The AIPS++ Prototyping Bible

or, How to interpret the Universe any way you like

The Smirkers and the Plodders

1 Introduction

This document attempts to combine together the AIPS++ design work that has been done in January - June, 1994 and which reached an apex during the Jodrell Bank design week of May 16 - 20. Like its namesake, the document is subdivided into a number of Gospels. These gospels give the views of AIPS++ designers as to how the various components of the AIPS++ system should fit together. Hopefully, unlike the Christian church, AIPS++ will converge toward one view of the Universe rather than diverge into Schisms!

2 The Gospel According to Dave

2.1 Introduction

This gospel describes the results of two workshops on uv-calibration and imaging held at Dwingeloo, 25th to 29th April 1994, and Jodrell Bank, 16th to 20th May 1994.

2.1.1 Basic aims and requirements

The aim of this effort is to produce a design which could be implemented as an object-oriented framework of uv-calibration and imaging in C++ – essentially a superstructure using classes from the AIPS++ infrastructure library as well as the high-level astronomical classes which we concentrate on here. This could lead to a moderately sophisticated prototype, and if the design of the framework is successful, further development ought to be able to proceed largely by sub-classing and minimal modification of the framework itself.

This approach is not without difficulty. We have uncovered areas where much work in infrastructure design and implementation is necessary. However, it is appropriate that the high-level design should drive such developments rather than vice-versa.

2.2 What's new?

2.2.1 Source Models

A notable development in these meetings has been a change of emphasis from imaging to modelling; we now see the main goal of our framework as being a generalised source model, rather than a simple image. This is not such a new idea, but we think this is the first time that this has been formally adopted.

Source modelling (generalised imaging) will employ a standalone modelling method[1] which acts to update a SourceModel; calibration/self-calibration will be based on an update method on the VisSet. The MeasurementModel and TelescopeModel still play their parts, but within the framework they are likely to be seen as tools used by the source model and data classes.

[1] This should probably be a pure function; we had originally envisaged this being a method of SourceModel, but there seems to be no compelling reason to do this at the moment. However, if this process ever requires access to any information private to a SourceModel, then it should be a method of SourceModel

A good example of this is in CLEAN imaging (with or without self-calibration; the latter merely adds another loop of iteration), where the source is modelled as a set of point-source components. Then the generation of an image is a simple extension of the basic framework for generalised modelling, *viz.,* convolving the CLEAN model components onto an image grid and adding residuals. Of course, there is no reason why the desired model should not be a conventional image; indeed, this is likely to be the case in implementations of many algorithms such as Maximum Entropy.

We see several benefits to our approach;

- The so-called A-matrix approach suggested by Tim Cornwell (Cornwell, 1993 *a,b*) should be possible to implement in such a framework; indeed, the *Imagers* described in the original Greenbank scheme (Shone & Cornwell, 1992) are essentially identical to our source model update functions.
- The general modelling framework might be applicable to a wide range of different forms of modelling of astronomical data. This includes, for instance, single-dish data reduction, although our primary motivation has been the reduction of radio-synthesis data.

We should not exclude the possibility of other frameworks for data reduction, although we believe that the guiding principles that have been used here (and at Green Bank) have a generality that should influence such frameworks.

2.2.2 MeasurementSet physical storage

A problem which regularly arises is the issue of how to divide raw data (particularly those which vary at different rates) intelligently when filling. This does not have a general solution, and may vary with instrument and observing mode. We conclude that we should not attempt such a solution, but should attempt to create MeasurementSets which have a common logical table interface, whilst being stored efficiently. A particularly useful physical form which we have selected is based on that used by MIRIAD, and is in turn derived from the physical format of data from Hat Creek. This stores data sequentially in a variable-name=value form, with the value of a variable staying fixed until it changes.

All instruments and observing modes should fit easily into this, whilst one implementation of the storage manager will provide a table interface. The disadvanage if this approach is that the data will be read-only, and sequential access is necessary unless an indexing scheme is used. Neither of these seem to present any great problem; it is unlikely that we should want to modify a MeasurementSet under normal circumstances, and even if we should, the data can be copied into a table which has a more conventional physical form.

2.3 Fundamental Classes

Although the basic classes we are using have been around for quite some time, we briefly describe them here in order to clarify our views of their nature and purpose, particularly for the benefit of those who may not be familiar with current ideas.

The goal of our scheme is the construction and refinement of a *SourceModel*, given raw observed data in a *MeasurementSet*, together with data describing the assumed nature and state of the observing instrument in the form of a *TelescopeModel.* The *MeasurementSet* is uncalibrated, and a calibrated *VisSet* is produced by applying corrections described by the *TelescopeModel* to the *MeasurementSet.* Thus a secondary goal, in order to calibrate, is the refinement of the state of the *TelescopeModel.*

We assume that the relationship between a "perfect" *SourceModel* and a "perfect" *MeasurementSet* is described by a *MeasurementModel*, and that any real *MeasurementSet* is corrupted by imperfections in the observing system described by the *TelescopeModel.* Such imperfections may apply within the domain of the *SourceModel* or of the *MeasurementSet;* eg primary-beam effects apply in *SourceModel* domain, while receiver gains apply in the domain of the *MeasurementSet.*

Thus the *MeasurementModel* embodies the forward and reverse transforms between sky-plane and visibilities in the case of radio-synthesis imaging – essentially, the *van Cittert–Zernicke theorem*, but in practice, this might be implemented as a two or three-dimensional Fourier transform, depending on the nature of the chosen *SourceModel* and/or assumptions about the instrument.

Note that the *TelescopeModel* will be responsible for containing and describing telescope effects (both those in the sky-plane and visibility domain). However it will only be responsible for corrected for those that are conveniently handled in the *uv* domain – the *MeasurementModel* will need to account for the telescope effects which are most conveniently handled in the image domain. These include primary beam and pointing effects, polarisation leakage and receptor gains which vary across the primary beam, etc. In this case, the *MeasurementModel* is implementing an augmented *van Cittert-Zernicke* equation. We do not presuppose that the *MeasurementModel* can correct everything when forming a dirty image (the *invert* method). However it can at least corrupt the sky-plane data in the appropriate fashion (the *predict* method).

The *MeasurementSet* and the *TelescopeModel* are likely to depend on the particular instrument; different interferometers generate different kinds of visibility data and use different calibration schemes – these must be supported at the level of the raw data. On the other hand, a *VisSet* is a canonical form, which does not display any dependence on a particular type of interferometer to the outside world, and allows access to data via a canonical set of interface classes.

2.3.1 Table-like interfaces to data

Most classes will have a `getTable()` method which returns some underlying table, in order to allow access for the Table Browser and any desperate acts of data manipulation. However, data access will normally be via other interfaces which reflect the semantic behaviour of the class.

2.3.2 Individual classes in detail

We shall attempt to describe the detailed attributes and methods of the major astronomical classes in more detail here. **N.B., THEY ARE STILL UNDER CONSTRUCTION!**

2.3.3 SourceModel

SourceModel is typically a model of the sky-brightness distribution, such as a set of CLEAN components, an image-array generated by Maximum Entropy Methods, or some other parameterised model (eg Gaussian components). SourceModel is probably an abstract base class, which contains methods to manipulate a list of parameterised components.

In forming a concrete class from the abstract base class, one of the more important methods to be overloaded may be `update`. This effectively implements a modelling or deconvolution operation. We envisage a number of these `update` methods (and corresponding different SourceModel types) which implement different algorithms.

In this framework, we believe we can implement the current conventional algorithms (both image- and visibility-base deconvolution algorithms). At the same time, there is enough flexibility to allow newer approaches, such as the A-matrix, to be implemented.

2.3.3.1 Attributes

A SourceModel will probably be internally implemented by a table of parameterised source components (eg CLEAN point sources, Gaussian models, *etc.*).

2.3.3.2 Methods

- `update()` – implements the basic modelling algorithm. There is a range of possibilities for what this method might do in practice, eg
  - Deconvolution of existing images to generate a CLEAN component model (CLEAN modelling of images).
  - CLEAN modelling of *uv*-data given a Measurement Model; This might be implemented in several ways, including MX-like ("Cotton-Schwab") CLEANing.
  - CLEAN modelling of uvdata as above, but including self-calibration in an MX-like CLEAN (difference mapping).
  - Most of the above should be possible using deconvolution algorithms other than CLEAN.

  Thus the method is likely to take several forms:
  - `update(VisSet uvdata, MeasurementModel mm, Deconvolver deconv);` models uvdata given mm & deconv.
  - `update(Image dirty, PSF psf, Deconvolver deconv);` deconvolves dirty using psf.
  - `update(SourceModel source1, SourceModel source2,...);` used in the combination of multiple SourceModels to form a new one.
- `Fourier` – Return the Fourier transform of the model at a particular location in the Fourier plane. One should note that this is distinct from the `predict` method of the MeasurementModel. However the MeasurementModel's `predict` will almost certainly make use of this method to achieve its job. Other methods will probably be needed to support the MeasurementModel's `predict` function.

2.3.4 MeasurementSet

The MeasurementSet is essentially a dump of the raw observed data into AIPS++ internal form.

2.3.4.1 Attributes

This may contain tables, which in turn contain the real data.

2.3.4.2 Methods

These would include data selection.

2.3.5 TelescopeModel

TelescopeModels will contain a collection of *uv*-domain telescope effects (receptor gains, bandpass functions, etc) which it can solve for and correct. It probably also contains classes which describe several sky-plane effects (instrumental primary beam, polarisation leakage and gain which varies within the primary beam, etc). It is unclear who (if anyone) solves for these effects. In many (most?) cases standard models will suffice.

Telescope Models are constructed with an ordered list of TelescopeComponents. In this initial state, the Telescope model is fairly unintelligent, and must extract information from a MeasurementSet (or multiple MeasurementSets) in order to model a Telescope. Alternatively, a set of FakeParameters (eg antenna positions etc - STATIONS.DAT in the terminology of the CalTech FAKE program) would be used. In any case, the TelescopeComponents will be given these parameters in turn.

Information extracted from the MeasurementSet or FakeParameters will be copied (stored) into the TelescopeModel or TelescopeComponents if required for use later.

The TelescopeModel and all TelescopeComponents have solve(), apply() and corrupt() methods. The solve() method in the TelescopeModel will take some additional parameter list (calparams) which it uses to locate the appropriate TelescopeComponent which to invoke.

Normally, the TelescopeComponents are instantiated with the parameters need to define the way in which they work (eg the solution scheme to be used, interpolation scheme etc), but if it may be necessary to change these, this should be possible via soemthing like a `setparameter` interface.

2.3.5.1 Methods

- `solve` – Given data and a model, this solves for instrumental parameters. The `solve` method probably does no more than determine the appropriate TelescopeComponent, and invoke its `solve` method.

2.3.6 VisSet

The VisSet contains a MeasurementSet and TelescopeModel. It will be responsible for producing objects via which most high level applications will access the visibility data. It will be responsible for orchestrating the correction of the data that the TelescopeModel may need to perform.

2.3.6.1 Attributes

These include a TelescopeModel and a MeasurementSet.

2.3.6.2 Methods

- `selection` – The VisSet will probably have methods to select a subset of visibility data. In many cases these selections would be simply implemented with the underlying MeasurementSet's corresponding selection methods. However selections which depended on 'derived parameters' may need to be handled by the VisSet.
- `update` – This updates the current state of the TelescopeModel, essentially by invoking the TelescopeModel's `solve` method to solve for instrumental parameters (e.g. receptor gains). After having updated the VisSet (by refining the TelescopeModel), the visibility data that it produces will immediately reflect the improved calibration parameters.
- `dup` – The dup method generates a duplicate VisSet which, in a number of respects, is a copy of the parent.

2.3.7 Data Access Classes

The VisSet as described above has not produced any data. The data required by individual applications and algorithms will require more specialised views of bulk-data classes. For example, self-calibration will the set of all baselines for a particular integration (or time range).

These specialised forms will be typically be accessed via iterators which return successive chunks as desired. We envisage that the construction of the iterator will involve any indexing or (heaven forbid!) sorting of the data required from the MeasurementSet; then successive iteration (probably with a next() method call on the iterator or overloaded "[]" operator) will result in the requested data being materialised with appropriate corrections applied. The chunks themselves will have methods which permit data to be accessed in generic ways via mathematical interface classes such as Vectors, Arrays and Matrices.

It is quite possible that these specialised data classes will be friends of a VisSet.

For example, the internal operation of TelescopeModel.solve() (or rather, a component to model receptor gains) might look something like the following:

```
// Create integration iterators for rawdata and modeldata, selecting
// chunks of data with length inttime, with the first chunk beginning
// at start_time and the last chunk ending at end_time
uvIntegrationIterator rawit(rawdata, start_time, end_time, inttime);
uvIntegrationIterator modit(modeldata, start_time, end_time, inttime);

// For each integration, adjust gains given vectors of raw and model
// visibilities
while (rawit.next() && modit.next())

  gain.solve(rawit.vis(), modit.vis());
```

2.3.8 MeasurementModel

2.3.8.1 Attributes

After instantiation, the MeasurementModel will know which sky-plane effects it needs to consider. These will either be set by arguments in the instantiation, of be fixed by the type of the MeasurementModel (assuming a hierarchy of such classes).

2.3.8.2 Methods

- `invert` – given a VisSet, `invert` produces a dirty image.
- `psf` – generate the point-spread function of a dirty image. In principle this can be a shift-variant point-spread function, to deal with effects such as primary beams and some mosaicing approaches.
- `predict` – Given a SourceModel, this predicts the visibility that would be observed for a idealised telescope. Although sky-plane effects are considered in the prediction process (e.g. primary beam), visibility-domain effects (e.g. receptor gains) are not. It could also include effects such as bandwidth and time smearing when it computes model visibilities.

2.3.9 PSF

2.3.9.1 Attributes

This implements a true point-spread function, which may be position-dependent. Such is needed when modelling sky-plane instrument effects. In the simplest case (no primary beam correction), the PSF would embody a simple beam pattern. The next level of complexity would be to include a primary beam pattern as well.

2.3.9.2 Methods

These would include a generalised convolve function (convolve a SourceModel by the point-spread function). It is generalised in the sense that the point-spread function is position-dependent. Other methods would be required to determine whether the point-spread function was position independent, as well as other aids to support traditional deconvolution algorithms, traditional processing approaches, as well as the A-matrix formalism. We do not envisage that all deconvolution algorithms will be able to cope with all sorts of point-spread functions. We must, however, support the cases which are common in present systems.

2.4 Symmetry?

There are some basic asymmetries and blemishes in the above scheme. These include

- Although we believe all telescope properties are stored in the TelescopeModel, the handling of these effects (sky-plane and uv-domain) are split between the TelescopeModel and the MeasurementModel in a rather unnatural fashion.
- Some may claim that there should be a symmetry between SourceModel and TelescopeModel – both are determined (to varying extents) by the observation – MeasurementSet

can be seem as determining (or at least refining) the two. Such a symmetry between SourceModel and TelescopeModel may be important in some applications (e.g. astrometry and VLBI). However, such a symmetry is not present in the above classes. In particular, the VisSet is very asymmetric in this context.

Our defence against such criticism is that our aim is conventional radio interferometric imaging. We are also interested in implementing the current standard algorithms (although with some eyes to the future). We believe the aims of our main customers and the realities of current algorithms justify these asymmetries. Furthermore, it may be that this asymmetry reflects our primary interest, *viz.,* determining the SourceModel; in other application domains, such as geodesy, a framework in which the TelescopeModel is emphasised might be more appropriate. We suggest that this approach, where the basic classes are reused in a different framework, might be more appropriate in order to handle diverse requirements, rather than attempt to develop a completely general, symmetrical framework.

2.5 Examples of schemes for calibration/self-calibration and source modelling

`SourceModel.update()` etc., is used for source modelling, including deconvolution. `VisSet.update()` is used to recalibrate *uv*-data. This typically involves updating an associated TelescopeModel by invoking the `TelescopeModel.solve()` method, and then using the Telescope Model to generate corrected *uv*-data.

In simple imaging schemes, the update of VisSet and SourceModel would probably take place independently. In more sophisticated schemes such as difference mapping, `VisSet.update()` might be invoked from within `SourceModel.update()`.

2.5.1 Simple imaging

```
// Simple CLEANing.  This example makes an image and deconvolves; this
// approach allows deconvolution of existing images when the original
// visibility data are no longer available.

// Instantiate uv-data set from extant MS and TM
VisSet uvdata(ms, tm);

// Instantiate Measurement model
MeasurementModel mm();

// Create dirty image and PSF - maybe merge these in practice
Image dirty = mm.invert(uvdata, gridparms...);
PSF psf   = mm.makePSF(uvdata, gridparms...);

// Instantiate a CLEAN deconvolver with appropriate parameters
CLEANDeconvolver deconv(niter, loopgain...);

// Construct a source model
SourceModel src();
```

```
// Update the source model by CLEANing
src.update(dirty, psf, deconv);
```

2.5.2 Imaging with self-calibration

```
// Self-calibration, starting with an initial source model.

SourceModel initial_source_model(fluxes, positions);
CLEANDeconvolver deconv(niter, loopgain);
SourceModel src();
MeasurementModel mm();

// Instantiate uvdata from extant MS and TM
uvData uvdata(ms, tm);

// Do self-cal with initial model
uvdata.update(initial_source_model, mm, calparams);

// Update source model using calibrated data
src.update(uvdata, mm, deconv);

// Self-cal/imaging loop
Bool converged;
do until (converged)

   // Update the uvdata...
   uvdata.update(src);

   //... and the source model
   src.update(uvdata, mm, deconv);

   // Some measure of convergence (Illustrative only!)
   converged = (src.OK() && uvdata.OK() && tm.OK());
```

2.5.3 Internals of `SourceModel.update()` in an MX-like scheme

```
SourceModel.update(
                   VisSet uvdata,
                   MeasurementModel mm,
                   Deconvolver deconv
                  )

// The internal details of src.update in the last example might look
// like the following in the case of MX.  This assumes that the
// deconvolver is a Clark CLEAN, but others could fit in this
// framework, so long as they do not depend on a positivity constraint.

// Declare residual uv-data and residual dirty image here, because
```

```
// they will first be used in the major cycle loop
VisSet resdata;
Image resdirty;

// Generate the point-spread function
PSF psf = mm.makePSF(uvdata, gridparms...);

// Major cycle loop until deconvolution finished.

while(!deconv.done())

   // Form residuals
   resdata = uvdata - uvdata.dup(*this, mm);

   // Make residual image
   resdirty = mm.invert(resdata, gridparms...);

   // Do image-plane deconvolution using deconvolver in image with psf
   this->update(resdirty, psf, deconv);
```

2.5.4 Internals of `VisSet.update()`

```
VisSet.update(
                 SourceModel srcmodel,
                 MeasurementModel mm,
                 CalibrationParameters calparams
                 )

// Self-calibration - VisSet.update() internals.
// Calibration for a particular effect is achieved by comparison of
// data predicted by a source model (which is nominally "perfect") and
// data which reflect the current state of calibration (in the case of
// incremental calibration), which may be raw, observed data.  The
// comparison and determination of corrections is performed by a
// Telescope Model which is associated with the uv-data.

// Model data are generated by "measurement" and corruption (including
// image plane effects) of a source model.
VisSet modeldata = this->dup(srcmodel, mm, calparams);

// These are to be compared with data corrected for effects up to and
// including the one to be calibrated; these are generated here.
VisSet cordata = this->dup(calparams);

// TelescopeModel.solve() is invoked to compare these and determine
// solve for corrections.
this->tm.solve(cordata, modeldata, calparams);
```

2.6 Some outstanding issues

- Our schemes need more breadth and depth, but the emphasis should initially be on depth, with an early prototype. The mechanism for handling individual components of the TelescopeModel should be tackled, fairly soon, with a few useful examples.
- Should take account of existing tools (eg for CLEANing, gridding), and use these if appropriate; otherwise specify changes in infrastructure.
- As we go deeper we have to consider interfaces to data; a number of issues have to be addressed and resolved here, such as to what extent can we use tables?

In broadening the scope of our schemes, we suggest exploration of the following problems in addition to those already described:

- Difference mapping and Mosaicing – we believe these are handled in a straightforward way by the above scheme.
- Combining instruments - a critical test. We envisage a VisSet which contains multiple MeasurementSets and TelescopeModels, although only a single MeasurementModel is required.
- Polarimetry

3 The Gospel According to Brian

Everything seems reasonable to me except the following:

- I assume that psf.response returns the sm convolved with the psf, and psf.residual subtracts the dirty image from that. Am I right?
- Bob asserts that tm.update() is always only called for one tc at a time. If so, it's clearer (IMO) to just call tm.gettc.solve.

Default constructors, assignment operators, copy constructors etc are all supposed to have been written, const is ignored, etc. This is only pseudo-C++ - i.e. it is only meant to be illustrative, not used directly.

```
// A MeasurementSet is just a table in which certain column names have
// predefined meanings. Examples of such columns include:
// time          Double    (midpoint of the integration? Start time?)
// duration      Double
// ...

class MeasurementSet : public ReadOnlyTable {
public:
    MeasurementSet(String filename);
    MeasurementSet(Table tabledata);
    //... other constructors ...
};
```

A standard interface to visibility data. It is important to decide what can vary per row. The more general, the more tedious it is to deal with.

```
class VisSet {
public:
    // All members are logically abstract (really would be letter envelope)
    MeasurementSet asTable();
    VisSet copy(Bool deep=False);
    VisSet select(TableExpr selection);

    // Inquiry
    uInt nRows();               // better name anyone?
    uInt nSignals();            // normally number of polarizations
    uInt nSpectralWindows();

    // Variable part (can vary per row)
    uInt nChannels(uInt rownr);
    Vector<SignalType> signals(uInt rownr);
    Matrix<Complex> visibility(uInt rownr); // nsignal*nchannel
    Matrix<Bool> flags(uInt rownr);         // nsignal*nchannel
    Vector<Double> frequencies(uInt rownr);
    Double frequency(uInt rownr, uInt channel);
    // ...
};
```

A MSVisSet is a VisSet in which calibration is applied "on the fly" from an associated telescope model. The data isn't actually inside the MSVS, instead the data is in an associated MS. Thus the MSVS is a lightweight "handle". MSVisSet is a pretty ugly name.

```
class MSVisSet : public VisSet { // calibrated on the fly
public:
    MSVisSet(MeasurementSet ms, TelescopeModel tm);
    const MeasurementSet &measurementSet(); // Can only read data
    TelescopeModel &telescopeModel();       // can modify tm
    void update(SourceModel, MeasurementModel, Calparams);
    void update(Calparams);
private:
    // exposition only
    MeasurementSet *ms;
    TelescopeModel *tm;
};
```

A TableVisSet is a VisSet into which the data has been "deeply" copied into a local table - for instance when the calibration has been irrevocably applied to the data, or from some simulation. Unlike other VisSet's, the TableVisSet may be modified (by getting and setting it's table). The Table that underlies the TableVisSet must contain columns that correspond to all the public data access functions of VisSet. For convenience, "set" functions would probably also be provided so you wouldn't have to write to the TableVisSet through the table interface.

```
class TableVisSet : public VisSet {
public:
    TableVisSet(Table data);
    Table getTable();
    void setTable(Table newvisdata);
```

```
private:
    // exposition only
    Table data;
};
```

A ModelVisSet presents a visibility interface to a (image-plane) source model (image or source list).

```
class ModelVisSet : public VisSet {
public:
    ModelVisSet(SourceModel, MeasurementModel, VisSet);
    ModelVisSet(SourceModel, MeasurementModel, TelescopeModel simulated,
parms); // simulated
    SourceModel sourceModel();
    MeasurementModel measurementModel();
    VisSet visSet();
    void reset(SourceModel, MeasurementModel,VisSet);
    // set returned VisSet to be "actual - model"
    void returnResiduals(Bool);
private:
    // exposition only
    SourceModel *model;
    MeasurementModel *predictor;
    VisSet *paraform;
};
```

A VisSetIterator steps a VisSet "cursor" through a larger VisSet. This class is written as if one class can step through in any order, an alternative is to express this constraint through derivation. The iteration semantics would really be richer.

```
class VisSetIterator {
public:
    VisSetIterator(VisSet container, Selection,
   VisSetIterationType = INTEGRATION);
    void start();
    Bool atEnd();
    void next();
    VisSet operator()(); // i.e. the cursor
};
```

A SourceModel represents astronomical sources before observation (i.e. not convolved with a PSF). An abstract base class (logically; it might be implemented as a letter/envelope). Time independent?

```
class SourceModel {
public:
    Image toImage(gridparms);
    Complex fourier(uvw, Frequency, Stokes, Time);     // UV-plane

    uInt ncomponents();
    ParamaterizedComponent operator()(int number);
};
```

Paramaterized source list; a CC list is a subset of this.

```
class ParamaterizedSourceModel : public SourceModel {
public:
    ParamaterizedSourceModel();
    void addSource(Position, Float totalFlux, Stokes=I, SourceType=POINT,
   SpectralIndex=FLAT);
private:
    // exposition only
    Vector<ParamaterizedComponent> sources;
};
```

A SourceList based on an image, e.g. using a maximum entropy image as a model.

```
class ImageSourceModel : public SourceModel {
public:
    ImageSourceModel(Image, SpectralIndex=FLAT);
    Image image();
    void setImage(Image);
private:
    // exposition only
    Image *data;
};
```

A SourceModelGroup is a SourceModel which consists of multiple other SourceModels.

```
class SourceModelGroup : public SourceModel {
public:
    SourceModelGroup(SourceModel, SourceModel, ...);

    uInt nmodels();
    SourceModel first();
    SourceModel next();
    void remove();              // Get rid of current sm
    void add(SourceModel);      // Add another one
private:
    // exposition only
    List<SourceModel> models;
};
```

The MeasurementModel encapsulates the interferometric image formation. There will probably be a hierarchy of such classes; from the simplest "UVMAP" type process, through more complicated mosaicing processes.

```
class MeasurementModel {
public:
    void invert(Image &dirtyImage, PointSpreadFunction &dirtyBeam,
VisSet uvdata, GridderParams);
    VisSet predict(SourceModel, VisSet paraform);
private:
    // exposition only
    GridTool gridder;
    FFTServer ffts;
};
```

We have a PSF class to allow for position (and time?) dependent PSFs. Otherwise it could just be an image.

```
class PointSpreadFunction {
public:
   Image operator()(Position);
   Image response(SourceModel);
   Image residual(SourceModel, Image dirtyImage);
};
```

A TelescopeModel consists of telescope state information (e.g., antenna stations) as well as a set of telescope components which contain calibration information and algorithms.

```
class TelescopeModel
public:
    TelescopeModel(Vector<TelescopeComponents> calibrations);
    // ize or ise? Punt
    void init(MeasurementSet);  // real tm
    void init(Table stations, Table observations, ...); // simulated tm

    void solve(MSVisSet, MeasurementSet, Calparms);
    void solve(Measurementset, Calparms);
    MSVisSet apply(MeasurementSet);
    MeasurementSet corrupt(VisSet); // Do we need otf corruption?

    Vector<Float> uvw(Time, uInt ant1, uInt ant2);
    Vector<Float> uvwLambda(Time, uInt ant1, uInt ant2, Float frequency);
    // ...

    uInt nTC();
    TelescopeComponentType tcIsA(uInt num);
    ReceptorGainTC receptorGain(uInt num);
    BandPassTC bandPass(uInt num);
    // ...

    Table stations(Time);
    //...

    friend class TelescopeComponent;  // Because it caches info in the TM
};
```

A TelescopeComponent encapsulates some sort of calibration. Probably an abstract base class.

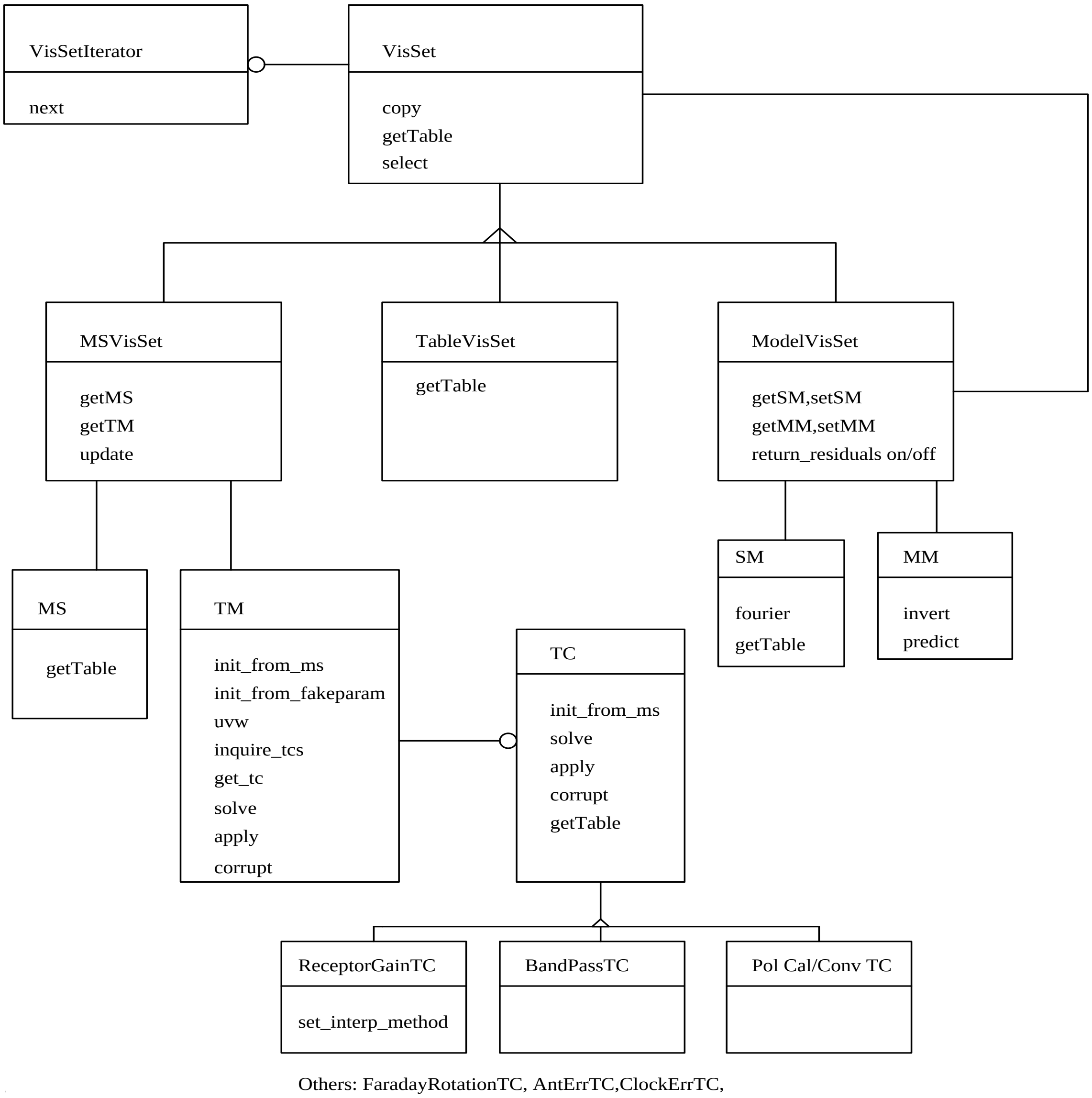
```
class TelescopeComponent {
public:
    initialize(MeasurementSet);
    initialize(Table);

    void solve(SourceModel, VisSet, calparams);
    MSVisSet apply(MeasurementSet);
    MeasurementSet corrupt(VisSet);

};
```

ReceptorGainTelescopeComponent is the usual antenna-based gain solution.

```
class ReceptorGainTelescopeComponent : public TelescopeComponent {
public:
    ReceptorGainTelescopeComponent(Interpolator, Float gainInterval);
    Table gainTable();
    void setGainTable(Table);
};
```

4 The Gospel according to Bob

4.1 The Universe in a Picture



4.2 An example of how we might calibrate ATCA data.

Fill the MS from tape.

```
MeasurementSet ms("/dev/nrst8");
```

Instantiate the telescope components that we want.

```
ReceptorGainTC   gain("interval=2min,interp=1st_order,method=L2");
LinearFeedTC     pol("interval=24hr");
BandPassTC       bp("interval=24hr);
```

For ATCA calibration, the receptor gain, polarisation solution and bandpass calibration are performed in a coupled fashion. Make a composite ATCA-specific TC, which contains the gain, polarisation and bandpass TC as its sub-components. An ATCA_TC also has the intelligence to solve for XY phase from on-line measurements.

```
ATCA_TC atca(gain,pol,bp,"xy_phase_interval=20min");
```

Create a telescope model and initialise it.

```
TM atca_tm(atca);
atca_tm.initialize(ms);
```

Make a visset.

```
MSVisSet vis(mm,atca_tm);
```

Solve for the XY phase from the on-line measurements in the measurement set.

```
vis.update("use atca;mode=xyphase");
```

Find the bandpass function, and then the polarisation solution. Both steps determine (unscaled) receptor gains as well. In these steps we assume that the calibrators are point sources at the field centre with flat spectra. I assume here that because this case is so common, that "update" can assume this to be a default model if a given model is not passed in.

For these updates, the polarisation of the bandpass calibrator is irrelevant. The polarisation of the secondary is determined from the data.

```
vis.select("source=bandpass_cal").update("use atca;mode=bp");
vis.select("source=secondary_cal").update("use atca;mode=pol;submode=qusolve");
```

We need to scale the gains and bandpass to set the true flux density scale (with frequency). We need to use a calibrator with known flux density (and spectrum) to determine the absolute flux scale. I assume we have a library of standard models, and that we can extract a model of the flux calibrator from this library.

```
SourceModel flux_cal_sm =
get_sm_from_library(vis.select("source=flux_cal").name);
MeasurementModel mm;
vis.select("source=flux_cal").update(flux_cal_sm,mm,"use atca;mode=flux");
```

4.3 Telescope Models

Telescope Models (TMs) are constructed by giving it an ordered list of Telescope Components (TCs). In this initial state, the TelescopeModel is fairly dumb. To give it some information on the actual telescope to be modelled, the TelescopeModel must be fed a Measurement Set (MS) (or possibly several MeasurementSets) or a set of fake parameters (e.g. fake parameters which define where the antennas are, where and when the antenna points at something, etc) – FakeParam. The TelescopeCompoents in turn will be fed these MeasurementSets or FakeParams.

The TelescopeModel and the TelescopeComponents will copy any data from the MeasurementSet and FakeParams that they think they need to remember.

The TelescopeModel, and all the TelescopeComponents have solve, apply and corrupt methods. The solve method in the TelescopeModel will take some extra parameter list (calparams) which it uses to locate the appropriate TelescopeComponent to invoke.

Normally the TelescopeComponents are instantiated with the parameters needed to define how they work (e.g. solution scheme to be used, if there is a choice, or the interpolation scheme, etc). However if it becomes necessary to change these, the TelescopeComponents can be fished out of the TelescopeModel, and their state changed (through some "set parameter" interface).

Most objects will have a getTable method, to return some underlying table. This interface is to allow browsing by the table browser, rather than being the normal access method to internal data.

4.4 Questions

- For implementation, is the MeasurementSet a pure table (i.e. not a distinct class).
- What are the parameters that would need to be specified to create a fake telescope model.
- For implementation, is the "update" method a member function of VisSet rather than MSVisSet. If so, the TableVisSet and ModelVisSet would throw exceptions if their "update" method was called.

5 The Gospel according to the Plodders

While the three saints were deliberating at Jodrell, a second group attempted to flesh out some of the methods suggested by them, The following section represent their combined efforts (this may possibly be nothing more than insane babbling!

5.1 Some Examples

Here is the first result.

We are expanding on the examples from p 8/9 of the april-94 Dwingeloo document.

The first example expands on some of the code from p 8.

```
  VisSet visset(ms,tm); // construct a VisSet from extant MS and TM

  Image image(imageparms);         // ???

  MeasurementModel mm( const &taperparms, const &image);
// MM does have a state now, viz.:
```

```
// data:imageparms = imageparms
// data:taperparms = taperparms

// here we can also perform some checks, e.g.
// - taper vs. cellsize
// - image primenumber FFT
```

Now let's work out `mm.invert(const &visset, &image)`

```
  GridTool gt(cellsize, imagesize, maxUVW);
  Array array(imageparms=Iposition); // ???
  Array wts(); // ???


  i=0;
  VisSetVecIter vsi(visset);
  vsi.origin();
  while (!vsi.pastEnd())
    // compute vis weights (from taper etc.)
    gt.grid(array, wts, vsi.VisandWt());

    FFTServer fft(array);
    image.outarray(i) = fft.fftmethod(array);

    vsi.next();
    i++;
```

In the second example we expand some of the self-cal code from p.9

```
  VisSet visset(ms,tm);
  MeasurementModel mm(...);      // there are two(?) possible constructors
  SourceModel src(flux,position); // let's take a central point source
  visset.update(src,mm,calparams);
```

Let's expand on `visdata = dup(src,mm,calparams);`

```
// let's expand on part of this (see also p.11)
//
VisData visdata();
VisDataIter vdi(VisData);
VisSetIter  vsi(*this); // refer to the current VisSet

  vsi.origin();
vdi.origin();
while(!vsi.pastEnd())
vdi.SetVis(mm.predict(src,vsi.uvw());
vsi.next();
vdi.next();
```

```
//  tm.solve(cordata,modeldata,calparams)


VisData visdiv = cordata/modeldata;
visdiv.solve(calparams, cordata.tm()->gains());


// if we expand on this:

VisData::solve(&calparams, &Vector<Complex>)

// solve a linear system (or something fancier)
// with the understanding that
// visdiv := cordata/modeldata := data_ij/model_ij
// (i,j) is an receptor (ant) pair
Arg(data_ij/model_ij) = phi_i - phi_j
|data_ij/model_ij| = ln(g_i) + ln(g_j)
```

5.2 Simulated Measurement Set

A SimulationMS (MeasurementSet) is needed for modelling. So, what are the basic elements needed in such a SimMS?

- MS Time,IntegrationTime,FreqId,SourceId,Vis,Flags,Weights (Note that by 'Time' we mean the centroid on the integration interval)

  E.g. an instatiation of an obvious derived class: ? wsrtMS jansTM(ms ,tc1,tc2,tc3);
- TM/TE telescope elements, meant for slowly varying variables: Time,FreqId,AntPos,BeamShape,PointerErrors, chars(MountType,PolTypes,....)
- TM/FR frequency table Freq(nchan) lookup table, or something simpler, optionally perhaps VLSR(nchan). Perhaps also ChannelWidth(nchan)?
- TM/SU Source lookup table RA,DEC,RA_0,DEC_0,Name,VLSR, etc.
- TM/GN Antenna based Gain Table - may vary fast Time, Gain
- TM/BL Baseline based Gain Table (Vout = (Vin+GA)*GM Time, GainAdditive, GainMultiplicative

5.3 The SourceModel (cf. pp8-10 of the Dwingeloo document)

SM member functions we need:

- - various constructorscan be used: an image, a clean-component list, a parameterized list of models (e.g. gaussians, lorenzians etc.)
- - convert to image (need an interpolation function, or a convolution; in the latter case this also requires an extra inverse tapering in the uv plane)

- - convert to list
- - transform to uv - plane (dft for SMList, fft for SMpixel)
- - merge (this compresses the list/adds the images, where possible)
- - link (this is meant for special cases where you want to preserve the detailed implementation of the SM<Type>, but may not be so efficient for same-type SM's)

an SMlist will contain a list of pixels,

an SMpixel will contain a grid/image

Image pixels are each a vector of data, e.g. Polarization Intensity, Position Angle, Total Intensity

5.4 BIMA windows

To remind ourselves what we mean by a spectrum being split up in windows, a window is defined as a contiguous section in the spectrum, that can be fully identified with a reference channel and frequency, channel spacing and bandwidth. There is no formal connection between individual windows.

Examples of windows can be to store the contiguous channels from a spectrum in the first window, and the wide band channel in another one. Or, as in the case of BIMA, store the USB (upper side band) and LSB in the the first two windows, followed by their wide bands. In other modes, the BIMA corellator is able to handle up to 16

In an advanced calibration method we will want to combine the gains from corresponding channels from the LSB and USB and first fit a so-called double-side band difference, followed by one to the sum (it can be proven one gains sqrt(2) in S/N).

Pseudo Code :

```
// define the telescope components
  List<TC> tclist(); // first define a list of TCs

  ReceptorGainTC tc1(); // instantiante as many as you want
  tc1.set_interpolation(Spline); // and set special properties we need
  tclist.add(tc1); // add it the list
// and repeat previous 3 lines for other TC's

// there is the question here how one syncs multiple
// TC's when iterating over a BIG dataset (SM)

  TelescopeModel tm(tclist); // now finally, make the TM from this lis

  MeasurementSet ms(); // this gets the standard columns
  fill_miriad(ms,fillpars); // fill it with data from a miriad dataset
// if we expand on this, we may find:
Table mstab = ms.getTable(); // get internal table
mstab.addCol("cormode",0,Iteger); // add BIMA specific
        mstab.addCol("corfs",1,MaxWin,Real); // columns

  tm.init_from_ms(ms); // how to do multiple ms's?
```

from the code:

```
TelescopeModel tm(tclist);
```

expand on the ReceptorGainTC member of this list:

```
tm.init_from_ms(ms)
{
// INSIDE: we're expanding from this member function here
tclist.origin();
while (!tclist.pastEnd())
   tclist.next()->init_from_ms(ms);
   // INSIDE: expanding on previous line with tcrg being the TC

      int maxchan = ms.get_maxchan(); // get global maxchan, since it
// may vary (hopefully not)
          Vector<String> rn = ms.get_receptor_names();
              Table gtable("...",Table::New);
      gtable.addcol("time",0,Double);
      for (int i=0; i,rn.len(); i++)
                  gtable.addcol(rn[i],1,maxchan,Complex);
      Vector<Complex> gain = tcrg.getgains(time);
      tcrg.putgains(time,gain)
// INSIDE this:
addrow(time,gain);
```

Here is an example how calibration is applied to a source.

```
// define the telescope components
  List<TC> tclist(); // first define a list of TCs

  ReceptorGainTC tc1(); // instantiante as many as you want
  tc1.set_interpolation(Spline); // and set special properties we need
  tclist.add(tc1); // add it the list
// and repeat previous 3 lines for other TC's

  MeasurementSet calsrc, progsrc;
  TelescopeModel tm(calsrc,tclist);
  MSVisSet vs(calsrc,tm);
  vs.update(sm,mm);
  MSVisSet ss(progsrc,tm);
```

6 Source Model

In most reduction packages, the Source Model (SM) used for SELFCAL is just a list of CLEAN components, or even an Image. Some packages, like NEWSTAR, OLAF (and the Caltech VLBI reduction package?), have pioneered the concept of a 'parametrised' source component, which has parameters for source shape, polarisation, spectral index, time-variability, etc. We think that there is little doubt that the AIPS++ Source Model must contain both concepts, and leave room for more.

At the highest level, the SM is fully deconvolved, and also otherwise independent of a particular telescope. (We have toyed with the idea of making such a Global SM into a different class from a telescope-dependent Local SM, but our reasons for doing so gradually evaporated during the design process). Such a 'top-level' SM does not necessarily have to be a complete representation of the observed field; for instance, the low-level extended emission may be missing, or all the point sources below 2 sigma.

The SM may be successively 'corrupted' with image-plane instrumental effects like primary beam attenuation (incl pointing errors), off-axis polarisation, non-isoplanaticity, etc. The corrupted SM may either be used to 'predict' a uv-model for SELFCAL, or to 'solve' for image-plane corrections by comparing it with a more or less dirty image.

6.1 Tentative list of methods

The description above leads immediately to a tentative list of methods for the SM class:

- **ConvertToImage:** Although a SM may contain an Image, it is certainly not an Image itself. However, it can always be converted to an Image. (NB: It may be convenient to define a default or 'preferred' grid for this purpose.)
- **Generate:** There will be many ways to generate or refine SM's, usually from Images (e.g. by a form of CLEAN), but quite possibly also in other ways (e.g. manual input by the user).
- **Merge:** One (itself) or more SM's may have to be be merged, for instance to collect multiple CLEAN components at the same position, or to make a full-polarisation SM, or to attach spectral indices etc.
- **FourierTransform:** This is used to predict the uv-model value for a particular VisSet, at the request of the Measurement Model. The relevant uvw-coordinates, observing bandwidth and integration time must somehow be available. **Etc:**

6.2 Source Components

An important element of the SM is the parametrised SourceComponent. Such an SC could have the following parameters:

- Position (l,m)
- One or more Flux indicators, which may be a linear combination of the 4 Stokes parameters I,Q,U,V. For instance I+Q.
- Frequency dependence (e.g. spectral index)
- Rotation measure (intrinsic to the source, not ionosphere)
- Time variability (intrinsic to the source).
- Etc

There are various types of SC's, for instance at least three types of Clean Components (see below), SC's with various shapes, etc.

NB: We are rather proud of our solution for the *Flux indicators*. Ideally, there should be 4 separate ones for the 4 Stokes parameters. Each indicator (column) would then be characterised by 4 complex numbers: 1.0.0.0 (I) 0.1.0.0 (Q), 0.0.1.0 (U) and 0.0.0.i (V). However, when a SM is generated from an WSRT 'XX-map', which represents I+Q, the Flux would be characterised by 1.1.0.0.

6.3 A word about Clean Components

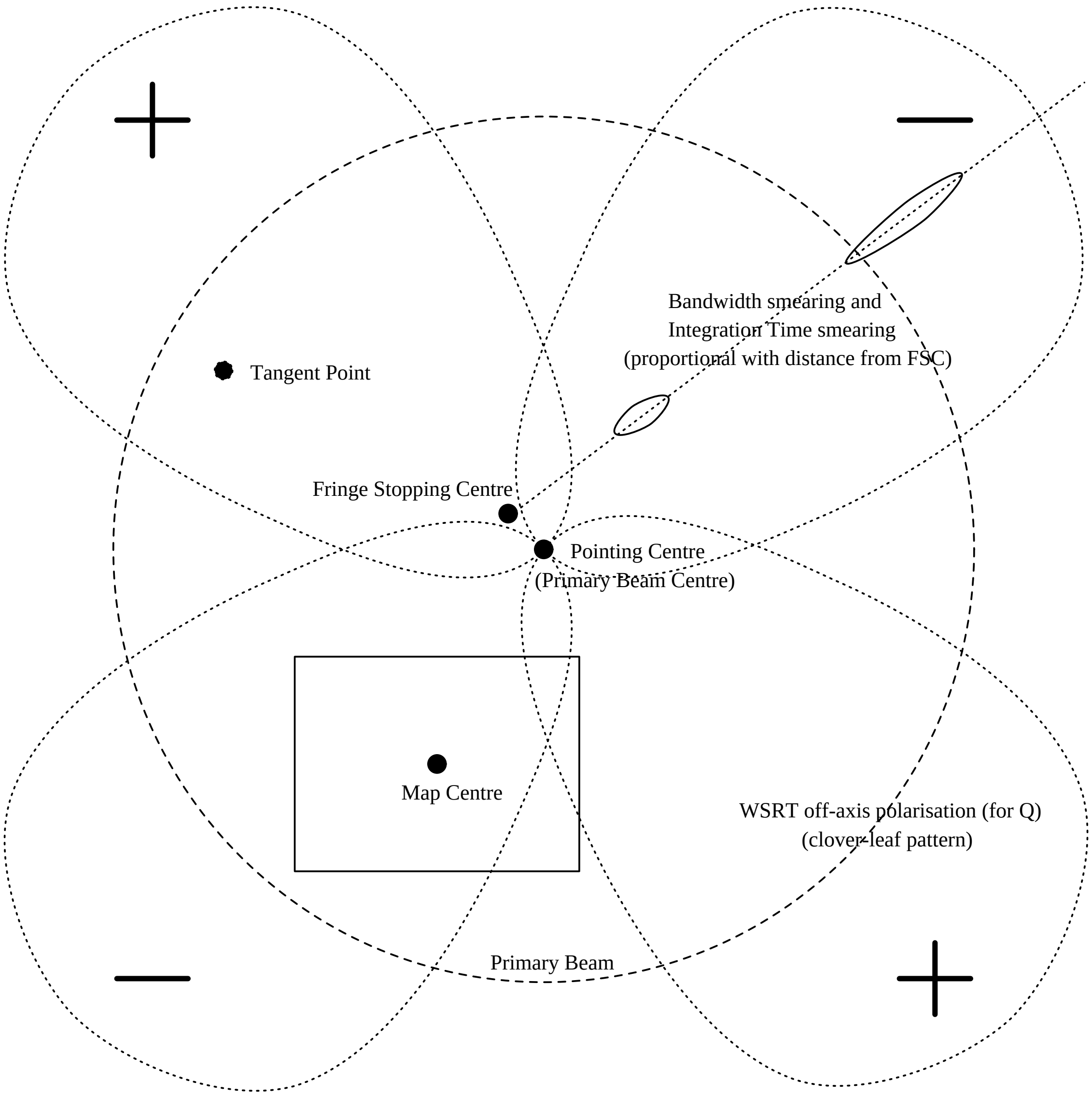
CLEAN components are easy to generate (with CLEAN), but they are not fully deconvolved. This means (roughly) that a point source, which should ideally be represented by a single SC, is usually represented by multiple Clean components. The two main causes for this are the following:

- Standard CLEAN (in which sources are subtracted in the image plane) will interpret point sources as extended sources if the PSF (beam) varies over the image. The latter is the case for for bandwidth smearing and integration-time smearing (see fig \refsm-visaid), or for non-coplanar arrays. The problem disappears if the sources are subtracted from the measured uv-samples (but not if they are subtracted from gridded uv-data).

- A point source between grid points has to be approximated by a number of Clean Components on grid points.

When SC's are used to 'predict' a uv-model for a particular uv-sample, the observing bandwidth and integration time are taken into account. Standard Clean Components have to be predicted assuming zero bandwidth and zero integration time.

Eventually, algorithms should be developed to convert (groups of) Clean Components into fully deconvolved SC's. Ideally, Clean Components should gradually be abolished as our deconvolution techniques improve, but they cannot be ignored yet.



The above figure provides a visual aid for thinking about the Source Model. The reader should not take the actual shape of things too literally. Centered on the Pointing Centre are the primary beam pattern and the 'clover-leaf' pattern for off-axis instrumental Q-polarisation for the (equatorially mounted) WSRT. The magnitude of this effect is rather frequency-dependent, which may be caused by standing waves. The pattern for Stokes U is similar, but rotated by 45 degr. Beam-smearing effects are centered on the Fringe Stopping Centre, which does not always have to coincide with the Pointing Centre. This can be accidental because of a pointing error, or on purpose because of a special observing mode. The Map Centre is arbitrarily specified when making a map. The Tangent Point is the point where the map plane 'touches' the celestial sphere.