The Imaging Model Design

April 18, 1992

S. Bhatnagar
M.A. Holdaway

1 Introduction

This is the first attempt at a detailed design of the Imaging Model object (see AIPS++ Memo 103, "Calibration, Imaging, and Datasystems for AIPS++ - Report of the meeting held at Green Bank, WV, Feb 1992"). This document ellucidates the design principles and goals which guide us in the development of various Imaging Models (IMs). We present a fairly detailed design of the Interferometer Imaging Model (IIM), and we show how the Non-Coplanar Imaging Model and the Non-isoplanatic Imaging Model can be built out of the IIM. We show that the Single Dish Imaging Model is applicable to data from only one sky pointing at a time, and that Mosaicing must be used to make single dish images. We argue that Mosaicing is not an Imaging Model, but an Imaging Model Manager, as it utilizes the Imaging Models associated with the YegSets it processes.

2 Design Principles

Some of the design principles were in our minds before we started this detailed design, but others emerged only after seeing commonalities between the various types of Imaging models. Our basic guiding principles and assumptions are:

- the Imaging Model must support DFT and FFT algorithms in a flexible and easy to use manner.
- the extra baggage that goes with the use of FFT (Gridding, Convolution function) should not get instantiated unless asked for. The details of the FFT should be out of sight but accessible.
- control of parameters which effect the output should be complete and flexible. Some of these parameters may exist in some of the objects used inside the IM, but still control from the IM should be provided.
- the IM should be able to handle Spectral Line data in a flexible manner and with in the frame work of the Image class
- the IM should provide an interface which is as general as possible so that it will be useful in more complicated cases like Non-isoplanatic Imaging and Mosicaing, but the default behavior should be an "easy to use" IM .
- if possible, a clean break should be made between the IM and the operation of selecting the visibilities.
- the data used by the IM (YegSets, Images) and the instantiation of the cooperating objects used by IM should be dynamic. Some heuristics must be at work to determine when best to instantiate FFTServer and when to delete it.
- a YegSet can generally have more than one associated IM (ie, a YegSet can be made up of data from a number of Telescopes).

- the YegSet or YegSetView which is passed to the Interferometer ImagingModel's `Invert` and `Predict` must contain data from a single pointing on the sky.

- images are based on arrays which are dynamically allocated. If the only information in an image is the header (ie, an image filled with 0.0), then no space need be allocated for the array. Similarly, spectral line cubes should allocate memory for each plane as the cubes are filled.

Using these guidelines, we designed the IntImagingModel which we present below. We later show that it is flexible and can be a fundamental building block in more complicated Imaging Models.

3 The Interferometer Imaging Model (IIM)

A draft of the IIM header is presented in Appendix A. Here we discuss some of the issues in our design.

3.1 General Layout of IIM

The primary features if the IIM are the `Predict` and `Invert` methods which were discussed at Green Bank. The `Predict` method takes a model Image and a template YegSet and predicts model Yegs. Similarly, `Invert` takes a data YegSet and a template image and produces an image:

```
void myIM.Predict( SkyModel, YegSet );
void myIM.Invert( myYegSet, Image );
void myIM.Invert( myYegSet, Image, PSF);
```

The second form of Invert will also procude the PSF.

The mechanism of getting the result used here is where the method takes in a template (Template Image for `Invert` and Template YegSet for `Predict`) and fills them with the result. The templates themselves have no data. In this approach:

- the Template and the actual Image (or YegSet) which carry the data in the rest of the code are the same instantiation of the object.

- provides a mechanism to get the PSF without resorting to Collection of Images at this level

- the return value of the method is left free from some more information that it may like to tell (some error condition etc.)

To generate a template sky plane image to be passed to `Invert`, we could implement a method of IIM:

```
Image IntImagingModel::TemplateImage( YegSet, Cellsize, Imagesize, Shift, PixType );
```

The header of the template image is all that is really needed (the array never needs to be allocated). We considered making Cellsize, Imagesize, Shift, and PixType data members of the IIM with the option of overriding the member values via the use of the template.

When `DoFFT` is true, `Invert` will call the private member function of IIM `DoInvertFFT`, which in turn will use the IIM's GridTool and FFTServer, and the Convolution Function's gridding corrector. But the first thing to do is to call a method of YegSet which requests the data to be ordered in an appropriate manner:

```
ysCursor.Order( coordOrder );
```

This request for order will be optional as disussed below. Next, the `FFTTool::ConjugateImage` method creates a Fourier plane image with proper header and axes from a template sky plane image. The input Template Image will be real. While making the Fourier plane, either a new temporary compelx image can be made and later (after FFT etc.) the real part copied in to the input Image, or the input Image be transformed to a complex image, used in Gridding, FFT etc. and later transformed back to a real Image before exiting. This may be a little memory effecient. (There will be a method of FFTServer which will do the complex-to-real FFT). This complex Fourier plane image can then be passed to `Gridtool::Resample` with the YegSet.

```
FilledImage FourierPlane FFTTool.ConjugateImage( TemplateSkyPlane );
myGridTool.Resample( myYegSet, FourierPlane);
```

The `Resample` method will automatically shift and rotate the (u, v, w) coordinates as specified by the header information in the YegSet and the Fourier plane image. This will be especially important for the more complicated "multi-plane" imaging methods such as fly. Currently, `Resample` goes both ways, gridding or degridding, depending upon the order of the arguments. The gridded visibilities on the Fourier plane image can then be FFTed and corrected for the gridding convolution function.

3.2 Some Details

In order to make an image from the data in the YegSet, the IIM needs to get the appropriate sky coordinate projection and the phase reference sky position for which the visibilities have been calibrated. This information can be covertly channeled to the appropriate images in the `TemplateImage` method.

YegSet selection is discussed in Dave Shones "YegSet Data Selection and Aggregation" Note. Most of this selection belongs outside of the IIM. In more complicated models, some selection YegSet selection must be carried out inside the Imaging Model, but still one level above the IIM.

The YegSet must provide a means of ordering the visibilities. While XY ordered visibilities will not be required for typical image sizes on typical computers, very large images on small memory computers may require XY ordering. Also, some operations will be more efficient if the Yegs are ordered by spectral channel, frequency, or Stokes.

In the absance of explicit ordering request, some heuristics can be used to determine the "best sort order" inside IIM.

The YegSet must provide a means of asserting that the visibilities are valid. For example, the IIM will need to use a service called `AssertOnePointing`. A member function of IIM called `IsValid` would miz and match the appropriate YegSet `Assert` methods.

YegSet must deliver vectors of calibrated visibilities, vectors of U, V, W, and vectors of Weight.

When `Predict` is called to predict a YegSet, the values of U, V, W, time, etc, will be the same as those in the template YegSet. Logically, the predicted YegSet is an independent entity. Physically, the predicted visibilities should be stored in parallel with the template visibilities so that all common data can be shared. This is essentially the same problem as the TelModel with several different kinds of gain tables; when one gain table has two versions, logically we have two different TelModels. but physically we have only one copy of the nonredundant gain tables.

FFTServer, an ImageTool, should be able to do the following:

- Determine the FT direction from image header information.
- Generate a correct header and image when no template output image is passed.

- Correctly deal with the case where the input and output images are of different size (this requires an output image to be passed in).
- An FFT object can be initialized for some size transform and precalculate several numerical factors and store them. This will save time when several FFT's of this size are performed.
- The FFT should have an internal switch which indicates if "many" FFTs will be performed, causing the above mentioned FFT object to be used, or "few" FFTs will be performed, probably resulting in a machine-optimized FFT to be called.

Aside from the dimensions, there is nothing fundamentally different between a 2D and a 3D Fourier transform. 3D code will treat a 2D case. It will be seen in Fly that it is convenient to implement the IIM as a 3D transform.

3.3 Spectral Line Data Sets

We stated that data selection and the IIM are separate concerns. However, spectral line data sets may require further specification.

Consider the `Invert` method. Given a spectral line YegSet only, the default is to produce a spectral line image cube. If a template cube is present, then images only for those channels or frequencies in the template will be created. If a list of frequencies is passed to `Invert`, then images only for those frequencies will be generated. If any member of the frequency list are arrays, then the data from all frequencies specified in the array will be gridded onto the same grid and transformed.

To deal with multiple frequencies (or, for that matter, multiple YegSets which use the same IIM) to produce a single image, we just need to get all the data on the same grid. `Invert` calls `DoInvertFFT`, which uses its GridTool's `Resample` to perform the gridding. `Resample` can be called a number of times as `DoInvertFFT` loops over all relevant frequencies (or YegSets) to get them all on the grid.

Since a single image plane might be generated from data measured at several frequencies, an array of frequencies must be indicated in the image header.

To handle the case of Spectral YegSet then, we envisage the following selection scheme. The "window" as suggested by Shone will be set outside the IIM to select the Channels that the user wants to process. This will mean that inside the IIM, the YegSet will present only those Channles which are to be used. The frequency list of the IIM is then set to determine "how" these channels are to be used.

So, if the channels 1,2,5,6 are selected for use and one wants to make three images - one of channel1 and channel6 each and third made by gridding the data from channel2 and 5 on the same grid, the Frequency List of the IIM will be set up as

```
FreqList[0] = 1;
FreqList[1] = 6;
FreqList[2][0] = 2;
FreqList[2][1] = 5;
```

It is obvious that the FreqList is a dynamic array of array. The IIM will step though this list and work accordingly. If the channels listed in the Freq List are absent in the YegSet, an exception will occur.

Finally, there may be some advantage to performing the FT's for all spectral channels en masse. Consider a DFT for example. If the frequencies are uniformly spaced, ie, $\nu = \nu_0 + k\Delta\nu$, then the values

of u are also evenly spaced for the different channels on one physical baseline. The Fourier transform then looks like:

$$I_k(\mathbf{x}) = \sum_j V_{j,k} e^{\frac{i2\pi(\nu_o + k\Delta\nu)}{c} \mathbf{b}_j \cdot \mathbf{x}} \quad (1)$$

$$= \sum_j V_{j,k} e^{\frac{i2\pi b_j}{c}(\nu_o x + k\Delta\nu x)} \quad (2)$$

$$= \sum_j V_{j,k} e^{iA_j \nu_o x} \left(e^{iA_j \Delta\nu x}\right)^k \quad (3)$$

$V_{j,k}$ is the visibility for baseline j and channel k and $\mathbf{b}_j$ is the physical baseline in meters, and the second step has assumed the baseline and x vectors are scalars. The *sin* and *cos* do not need to be calculated for each channel's value of u_j if we store calculated values of $e^{iA_j \nu_o x}$ and $e^{iA_j \Delta\nu x}$. This is a strong argument for a DFT which performs the transforms on the channels all at once rather than one at a time. It seems that there is no analagous advantage for an FFT.

As a simple example discussed above, to use the IIM for Spectral Line case, one does the following on a Spectral line YegSet.

```
Set up the window on the YegSet to select
channels 1,2,5,6;

YegSet.Window(MyWin);
:
:
IntImagingModel IIM();              // DFT version
Array<Array<int> FreqList;
Image   MyImage;

FreqList[0] = 1;
FreqList[1] = 6;
FreqList[2][0] = 2;
FreqList[2][1] = 5;
:
:
Set up the header of MyImage for ImgSize, CellSize etc.
:
:
IIM.Invert(YegSet, MyImage, FreqList);  // Will blow the Image
                                        // to three planes

ConvFunc MyCF(6,256,256)    // Set up ''a'' ConvFunc with
                            // SupportSize=6 and MathTable
                            // of size 256

IIM.Reset(TRUE, MyCF);      // Tell the IIM to go to FFT mode
```

```
                                    // and use this CF.  Inside IIM
                                    // the FFTServer comes into being now.
IIM.Invert(YegSet, MyImage, FreqList);
                                    // Generate 3 more planes
                                    // using FFT now

IIM.Reset(CF, 4)                    // Use a support size of 4
FreqList.Init();
FreqList[0][0]=1; FreqList[0][1] = 2;
IIM.Invert(YegSet, MyImage, FreqList)
                                    // Produce one plane
```

4 The East-West Interferometer Imaging Model (EWIIM)

Data from E-W arrays can be treated the same as a generic interferometer, so the EWIIM needs to reproduce the functionality of the IIM. But there are some imaging operations which are unique to E-W arrays. For example, images can be accumulated in real time by radial Fourier transforms. Hence, the EWIIM can be derived from the IIM.

If the visibilities are calculated in a polar projection rather than in a SIN projection, the the IIM `Invert` will correctly produce an image which is not distorted by noncoplanar effects.

While the requirements placed on the YegSet are very similar to the requirements the IIM places on the YegSet, the YegSet may want to handle the data differently; for example, (u,v,w) may be calculated on the fly rather than stored. This does not concern the IM, as the `GetUVW` methods can be polymorphic if the (u,v,w) are not stored, they are calculated and returned.

An E-W array specialist needs to consider if there are any fundamental requirements put on other classes by the EWIIM.

5 The Fly Interferometer Imaging Model (FIIM)

FLY inverts one given YegSet onto different tangent planes on the celestial sphere and thus treats the problem of non-coplanar baselines to a good approximation. To keep the image together, some book keeping has to be done (determine the shift to be applied to the UV data set for each facet, the PS associated with each facet, the info. for each facet so that they can be later projected to one plane, etc. We therefore argue that Fly requires its own IM. The algorithm used to invert on one given facet is however the IIM. (Consider the case of a Mosaic of low frequency VLA data. The Mosaic Imaging Model Manager will look to the YegSet's associated ImagingModel to determine how to invert a single pointing's worth data, which is the Fly method.)

The Fly Interferometery Imaging Model (FIIM) will use the generic IIM's `Invert` to image the visibilities, appropriately rotated upon gridding, onto each facet, and the IIM's `Predict` to calculate each facet's contribution to the visibilities. These will be cast into the framework of the FIIM's own `Invert` and `Predict`. However, these FIIM methods would generate and operate on a collection of facets rather than single image. Each facet must have an independent header and could be organized into a new kind of cube if they are all the same size. However, the facets should probably be stored in a "collection of images" which would allow the facets to be of different sizes, required by user-specified facets. In general, the facets

can overlap, leading to a new (solvable) problem of image weights. The FIIM then becomes responsible for managing its facets, facet weight images, and its facet-wise calls to the IIM's methods.

In many cases, fly's facets are 2D images. A case intermediate between fly and the full 3D imaging consists of a fly of several thiner 3D facets. This is another argument for the generic IIM being able to handle straight 3D transforms.

An implementation of the FIIM's Invert method will look something like:

```
FIIM::FIIM(int NoOfFacets, FilledImage Template, FacetImSize,
        FacetCellSize...)
{
   ImageCollection Collection( 0, 0 );

   for (i=0; i < NoOfFacets; i++ )  {
//    figure out tangent point, make i facet
      FilledImage Window = figure it out ( Template,
             FacetImSize, FacetCellSize );
      FilledImage Dirty( Window, 0.0 );
      Collection.Put (i, ''Window'', Window);
      Collection.Put (i, ''Dirty'', Dirty);
                  :
                  :
    }

    IntImagingModel MyIIM(TRUE, CF);   // The FFT version
    :
    :
};

FIIM::Invert(YegSet myYegSet, ImageCollection Collection)
{
 for(int i=0; i<NoOfFacets; i++)
 {
       FilledImage &Dirty = Collection.Get(i, ''Dirty'');
       FilledImage &PSF = Collection.Get(i, ''PSF'');
       MyIIM.Invert (myYegSet, Dirty, PSF);
// GRIDTool in Invert automatically shifts Yeg's u,v,w to match Dirty's header
 }
}
```

6 The Non-Isoplanatic Interferometer Imaging Model (NIIIM)

As discussed in AIPS++ Note 132 (April 7, 1992, Holdaway and Bhatnagar), the NIIIM will need to deal with some sort of window on the sky: the NIIIM must determine which gains are appropriate for a particular window, apply those gains to the visibilities, and `Invert`; and the NIIIM must multiply a model image by a window, `Predict`, the contribution of the source structure in this window to the measured visibilities, and un-apply the gains which are appropriate for this window from the visibilities. The window-wise

Invert and Predict operations are the same as IIM's Invert and Predict. Since the different windows require some bookkeeping external to the IIM, the NIIIM is a separate Imaging Model which *has an* Interferometer Imaging Model. The NIIIM will also have to coordinate the TelModel's ApplyGains and interpolation methods with the windows.

7 The Mosaic Imaging Model *Manager* (MIMM)

There is no such thing as a Mosaic Imaging Model. There is no YegSet which will be associated with a Mosaic Imaging Model. When a mosaic image is to be made from a number of YegSets, the Imaging Models which are associated with these YegSets must be used. Hence, we coin the term *Mosaic Imaging Model Manager.*

The Mosaic IMM uses the PB associated with each YegSet to produce a mosaic of dirty maps, a mosaic of clean maps, or a mosaic gradient image for MEM imaging from all appropriate YegSets. Conversely, given a YegSet or YegSets with multiple pointings on the sky plus an estimate of the sky brightness distribution and the PBs, the Mosaic IMM can predict the yegs in the YegSets.

Single pointing, single telescope data which need a PB correction can either be imaged via the IIM with the PB applied to the final deconvolved image by the PB ImageTool, or can be treated as a single pointing mosaic.

Mosaic requires that the data be selected by Telescope and Observed Position on the sky (TOP). This could define an aggregate which would need to be iterated through. If there were different YegSets, we would need to iterate through each of them as well.

The following is the sort of code which might be written to perform the linear mosaic algorithm, in which dirty images from a number of pointings on the sky are added together:

```
FilledImage  Mosaic::Invert( YegSet myYegs, FilledImage Template ) {
        CoordWindow Telwindow( aCoordSys );
        Telwindow.Set( "Telescope", 0 );
        YegSetCursor TelCursor( myYegs );              // make a cursor
        TelCursor.SetStep( "Telescope", 0 );           // step through Telescopes
        YegSetView  OneTel = myYegs.View( Telwindow );

        MosAccumulator LinMosAcc( Template );

        do {
            ImagingModel  myIM = OnePointing.getImagingModel();

            CoordWindow Pointwindow( aCoordSys );
            Pointwindow.Set( "Pointing", 0 );
            YegSetCursor PointCursor( OneTel );
            PointCursor.SetStep( "Pointing", 0 );      // step through Pointings
            YegSetView  OnePoint = OneTel.View( Pointwindow );

            do {
                FilledImage  Dirty( Template, 0.0 );
                myIM.Invert( OnePoint, Dirty );
                linMosAcc.AddIm( Dirty );
```

```
        } while ( PointCursor.Next( "Pointing" ));

    } while ( TelCursor.Next( "Telescope" ) );

    return  linMosAcc.Mosaic();
}
```

The image which is passed to MosAccumulator::AddIm must have access to the primary beam associated with the YegSet from which it was made. Hence, IIM::Invert must somehow make that association between the dirty image and the TelModel associated with the generating YegSet. IIM::Invert also needs to copy over the Telescope sky pointing position.

The Mosaic Accumulator class is described in AIPS++ Note 123, "Mosaicing Report for the Prototype", Mar 24 1992. Its task is to accumulate the numerator and denominator of the least squares mosaic equation:

$$I(\mathbf{x}) = \frac{\sum_{i=1}^{N pointings} A(\mathbf{x}-\mathbf{x}_i) I_i(\mathbf{x})}{\sum_{i=1}^{N pointings} A^2(\mathbf{x}-\mathbf{x}_i)}. \tag{4}$$

"Pointing" represents a coordinate axis which can either be enumerated (pointings 1-20) or continuous (RAstart = 17.00, RAstop = 17.20). Both must be implemented. The frequency coordinate will have the same dual nature.

Now lets say we are involved in the iterative nonlinear mosaic. We wish to calculate an image which is the gradient of χ^2. First, we must generate the dirty images and point spread functions:

```
Mosaic::Invert( YegSet myYegs,  ImageCollection Template ) {
        CoordWindow Telwindow( aCoordSys );
        Telwindow.Set( "Telescope", 0 );
        YegSetCursor TelCursor( myYegs );              // make a cursor
        TelCursor.SetStep( "Telescope", 0 );           // step through Telescopes
        YegSetView  OneTel = myYegs.View( Telwindow );

        int i = 0;
        do {
            ImagingModel  myIM = OnePointing.getImagingModel();

            CoordWindow Pointwindow( aCoordSys );
            Pointwindow.Set( "Pointing", 0 );
            YegSetCursor PointCursor( OneTel );
            PointCursor.SetStep( "Pointing", 0 );     // step through Pointings
            YegSetView  OnePoint = OneTel.View( Pointwindow );

            do {
//myIM.Invert gets the Dirty and PSF from the Template and fills them
                myIM.Invert( OnePoint, Template.Get(i,''Dirty''),
                                       Template.Get(i,''PSF'' ) );
                i++;
            } while ( PointCursor.Next( "Pointing" ));
        } while ( TelCursor.Next( "Telescope" ) );
```

```
        return  TheBank;
}
```

Then, to get the gradient image, we need to pass `TheBank`, which is a collection of dirty images and PSFs, and a model image to `Mosaic::Gradient`:

```
FilledImage Mosaic::Gradient( ImageCollection TheBank, FilledImage Model ) {
        int N = TheBank.HowMany();
        MosAccumulator gradient( Model );

        for (int i=0; i < N; i++) {
            FilledImage &Dirty = TheBank.GetIm( i, "Dirty");
            FilledImage &PSF = TheBank.GetIm( i, "PSF");
            gradient.AddIm( Dirty, PSF, Model );
        }
        return gradient.MosaicNumerator();
}
```

This example presents the `ImageCollection`, still very much in its conceptual stage. There are a number of applications in which each of several regions on the sky have several associated sky or Fourier plane images. Something like the `ImageCollection` would save time. See below.

If some of the slots in the `TheBank` were generated from single dish data, the "dirty image" could be represented as the average of the measured total power values, and the "point spread function" (the Fourier transform of the (u,v) coverage) is everywhere unity, and need not be stored. One weakness in this design is that in some sense `MosAccumulator::AddIm` needs to know the difference between SD and interferometer data.

8 The Single Dish Imaging Model (SDIM)

The primary methods of the IM are `Predict` and `Invert`. For a single dish imaging experiment, there may be some confusion as to what the products of predict and invert are. By analogy to the IIM, which deals with single pointing interferometer observations, the SDIM `Predict` and `Invert` should deal with one pointing on the sky. Given a model map in units of Jy/Pixel, the model is inverse Fourier transformed to the point $(u, v) = (0, 0)$; in other words, the image is summed. To invert, the measured total power (implicitly at (u,v) = (0,0)) is Fourier transformed onto an image (ie, each pixel is set equal to the measured total power value).

The SDIM `Predict` and `Invert` methods correspond directly to the IIM methods for (u,v) = (0,0). To make a real map, one needs to mosaic these observations. As discussed in Section 7, Mosaic is not an Imaging Model, but something which we refer to as an Imaging Model Manager (IMM). The Mosaic IMM will be able to produce a single image from a YegSet with data from several different Telescopes (different associated Imaging Models and TelModels). Hence, the Mosaic IMM will coordinate the use of each Imaging Model's `Predict` and `Invert` in generating an image or predicting data. The Mosaic IMM can also deal with SD data alone to produce a full Image.

As an example, consider a multiple pointing SD observation. To `Invert`, the values of the total power are traditionally placed in the appropriate pixel of a uniform grid. But what if the observation was not

made on a uniform grid? The Mosaic IMM can deal with this general case. For each pointing, perform the invert, apply the primary beam (PB), add this to an accumulator image, and normalize appropriately. Shortcuts can be made for the special case of observations on a regular grid. For `Predict`, the value of the appropriate pixel in the above mentioned uniform grid is returned. The Mosaic IMM predicts the measured total power from a *deconvolved* image by multiplying by the PB and Fourier transforming for (u,v) = (0,0), ie, integrating. This is equivalent to convolving the deconvolved image with the PB and sampling, or just sampling the un-deconvolved uniform grid map. While using the Mosaic IMM formalism with the simple SDIM may seem cumbersome, it is more general than the traditional methods. Furthermore, if the traditional case does apply, simple polymorphic `Predict` and `Invert` methods could be used.

To summarize, SDIM and IIM look very similar, except that SDIM will not need the values of (u,v) as they are assumed to be zero, and the Fourier transform would be replaced by a sum in `Predict` and by an operator which fills in an image with a single value equal to the total power normalized. To make a map, the Mosaic Imaging Model Manager's `Invert` would use the SDYegSet's associated IM to piece together the data from all pointings.

9 Common Issues for Images

9.1 Image Collections

A concept which appears in Mosaicing, Fly, and Non-isoplanatic imaging is the "collection of images". These are not image cubes because they may be of varying size. It seems that these are not "images of images" because the collections do not necessarily have the properties of an image; for example, in the Mosaic case, each of N images could correspond to the exact same region of the sky, so sometimes it would be more like a stack or a cube (but not generally). This collection needs to be a dynamic entity. Something built on an `Array<Array<FilledImage>>` is a possibility. Unevenly filled positions would not waste space if the Images are dynamically allocated.

9.2 Image SubSelection

Windows and boxes in images are a common theme through many of the imaging methods discussed here. The Image class should provide the appropriate services to utilize these different windowing concepts. Some windowing concepts which would be good to treat are:

- Boxes specified by BLC and TRC. They are assumed to be 1 inside (outside) the enclosed region and 0 outside (inside). These boxes should deal with N-dimensional regular images. They are easy to specify, do not require much storage, and are easy to implement. They are not very flexible and can cause trouble if they overlap.

- Masks require a full weight image and can be any shape and can be continuously varying in value.

- Windows are like masks in that they have an associated weight image, but windows imply a collection of masks related in a special manner. The weight of non-overlapping internal window regions is 1, the weight of external regions is 0, and the weight of a pixel which is in N windows is $1/N$. Windows are useful in Clean, Fly and in nonisoplanatic imaging. A set of clean components from some facet can be multiplied by its associated window weight image before the clean components are subtracted from the data visibilities. It doesn't matter that a given pixel is found in multiple windows.

9.3 Images and Associated TelModels

Images must also be entered in the Associator table to indicate which TelModel and which ImagingModel was associated with the YegSet from which they were formed. Some mosaic images will not have a unique TelModel or ImagingModel; either NULL models or list of models could be used.

10 Dangling Pointers!

Finally, here are some of the issues we are not so sure of.

- The YegSet/Image templates that the Imaging Models will use can be either provided as empty YegSet/Image with only the header, or all the relevant information be given to the Imaging Model which fills in the correct header. We currently feel that both the methods could be provided, with a reasonable default and a reasonable rule to override one of them in case both are provided.

- As mentioned in the text before, we are not clear about the data structures which will manage a "collection of Images" and association of Images with other Images.

- The ConvFunc is an object which is instantiated to hold a given convolution function, its FT and a method which will apply the gridding correction to an Image. Internally it uses two Math Tables to hold the two functions (the Grid Convolution Function and its FT) and an interpolation scheme. The tabulated mathematical functions that it represents are held in the MathTab object. Mathematically there can be various ConvFunc.

  There are two ways of having these different ConvFunc. One is to have different object for different function. The other is to have a generic MathTab and ConvFunc object which takes in a pointer to a function which it will use to fill its internal tables.

  The former method is simple but it also implies that somewhere in the code we will have to have a case statement which will look, may be, at a char* to determine which ConvFunc to use. This is clearly not very flexible and extendable.

  The latter method looks elegant and flexible in the sense that, to realize a new, exotic function, one only needs to code a function which will return the value of the function. But it is also more cumbersome. If the function is a parameterized function, that can be encoded in this function and the MathTab/ConvFunc need not know about it. This will also mean that the interface of this function, the pointer of which is passed to the MathTab/ConvFunc, is fixed.

- To handle the spectral line case with in the framework of IIM, we have used the concept of Views. The FreqList, given to the IIM holds the various frequencies to be used (as explained above). Each element of FreqList it self can be a list, in which case all those frequencies will be gridded onto one grid. To get the data for all these frequencies from the YegSet inside the Grid::Resample, we set up a "View" of the YegSet which presents only these frequencies.

  This in general, then requires mechanism to set up a "view" which includes non-contiguous bits of the underlaying data.

  The other way of doing the spectaral line inversion without setting up this "non-contiguous view" is to have a inner loop. The `Invert` method may then look like:

```
IIM::Invert(YegSet& myYegSet, FilledImage& myImageCube,
                              FilledImage& PSFCube,
                              FreqList& myFreqList)
{
        for (int i=0; i<FreqList.Length(); i++)
        {
//
// Set up the View (Window?), etc. for the Grid
//
           FilledImage Grid = FFTTool.ConjugateImage( myImageCube,
                                                      myFreqList[i] );
           FilledImage GridWT( Grid );
//
// Set up the View (Window?) for the output Image/PSF.  This will make
// sure that the right plane in the cube is operated upon.
//
           myImageCube.View( myFreqList[i] );
           PSFCube.View( myFreqList[i] );

           for (int j = 0; j<FreqList[i].Length(); j++)
           {
                    YegSet.Cursor(''Freq'',FreqList[i][j]);
                    FilledImage Image Grid = FT.Template(myImageCube);
                    FilledImage GridWT     = FT.Template(PSFCube);
                    GridTool.Resample(YegSet, Grid, GridWT);
           }
           FT.DoFFT(Grid, myImageCube);
           Ft.DoFFT(GridWT, PSFCube);
           ConvFunc.Correct( myImageCube );
           ConvFunc.Correct( PSFCube );
        }
};
```

Either of the methods is acceptable. However, the demand of "non-contiguous view" does look more general and flexible. Both the methods need to be subjected to complexity analysis.

- In methods like `FFTTool.DoFFT(Grid, myImage)`, at some stage, the data will exist in Grid as well as in myImage. This will be highly memory ineffecient.

  To do an in-place FFT effeciently, we may first make myImage a complex image, use it as a Grid in `GridTool::Resample(YegSet, Grid)`. This Grid then can be passed to FFTTool, which will do an in-place FFT on Grid, and make the Grid a real upon exit (remove the Imag. part of the complex storage). No redundent extra memory is used anywhere in the process.

  The code as used here is however more readable. To retain the readibility of the code, we can alternatively perform a "move" kind of an operation inside DoFFT on the FFTed data, which is hanging off the Grid (due to in-place FFT). The pointer to the data storage can be give to myImage

(after making it real) and detached from Grid (this detachment will be required since the Grid may be used again).

A IntImagingModel's Header

The following is a proposed header for the IntImagingModel class.

```
class IntImagingModel{
public:


//
// Constructour
//
//  Will behave as a DFT Invert/Predict by default
//
  IntImagingModel()

//
// Will behave as a FFT Invert/Predict
//
  IntImagingModel(ConvFunc& myCF, int mySupportSize=6,
                  Array<float> myShift=0,
                  boolean mySlowZ=TRUE,
                  boolean myDoUniform=TRUE,
                  const DynArray<DynArray<int>> myFreqList = 0,
//
// Destructor
//
  ~IntImagingModel() { delete ConvFunc; delete FreqList;};


//
// Services
//

  void Invert( const YegSet&, Image& );
  void Invert( const YegSet&, Image& , Image& );
  void Predict(const Image& , YegSet& );

//
// The various Reset functions - to change the state of the IIM after
// instantiation.
// The SupportSize of the ConvFunction is part of the ConvFunc.
//

  Reset(ConvFunc &NewCF)
   {CF=NewCF; Gridder.ResetConvFunc(CF); };
```

```
  Reset(DynArray<DynArray<int>> NewFreqList) {FreqList = NewFreqList;};
//
// Should the follwing data members be allowed direct access?
//      IIM.DoFFT = TRUE;
// etc.
//
  Reset(boolean  NewDoFFT, boolean NewDoSloZ, boolean NewWeighting)
    { DoFFT = NewDoFFT; DoSlowZ = NewDoSlowz; DoUniform = NewWeighting};


private:
  Boolean ConvFuncSet, FFTServerSet;
  Boolean DoFFT, DoSlowZ, DoUniform;
  ConvFunc &CF;
  FFTServer FFTTool=0;
  GridTool Gridder = 0;
  DynArray FreqList;
//
// The lower level routines which perform the FT
//
  YegSet DoPredictDFT ( ... );
  Image DoInvertDFT  ( ... );
//
// The FFT servers will be instantiated if DoFFT = TRUE and will be
// destroyed when DoFFT becomes FLASE.  This is to make sure:
//     - the extra baggage of FFT is not carried if only DFT is
//       intented to be used
//     - if FFT is to be used, it should instantiated once (it will
//       calculate its sin/cos tables once)
//


  Image  DoInvertFFT (....);
  YegSet DoPredictFFT( ...);
};
```

The gridding convolution function, or ConvFunc object, has the convolution function and the correction function in lookup tables and a method for applying the grid correction to an image. If not supplied, the ConvFunc defaults to something reasonable.

A version of Invert which treats spectral line data might look like this (this can be the generic implementation which will handle both, Spectral Line data as well as Continum data):

```
void IIM::Invert(YegSet& myYegs, FilledImag& myImageCube,
                                    FilledImage& PSFCube,
                                    FreqList& myFreqList)
{
        CoordWindow FreqWindow( acoordsys );
        YegSetView Someyegs = myYegs( FreqWindow );
```

```
        for (int i=0; i < myFreqList.Length(); i++)
        {
//
// This will setup a View for myImageCube and the header for the Grid
// so that the right plane in the Cube is used for later operations
//
            myImageCube.View( myFreqList[i] );
            PSFCube.View() = myImageCube.View();

            FilledImage Grid = FFTTool.ConjugateImage( myImageCube,
                                                       myFreqList[i] );
            Grid.Fill( 0.0 );
            FilledImage GridWT( Grid );

//  select a vector of frequencies (myFreqList is 2-D)

            FreqWindow.Set( "Freq", myFreqList[i] );
            Gridder.Resample( SomeYegs, Grid, GridWT );
//
// In place FFT.  On the output, the Grids will have the FFTed
// data.  The Views of myImageCube and PSFCube determine which
// plane to fiddle with.
//
            FFTTool.DoFFT( Grid, myImageCube);
            FFTTool.DoFFT( GridWT, PSFCube );

            CF.Correct( myImageCube );
            CF.Correct( PSFCube );
            myImage.Deal_With_Associator_Details( SomeYegs );
        }
}
```