ChannelWeight	Float	Per-channel version of Weight
ChannelFlag	Bool	Per-channel version of Flag

All of the above are vectors that must be the same length as the data vector. There must be only one of Sigma and Weight or ChannelSigma and ChannelWeight in a given `MeasurementSet`. ChannelFlag can exist in conjunction with Flag. Flag turns "on" or "off" the entire row while ChannelFlag turns "on" or "off" individual values in the data vector.

Additional columns will be registered and added as needed.

2.1.3 Implementation

`SDMeasurementSet` has been implemented as having a `Table` rather than derived from `Table`. The primary motivation for this was the design of `Table` at the time of this implementation. `Table` then consisted of `ROTable` (a read-only, or constant, table) and `Table` (a read/write table), which was derived from `ROTable`. In order to implement `SDMeasurementSet` as a `Table` following the above would require an `ROSDMeasurementSet` for read-only data which would be derived from `ROTable`. `SDMeasurementSet` would then need to be derived from `ROSDMeasurementSet` as well as from `Table`. This inheritance, with `ROTable` being in the inheritance tree twice, was felt to be a potential maintenance headache. It was decided to avoid the problem entirely by implementing `SDMeasurementSet` with a private `Table` data member. The most recent version of the `Tables` module eliminates `ROTable`. Since a `MeasurementSet` seems to clearly be a `Table`, we will use this opportunity to implement `MeasurementSet` as a `Table` in the near future.

An `SDMeasurementSet` is constructed from an existing table. During construction, the table is checked to see that the required keywords and columns as outlined above are present. A static member function, `miminumDescriptor()` provides the minimum table descriptor (*i.e.*,one that consists only of the required keywords and columns). This can be used to construct an empty table which would then be used to construct a valid `SDMeasurementSet`.

Normal table access to the columns would require that the correct spelling of the column name be used. For example, to construct an `ArrayColumn` of Float from a column named "Data" in an existing `Table` named `tab`, one would use:

```
ArrayColumn<Float> data(tab, "Data");
```

Mistakes in the string containing the column name would only be caught at run time (and then only if a column of the miss-typed name did not exist with the specific type).

In this implementation, we have provided a member function that returns either a reference or a pointer

to the appropriate `ArrayColumn` or `ScalarColumn` object for each of the required and optional column names. Pointers are returned for the optional columns while references are returned for the required columns. Here are some examples from the `SDMeasurementSet` header file:

```
// Columns that are GUARANTEED to be in all SDMeasurementSets
ScalarColumn<Int> &scanNumber();
ScalarColumn<Int> &timeStamp();
ArrayColumn<Float> &data();

// Optional columns
ScalarColumn<Double> *uTime();
ScalarColumn<Float> *integrationTime();
```

The advantage of this scheme is that mistakes in the column names are caught at compile time rather than run time. We feel that this is an important advantage that should be retained in any future `MeasurementSet`. One clear disadvantage is that some columns (the required columns) are returned as references while the optional columns are returned as pointers. The current thoughts on the future `MeasurementSet` (to be shared by all types of data) are outlined later in this document and summarized here. We intend to provide an enumeration, with each registered column name having a corresponding entry in the enumeration. Member functions will provide a translations between the enumerated symbolic names and the registered column names. This will eliminated the cumbersome need to maintain a member function for each registered column name.

There are a few additional functions that `SDMeasurementSet` provides. The first 3 of these would be unnecessary if `SDMeasurementSet` was a Table, the last 2 are more typical of the types of convenience functions likely to be necessary in a `MeasurementSet`.

`void attachNewTable(Table &attachToMe);`

This allows one to attach a `SDMeasurementSet` to a different table from the one that was used when the class was constructed (thereby allowing the same `SDMeasurementSet` object to be used with several tables, although not at the same time).

`Table &asTable();`

This returns a reference to the currently attached `Table`. This allows the user to do `Table` operations (*e.g.*,sort, select, etc.). The results from such operations might then be used to construct a new `SDMeasurementSet`.

`uInt nrow();`

A convenience function that returns the number of rows in the attached `Table`.

`static Bool isSDMeasurementSet(const TableDesc &desc);`

This allows a user to test if a table descriptor is a valid for use by a `SDMeasurementSet` (this make sure that the required columns and keywords all exist).

`Double referenceDate() const;`

A convenience function that returns the value of the ReferenceDate keyword.

There is no substantial difference between `SDMeasurementSet` and ROSDMeasurementSet other than that const or RO objects are used.

2.1.4 Filling a `SDMeasurementSet`

A `MeasurementSet` must be filled. At the present time, the only access to astronomical data is through FITS and the current set of AIPS++ FITS classes. For single dish data from NRAO telescopes, UniPOPS provides a FITS writer that writes Single Dish FITS files. Single Dish FITS is standard binary table FITS with some conventions for describing single dish data (standard column names). A FITS binary table (and hence single dish FITS data) is easily placed into an AIPS++ table. Eventually AIPS++ will be able to fill a MeasurementSet from a telescope specific data format.

There are two facts about using FITS as the input data that impact the filler. First, the FITS classes only read sequentially from the input file. Second, single dish FITS binary tables require that the length of each data array in each row of the FITS table be the same length. For NRAO 12-m on-the-fly data, this

means that the OTF rows are found in a separate FITS table from the calibration data. Additionally, the 12-m gain calibration data is contained in a separate file from the OTF and OFF data, which must also be written to a separate FITS binary table. So, in order to fill a single `SDMeasurementSet` with all of the data necessary to calibrate and image an OTF observing session, three input single dish FITS binary tables are required (the OTF data, the OFF source data and the gain calibration data).

In order to facilitate the merging of these three input FITS files into a single `MeasurementSet`, a `SDMSSource` class was developed. This class can also be used to fill data from a single FITS file. It is planned that this will ultimately be a base class from which specific `SDMSSource` classes will be derived to handle input from sources other than FITS (*i.e.*,telescope specific data formats). Filling a `MeasurementSet` from multiple data streams will be a useful option for other fillers.

The following background information is required to understand the excerpts from the `SDMSSource` header file which follows.

`BinaryTable` is the AIPS++ FITS class that converts a FITS binary table to an AIPS++ `Table`. A `BinaryTable` object is used to construct a `SDMSSource`.

Several `SDMSSource` objects are attached to a single `SDMeasurementSet`. They can fill the attached `SDMeasurementSet` until the universal time (UTime) of the data in the `BinaryTable` object changes or they can fill until they reach the end of the `BinaryTable` object.

A number of member functions are provided which are used to help an application coordinate several `SDMSSource` objects so that the single `SDMeasurementSet` is filled in the desired order.

```
class SDMSSource {
public:
    // Construct one from a BinaryTable, the second argument
    // is used to indicate whether or not this object should keep an
    // index of the mapping of scan number to time stamp.  Since
    // The OFF and GAIN calibration information is referenced by
    // scan number in the input FITS data and this information is
    // reference by TimeStamp in an SDMeasurementSet, this index is
    // available to the filling program so that the correct reference
    // information can be filled into the SDMeasurementSet.  This is
    // important when filling from separate FITS files where the OFF
    // and GAIN TimeStamps are not available to each individual
    // SDMSSource.  The default is to NOT keep an index.
    SDMSSource(BinaryTable& bintab, Bool indexed = False);
    ~SDMSsource();

    // provide a TableDesc appropriate for a SDMS.  The default
    // values for each column and keyword are derived from the first
    // row of the attached BinaryTable.  This can be used to construct
    // the initial empty SDMS.
    TableDesc &makeDesc();

    // test to see if a SDMS is attached (it must be attached after
    // the object is constructed).
    Bool isAttached();

    // attach the SDMSSource to an SDMeasurementSet
    Bool attachSDMS(SDMeasurementSet& attachToMe);

    // return a reference to the currently attached SDMeasurementSet
    SDMeasurementSet& getSDMS();

    // fill the SDMS until the time in the BinaryTable changes from
```

```
    // its current value.  Each time change generates a new TimeStamp.
    // Start filling at the indicated starting row of the attached
    // SDMS.  The returned value is the next available row of the
    // attached SDMS that could be filled.  If the returned value is
    // equal to the input value, nothing was filled to the SDMS
    // (likely due to the attached BinaryTable being at the end of the
    // FITS table).
    uInt fill(uInt startRow);

    // fill the SDMS until all the remaining data in the
    // BinaryTable has been used.  Start filling at the indicated
    // starting row of the attached SDMS.  The returned value is the
    // next available row of the SDMS.  If the returned value is
    // equal to the input value, nothing could be filled to the SDMS.
    uInt fillAll(uInt startRow);

    // return the time associated with the next row of the
    // BinaryTable.  The value returned is UT seconds since the
    // ReferenceDate of the SDMS.  Returns 0 if not attached to an SDMS.
    Double nextUTime();

    // For an indexed SDMSSource object, return the timeStamp
    // associated with a given scan number.  Returns 0 if that scan
    // number is not found in the index or the SDMSSource is not indexed.
    uInt timeStamp(uInt scanNumber);

    // The shape of the data column of the BinaryTable.
    IPosition dataShape();

    // The maximum number of rows this could potentially fill.
    // For most FITS binary tables this is simply the number of rows
    // in the binary table.  For 12-m OTF binary tables, each row is
    // really several individual spectra, each one of which will be
    // a single row in the SDMS.
    uInt nrows();

     // Other information valid for the full SD-FITS binary table.
     // The ReferenceDate, may not be the ReferenceDate of the SDMS.
     Double refDate();
     // The Telescope
     String telescope();
     // The observer
     String observer();
     // The project
     String project();
     // Is the BinaryTable 12-m OTF data
     Bool isOTF(); private:
    // this is not shown in this sample, an implementation detail.
};
```

The SDMS filler then has the following form:

- Get the names of the input single dish FITS files.

- Open and construct a `BinaryTable` object for each file.
- Construct a `SDMSSource` from each `BinaryTable`.
- Using `makeDesc()` from one of the `SDMSSource`, construct an empty `SDMS`. Set the size (number of rows) of the `SDMS` based on the returned values from `nrows()` from each `SDMSSource`.
- Attach the newly created `SDMS` to each `SDMSSource`.
- Fill the `SDMS`, this will be sorted in increasing UTime since each input `BinaryTable` is itself sorted (alternatively, the final filled table could be sorted and time stamps adjusted if the input streams were unsorted).
- Find the `SDMSSource` with the smallest `nextUTime()` and use `fill()` to fill until `UTime()` changes.
- Put the appropriate values for OffTimeStamp and GainTimeStamp in the `SDMS` based on the index associated with each `SDMSSource`.

Most of the steps above are not specific to FITS or even to OTF data suggesting that `SDMSSource` is a useful concept that can be generalized. Specific input data streams will likely need specific derived versions of this base class. Filling from multiple data streams will also be common.

The execution time to convert the trial data set described above from the on-line data files to FITS is roughly 5 minutes. The time to fill a `SDMeasurementSet` from these FITS files is approximately 10 minutes.

2.2 Calibration

Once filled, then next step before the data is imaged is calibration. In AIPS++ , calibration is accomplished through a `TelescopeModel`. A `TelescopeModel`l in general will consist of multiple `TelescopeComponent` objects. Each `TelescopeComponent` in turn consists of the information and methods required to perform a specific calibration step. A `TelescopeComponent` has an `apply()` method that actually calibrates the data and a `solve()` method that uses a `SourceModel` and the data to solve for some calibration parameters (that might then be used to calibrate the data through `apply()`). There is also a `corrupt()` method that is the inverse of `apply()`. The `TelescopeModel` then coordinates the methods of the various `TelescopeComponent`s contained within it.

2.2.1 `PositionSwitchedTC`

Our goal of imaging 12-m OTF data only requires one `TelescopeComponent`. We use this directly without any `TelescopeModel`. On-the-fly data at the 12-m is position switched data. This is an observing technique that is common at single dish telescopes. It is described in more detail earlier in this document.

A `PositionSwitchedTC` (position switched telescope component) is used to apply "on minus off" style calibration (and decalibration, ultimately, through `corrupt()`). There are two styles of such calibration within NRAO single dish telescopes (ON is the "on source" data - this is really what needs to be calibrated, OFF is the "off source" data).

- Scalar Normalization: The output (difference) spectrum is calculated by:

  $$\texttt{difference\_spectrum} = ((\texttt{ON} - \texttt{OFF})/\texttt{OFF})\,\texttt{TSYS}$$

  where TSYS is a single scalar number (the system temperature). This is referred to as the "Green Bank style" calibration since it is used at the 140' telescope in Green Bank.

- Vector Normalization: The output (difference) spectrum is calculated by:

  $$\texttt{difference\_spectrum} = ((\texttt{ON} - \texttt{OFF})/\texttt{OFF})\,\texttt{GAIN}$$

  where GAIN is a spectrum (Vector) of numbers. This is referred to as the "Tucson style" calibration since it is ' used at the 12-m telescope.

The preferred normalization when both are possible is vector normalization. The normalization can be forced to either style assuming all of the required information is present.

The OFF and GAIN data are determined by the values of the OffTimeStamp and GainTimeStamp columns for the row to be calibrated. Currently, this information is always taken directly from the `SDMS` being calibrated. Eventually, a user will be able to construct a `PositionSwitchedTC` that will maintain separate tables of where the OFFs and GAINs are. This will allow the user to alter those values without altering the underlying data. These tables will initially consist of the values found in the `MeasurementSet`. This is referred to as a "sucking" `TelescopeComponent` because the internal calibration information is "sucked" from the original data.

At the moment, `apply()` creates a new table. In the future, when virtual tables are available, it will perform on-demand (also called on-the-fly calibration, not to be confused with on-the-fly data). On-demand calibration simply means that `apply()` doesn't actually make a new copy of the data (as the current implementation does). The use of virtual tables means that the actual calibration may not occur until the user requests a specific value - the calibration is done "on-demand" as needed by the user.

We have not yet implemented `corrupt()`. `solve()` does nothing for this `TelescopeComponent` since there is nothing to solve for.

There are also a few utility functions that allow a user to set the type of normalization, query the type of normalization, and verify that the `SDMS` contains the required columns for calibration.

The `PScalibrate` application. Since the current implementation of `PositionSwitchedTC` creates a new table when the `apply()` function is used, it is simplest to have a separate application perform the calibration. In the future, applications will be able to calibrate "on-demand" and it will be possible to go directly from a `SDMS` through `apply()` to an imager (or other object requiring calibrated data) within a single application.

Our calibrator is extremely simple. Here is the full calibrator:

```
#include <iostream.h>
#include <aips/Tables.h>
#include <aips/Error.h>
#include <aips/Input.h>
#include <dish/SDMeasurementSet/SDMeasurementSet.h>
#include <dish/SDTelModel/PosSwitchTC.h>

main(int argc, char **argv) {
    try {
        Input inputs(1);
        inputs.Create("raw","","Input: raw, uncalibrated data");
        inputs.Create("calibrated","calibrated.data","Output:calibrated.data");
inputs.ReadArguments(argc, argv);
        String rawFile(inputs.GetString("raw");
        String calibratedFile(inputs.GetString("calibrated");
        ROTable rawTable(rawFile);
        ROSDMeasurementSet raw(rawTable);
        PositionSwitchedTC fixer(PositionSwitchedTC::VECTOR);
        SDMeasurementSet calibrated = fixer.apply(raw, calibratedFile);
    } catch (AipsError x) {
        cerr << "Error - " << x.getMesg() << endl;
    } end_try;

    return 0;
}
```

The `PositionSwitchedTC` is constructed so that it attempts to use vector normalization. As currently implement, the calibrator attempts to calibrate the entire table. Applications will eventually need to do any selection options on the table so that only those portions that need to be calibrated are calibrated.

The time to calibrate the test data set is approximately 3 minutes.

2.3 Imaging

The final step in our initial single dish implementation is imaging. Eventually this will be handled by a `MeasurementModel`. A `MeasurementModel` consists of an `invert()` and a `predict()` method. `invert()` "inverts" the data to obtain a model of the sky. Formally this can be modeled as an actual matrix inversion although in practice for typical single dish `MeasurementModel`s, there will be no need to do any matrix inversion. In fact, it should be possible to implement a number of common single dish `MeasurementModel`s in glish. `predict()` takes a model of the sky and "predicts" what the data would look like given the assumptions of the `MeasurementModel`.

2.3.1 `sdmsgridder`

We have not attempted to build a `MeasurementModel`. We have simply built an application that grids the irregularly spaced and over sampled 12-m OTF data onto a regular, appropriately sampled grid. This application will become the `invert()` method of a specific `MeasurementModel`.

The basic gridding functions required for the OTF imaging problem are the same as those required for gridding UV data from an interferometer. In the single dish case, we are gridding directly to the image plane and no fft of the gridded data is required. The gridding is really a convolution at each of the observed data points on the sky and is independent of frequency. The `GridTool` class provides the functionality that we require.

Much of our gridder is devoted to determining the parameters appropriate to the input calibrated `SDMeasurementSet` so that all of the values will be gridded. There is currently no way to select a portion of an `SDMeasurementSet`, although such an operation is trivial to implement. The user can indicate through command line inputs the gridding parameters they wish to use (the size of the cube, the center position of the cube on the sky, the dimensions of each pixel on the sky, etc), if they need to override the default values or the values that the gridder will choose based on the `MeasurementSet`.

The `DummyImage` class is used to hold the gridded values. Eventually the new `Image` class will be used, but this was not available when the gridder was written. This should be a rather smooth transition. `GridTool` is used to do the actual gridding or convolution onto the cube. The resulting `DummyImage` is written out as a FITS file. This last step (conversion to FITS) will not be necessary once the `Image` class is used (unless requested by the user to transport their data to another analysis package or for long term storage).

2.3.2 Enhancements to `GridTool` for Efficiency

Our first implementation of gridder ran extremely slowly. This single step in the process of converting 12-m OTF data into an image took longer than the time spent taking the data and this first image was only from 1 polarization and 1 spectral window (1/4 of the total data set). This is clearly unacceptable.

The bootle neck is in the `GridTool` class. Indexing into the cube holding the gridded data and the cube holding the sum of the weights at each data point is through virtual function calls. These calls are extremely expensive. We have replaced the inner-most loop (where the values are multiplied by the appropriate weights and placed into the two cubes) with a simple 10 line fortran routine (not counting comments and variable declarations). This step has increased the execution time so that the gridding time is now roughly comparable to the equivalent step in classic AIPS.

The time to grid the calibrated test data after implementing these enhancements is just under 4 minutes for 1 polarization of the 128 channel spectral window.

3 Plans

We now summarize our immediate, short range, and long range plans for Single Dish Calibration and Imaging.

3.1 Improvements

All of the classes we have implemented for this initial single dish development and calibration effort need improvements. Many of these have been suggested in previous sections of this document.

3.1.1 MeasurementSet

Building upon our experience in implementing SDMeasurementSet, Peter Teuben is outlining a more general MeasurementSet. This MeasurementSet will be used by all of AIPS++ . While it may be necessary to derive more specific MeasurementSets from the base MeasurementSet, it is our feeling that this should be avoided when practical. Initially we plan that both interferometric and single dish data will share the same MeasurementSet.

Sharing the same MeasurementSet does not mean that the table organization will be the same. One of the strengths of the planned MeasurementSet is that it is a repository of standard column names. A few columns will likely be required while most will be optional. Rather than derive specific MeasurementSets that require more column names, it is felt that the same result can be accomplished through member functions of the MeasurementSet. For example, one might have

```
Bool isSingleDish();
```

to verify that the MeasurementSet is a valid single dish MeasurementSet (it has the appropriate required column names) or

```
Bool isVLASet();
```

to verify that the MeasurementSet is a valid VLA MeasurementSet. If there are other member functions required by a specific type of MeasurementSet, then this suggests that a new class should be derived. However, since the MeasurementSet is primarily a Table that holds the data with little additional functionality, the need for specific derived MeasurementSets seems remote at this time.

As described earlier in this document, the key aspect of the new MeasurementSet is the use of an enumeration listing the columns with a predefined meaning. This replaces the multiple member functions, one for each column, that are found in the initial SDMeasurementSet. Here is one possible implementation that is being considered (using pseudo-C++).

```
class MeasurementSetInfo {
public:
    // Columns with a predefined meaning
    enum PredefinedColumns {
       NON_EXISTENT=0, // "TRUE" columns exist.
       DATA, // Data vector (spectrum, time series, ...)
       ROW_FLAG, // The per row flag
       CHANNEL_FLAG, // The vector of channel flags
       ...
       OTHER};

    virtual String columnName(PredefinedColumns which); // avoid typos
    virtual PredefinedColumns columnType(String name);
    virtual Bool isScalar(PredefinedColumns which);
    virtual Bool isArray(PredefinedColumns which);

    virtual Bool exists(PredefinedColumns which);

    virtual Fallible< ScalarColumn<Int> >getIntColumn(PredefinedColumns which);
    // ... for all column types.
};
```

MeasurementSetInfo defines all the standard MS columns and allows access to them in a way which catches typos at compile time. This could then be used within a MeasurementSet.

Once the basic form of `MeasurementSet` is agreed upon, the larger task of agreeing on standard definitions for column names will need to be addressed. In order for the UV Calibration and Imaging group to proceed early next year, both of the above will need to be in place. We expect that the next version of the `MeasurementSet` will be available for initial use by the end of December, 1994.

3.1.2 Filling

It is imperative that AIPS++ be able to fill directly from any of a number of telescope specific data formats. For 12-m OTF data, this is important for two reasons. First, the time spent converting 12-m data (which is in the UniPOPS Single Dish Data (SDD) format), is a significant part of the time spent creating an OTF image. It is unlikely that the calibration and imaging will be able to keep up with the projected data rates given the current time required for conversion from SDD to SD-FITS and from SD-FITS to an AIPS++ Table. Second, observers need to see the image as soon as possible, preferably as the data arrives so that the gridded image can build up as the data is deposited on disk. Mistakes in the observing parameters, pointing problems, unexpected emission (requiring the observer to map a larger region), atmospheric problems, etc, need to be dealt with as soon as possible so as to make the most efficient use of limited observing time. At 40 or more minutes for a modest OTF image, the observer simply can not afford to wait until the data has been taken to grid the data (wasting even more time) and finally to see the results. A simple mistake has the ability to waste up to an hour of valuable observing time.

The existing `SDMSSource` will be used as a starting point for development of a more general `MSSource` base class. From this base class the single dish FITS and SDD specific versions will be derived. The SDD specific version will be able to handle data from the 12-m as well as from the 140' telescope. Alternative binary table FITS fillers should be easily derivable from this base class using the single dish FITS filler as an example. We expect to be able to fill the general `MeasurementSet` from SDD data by the end of January, 1995.

3.1.3 Calibration

We expect to be able to implement on-demand calibration of position switched data using virtual columns shortly. It is likely to be a few days worth of work and should be easily completed by the end of December, 1994.

The other planned improvement is to implement a true "sucking" `TelescopeComponent` constructor. As described previously, this would allow the user to edit the calibration tables without altering the raw `MeasurementSet`. This is estimated to be about 1 weeks worth of work. However, the ability to truly edit these tables depends on a working table browser. Furthermore, the coordination between a running application which is doing calibration on-demand and a table editor that would allow the user to alter the calibration tables used by the application (allowing feedback to the application) needs to be worked out.

We will develop additional `TelescopeComponent`s as required. The simple nature of most current single dish calibration techniques means that this step is not difficult for most planned calibration techniques. Some `TelescopeComponent`s may be sufficiently simple that they can be implemented as a glish script. This ability provides the flexibility that single dish observers require (where a new observing mode or a slightly different calibration technique are often developed during an observing session) while maintaining the framework of the `TelescopeComponent` and `TelescopeModel` approaches to calibration.

3.1.4 OTFtool

The ultimate near-term goal of the single dish calibration and imaging effort is to produce a tool for gridding 12-m OTF data as it arrives on disk in the raw telescope file during observation. The user will be able to see the gridded image build up as the data is taken, row by row. Some analysis of the data will be possible (the user will be able to capture snapshots by getting copies of the data cube at any phase of the imaging). The user will have control over the calibration (both the type of calibration, pre-imaging as is now down and post-imaging, described below, as well as the calibration parameters as described above).

Once the capability to read SDD is available, most of the above functionality should be achievable in a few weeks. We expect to have a useful OTFtool by the end of February which will display the gridded image

as the data arrives and allow the user to analyze a snapshot of the gridding process at any time during the process. Control of the calibration step will likely take some additional time (this depends on an addition described in the next section).

3.2 Additions

3.2.1 Spectrum and SpectrumSet

The `MeasurementSet` is flexible. The price for flexibility is that applications must do a fair amount of checking to verify that a specific `MeasurementSet` has the characteristic they desire and to deal with missing information appropriately. In other words, the fundamental telescope-specific nature of a `MeasurementSet` is not convenient for general, non-telescope-specific applications. We find a need for a more general class to simplify the interface to `MeasurementSet`. Some non-telescope-specific applications that would use such a class include imaging applications (thus allowing these imaging application to image data from different telescopes), data editors and displayers where the basic astronomical information is to be edited and displayed.

Our current term for such an object is a `Spectrum`, although with a suitable renaming, this object might be general enough to encompass time series and other vector objects. A `SpectrumSet` is a collection of `Spectra` presented in some order. A `SpectrumIterator` may be a more appropriate name. These concepts are not specific to single dish data.

The `SpectrumSet` as a whole has

- Units
- A Coordinate System
- "Arbitrary" information (keyword = value)

Each `Spectrum` has

- a vector of values
- a characteristic frequency for each channel (if this is to be more general, then this statement simply requires a characteristic value for each channel)
- A scalar flag (flagging the entire spectrum)
- A vector flag (one for each channel)
- Weights and theoretical sigma
- Location (Sky - Ra/Dec, UV, whatever)
- Stokes info

These would be base classes. The obvious implementation is to attach them to a `MeasurementSet`. However, they might also be attached to an on-line system or even to an image cube for single dish data.

The later observation, that a single dish `SpectrumSet` can be attached to an `Image`, will allow us to do post-imaging calibration. The `TelescopeModel` can be instructed to apply a calibration to a `SpectrumSet`. If the `SpectrumSet` is attached to an `Image`, this would immediately alter the image.

3.2.2 Continuum Data

Most of the discussion in this document concerns spectral line calibration and imaging. We have been driven by need rather than complete functionality. UniPOPS' most obvious deficiency is in dealing with complex spectral line data sets and we have chosen one clear example of that to focus on, 12-m OTF spectral line data. The `MeasurementSet` is sufficiently flexible to allow for continuum data. Currently it is not high on our priority. We would, however, like to fill a `MeasurementSet` with continuum data shortly after we have a useful OTFtool simply to demonstrate that it can be done.

3.2.3 UniPOPS and glish

One of the strongest aspects of UniPOPS is it's versatile procedure language. The 10 1-dimensional arrays in UniPOPS can be manipulated directly, element-by-element, within a procedure or through a number of specific verbs, which operate on an entire vector. glish is more versatile since vectors (and object of higher dimensionality) are built in to the language. It will be possible to replace the functionality of many UniPOPS verbs with glish scripts.

One long term task is to mirror as much of the functionality of UniPOPS as possible with glish scripts. Specific applications (*i.e.*,using C++rather than glish) will be created where required (by poor execution times). This is a time consuming but not particularly challenging task.

3.3 NRAO

AIPS++ will gradually replace UniPOPS as the single dish package of choice within NRAO. In order for this to be a replacement of choice, AIPS++ must contain all of the functionality of UniPOPS. As stated above, this is a time consuming but not particularly challenging task. We expect to have most of the key functionality of UniPOPS available within an AIPS++ environment within the next year.

Users of UniPOPS as well as other analysis packages will not choose to use AIPS++ unless it does something not available in their current package or if AIPS++ simply does that thing better. UniPOPS is adequate for most single dish users. It is unlikely that simply supplying AIPS++ with the functionality of UniPOPS will be sufficient to draw users to AIPS++ . This is especially true for experienced UniPOPS users. The payoff for learning a new "language" must be worth the price. Consequently, the primary focus of the single dish calibration and imaging effort within AIPS++ is initially on providing tools to do tasks not available or difficult within UniPOPS. One such example is the OTFtool described previously.

3.3.1 GBT

The GBT is relying on using AIPS++ for it's data analysis needs. There is sufficient time that this should not be a problem. However, there are specific near-term analysis needs for the GBT that can be met by AIPS++ . One specific example is the holography test planned for early 1995. We believe that there are sufficient parts of the AIPS++ infrastructure currently in place and sufficient experience with single dish data for AIPS++ to provide the analysis needs for this test. We will work with the GBT to formalize the list of requirements for this test and should begin work on this as soon as possible.

3.3.2 12-m

The next analysis software need at the 12-m after OTFtool is support for the 8-beam receiver. This will be available for use sometime during 1995. As with the GBT, a fair amount of planning needs to occur so that we can anticipate the analysis needs rather than react to the analysis needs as we are doing with OTFtool.

3.4 General AIPS++ SDCI Plans

Users will use AIPS++ only by choice. They payoff must be worth the price of learning the new language. Single dish analysis needs are generally not well supported. The support staff frequently have their hands full maintaining an existing package and providing support to new instruments and observing techniques.

There is one primary reason why single dish analysis will migrate to AIPS++ (at least from UniPOPS, although this same reason likely applies to other packages). UniPOPS is difficult to maintain. New instruments and observing techniques are increasingly difficult to shoehorn into the old analysis package. UniPOPS deals well with moderately sized, 1-dimensional chunks of data. It offers little help in managing the data base (keeping track of which objects were observed when with what instrument, etc). It is increasingly common for single dish telescopes to generate huge data bases during a single observing session. Large regions of the sky can be mapped with ease. Multi-feed (multi-beam) arrays are becoming common. Even the individual spectra are getting quite large (in terms of number of channels). These advances in instrumentation and observing techniques strain or break UniPOPS.

One solution to these advances would be to use a different package to handle the new techniques while retaining the old package for the more mundane tasks it does so well. This is the immediate solution being used at the 12-m to handle OTF spectral line data. It is also the technique they have used for quite some time to handle complex continuum mapping observations. OTF data from the 12-m is currently handled within classic AIPS. It is adequate although it does not allow the flexibility of calibration that is desired. It is also awkward for the observer to use two different analysis environments. Clearly it is preferable for all analysis to occur within the same system.

One major obstacle that will keep single dish observers from using AIPS++ is filling a `MeasurementSet` from their native data format. A primary job of the single dish development will be to assist consortium members and ultimately non-consortium members in getting their data into a `MeasurementSet`. A large portion of single dish observers use the CLASS reduction package. The single dish calibration and imaging group will have as a goal the conversion of CLASS data into AIPS++ as well as AIPS++ data into CLASS format. This will make it easier for a number of other groups to begin to explore AIPS++ .

3.5 Summary of Plans

- `MeasurementSet` - this must be in place before the UVCI design effort begins. End of December 1994.
- Fill `MeasurementSet` from NRAO single dish format (SDD). End of January 1995.
- `OTFtool` - a useful tool for 12-m observers. Requires both of the above. Also requires simple image display and analysis capabilities). End of February 1995.
- GBT holography test. This will require close coordination with the GBT group. A specific list of analysis requirements is needed as soon as possible (End of December 1994). Will require filling of a `MeasurementSet` from the spectral processor (this will likely be from FITS). Early spring, 1995.
- 8-beam receiver. Definition of initial analysis requirements.
- glish scripts for mundane UniPOPS functionality. As time permits with the goal of duplicating most UniPOPS verbs by the end of 1995.
- Parkes data into a `MeasurementSet`. This is the non-NRAO single dish telescope that is represented in the consortium. Parkes has used UniPOPS in the past so this step may not be too difficult. End of 1995.
- CLASS data into `MeasurementSet`. End of 1995.

A Summary of demonstration.

This is a brief summary of the demonstration of the OTF gridder that was presented at the AIPS++ review. The demonstration consists of the following glish clients:

- **OTF gridder.** This is the application that does the gridding of calibrated data. It recognizes the following events
  - **grid** : start gridding. The input parameters are passed to the gridder client in a glish record.
  - **suspend** : stop gridding. The client writes out the current image and waits for the next **grid** event.
  - **getImage** : write out the current image. The client temporarily suspends gridding in order to write out the current image. Gridding resumes immediately after the image is written.

  The gridder client generates three glish events:
  - **progress.** This is generated occasionally to indicate the fraction of the data that has been gridded.
  - **imageSaved.** This indicates that the current image has been saved. It only occurs in response to a **getImage** event.
  - **grid_result.** The gridding has finished. The gridder is now waiting for another **grid** event.
- **otfClient.** This is a simple glish client that generates the **suspend** and **getImage** events from buttons that the user can click on (these events are then passed on to the the gridder client). The **otfClient** client also recognizes **progress** events. These are translated into a displayed bar indicating the progress value.
- **imager.** The image display client. This displays a 2-D AIPS++ `Image`. It has some simply knowledge of the image coordinates and one can choose and manipulate the colormap. It recognizes the **imageFile** event, generated in response to an **imageSaved** or **grid_result** event, and displays the indicated AIPS++ image (one plane of the image cube is hard-coded as the plane to display). Additionally, one can click on a button (labeled "spectrum") and then select a pixel in the image. This generates an **xy_plot** event.
- **plotter.** This recognizes the **xy_plot** events generated by the **imager**. The value associated with that event indicates the xy pixel selected. From that information, the **plotter** displays the values of the image cube in the third dimension at the the indicated xy pixel.

The gridder event flow is indicated in the following pseudo-glish code:

```
whenever gridder->progress do
    send otfClient->progress([percent=$value * 100])

whenever gridder->grid_result, gridder->imageSaved do
    send imager->ImageFile([name=$value.file, plane=64])

whenever otfClient->suspend, otfClient->getImage do
    send gridder->[$name]($value)

whenever imager->pixel_result do
    send plotter->xy_plot($value)
```

The demonstration begins by starting up glish and initializing the four glish clients described above (accomplished by including a glish script).

```
include ``SDGridder.g''
```

The inputs are retrieved from a mosaic window (although they could also be set from the command line - this was simply one example of some early thoughts on how inputs might be set and passed to an application).

```
inputs := mosaic_inputs(‘‘SDGridder’’)
```

This glish record, inputs, now contains the inputs for the gridder client. It is passed on to the gridder through a glish function which generates a “grid” event (after some translation of the inputs record to set the variable types correctly).

```
SDGridder(inputs)
```

At this point, the otfClient window appears and the status bar shows zero progress. The gridder reads the data and begins gridding according to the inputs. As the gridding progresses, the status bar in otfClient shows the value associated with the most recent “progress” event generated by the gridder.

At some point the user wishes to see an intermediate image (*i.e.*,before all the data have been gridded). The user clicks on the “Get Image” button in the otfClient, which generates a “getImage” event, which is passed to the gridder. At a time convenient for the gridder, the intermediate image is written to disk and gridding resumes. The gridder generates an “imageSaved” event, which is translated to an “ImageFile” even which is passed to the imager client. The intermediate image is displayed. The user can switch to “spectrum” mode in the imager in order to display the values in the third dimension of the cube at a specific xy pixel.