NOTES ON OVERALL CASA DESIGN, SOFTWARE FRAMEWORK, IMAGER LAYOUT, AND CONTROL-FLOW (MOST RELEVANT TO IMPLEMENTAION OF A-PROJECTION): STATIC ANALYSIS

1. (High Level) Overview

In CASA, the Measurement Equation (ME) is manipulated and evaluated by breaking it down into two parts: the **VisEquation** C++ base class (with derived concrete classes or variants), which expresses the visibility-plane part (direction-independent effects) of the ME, and the **SkyEquation** C++ base class (with derived concrete classes / variants such as **CubeSkyEquation**) which expresses the sky-plane part (image-plane effects – direction- dependent effects (DDE)) of the ME.

The **SkyEquation** base class and its derived concrete classes such as **CubeSkyEquation** deal with the Fourier Transform part of the ME, so they are given the **FTMachine** base class (with derived concrete classes such as **nPBWProjectFT** and **GridFT**) , which is used for performing forward and inverse Fourier transforms, for this purpose. For example, the class **GridFT** implements a straightforward gird and de-grid FFT-based Fourier transform while **nPBWProjectFT** does grid-based Fourier transforms and also includes the effects of primary beam and antenna pointing offsets.

The sky brightness is modeled by the base class **SkyModel** (which has derived concrete classes such as **ImageSkyModel**). The **SkyModel (ImageSkyModel)** class has an interface to **SkyEquation (CubeSkyEquation)** via the **PagedImage** class**.**

The Abstract Base Class **SkyJones** models sky-plane instrumental effects such as the primary beam for the **SkyEquation. BeamSkyJones** , derived from **SkyJones**, describes the interface to beam-based **SkyJones** objects. Like **SkyJones** , it too is an Abstract Base Class, but implements primary beam-related methods (i.e., primary beam-like sky-plane effects). The derived concrete class **VPSkyJones** only models and deals with only the voltage pattern Jones matrics. The **PBMathInterface** virtual base class defines the interface to PB Math objects, i.e., the encapsulation of the PB mathematical functioning. Derived concrete classes and base classes such as **PBMath1D** carry out the mathematical operations for the primary beams and voltage patterns. **BeamSkyJones, VPSkyJones, PBMathInterface, PBMath1D**, etc. are directly invoked and implemented during the simulation stage (before imaging) and to a lesser extent during imaging (e.g., **VPSkyJones**).

The visibility data is held in the **MeasurementSet Table,** which is the main interface class to a read/write table. To expedite processing, the **VisibilityIterator** class is used to iterate through a writeable **MeasurementSet** as needed. The **VisibilityIterator** is stored in the **VisSet** such that one can construct **VisSet** and then use the iterator method to retrieve the iterator. Once one has **VisibilityIterator**, it can be used to access actual visibility data in chunks by using the **VisBuffer** class.

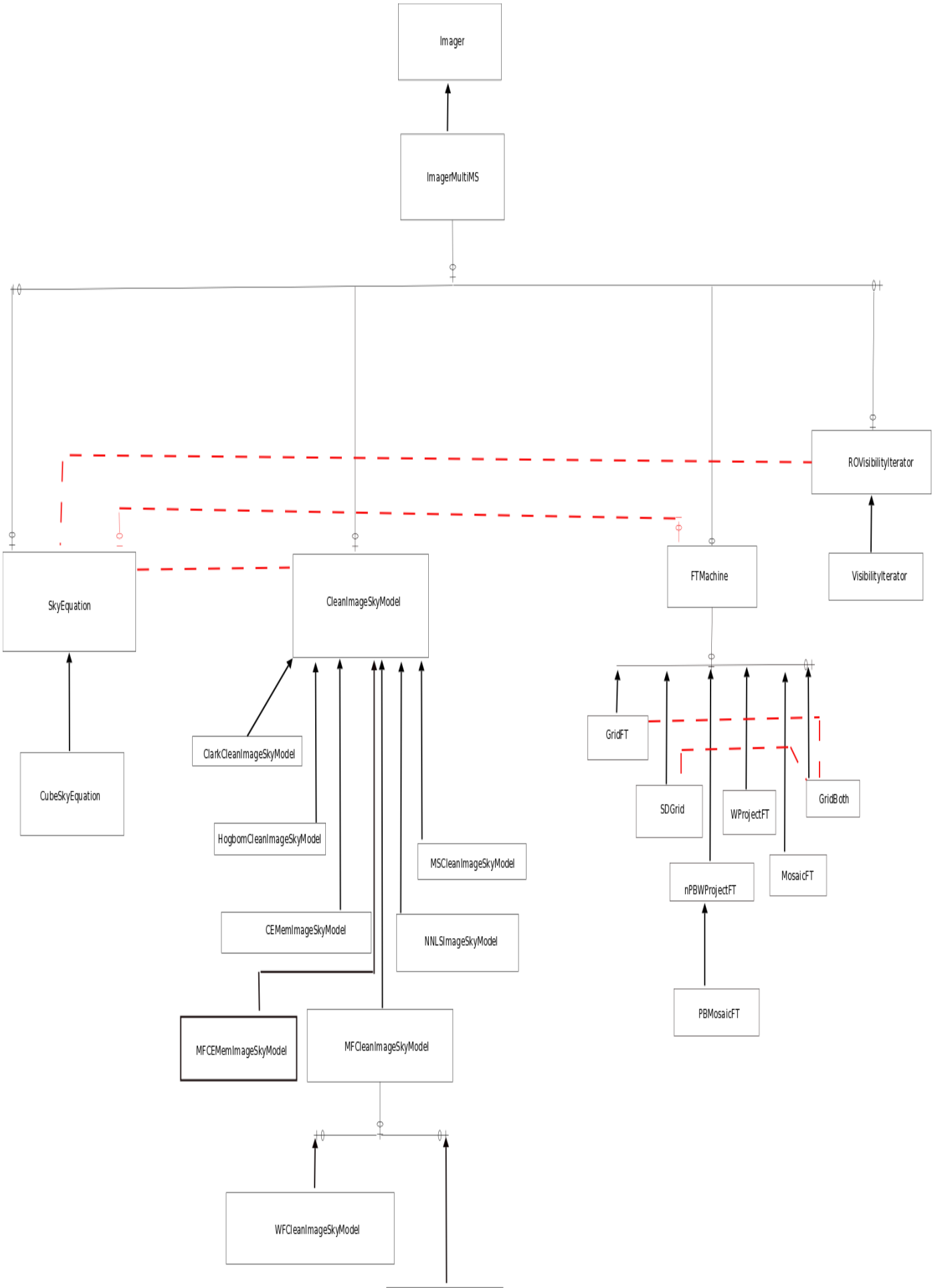
nPBWProjectFT is an FTMachine for gridded Fourier transforms including the effects of PBs, pointing offsets, and the w-term. **nPBWProjectFT** uses and implements gridding convolution function (GCF)

operations via the **ConvovleGridder** class, for example, and underlying gridders written in Fortran (i.e., **fpbwproj.f**). **nPBWProjectFT** makes use of the base class **ConvolutionFunction** to encapsulate and compute convolutional gridding and derived classes such as **IlluminationConvFunc** and **VLACalcIlluminationConvFunc** (for the specific case of the VLA) for the calculation of the Aperture Illumination Function (AIF) for use in combination with gridding convolution during iterative deconvolution when the A-Projection algorithm is used to correct for DDEs (i.e., PBs).

At the highest levels, including the user level, iterative deconvolution algorithms such as Clark CLEAN or Hogbom CLEAN and algorithms which correct for DDEs in the Fourier plane (such as the A-Projection algorithm) can be used by setting the **ftmachine** parameter of the **imager** and **clean** tools and tasks (which include python and xml code such as **task_clean.py**, **cleanhelper.py**, **clean.xml**, **imager.xml**, etc.) to "pbwproject". At the highest levels of C++ code, the **Simulator** class simulates the MS from **SkyModel** and **SkyEquation** and the **Imager** class contains functions needed for the **imager** tool and seems to act as a 'wrapper' for all imaging functions and tasks. At a lower C++ level, the **ClarkCleanModel** and **ClarkCleanImageSkyModel** classes, for example, perform the Clark CLEAN algorithm on arrays, and the **HogbomCleanImageSkyModel** derived concrete class (derived from **SkyModel**) implements the Hogbom CLEAN algorithm.

1.1 Imager/ImageSkyModel/FTMachine Class Hierarchy Diagram

On the following page is a summary of the CASA Imager/ImageSkyModel/FTMachine class hierarchy (from K.Golap, May 2010 and R.V. Urvashi, May 2011, http://www.aoc.nrao.edu/~rurvashi/ImagingAlgorithmsInCasa/node6.html). This diagram can be seen more clearly at http://www.aoc.nrao.edu/~rurvashi/ImagingAlgorithmsInCasa/KG_Imager_main.png.


Imager
ImagerMultiMS
ROVisibilityIterator
SkyEquation
CleanImageSkyModel
FTMachine
VisibilityIterator
CubeSkyEquation
ClarkCleanImageSkyModel
HogbomCleanImageSkyModel
MSCleanImageSkyModel
CEMemImageSkyModel
NNLSImageSkyModel
MFCEMemImageSkyModel
MFCleanImageSkyModel
WFCleanImageSkyModel
GridFT
SDGrid
WProjectFT
GridBoth
nPBWProjectFT
MosaicFT
PBMosaicFT

1.2 Relatively High- Level Illustrative/Pseudo-Code Example

Here we present simplified illustrative/pseudo-code high-level CASA C++ example of the use and implementation of some of the classes discussed in the above overview.

```
// Read the VisSet from disk

VisSet vs ("3c84.MS");

// Create an ImageSkyModel from an image on disk

ImageSkyModel(ism(PagedImage<Float>("3c84.modelimage"));

// Make an FTMachine: here we use a simple Grid and Fourier Transform.

GridFT ft;

SkyEquation se (ism,vs,ft);   // For imaging, we need SkyEquation, ImageSkyModel, and FTMachine

// Predict the visibility set for the model

se.predict();

// Make a Clean Image and write it out

HogbomCleanImageSkyModel csm(ism);

if (csm.solve()) {

PagedImage<Float> cleanImage=csm.image(0);

cleanImage.setName("3c84.cleanimage");

}
```

2. More detailed (mostly static) analysis of the SkyEquation, SkyModel, SkyJones, and FTMachine classes and their concrete derived classes and variants.

SkyEquation and its derived classes are responsible for evaluating the sky-plane part of the ME, χ^2, and gradients of χ^2 used for the evaluation of generalized dirty images. **ViSet** is responsible for providing visibility data to **SkyEquation** (**CubeSkyEquation)** and storing the results of predictions. Thus, it's a convenient interface to an MS. **SkyModel** (and its derived concrete classes and variants) supplies a set of images to the **SkyEquation** (**CubeSkyEquation**). **SkyJones** (and its derived classes and variants) supplies sky-plane-based calibration effects to **SkyEquation** by multiplying a given image by matrices such as $\mathbf{J^{sky}}$**.**

As an example, the following code fragment from **SkyEquation.cc** defines a **SkyEquation** linking a **VisSet vs** with a **SkyModel sm** using an **FTMachine ft** for forward transfroms (Sky→Vis) and ift for inverse transforms (Vis→Sky):

SkyEquation::SkyEquation(SkyModel& sm, VisSet& vs, FTMachine& ft, FTMachine& ift) ….

2.1 SkyEquation and CubeSkyEquation

The **SkyEquation** class and its derived/variant classes (e.g., **CubeSkyEquation**) encapsulates the equation between sky brightness and visibility measured by a radio interferometer. The principal use of **SkyEquation** is that gradients of χ^2 may be calculated and returned as an image. The following components (and their variants / derived classes) can be plugged into **SkyEquation:**

- Antenna-based DDE terms: **SkyJones**
- Sky Brightness Model: **SkyModel (ImageSkyModel)**
- Fourier transform machine: **FTMachine (nPBWProjectFT)**

SkyEquation (CubeSkyEquation) calls the **get** method of **SkyModel** (**ImageSkyModel**, etc.) to get a set of images for which it can do a prediction of the visibilities in a **VisSet** and/or **VisBuffer**, e.g.,

```
// Add the sky visibility for this coherence sample
VisBuffer& SkyEquation::get(VisBuffer& result, Int model,
                        Bool incremental,
                        MS::PredefinedColumns Type) {….
```

To obtain the predicted visibilities, one uses the **predict** method of **SkyEquation (CubeSkyEquation)**, e.g.,

```
void SkyEquation::predict(Bool incremental,  MS::PredefinedColumns Type) {…
```

To solve for a **SkyModel (ImageSkyModel,** etc.) one uses the **solve** of **SkyEquation**. This calls the **solve** method of the **SkyModel**, e.g.,

```
// Solve for a SkyModel
Bool SkyEquation::solveSkyModel() {
  return sm_->solve(*this);
}
```

This will usually be an iterative procedure that in turn calls the **gradientsChiSquared** method of **SkyEquation (CubeSkyEquation)** which accumulates and increments χ^2 and the first and second gradients of χ^2 into the **SkyModel**, e.g.,

```
void SkyEquation::gradientsChiSquared(const Matrix<Bool>& required,
                                      SkyJones& sj) {
  // Keep compiler happy
  if(&sj) {};
  if(&required) {};
  throw(AipsError("SkyEquation:: solution for SkyJones not yet implemented"));
}
```

and

```
void SkyEquation::incrementGradientsChiSquared() {…
```

The gradients are with respect to the set of model images. Conversion of the gradients to be with respect to the internal variables of the **SkyModel** is the responsibility of the particular **SkyModel** and is not visible outside of that **SkyModel.** So, for example, **ComponentSkyModel** has to take the image form of the gradient of χ^2 and convert it to be with respect to the parameters of the various components. In conjunction with this, **SkyEquation** evaluates the update direction and implements Newton-Raphson minimization. The image of **SkyModel** must contain the increment to the image. For each model, a collection of complex transfer functions are used to avoid gridding and degridding all of the visibilities.

SkyEquation also makes an approximate point spread function (PSF) for each image model, i.e.,

```
// We make an approximate PSF for each plane.We only do this per model
// since we may not need all PSFs.
// ************* Note that this overwrites the model! *********
void SkyEquation::makeApproxPSF(Int model, ImageInterface<Float>& psf) {
  LogIO os(LogOrigin("SkyEquation", "makeApproxPSF")); ….
```

This PSF is approximate in the sense that it's a shift-invariant approximation.

SkyEquation (**CubeSkyEquation**) needs to be able to perform Fourier Transforms on visibility data, so there is a means within it to call or interface with FTMachines such as **nPBWProjectFT** (which implements A-Projection). For example, see the following code fragment below from **SkyEquation.cc**:

```
void SkyEquation::gradientsChiSquared(Bool incremental, Bool commitModel) {
  AlwaysAssert(ok(),AipsError);
```

```
 if ((ft_->name() == "PBWProjectFT"))
    {
      ft_->setNoPadding(False);
      fullGradientsChiSquared(incremental);
    }
  else …
```

and from **CubeSkyEquation.cc**:

```
else if(ft.name()== "PBWProjectFT"){
     ft_=new nPBWProjectFT(static_cast<nPBWProjectFT &>(ft));
     ift_=new nPBWProjectFT(static_cast<nPBWProjectFT &>(ft));
     ftm_p[0]=ft_;
     iftm_p[0]=ift_;
     if(nmod != (2 * sm_->numberOfTaylorTerms() - 1)) /* MFS */
       throw(AipsError("No multifield with pb-projection allowed"));
     for (Int k=1; k < (nmod); ++k){
      ftm_p[k]=new nPBWProjectFT(static_cast<nPBWProjectFT &>(*ft_));
      iftm_p[k]=new nPBWProjectFT(static_cast<nPBWProjectFT &>(*ift_);
```

2.2 SkyModel and its variants and derived concrete classes

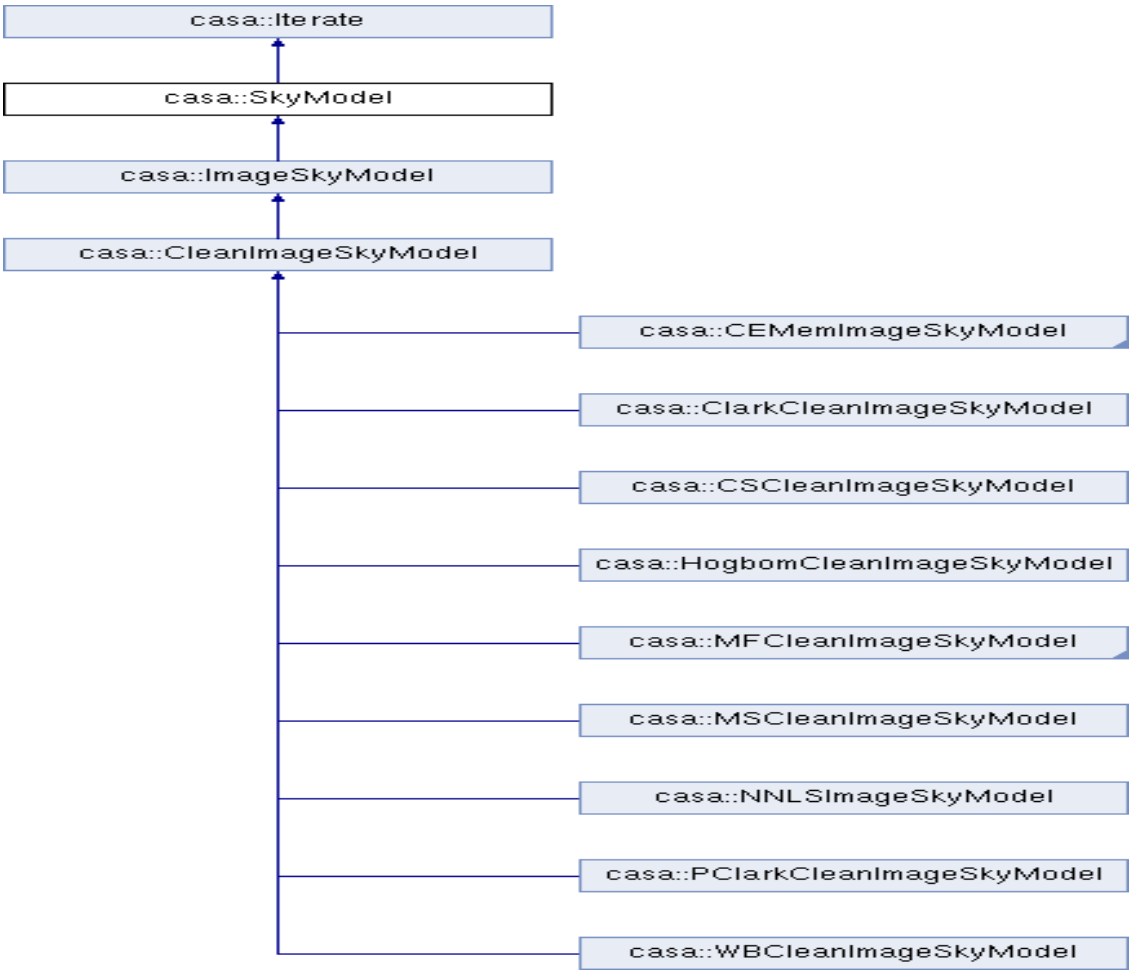
SkyModel describes an interface for model images to be used with **SkyEquation.** It is an Abstract Base Class: most methods must be defined in derived classes such as **ImageSkyModel**, **CleanImageSkyModel, ClarkCleanImageSkyModel**, etc. (Note that there is only a **SkyModel.h** header file and no **SkyModel.cc** source code file).

A **SkyModel** contains a number of separate models. The interface to **SkyEquation** is via an image per model. **SkyEquation** uses this image to calculate Fourier transforms, etc. Some (most) **SkyModels** are solvable: the **SkyEquation** can be used by the **SkyModel** to return gradients w.r.t. itself (via the image interface). Thus for a **SkyModel** to solve for itself, it calls the **SkyEquation** methods to get gradients of χ^2 w.r.t. the image pixel values (thus returning, essentially, a residual image). The **SkyModel** then uses these gradients as appropriate to update itself. Thus, the gradient of χ^2 is

calculated with **SkyEquation** and **SkyModel (ImageSkyModel**, etc.) then uses it to update the estimate image. The following examples illustrate how a **SkyModel** can be used:

- Simple cleaning: one model image. The gradient gives the residual image and a special method gives a PSF.
- Cleaning with visibility-based subtraction: one model image. The gradient can be calculated as needed (using **SkyEquation**) to produce the correct residual image.
- Wide-field imaging: one image model per patch of sky that is to be imaged.

The figure below is the inheritance diagram for the **SkyModel** class:



2.2.1 ImageSkyModel (and derived class CleanImageSkyModel)

ImageSkyModel (CleanImageSkyModel) describes an interface for model images to be used in the **SkyEquation**. It is derived from **SkyModel.**

An **ImageSkyModel** contains a number of separate models. The interface to **SkyEquation** is via an image per model. **SkyEquation** uses this model image to calculate Fourier transforms, etc. Some **SkyModels** are solvable: **SkyEquation** can be used by **ImageSkyModel** to return gradients w.r.t. itself (via the image interface). Thus, for **ImageSkyModel** to solvefor itself, it calls the **SkyEquation** methods to get gradients of χ^2 w.r.t. the image pixel valus (thus essentially returning a residual image). **ImageSkyModel** then uses these gradients as appropriate to update itself.

In conjunction with **SkyEquation (CubeSkyEquation)** and **SkyModel, ImakeSkyModel** carries out a number of tasks, including:

- Check to add a component list (where the **ComponentList** class is a container that allows many sky components to be grouped together and manipulated as one large compound component, e.g.,

```
Bool ImageSkyModel::add(ComponentList& compList)

{

  if(componentList_p==0) {

    componentList_p=new ComponentList(compList);

    return True;

  }

  return False;

}
```

- Check to update the model image, e.g.,

```
Bool ImageSkyModel::updatemodel(const Int thismodel, ImageInterface<Float>&
image){

  if(nmodels_p < thismodel)

    throw(AipsError("Programming error " + String::toString(thismodel) + "
is larger than the number of models"));

  image_p[thismodel]=&image;

  AlwaysAssert(image_p[thismodel], AipsError);

  image_p[thismodel]->setUnits(Unit("Jy/pixel"));

  modified_p=True;

  return True;

}
```

- Add an image and use the maximum number of transfer functions (maxNumxfr) that we might want to associate with this image, e.g.,

```
Int ImageSkyModel::add(ImageInterface<Float>& image, const Int maxNumXfr)
```

```
{
  Int thismodel=nmodels_p;
  nmodels_p++;
  if(nmodels_p>maxnmodels_p) maxnmodels_p=nmodels_p;
  maxNumXFR_p=maxNumXfr;
  image_p.resize(nmodels_p);
  cimage_p.resize(nmodels_p);
  cxfr_p.resize(nmodels_p*maxNumXFR_p);
  residual_p.resize(nmodels_p);
    ……
```

- Check to add a residual image, e.g.,

```
Bool ImageSkyModel::addResidual(Int thismodel, ImageInterface<Float>& residual)
{
  LogIO os(LogOrigin("ImageSkyModel", "addResidual"));
  if(thismodel>=nmodels_p||thismodel<0) {
    os << LogIO::SEVERE << "Illegal model slot" << thismodel << LogIO::POST;
    return False;
  }
  residual_p[thismodel] = &residual;
  AlwaysAssert(residual_p[thismodel], AipsError);
  if(residualImage_p[thismodel]) delete residualImage_p[thismodel];
  residualImage_p[thismodel]=0;
  return True;
}
```

- Make the approximate point spread function needed for each model image, e.g.,

```
void ImageSkyModel::makeApproxPSFs(SkyEquation& se) {
  LogIO os(LogOrigin("ImageSkyModel", "makeApproxPSFs"));
  if(!donePSF_p){
    for (Int thismodel=0;thismodel<nmodels_p;thismodel++) {
      //make sure the psf images are made
      PSF(thismodel);
    }
```

```
    se.makeApproxPSF(psf_p);
```

- Initialize for the gradient search, e.g.,

```
void ImageSkyModel::initializeGradients() {
  sumwt_p=0.0;
  chisq_p=0.0;
  for (Int thismodel=0;thismodel<nmodels_p;thismodel++) {
    cImage(thismodel).set(Complex(0.0));
    gS(thismodel).set(0.0);
    ggS(thismodel).set(0.0);
  }
}
```

- Check to make the Newton Raphson step internally, in conjunction with or related to the making of the residual image in, e.g., **ClarkCleanImageSkyModel** (see next section), e.g.,

```
Bool ImageSkyModel::makeNewtonRaphsonStep(SkyEquation& se, Bool incremental,
                                          Bool modelToMS) {
  LogIO os(LogOrigin("ImageSkyModel", "makeNewtonRaphsonStep"));
  se.gradientsChiSquared(incremental, modelToMS);
  // Now for each model, we find the recommended step
  if(numberOfModels()>0) {
    for(Int thismodel=0;thismodel<nmodels_p;thismodel++) {
      if(isSolveable(thismodel)) {
    LatticeExpr<Float> le(iif(ggS(thismodel)>(0.0), -gS(thismodel)/ggS(thismodel),
0.0));
    residual(thismodel).copyData(le);
      }
    }
  }
  modified_p=False;
  return True;
}
```

- Check to see if this **SkyModel** is solved for. This replaces the dirty image with the residual image, e.g.,

```
// Simply finds residual image: i.e. Dirty Image if started with zero'ed image. We
work from corrected visibilities only!

Bool ImageSkyModel::solve(SkyEquation& se) {

  return solveResiduals(se);

}
```

- Check to see if the residuals are explicitly solved for, e.g.,

```
// Simply finds residual image: i.e. Dirty Image if started with zero'ed image. We
work from corrected visibilities only!

Bool ImageSkyModel::solveResiduals(SkyEquation& se, Bool modelToMS) {

  makeNewtonRaphsonStep(se, False, modelToMS);

  return True;

}
```

- Get the current residual image: this is either that image specified via addResidual, or a scratch image.

```
// Return residual image: to be used by callers when it is known that the

// current residual image is correct

ImageInterface<Float>& ImageSkyModel::getResidual(Int model) {

  return ImageSkyModel::residual(model);

}
```

- Is this model solvable?

```
Bool ImageSkyModel::isSolveable(Int model)

{

  AlwaysAssert(nmodels_p>0, AipsError);

  AlwaysAssert((model>-1)&&(model<nmodels_p), AipsError);

  return solve_p[model];

};
```

2.2.2 The Clark CLEAN Aglorithm: ClarkCleanModel, MFCleanImageSkyModel

- **ClarkCleanModel** is a class for performing the Clark Clean Algorithm on Arrays. This class uses the Clark CLEAN algorithm to deconvolve model images . Only the deconvolved models of the sky are directly stored by this class. The PSF and convolved (dirty) image are stored in a companion class which must be derived from **ResidualEquation**. Note that **ClarkCleanModel** is **not** derived from the **SkyModel** class.

The cleaning works as follows: The user constructs a **ClarkCleanModel** by specifying an initial model of the sky. This can by be one,two,three... dimensional depending on the dimension of the PSF. The user then constructs a class which implements the forward transform equation (FFT) between the model and the dirty image for predicting model visibilities. Typically this will be the **ConvolutionEquation** class.

The user then calls the **solve()** function (with the appropriate equation class as an argument, e.g., **ConvolutionEquation**), and this class will perform the Clark clean, e.g.,

```
Bool ClarkCleanModel::solve(ConvolutionEquation & eqn){

  theLog << LogOrigin("ClarkCleanModel", "solve");

  AlwaysAssert(theModel.ndim() >= 2, AipsError);

  const IPosition dataShape = theModel.shape();

  Int npol = 1;

  if (theModel.ndim() >= 3)

    npol = dataShape(2);

  AlwaysAssert(npol == 1 || npol == 2 || npol == 4, AipsError);

  // compute the current residual image (using an FFT)

  Array<Float> residual;

  eqn.residual(residual, *this);
```

The various clean parameters are set (prior to calling **solve**) using the functions derived from the **Iterate** class, in particular **setGain()**, **setNumberIterations()** & **setThreshold()** (to set a flux limit).

The **solve()** function does not return either the deconvolved model or the residuals. The solved model can be obtained using the getModel() function (derived from ArrayModel()), e.g.,

```
void ClarkCleanModel::getModel(Array<Float>& model) const{

  model = theModel;

};
```

and the residual can be obtained using the residual() member function of the ConvolutionEquation Class, e.g.,

```
Bool ConvolutionEquation::

residual(Array<Float> & result,

     const LinearModel< Array<Float> > & model) {

  if (evaluate(result, model)) {

    result = theMeas - result;
```

```
    return True;
  }
  else
    return False;
}
```

A semi-pseudo-code example of the use of the **ClarkCleanModel** and **ConvolutionEquation** classes is as follows:

```
Matrix<Float> psf(12,12), dirty(10,10), initialModel(10,10);
// ...put appropriate values into psf, dirty, & initialModel....
ClarkCleanModel<Float> deconvolvedModel(initialModel);
ConvolutionEquation convEqn(psf, dirty);
deconvolvedModel.setGain(0.2);
deconvolvedModel.setNumberIterations(1000);
Bool convWorked = deconvolvedModel.solve(convEqn);
Array<Float> finalModel, residuals;
if (convWorked){
   finalModel = deconvolvedModel.getModel();
   ConvEqn.residual(deconvolvedModel, finalResidual);
 }
```

Using a Clark clean deconvolution procedure, one can solve for an improved estimate of the deconvolved object. (Note that the convolution equation contains the psf and dirty image), i.e.,

```
  Bool solve(ConvolutionEquation & eqn);
  Bool singleSolve(ConvolutionEquation & eqn, Array<Float>& residual);
```

These functions set various "knobs" that the user can tweak and are specific to the Clark clean algorithm. The more generic parameters, i.e., clean gain, and maximum residual flux limit, are set using functions in the **Iterate** base class. The **get** functions return the value that was actually used after the cleaning was done.

Once can set the maximum number of minor iterations to perform for each major cycle, i.e.,

```
   virtual void setMaxNumberMinorIterations(const uInt maxNumMinorIterations);
  virtual uInt getMaxNumberMinorIterations();
```

and set and get the initial number of iterations,

```
   virtual void setInitialNumberIterations(const uInt initialNumberIterations);
```

```
virtual uInt getInitialNumberIterations();
```

The maximum number of major cycles to perform are set as follows:

```
virtual void setMaxNumberMajorCycles(const uInt maxNumMajorCycles);

virtual uInt getMaxNumberMajorCycles();
```

All of the minor cycle iterations are carried out for one major cycle. Cleaning stops when the flux or iteration limit is reached, i.e.,

```
void doMinorIterations(Array<Float> & model,
                       Matrix<Float> & pixelValue,
                       const Matrix<Int> & pixelPos,
                       const Int numPix,
                       Matrix<Float> & psfPatch,
                       Float fluxLimit,
                       uInt & numberIterations,
                       Float Fmn,
                       const uInt totalIterations,
                       Float& totalflux);
```

- **MFCleanImageSkyModel,** which implements the multi-field CLEAN algorithm, is a class derived from **SkyModel(ImageSkyModel)**. It is implemented using the **ClarkCleanModel** class. Multi-field CLEAN is an FFT-based CLEAN algorithm where cleaning is split into major and minor cycles. In a minor cycle, the brightest pixels are cleaned using only the strongest sidelobes (and main lobe) of the PSF. In the major cycle, a fully correct subtraction of the PSF is done for all points accumulated in the minor cycle using an FFT-based convolution for speed.

2.3 SkyJones, BeamSkyJones, VPSkyJones, and examples

The **SkyJones** abstract base class models and implements sky-plane based instrumental/calibration effects such as voltage beams / primary beams for the **SkyEquation (CubeSkyEquation)** class and it describes an interface for Components to be used in **SkyEquation**. Most methods must be defined in derived classes such as **BeamSkyJones** and **VPSkyJones.** Note that in CASA, it appears as if visibilities are corrupted by the primary beam model more in the simulation stage (via derived classes of **SkyJones** such as **BeamSkyJones** and **VPSkyJones**) than at the imaging stage.

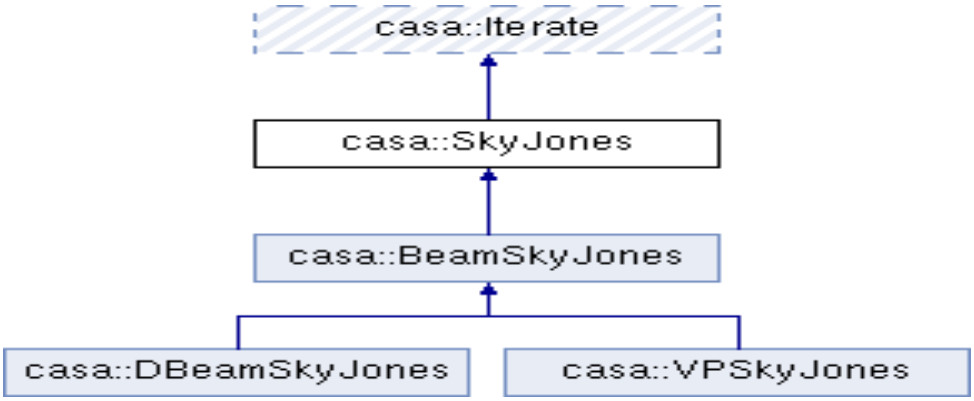
Conceptually, **SkyJones** applies an image of Jones matrices to an image. For example, it takes an image of the sky brightness and applies the complex primary beam for a given interferometer. Only

the interface is defined here in the Abstract Base Class (**SkyJones)**. Actual concrete classes, such as **BeamSkyJones** and **VPSkyJones** must be derived from **SkyJones**. Some (most) SkyJones are solvable: the **SkyEquation** class can be used by the **SkyJones** to return gradients with respect to itself (via the image interface). Thus for a **SkyJones** to solve for itself, it calls the **SkyEquation** methods to get gradients of chi-squared with respect to the image pixel values. The **SkyJones** then uses these gradients as appropriate to update itself.

The following high-level code example illustrates how the **SkyModel**-derived class **VPSkyJones** can be used when creating an **ImageSkyJones** from an image on disk:

```
ImageSkyModel ism(PagedImage<Float>("3C273XC1.modelImage"));
// Make an FTMachine: here we use a simple Grid and FT.
GridFT ft;
SkyEquation se(ism, vs, ft);
// Make a Primary Beam Sky Model
VPSkyJones pbsj(ms);
// Add it to the SkyEquation
se.setSkyJones(pbsj);
// Predict the visibility set
se.predict();
// Read some other data
VisSet othervs("3c84.MS.Otherdata");
// Make a Clean Image and write it out
HogbomCleanImageSkyJones csm(ism);
if (csm.solve(othervs)) {
PagedImage<Float> cleanImage=csm.image(0);
cleanImage.setName("3c84.cleanImage");
}
```

The inheritance diagram for the **SkyJones** class is as follows:



2.3.1 BeamSkyJones

The **BeamSkyJones** class, derived from **SkyJones**, describes an interface to primary beam-based **SkyJones** objects. Like **SkyJones**, it too is an abstract base class, but implements the primary beam-related methods. The **BeamSkyJones** class encapsulates the antenna beam-based aspects which are present in at least a few other specific **SkyJones** classes (**VPSkyJones** and **DBeamSkyJones**) for use with the **SkyEquation** class. It is used in the CASA simulation stage when the model visibilities are corrupted by the primary beam model.

2.3.2 VPSkyJones

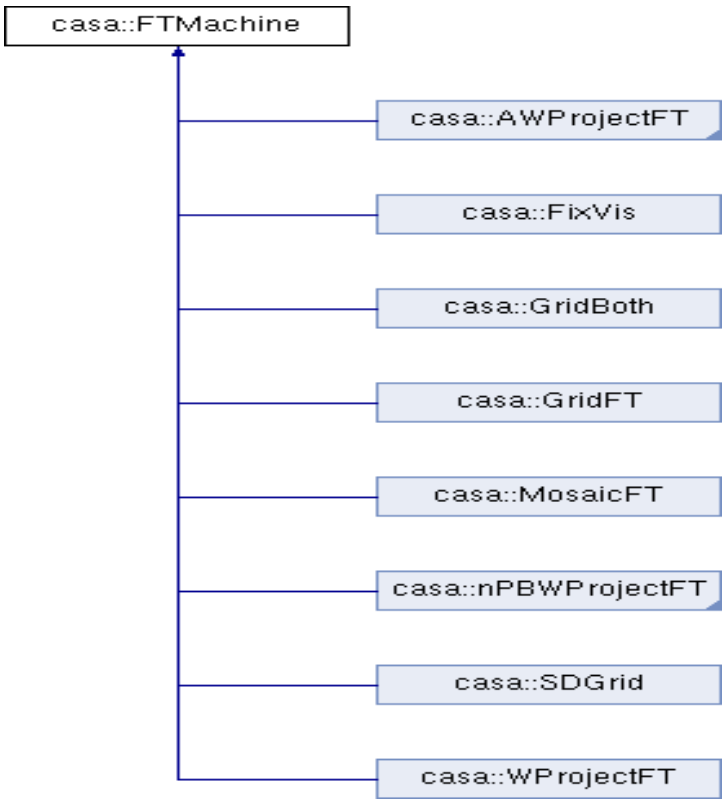
The **VPSkyJones** class models the diagonal elements of the voltage pattern Sky Jones matrices (E Jones). This class only deals with the diagonal elements of the voltage pattern jones matrices so as to deal with the non-leakage voltage pattern as applied to all Stokes, and beam squint (i.e, errors in Stokes V caused by differing RR and LL beams). Polarization leakage beams are dealt with in the **DBeamSkyJones** class. The motivation for this split between **VPSkyJones** and **DBeamSkyJones** is differing storage requirements for the underlying **PBMath** types (e.g., **PBMath1D**, **PBMath1Dairy**), which do the mathematical operations of the primary beams and voltage patterns, and different methods available to **VPSkyJones** and **DBeamSkyJones**.

2.4 FTMachine, ConvolutionFunction, VLACalcIlluminationConvFunc, IlluminationConvFunc, and nPBWProjectFT: the A-Projection Algorithm

The **FTMachine** class defines the interface for the Fourier Transform Machine. The **SkyEquation** (**CubeSkyEquation**) class needs to be able to perform Fourier transforms on visibility data. **FTMachine** allows efficient Fourier Transform processing using a **VisBuffer** which encapsulates a chunk of visibility (typically all baselines for one time) together with all the information needed for processing (e.g., UVW coordinates).

Note that the image must be Complex. It must contain the Complex Stokes values (e.g. RR,RL,LR,LL). **FTMachine** uses the image coordinate system to determine mappings between the polarization and frequency values in the **PagedImage** and in the **VisBuffer** classes.

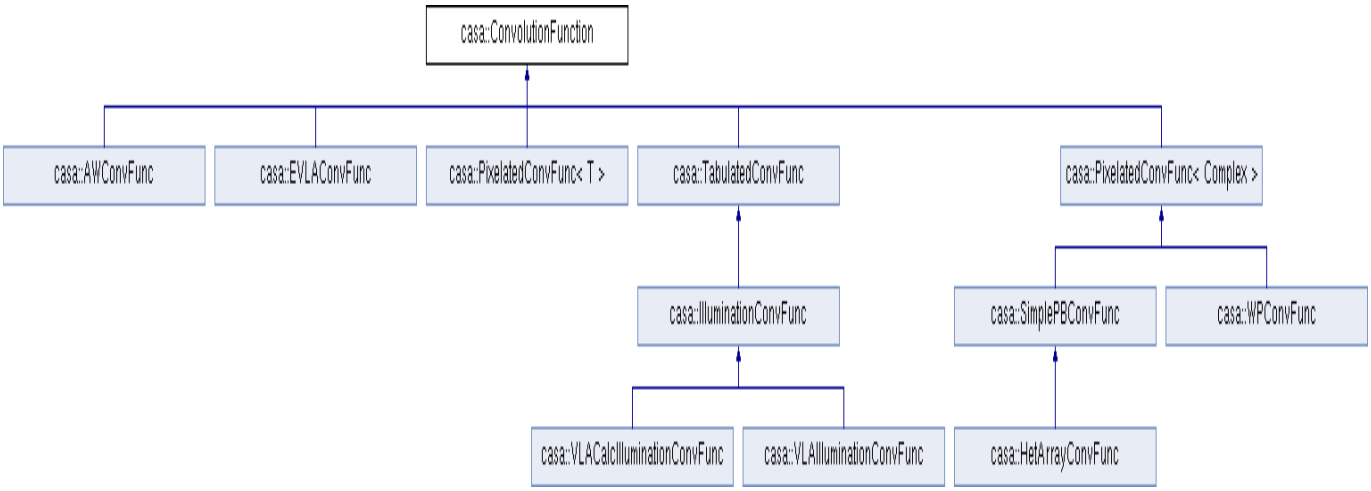
The inheritance diagram for **FTMachine** is as follows:



2.4.1 ConvolutionFunction

The **ConvolutionFunction** base class computes/enacpsulates convolution functions for convolutional gridding. The standard method of re-sampling data to or from a regular grid is done by convolutional gridding. This requires a convolution function with a finte support size and a well-behaved function in the Fourier domain. For standard gridding, the Prolate Spheroidal functions are used. **Convolution functions used in Projection algorithms (such as A-Projection, W-Projection, etc. and their combinations) each require potentially different mechanisms to compute**. These are implemented in separate classes (e.g., **VLACalcIlluminationConvFunc, IlluminationConvFunc**) . Since these are used in a common framework for gridding and de-gridding, they are all derived from the common **ConvolutionFunction** base class.

The inheritance diagram for the **ConvolutionFunction** class is as follows:



In the **ConvolutionFunction** class, the **makeConvFunction(…)** method/member function computes the convolution function and the convolution function used for gridding the weights (typically these are the same) and returns them in the **cfs** and **cfwts** parameters. The required information about the image and visibility parameters is dervided from the given image and **VisBuffer** objects. **wConvSize** is the number of w-term planes and **pa** is the Parallactic Angle in radians for which the convolution function(s) are computed, i.e.,

```
virtual void makeConvFunction(const ImageInterface<Complex>& image,
                      const VisBuffer& vb,
                      const Int wConvSize,
                      const Float pa,
                      CFStore& cfs,
                      CFStore& cfwts) = 0;
```

The **makeConvFunction(…)** method is implemented in **nPBWProjectFT**.

The **makeAverageResponse(…)** method computes the average response function. This is typically image-plane equivalent of the convolution functions, averaged over various axis. The precise averaging will be implementation-dependent in the derived classes, i.e.,

```
    virtual Bool makeAverageResponse(const VisBuffer& vb,
                         const ImageInterface<Complex>& image,
                         ImageInterface<Float>& theavgPB,
                         Bool reset=True) = 0;
    virtual Bool makeAverageResponse(const VisBuffer& vb,
                         const ImageInterface<Complex>& image,
                         ImageInterface<Complex>& theavgPB,
                         Bool reset=True) = 0;
```

The **makeAverageReponse(…)** method is implemented in **IlluminationConvFunc** and re-implemented in **VLACalcIlluminationConvFunc**.

2.4.2 IlluminationConvFunc

The **IlluminationConvFunc** class is a derived class of **ConvolutionFunction** that implements, e.g., the **makeAverageResponse(…)** method as well as the **makeConvFunction(…) method** described in the previous section, i.e,

```
virtual Bool makeAverageResponse(const VisBuffer& vb,
                            const ImageInterface<Complex>& image,
                            ImageInterface<Float>& theavgPB,
                            Bool reset=True)
      {throw(AipsError("IlluminationConvFunc::makeAverageRes() called"));};

    virtual Bool makeAverageResponse(const VisBuffer& vb,
                            const ImageInterface<Complex>& image,
                            ImageInterface<Complex>& theavgPB,
                            Bool reset=True)
      {throw(AipsError("IlluminationConvFunc::makeAverageRes() called"));};
```

and

```
void makeConvFunction(const casa::ImageInterface<std::complex<float> >&,
                  const casa::VisBuffer&, casa::Int, casa::Float,
                  casa::CFStore&, casa::CFStore&) {};
```

IlluminationConvFunc is used/called in **VLACalcIlluminationConvFunc** as well as **nPBWProjectFT**.

2.4.3 VLACalcIlluminationConvFunc

The **VLACalcIlluminationConvFunc** class is another derived class of **ConvolutionFunction** and it also re-implements the **makeAverageResponse(…)** method and makes use of **IlluminationConvFunc,** as discussed in Section 2.4.2. **VLACalcIlluminationConvFunc** is involved in calculating the aperture illumination function (AIF) (the Fourier transform of which gives the voltage pattern) of the VLA that is used in the construction of the gridding convolution function (GCF) by the **nPBWProjectFT** class in the implementation of the A-Projection algorithm.

In conjunction/addition to carrying out AIF computations, **VLACalcIlluminationFunc** carrys out the following tasks:

- Write the primary beam (PB) to the pbImage:

```
void VLACalcIlluminationConvFunc::applyPB(ImageInterface<Float>& pbImage,
                                  const VisBuffer& vb, const Vector<Float>& paList,
                                  Int bandID, Bool doSquint)
  {
    CoordinateSystem skyCS(pbImage.coordinates());
    IPosition skyShape(pbImage.shape());
    TempImage<Complex> uvGrid;
    if (maximumCacheSize() > 0) uvGrid.setMaximumCacheSize(maximumCacheSize());
    //    regridAperture(skyCS, skyShape, uvGrid, vb, paList, False, bandID);
    regridAperture(skyCS, skyShape, uvGrid, vb, paList, doSquint, bandID);
    fillPB(*(ap.aperture),pbImage);  } ….
```

- Write PB^2 to the pbImage:

```
void VLACalcIlluminationConvFunc::applyPBSq(ImageInterface<Float>& pbImage,
```

```
                                        const VisBuffer& vb,
                                        const Vector<Float>& paList,
                                        Int bandID,
                                        Bool doSquint)
    {
      CoordinateSystem skyCS(pbImage.coordinates());
      IPosition skyShape(pbImage.shape());
      TempImage<Complex> uvGrid;
      if (maximumCacheSize() > 0) uvGrid.setMaximumCacheSize(maximumCacheSize());
      //    regridAperture(skyCS, skyShape, uvGrid, vb, paList, False, bandID);
      regridAperture(skyCS, skyShape, uvGrid, vb, paList, doSquint, bandID);

      fillPB(*(ap.aperture),pbImage, True);
    } …..
```

- Fourier transform the re-gridded Fourier plane to get the PB:

```
  ap.aperture->setCoordinateInfo(uvCoords);

     ftAperture(*(ap.aperture));
     // if (doSquint==True)
     //   {
     //     String name("ftapperture.im");
     //     storeImg(name,*(ap.aperture));
     //   }
       }
    }
    void VLACalcIlluminationConvFunc::regridAperture(CoordinateSystem& skyCS,
                                        IPosition& skyShape,
                                        TempImage<Complex>& uvGrid,
                                        const VisBuffer &vb,
                                        const Vector<Float>& paList,
                                        Bool doSquint, Int bandID)
    {
      CoordinateSystem skyCoords(skyCS);  …..
```

- Make **SkyJones** (for modeling the VP/PB) and SkyMuller:

```
void VLACalcIlluminationConvFunc::ftAperture(TempImage<Complex>& uvgrid)
  {
    //
    // Make SkyJones
    //
    LatticeFFT::cfft2d(uvgrid);
    //
    // Now make SkyMuller
    //
    skyMuller(uvgrid);
  }
void VLACalcIlluminationConvFunc::loadFromImage(String& fileName)
  {
    throw(AipsError("VLACalcIlluminationConvFunc::loadFromImage() not yet
supported."));
  }; ……
```

- Do pixel-by-pixel multiplications of the Jones planes, computing only the diagonal of the SkyMuller.

Note that through the use and implementation of the **BeamCalc** class, **VLACalcIlluminationConvFunc** also makes use of a feed pattern text file (~/casa/data/nrao/VLA/VLA.surface) that contains 3 columns of angle (in degrees) and power (in dBi) VLA data. This data is used in the (VLA) aperture illumination (convolution) function calculations.

2.4.4 nPBWProjectFT: Implementation of the A-Projection Algorithm

nPBWProjectFT does Grid-based Fourier transforms which also includes the effects of primary beam and antenna pointing offsets. The **SkyEquation** needs to be able to perform Fourier transforms on visibility data and **nPBWProjectFT** allows efficient handling of direction dependent effects due to the primary beam and antenna pointing offsets using a **VisBuffer** which encapsulates a chunk of visibility (typically all baselines for one time) together with all the information needed for processing (e.g. UVW coordinates). The **nPBWProjectFT** class encapsulate the correction of direction dependent effects via visibility plane convolutions with a potentially different gridding convolution function for each baseline.

Basically, **FTMachine** produces the model image, and **nPBWProjectFT** ist the FTMachine that is capable of dealing with primary beam corruptions and pointing offsets. **nPBWProjectFT** uses an aperture illumination function model for an antenna to calculate the convolution function used in gridding. In CASA, this gridding convolution function includes a complex spheirodal function that enables the inclusion of pointing offsets as phase offsets.

Using the **nPBWProjectFT** FTMachine, errors due to antenna pointing offsets can be corrected during deconvolution. One form of antenna pointing error which is known a-priori is the VLA polarization squint (about 6% of the Primary beam width at any frequency). For Stokes imaging, using this FTMachine, the VLA polarization squint and beam polarization can also be corrected. Also since the effects of antenna pointing errors is strongest in the range of 1-2GHz band (where the sky is not quite empty while the beams are not too large either), this FTMachine can also be se tup to correct for the w-term, though this may not be fully implemented and functional as of CASA 3.2.0 active developer release.

Switches are provided in the **get()** method to compute the derivatives with respect to the parameters of the primary beam (only pointing offsets for now). This is used in the pointing offset solver.

The A-Projection algorithm and the **nPBWProjectFT** FTMachine is invoked/called by python clean tasks (e.g., cleanhelper.py) via lines such as

```
def getFTMachine(gridmode, imagermode, mode, wprojplanes, userftm):
    """
    A utility function which implements the logic to determine the
    ftmachine name to be used in the under-laying tool.
    """
#    ftm = userftm;
    ftm='ft';
    if ((gridmode == 'widefield') and(wprojplanes > 1)): ftm = 'wproject';
    elif (gridmode == 'aprojection'):                    ftm = 'pbwproject';
    elif (imagermode == 'csclean'):                      ftm = 'ft';
    elif (imagermode == 'mosaic'):                       ftm = userftm;
    return ftm;
```

and via the **Imager** class (**Imager2.cc),** i.e.,

```
// Make PBWProject FT machine (for non co-planar imaging with
  // antenna based PB corrections)
  //
  else if (ftmachine_p == "pbwproject"){
    if (wprojPlanes_p<=1)
      {
        os << LogIO::NORMAL
```

```
        << "You are using wprojplanes=1. Doing co-planar imaging (no w-projection needed)"
        << LogIO::POST;
     os << LogIO::NORMAL << "Performing pb-projection" << LogIO::POST; // Loglevel PROGRESS
   }
  if((wprojPlanes_p>1)&&(wprojPlanes_p<64))
    {
     os << LogIO::WARN
        << "No. of w-planes set too low for W projection - recommend at least 128"
        << LogIO::POST;
     os << LogIO::NORMAL << "Performing pb + w-plane projection" // Loglevel PROGRESS
        << LogIO::POST;
    }


  if(!gvp_p)
    {
     os << LogIO::NORMAL // Loglevel INFO
         << "Using defaults for primary beams used in gridding" << LogIO::POST;
     gvp_p = new VPSkyJones(*ms_p, True, parAngleInc_p, squintType_p,
                            skyPosThreshold_p);
    }

  ft_p = new nPBWProjectFT(//*ms_p,
                           wprojPlanes_p, cache_p/2,
                           cfCacheDirName_p, doPointing, doPBCorr,
                            tile_p, paStep_p, pbLimit_p, True);
((nPBWProjectFT *)ft_p)->setPAIncrement(parAngleInc_p);
  if (doPointing)
    {
```

etc…

Described below are the various tasks carried out by **nPBWProjectFT** and, in many cases, the various methods/member functions (including relevant samples and examples of code from **nPBWProjectFT.{h,cc}**) that are involved:

- Get various parameters from the visibilities:

```
bandID_p=-1;
if (applyPointingOffset) doPointing=1; else doPointing=0;

convFuncCacheReady=False;
PAIndex = -1;
maxConvSupport=-1;
```

- Set up the Convolution Funcion disk cache manager object.
- Set up image cache needed for gridding.

```
if(imageCache) delete imageCache;   imageCache=0;
```

- The tile size should be large enough that the extended convolution function can fit easily…
- Make the primary beam part of the convolution function and make an image with circular polarization axis:

```
Int nPBWProjectFT::makePBPolnCoords(CoordinateSystem& squintCoord,const
VisBuffer&vb)
  {
    //
    // Make an image with circular polarization axis.
    //
    Int NPol=0,M,N=0;
    M=polMap.nelements();
    for(Int i=0;i<M;i++) if (polMap(i) > -1) NPol++;
    Vector<Int> poln(NPol);
```

```
    Int index;
    Vector<Int> inStokes;
    index = squintCoord.findCoordinate(Coordinate::STOKES);
    inStokes = squintCoord.stokesCoordinate(index).stokes();
    N = 0;
```

etc..

- Make an object to hold the observatory position information, make a frame object out of the observatory position and time objects, and, finally, make the convert machine.
- Make a Coordinate Conversion Machine to go from (Az,El) to (HA,Dec).
- Compute reference (HA,Dec)
- Convert reference (HA,Dec) to reference (Az,El)
- Convert the antenna pointing offsets from (Az,El)-->(RA,Dec)-->(l,m)
- From (Az,El) -> (HA,Dec); add (Az,El) offsets to the reference (Az,El)
- Convert offsetted (Az,El) to (HA,Dec) and then to (RA,Dec)
- Convert offsetted (RA,Dec) -> (l,m)
- Make a sensitivity image (sensitivityImage), given the gridded weights (wtImage). These are related to each other by a Fourier transform and normalization by the sum-of-weights (sumWt) and normalization by the product of the 2D FFT size along each axis. If doFFTNorm=False, normalization by the FFT size is not done. If sumWt is not provided, normalization by the sum of weights is also not done.

```
void nPBWProjectFT::makeSensitivityImage(Lattice<Complex>& wtImage,
                                 ImageInterface<Float>& sensitivityImage,
                                 const Matrix<Float>& sumWt,
                                 const Bool& doFFTNorm)
  {
    Bool doSumWtNorm=True;
    if (sumWt.shape().nelements()==0) doSumWtNorm=False;

    if ((sumWt.shape().nelements() < 2) ||
    (sumWt.shape()(0) != wtImage.shape()(2)) ||
    (sumWt.shape()(1) != wtImage.shape()(3)))
      throw(AipsError("makeSensitivityImage(): "
                "Sum of weights per poln and chan required"));
    Float sumWtVal=1.0;

    LatticeFFT::cfft2d(wtImage,False);
    Int sizeX=wtImage.shape()(0), sizeY=wtImage.shape()(1);
    sensitivityImage.resize(wtImage.shape());
    Array<Float> senBuf;
    sensitivityImage.get(senBuf,False);
    ArrayLattice<Float> senLat(senBuf, True);
```

- Copy one 2D plane at a time, normalizing by the sum of weights and possibly 2D FFT. Also, set up Lattice iterations on wtImage and sensitivityImage

```
    IPosition axisPath(4, 0, 1, 2, 3);
    IPosition cursorShape(4, sizeX, sizeY, 1, 1);
    LatticeStepper wtImStepper(wtImage.shape(), cursorShape, axisPath);
    LatticeIterator<Complex> wtImIter(wtImage, wtImStepper);
    LatticeStepper senImStepper(senLat.shape(), cursorShape, axisPath);
    LatticeIterator<Float> senImIter(senLat, senImStepper);
```

- Iterate over channel and polarization axis

```
    if (!doFFTNorm) sizeX=sizeY=1;
    for(wtImIter.reset(),senImIter.reset();  !wtImIter.atEnd(); wtImIter++,senImIter++)
      {
    Int pol=wtImIter.position()(2), chan=wtImIter.position()(3);
    if (doSumWtNorm) sumWtVal=sumWt(pol,chan);
    senImIter.rwCursor() = (real(wtImIter.rwCursor())
```

```
                        *Float(sizeX)*Float(sizeY)/sumWtVal);
      }
```

- The following code is averaging RR and LL planes and writing the result back to both planes. This needs to be generalized for the full polarization case.

```
  IPosition start0(4,0,0,0,0), start1(4,0,0,1,0), length(4,sizeX,sizeY,1,1);
Slicer slicePol0(start0,length), slicePol1(start1,length);
Array<Float> polPlane0, polPlane1;
senLat.getSlice(polPlane0,slicePol0);
senLat.getSlice(polPlane1,slicePol1);
polPlane0=(polPlane0+polPlane1)/2.0;
polPlane1=polPlane0;
```

- Normalization of the position angle averaged (and potentially antenna averaged) primary beam

```
void nPBWProjectFT::normalizeAvgPB()
  {
    if (!pbNormalized)
      {
    pbPeaks.resize(avgPB.shape()(2),True);
    if (makingPSF) pbPeaks = 1.0;
    else pbPeaks /= (Float)noOfPASteps;
    pbPeaks = 1.0;
    logIO() << LogOrigin("nPBWProjectFT", "normalizeAvgPB")
            << "Normalizing the average PBs to " << 1.0
            << LogIO::NORMAL
            << LogIO::POST;

    IPosition avgPBShape(avgPB.shape()),ndx(4,0,0,0,0);
    Vector<Float> peak(avgPBShape(2));

    Bool isRefF;
    Array<Float> avgPBBuf;
    isRefF=avgPB.get(avgPBBuf);

    Float pbMax = max(avgPBBuf);

    ndx=0;
    for(ndx(1)=0;ndx(1)<avgPBShape(1);ndx(1)++)
      for(ndx(0)=0;ndx(0)<avgPBShape(0);ndx(0)++)
        {
          IPosition plane1(ndx);
          plane1=ndx;
          plane1(2)=1; // The other poln. plane
          avgPBBuf(ndx) = (avgPBBuf(ndx) + avgPBBuf(plane1))/2.0;
        }
    for(ndx(1)=0;ndx(1)<avgPBShape(1);ndx(1)++)
      for(ndx(0)=0;ndx(0)<avgPBShape(0);ndx(0)++)
        {
          IPosition plane1(ndx);
          plane1=ndx;
          plane1(2)=1; // The other poln. plane
          avgPBBuf(plane1) = avgPBBuf(ndx);
        }
    if (fabs(pbMax-1.0) > 1E-3)
      {
        //      avgPBBuf = avgPBBuf/noOfPASteps;
        for(ndx(3)=0;ndx(3)<avgPBShape(3);ndx(3)++)
          for(ndx(2)=0;ndx(2)<avgPBShape(2);ndx(2)++)
            {
              peak(ndx(2)) = 0;
              for(ndx(1)=0;ndx(1)<avgPBShape(1);ndx(1)++)
                for(ndx(0)=0;ndx(0)<avgPBShape(0);ndx(0)++)
                  if (abs(avgPBBuf(ndx)) > peak(ndx(2)))
                    peak(ndx(2)) = avgPBBuf(ndx);

              for(ndx(1)=0;ndx(1)<avgPBShape(1);ndx(1)++)
                for(ndx(0)=0;ndx(0)<avgPBShape(0);ndx(0)++)
                  avgPBBuf(ndx) *= (pbPeaks(ndx(2))/peak(ndx(2)));
            }
        if (isRefF) avgPB.put(avgPBBuf);
      }
      }
```

```
    pbNormalized = True;
  }
```

- Resize the average PB if this is the first time

```
     if (resetPBs)
      {
    logIO() << "Initializing the average PBs"
           << LogIO::NORMAL
           << LogIO::POST;
    theavgPB.resize(localPB.shape());
    theavgPB.setCoordinateInfo(localPB.coordinates());
    theavgPB.set(0.0);
    noOfPASteps = 0;
    pbPeaks.resize(theavgPB.shape()(2));
    pbPeaks.set(0.0);
    resetPBs=False;
      }
```

- Make the Stokes PB

```
     localPB.set(1.0);

    {
      VLACalcIlluminationConvFunc vlaPB;
      if (bandID_p == -1) bandID_p=getVisParams(vb);
      vlaPB.applyPB(localPB, vb, bandID_p);
    }

    IPosition twoDPBShape(localPB.shape());
    TempImage<Complex>
localTwoDPB(twoDPBShape,localPB.coordinates());
    localTwoDPB.setMaximumCacheSize(cachesize);
    Float peak=0;
    Int NAnt;
    noOfPASteps++;
    NAnt=1;

     for(Int ant=0;ant<NAnt;ant++)
       { //Ant loop
   {
     IPosition ndx(4,0,0,0,0);
     for(ndx(0)=0; ndx(0)<twoDPBShape(0); ndx(0)++)
       for(ndx(1)=0; ndx(1)<twoDPBShape(1); ndx(1)++)
         for(ndx(2)=0; ndx(2)<twoDPBShape(2); ndx(2)++)
          for(ndx(3)=0; ndx(3)<twoDPBShape(3); ndx(3)++)
            localTwoDPB.putAt(Complex((localPB(ndx)),0.0),ndx);
```

- Accumulate the shifted PBs. (If antenna pointing errors are not applied, no shifting (which can be expensive) is required). Then, an average PB is made and put in the memory cache.

```
   {
     Bool isRefF,isRefC;
     Array<Float> fbuf;
     Array<Complex> cbuf;
     isRefF=theavgPB.get(fbuf);
     isRefC=localTwoDPB.get(cbuf);

     IPosition fs(fbuf.shape());
     {
       IPosition ndx(4,0,0,0,0),avgNDX(4,0,0,0,0);
       for(ndx(3)=0,avgNDX(3)=0;ndx(3)<fs(3);ndx(3)++,avgNDX(3)++)
       {
       for(ndx(2)=0,avgNDX(2)=0;ndx(2)<twoDPBShape(2);ndx(2)++,avgNDX(2)++)
         {
          for(ndx(0)=0,avgNDX(0)=0;ndx(0)<fs(0);ndx(0)++,avgNDX(0)++)
            for(ndx(1)=0,avgNDX(1)=0;ndx(1)<fs(1);ndx(1)++,avgNDX(1)++)
```

```
            {
              Float val;
              val = real(cbuf(ndx));
              fbuf(avgNDX) += val;
              if (fbuf(avgNDX) > peak) peak=fbuf(avgNDX);
            }
          }
        }
      }
      if (!isRefF) theavgPB.put(fbuf);
      pbPeaks += peak;
    }
        }
      theavgPB.setCoordinateInfo(localPB.coordinates());
      return True; // i.e., an average PB was made and is in the memory cache
```

- Locate a gridding convlution function in either mem. or disk cache. Return 1 if found in the disk cache. Return 2 if found in the memory cache. Return <0 if not found in either cache. In this case, absolute of the return value corresponds to the index in the list of convolution functions where this convolution function should be filled.

```
Int nPBWProjectFT::locateConvFunction(Int Nw, Int polInUse,
                            const VisBuffer& vb, Float &pa)
```

  etc …

- Given a polMap (mapping of which visibility polarization is gridded onto which grid plane), make a map of the conjugate planes of the grid, e.g, for Stokes-I and -V imaging, the two planes of the uv-grid are [LL,RR]. For input VisBuffer visibilites in order [RR,RL,LR,LL], polMap = [1,-1,-1,0]. The conjugate map will be [0,-1,-1,1].

```
void nPBWProjectFT::makeConjPolMap(const VisBuffer& vb,
                           const Vector<Int> cfPolMap,
                           Vector<Int>& conjPolMap)
```

  etc. ..

- Make a two dimensional image to calculate the auto-correlation of the ideal illumination pattern. We want this on a fine grid in the UV plane. Note that the visibility plane filter describing the effects of the antenna PBs is the autocorrelation of the antenna AIF.

```
Int directionIndex=coords.findCoordinate(Coordinate::DIRECTION);
AlwaysAssert(directionIndex>=0, AipsError);
DirectionCoordinate dc=coords.directionCoordinate(directionIndex);
directionCoord=coords.directionCoordinate(directionIndex);
Vector<Double> sampling;
sampling = dc.increment();
sampling*=Double(convSampling);
sampling*=Double(nx)/Double(convSize);
dc.setIncrement(sampling);
Vector<Double> unitVec(2);
unitVec=convSize/2;
dc.setReferencePixel(unitVec);
```

- Set the reference value to that of the image

```
coords.replaceCoordinate(dc, directionIndex);
```

- Make an image with circular polarization axis. Return the number of visibility polarization planes that will be used in making the user-defined Stokes image.

```
    polInUse=makePBPolnCoords(coords,vb);

    Float pa;
    Int cfSource=locateConvFunction(wConvSize, polInUse, vb, pa);

    lastPAUsedForWtImg = currentCFPA = pa;

    Bool pbMade=False;
    if (cfSource==1) // CF found and loaded from the disk cache
      {
   polInUse = convFunc.shape()(3);
   wConvSize = convFunc.shape()(2);
   try
     {
       cfCache.loadAvgPB(avgPB);
       avgPBReady=True;
     }
   catch (AipsError& err)
     {
       logIO() << "Average PB does not exist in the cache.  A fresh one will be
made."
               << LogIO::NORMAL << LogIO::POST;
       pbMade=makeAveragePB0(vb, image, polInUse, avgPB);
       pbNormalized=False; normalizeAvgPB(); pbNormalized=True;
     }
       }
    else if (cfSource==2)  // CF found in the mem. cache
      {
      }
    else                       // CF not found in either cache

      {
```

- **Make the Gridding Convolution Function, update the average PB, and update the Convolution Function and the avgPB disk cache.**

```
   PAIndex = abs(cfSource);
```

  - Load the average PB from the disk since it's going to be updated in memory and on the disk. Without loading it from the disk (from a potentially more complete existing cache), the average PB can get inconsistent with the rest of the cache.

```
   makeConvFunction(image,vb,pa);
   try
     {
       cfCache.loadAvgPB(avgPB);
       resetPBs = False;
       avgPBReady=True;
     }
   catch(SynthesisFTMachineError &err)
     {
       logIO() << LogOrigin("nPBWProjectFT::findConvFunction()","")
               << "Average PB does not exist in the cache.  A fresh
one will be made."
               << LogIO::NORMAL
               << LogIO::POST;
       pbMade=makeAveragePB0(vb, image, polInUse,avgPB);
     }

   //     makeAveragePB(vb, image, polInUse,avgPB);
   pbNormalized=False;
   normalizeAvgPB();
   pbNormalized=True;
   Int index=coords.findCoordinate(Coordinate::SPECTRAL);
   SpectralCoordinate spCS = coords.spectralCoordinate(index);
```

```
Vector<Double> refValue; refValue.resize(1);refValue(0)=cfRefFreq_p;
spCS.setReferenceValue(refValue);
coords.replaceCoordinate(spCS,index);
cfCache.cacheConvFunction(PAIndex, pa, convFunc, coords, convFuncCS_p,
                          convSize, convSupport,convSampling);
Cube<Int> convWtSize=convSupport*CONVWTSIZEFACTOR;
cfCache.cacheConvFunction(PAIndex, pa, convWeights, coords, convFuncCS_p,
                          convSize, convWtSize,convSampling,"WT",False);
cfCache.finalize(); // Write the aux info file
if (pbMade) cfCache.finalize(avgPB); // Save the AVG PB and write the aux info.
    }

  verifyShapes(avgPB.shape(), image.shape());

  Int lastPASlot = PAIndex;

  if (paChangeDetector.changed(vb,0)) paChangeDetector.update(vb,0);
```

- Show the list of support sizes along the w-axis for the current PA.
- **Method for making the GCF**

```
   void nPBWProjectFT::makeConvFunction(const ImageInterface<Complex>& image,
                                        const VisBuffer& vb,Float pa)
     {
       if (bandID_p == -1) bandID_p=getVisParams(vb);
       Int NNN=0;
       logIO() << LogOrigin("nPBWProjectFT", "makeConvFunction")
          << "Making a new convolution function for PA="
          << pa*(180/C::pi) << "deg"
          << LogIO::NORMAL
          << LogIO::POST;

       if(wConvSize>NNN) {
         logIO() << "Using " << wConvSize << " planes for W-projection" <<
   LogIO::POST;
         Double maxUVW;
         maxUVW=0.25/abs(image.coordinates().increment()(0));
         logIO() << "Estimating maximum possible W = " << maxUVW
            << " (wavelengths)" << LogIO::POST;

         Double invLambdaC=vb.frequency()(0)/C::c;
   //       logIO() << "Typical wavelength = " << 1.0/invLambdaC
   //       << " (m)" << LogIO::POST;
         Double invMinL = vb.frequency()((vb.frequency().nelements())-1)/C::c;
         logIO() << "wavelength range = " << 1.0/invLambdaC << " (m) to "
            << 1.0/invMinL << " (m)" << LogIO::POST;
         if (wConvSize > 1)
       {
         uvScale(2)=Float((wConvSize-1)*(wConvSize-1))/maxUVW;
         logIO() << "Scaling in W (at maximum W) = " << 1.0/uvScale(2)
               << " wavelengths per pixel" << LogIO::POST;
       }
        }

        // Get the coordinate system

        CoordinateSystem coords(image.coordinates());
```

- **Set up the GCF**.

```
        if(wConvSize>NNN)
          {
        if(wConvSize>256)
          {
            convSampling=4;
            convSize=min(nx,512);
```

```
  }
else
  {
    convSampling=4;
    convSize=min(nx,2048);
  }
  }
 else
   {
convSampling=4;
convSize=nx;
   }
 convSampling=OVERSAMPLING;
 convSize=CONVSIZE;
```

- Make a two dimensional image to calculate auto-correlation of the ideal illumination pattern. We want this on a fine grid in the UV plane.
- Set the reference value to that of the image.
- Make an image with circular polarization axis. Return the number of visibility polarization planes that will be used in making the user defined Stokes image.
- **Make the sky Stokes PB**. This will be used in the gridding correction step.

```
IPosition pbShape(4, convSize, convSize, polInUse, 1);
TempImage<Complex> twoDPB(pbShape, coords);

IPosition pbSqShp(pbShape);
  //    pbSqShp[0] *= 2;    pbSqShp[1] *= 2;
  unitVec=pbSqShp[0]/2;
  dc.setReferencePixel(unitVec);
  // sampling *= Double(2.0);
  // dc.setIncrement(sampling);
  coords.replaceCoordinate(dc, directionIndex);
  TempImage<Complex> twoDPBSq(pbSqShp,coords);
  twoDPB.setMaximumCacheSize(cachesize);
  twoDPB.set(Complex(1.0,0.0));
  twoDPBSq.setMaximumCacheSize(cachesize);
  twoDPBSq.set(Complex(1.0,0.0));
```

- Accumulate the various terms that constitute the **gridding convolution function**.

```
Bool writeResults;
    writeResults=False;

    Int inner=convSize/convSampling;
    //    inner = convSize/2;

    Vector<Double> cfUVScale(3,0),cfUVOffset(3,0);

    cfUVScale(0)=Float(twoDPB.shape()(0))*sampling(0);
    cfUVScale(1)=Float(twoDPB.shape()(1))*sampling(1);
    cfUVOffset(0)=Float(twoDPB.shape()(0))/2;
    cfUVOffset(1)=Float(twoDPB.shape()(1))/2;
    ConvolveGridder<Double, Complex>
      //      ggridder(IPosition(2, inner, inner), cfUVScale, cfUVOffset, "SF");
      ggridder(IPosition(2, inner, inner), uvScale, uvOffset, "SF");
// Each convolution function itself is a complex 3D array
(U,V,W) per parallactic angle (PA).
convFuncCache[PAIndex] = new Array<Complex>(IPosition(4,convSize,convSize,
                                                      wConvSize,polInUse));
    convWeightsCache[PAIndex] = new Array<Complex>(IPosition(4,convSize,convSize,
                                                             wConvSize,polInUse));
    convFunc.reference(*convFuncCache[PAIndex]);
    convWeights.reference(*convWeightsCache[PAIndex]);
```

```
convFunc=0.0;
convWeights=0.0;

IPosition start(4, 0, 0, 0, 0);
IPosition pbSlice(4, convSize, convSize, 1, 1);

Matrix<Complex> screen(convSize, convSize);
if (paChangeDetector.changed(vb,0)) paChangeDetector.update(vb,0);
VLACalcIlluminationConvFunc vlaPB;
vlaPB.setMaximumCacheSize(cachesize);

for (Int iw=0;iw<wConvSize;iw++)
  {
screen = 1.0;

/*
screen=0.0;
// First the complex prolate spheroidal function
//
//      inner=convSize/2;
//     screen = 0.0;
Vector<Complex> correction(inner);
for (Int iy=-inner/2;iy<inner/2;iy++)
  {
    ggridder.correctX1D(correction, iy+inner/2);
    for (Int ix=-inner/2;ix<inner/2;ix++)
      screen(ix+convSize/2,iy+convSize/2)=correction(ix+inner/2);
    // if (iy==0)
    //   for (Int ii=0;ii<inner;ii++)
    //  cout << ii << " " << correction(ii) << endl;
  }
*/
//
// Now the w term
//
if(wConvSize>1)
  {
    logIO() << LogOrigin("nPBWProjectFT", "")
            << "Computing WPlane " << iw  << LogIO::POST;

    Double twoPiW=2.0*C::pi*Double(iw*iw)/uvScale(2);

    for (Int iy=-inner/2;iy<inner/2;iy++)
      {
        Double m=sampling(1)*Double(iy);
        Double msq=m*m;
        for (Int ix=-inner/2;ix<inner/2;ix++)
          {
            Double l=sampling(0)*Double(ix);
            Double rsq=l*l+msq;
            if(rsq<1.0)
              {
                Double phase=twoPiW*(sqrt(1.0-rsq)-1.0);

screen(ix+convSize/2,iy+convSize/2)*=Complex(cos(phase),sin(phase));
              }
          }
      }
  }

// Fill the complex image with the w-term...

IPosition PolnPlane(4,0,0,0,0);
IPosition ndx(4,0,0,0,0);

for(Int i=0;i<polInUse;i++)
  {
    PolnPlane(2)=i;
    twoDPB.putSlice(screen, PolnPlane);
    twoDPBSq.putSlice(screen, PolnPlane);
  }
// {
//   Vector<IPosition> posMax(twoDPB.shape()(2));
//   posMax(0)(0)=pbShape(0)/2;
//   posMax(0)(1)=pbShape(1)/2;
//   posMax(1)(0)=pbShape(0)/2;
```

```
    //   posMax(1)(1)=pbShape(1)/2;
    //   getVisParams(vb);
    //   applyAntiAliasingOp(twoDPB,posMax,0);
    // }
    //
    // Apply the PB...
    Bool doSquint=True;
    vlaPB.applyPB(twoDPB, vb, bandID_p, doSquint);
    doSquint = False;
    //      vlaPB.applyPBSq(twoDPBSq, vb, bandID_p, doSquint);
    vlaPB.applyPB(twoDPBSq, vb, bandID_p, doSquint);
    /*
//  twoDPB.put(abs(twoDPB.get()));
//  twoDPBSq.put(abs(twoDPBSq.get()));
        */

    // {
    //   String name("twoDPB.before.im");
    //   storeImg(name,twoDPB);
    // }
    // {
    //   //
    //   // Apply (multiply) by the Spheroidal functions
    //   //
    //   Vector<Float> maxVal(twoDPB.shape()(2));
    //   Vector<IPosition> posMax(twoDPB.shape()(2));

    //   SynthesisUtils::findLatticeMax(twoDPB,maxVal,posMax);
    //   posMax(0)(0)+=1;
    //   posMax(0)(1)+=1;
    //   posMax(1)(0)+=1;
    //   posMax(1)(1)+=1;
    //   //   applyAntiAliasingOp(twoDPB,posMax,1);

    //   SynthesisUtils::findLatticeMax(twoDPBSq,maxVal,posMax);
    //   posMax(0)(0)+=1;
    //   posMax(0)(1)+=1;
    //   posMax(1)(0)+=1;
    //   posMax(1)(1)+=1;
    //   //   applyAntiAliasingOp(twoDPBSq,posMax,1,True);
    // }

    Complex cpeak=max(twoDPB.get());
    twoDPB.put(twoDPB.get()/cpeak);
    cpeak=max(twoDPBSq.get());
    twoDPBSq.put(twoDPBSq.get()/cpeak);
    //      twoDPBSq.set(1.0);
    // {
    //   String name("twoDPB.im");
    //   storeImg(name,twoDPB);
    // }

    CoordinateSystem cs=twoDPB.coordinates();
    Int index= twoDPB.coordinates().findCoordinate(Coordinate::SPECTRAL);
    SpectralCoordinate SpCS = twoDPB.coordinates().spectralCoordinate(index);

    cfRefFreq_p=SpCS.referenceValue()(0);

// Now FFT and get the result back
    //
    // {
    //   String name("twoDPB.im");
    //   storeImg(name,twoDPB);
    // }
    LatticeFFT::cfft2d(twoDPB);
    LatticeFFT::cfft2d(twoDPBSq);
    // {
    //   String name("twoDPBFT.im");
    //   storeImg(name,twoDPB);
    // }
// Fill the gridding convolution function planes with the result.

    {
      // IPosition start(4, convSize/4, convSize/4, 0, 0),
      //   pbSlice(4, convSize/2-1, convSize/2-1, polInUse, 1);
      // IPosition sliceStart(4,0,0,iw,0),
```

```
            //   sliceLength(4,convSize/2-1,convSize/2-1,1,polInUse);

            IPosition start(4, 0, 0, 0, 0),
              pbSlice(4, twoDPB.shape()[0]-1, twoDPB.shape()[1]-1, polInUse, 1);
            IPosition sliceStart(4,0,0,iw,0),
              sliceLength(4,convFunc.shape()[0]-1,convFunc.shape()[1]-1,1,polInUse);

            convFunc(Slicer(sliceStart,sliceLength)).nonDegenerate()
              =(twoDPB.getSlice(start, pbSlice, True));

            IPosition shp(twoDPBSq.shape());
            Int bufSize=convWeights.shape()[0], Org=shp[0]/2;
            // IPosition sqStart(4, Org-bufSize/2, Org-bufSize/2, 0, 0),
            //   pbSqSlice(4, bufSize-1, bufSize-1, polInUse, 1);
            // IPosition sqSliceStart(4,0,0,iw,0),
            //   sqSliceLength(4,bufSize-1,bufSize-1,1,polInUse);

            IPosition sqStart(4, 0, 0, 0, 0),
              pbSqSlice(4, shp[0]-1, shp[1]-1, polInUse, 1);
            IPosition sqSliceStart(4,0,0,iw,0),
              sqSliceLength(4,shp[0]-1,shp[1]-1,1,polInUse);

            convWeights(Slicer(sqSliceStart,sqSliceLength)).nonDegenerate()
              =(twoDPBSq.getSlice(sqStart, pbSqSlice, True));

          }
           }

          {
            Complex cpeak = max(convFunc);
            convFunc/=cpeak;
            //      cout << "#### max(convFunc) = " << cpeak << endl;
            cpeak=max(convWeights);
            //      cout << "#### max(convWeights) = " << cpeak << endl;
            convWeights/=cpeak;
            //      cout << "#### max(convWeights) = " << max(convWeights) << endl;
          }
```

// Find the gridding convolution function support size. No assumption about the symmetry of the GCF can be made (except that they are same for all polarization planes).

```
    Int lastPASlot=PAIndex;
    for(Int iw=0;iw<wConvSize;iw++)
      for(Int ipol=0;ipol<polInUse;ipol++)
  convSupport(iw,ipol,lastPASlot)=-1;
```

//Caution: This assumes that the support size at each Poln. is
// the same, starting from the center pixel (in pixel
// co-ordinates). For large pointing offsets, this might not be
// true.

```
    //
    //    Int ConvFuncOrigin=convSize/4;  // Conv. Func. is half that size of convSize
    Int ConvFuncOrigin=convFunc.shape()[0]/2;  // Conv. Func. is half that size of convSize
    IPosition ndx(4,ConvFuncOrigin,0,0,0);
    //    Cube<Int> convWtSupport(convSupport.shape());
    convWtSupport.resize(convSupport.shape(),True);
    Int maxConvWtSupport=0, supportBuffer;
    for (Int iw=0;iw<wConvSize;iw++)
      {
  Bool found=False;
  Float threshold;
  Int R;
  ndx(2) = iw;

  ndx(0)=ndx(1)=ConvFuncOrigin;
  ndx(2) = iw;
  //      Complex maxVal = max(convFunc);
  threshold = abs(convFunc(ndx))*THRESHOLD;
```

// Find the support size of the GCF in pixels

```
  //
  Int wtR;
  found =findSupport(convWeights,threshold,ConvFuncOrigin,wtR);
  found = findSupport(convFunc,threshold,ConvFuncOrigin,R);
```

// Set the support size for each W-plane and for all polarization planes, assuming that the support size for all polarization-planes is the same.

```
  //
```

```
   if(found)
     {
       //      Int maxR=R;//max(ndx(0),ndx(1));
       for(Int ipol=0;ipol<polInUse;ipol++)
         {
           convSupport(iw,ipol,lastPASlot)=Int(R/Float(convSampling));
           convSupport(iw,ipol,lastPASlot)=Int(0.5+Float(R)/Float(convSampling))+1;
           //              convSupport(iw,ipol,lastPASlot) +=
(convSupport(iw,ipol,lastPASlot)+1)%2;
           convWtSupport(iw,ipol,lastPASlot)=Int(R*CONVWTSIZEFACTOR/Float(convSampling));

   convWtSupport(iw,ipol,lastPASlot)=Int(0.5+Float(R)*CONVWTSIZEFACTOR/Float(convSampling))+
1;
           //              convWtSupport(iw,ipol,lastPASlot)=Int(wtR/Float(convSampling));
           //              convWtSupport(iw,ipol,lastPASlot) +=
(convWtSupport(iw,ipol,lastPASlot)+1)%2;
           if ((lastPASlot == 0) || (maxConvSupport == -1))
             if (convSupport(iw,ipol,lastPASlot) > maxConvSupport)
               maxConvSupport = convSupport(iw,ipol,lastPASlot);
           maxConvWtSupport=convWtSupport(iw,ipol,lastPASlot);
         }
     }
      }

    if(convSupport(0,0,lastPASlot)<1)
      logIO() << "Convolution function is misbehaved - support seems to be zero"
         << LogIO::EXCEPTION;

    logIO() << LogOrigin("nPBWProjectFT", "makeConvFunction")
       << "Re-sizing the convolution functions"
       << LogIO::POST;

    {
      supportBuffer = OVERSAMPLING;
      Int bot=ConvFuncOrigin-convSampling*maxConvSupport-supportBuffer,//-convSampling/2,
      top=ConvFuncOrigin+convSampling*maxConvSupport+supportBuffer;//+convSampling/2;
      bot = max(0,bot);
      top = min(top, convFunc.shape()(0)-1);
      {
   Array<Complex> tmp;
   IPosition blc(4,bot,bot,0,0), trc(4,top,top,wConvSize-1,polInUse-1);

   tmp = convFunc(blc,trc);
   (*convFuncCache[lastPASlot]).resize(tmp.shape());
   (*convFuncCache[lastPASlot]) = tmp;
   convFunc.reference(*convFuncCache[lastPASlot]);
      }

      supportBuffer = (Int)(OVERSAMPLING*CONVWTSIZEFACTOR);
      bot=ConvFuncOrigin-convSampling*maxConvWtSupport-supportBuffer;
      top=ConvFuncOrigin+convSampling*maxConvWtSupport+supportBuffer;
      bot=max(0,bot);
      top=min(top,convWeights.shape()(0)-1);
      {
   Array<Complex> tmp;
   IPosition blc(4,bot,bot,0,0), trc(4,top,top,wConvSize-1,polInUse-1);

   tmp = convWeights(blc,trc);
   (*convWeightsCache[lastPASlot]).resize(tmp.shape());
   (*convWeightsCache[lastPASlot]) = tmp;
   convWeights.reference(*convWeightsCache[lastPASlot]);
      }
    }
```

// Normalize such that plane 0 sums to 1 (when jumping in steps of
// convSampling). This is really not necessary here since we do
// the normalizing by the area more accurately in the gridder
// (fpbwproj.f).

```
   ndx(2)=ndx(3)=0;


   Complex pbSum=0.0;
   IPosition peakPix(ndx);
```

```
    Int Nx = convFunc.shape()(0), Ny=convFunc.shape()(1);

    for(Int nw=0;nw<wConvSize;nw++)
      for(Int np=0;np<polInUse;np++)
   {
     ndx(2) = nw; ndx(3)=np;
```

// Locate the pixel with the peak value. That's the origin in pixel co-ordinates.

```
       Float peak=0;
       peakPix = 0;
       for(ndx(1)=0;ndx(1)<convFunc.shape()(1);ndx(1)++)
         for(ndx(0)=0;ndx(0)<convFunc.shape()(0);ndx(0)++)
           if (abs(convFunc(ndx)) > peak) {peakPix = ndx;peak=abs(convFunc(ndx));}
     }

     ConvFuncOrigin = peakPix(0);
    //       ConvFuncOrigin = convFunc.shape()(0)/2+1;
     //      Int thisConvSupport=convSampling*convSupport(nw,np,lastPASlot);
     Int thisConvSupport=convSupport(nw,np,lastPASlot);
     pbSum=0.0;

     for(Int iy=-thisConvSupport;iy<thisConvSupport;iy++)
       for(Int ix=-thisConvSupport;ix<thisConvSupport;ix++)
         {
           ndx(0)=ix*convSampling+ConvFuncOrigin;
           ndx(1)=iy*convSampling+ConvFuncOrigin;
           pbSum += real(convFunc(ndx));
         }
     /*
     for(Int iy=0;iy<Ny;iy++)
       for(Int ix=0;ix<Nx;ix++)
         {
           ndx(0)=ix;ndx(1)=iy;
           pbSum += convFunc(ndx);
         }
     */
     if(pbSum>0.0)
       {
```

// Normalize each polarization plane by the area under its gridding convolution function.

```
Nx = convFunc.shape()(0), Ny = convFunc.shape()(1);
          for (ndx(1)=0;ndx(1)<Ny;ndx(1)++)
            for (ndx(0)=0;ndx(0)<Nx;ndx(0)++)
              {
                convFunc(ndx) /= pbSum;
              }

          Nx = convWeights.shape()(0); Ny = convWeights.shape()(1);
          for (ndx(1)=0;  ndx(1)<Ny;  ndx(1)++)
            for (ndx(0)=0;  ndx(0)<Nx;  ndx(0)++)
              {
                convWeights(ndx) /= pbSum*pbSum;
                // if ((ndx(0)==Nx/2+1) && (ndx(1)==Ny/2+1))
                //   {
                //  convWeights(ndx)=1.0;
                //  cout << ndx << " " << convWeights(ndx) << endl;
                //   }
                // else
                //   convWeights(ndx)=0.0;
              }
       }
     else
       throw(SynthesisFTMachineError("Convolution function integral is not positive"));

     Vector<Float> maxVal(convWeights.shape()(2));
     Vector<IPosition> posMax(convWeights.shape()(2));
     SynthesisUtils::findLatticeMax(convWeights,maxVal,posMax);
     //      cout << "convWeights: " << maxVal << " " << posMax << endl;
   }

  }
```

- Initialize the FFT to the visibility plane using the image as a template. The image is loaded and Fourier transformed.

```
void nPBWProjectFT::initializeToVis(ImageInterface<Complex>& iimage,
                                    const VisBuffer& vb)
  {
    image=&iimage;

    ok();

    init();
    makingPSF = False;
    initMaps(vb);

    findConvFunction(*image, vb);
```

- Initialize the maps for polarization and channel. These maps translate visibility indices into image indices.
- If we are memory-based then read the image in and create an ArrayLattice; otherwise, just use the PagedImage.
- Do the Grid-correction

```
      {
        normalizeAvgPB();

        IPosition cursorShape(4, nx, 1, 1, 1);
        IPosition axisPath(4, 0, 1, 2, 3);
        LatticeStepper lsx(lattice->shape(), cursorShape, axisPath);
        LatticeIterator<Complex> lix(*lattice, lsx);

        verifyShapes(avgPB.shape(), image->shape());
        Array<Float> avgBuf; avgPB.get(avgBuf);
        if (max(avgBuf) < 1e-04)
      throw(AipsError("Normalization by PB requested but either PB not found in the
    cache "
                      "or is ill-formed."));

        LatticeStepper lpb(avgPB.shape(),cursorShape,axisPath);
        LatticeIterator<Float> lipb(avgPB, lpb);

        Vector<Complex> griddedVis;
```

- Grid correct in anticipation of the convolution by the GCF (convFunc). Each polarization plane is corrected by the appropraite primary beam.

```
        for(lix.reset(),lipb.reset();!lix.atEnd();lix++,lipb++)
      {
        Int iy=lix.position()(1);
        gridder->correctX1D(correction,iy);
        griddedVis = lix.rwVectorCursor();

        Vector<Float> PBCorrection(lipb.rwVectorCursor().shape());
        PBCorrection = lipb.rwVectorCursor();
        for(int ix=0;ix<nx;ix++)
          {
            // PBCorrection(ix) = (FUNC(PBCorrection(ix)))/(sincConv(ix)*sincConv(iy));

            //
            // This is with PS functions included
            //
            // if (doPBCorrection)
            //       {
            //         PBCorrection(ix) =
    FUNC(PBCorrection(ix))/(sincConv(ix)*sincConv(iy));
            //         //PBCorrection(ix) =
    FUNC(PBCorrection(ix))*(sincConv(ix)*sincConv(iy));
            //         if ((abs(PBCorrection(ix)*correction(ix))) >= pbLimit_p)
            //           {lix.rwVectorCursor()(ix) /=
    (PBCorrection(ix))*correction(ix);}
            //         else
            //           {lix.rwVectorCursor()(ix) *= (sincConv(ix)*sincConv(iy));}
            //       }
            // else
            //       lix.rwVectorCursor()(ix) /=
    (correction(ix)/(sincConv(ix)*sincConv(iy)));
            //
```

```
            // This without the PS functions
            //
            if (doPBCorrection)
              {
                // PBCorrection(ix) =
    FUNC(PBCorrection(ix))/(sincConv(ix)*sincConv(iy));
                PBCorrection(ix) = FUNC(PBCorrection(ix))*(sincConv(ix)*sincConv(iy));
    //          PBCorrection(ix) = (PBCorrection(ix))*(sincConv(ix)*sincConv(iy));
                if ((abs(PBCorrection(ix))) >= pbLimit_p)
                  {lix.rwVectorCursor()(ix) /= (PBCorrection(ix));}
                else
                  {lix.rwVectorCursor()(ix) *= (sincConv(ix)*sincConv(iy));}
              }
            else
              lix.rwVectorCursor()(ix) /= (1.0/(sincConv(ix)*sincConv(iy)));
          }
      }
      }
      // {
      //   ostringstream name;
      //   cout << image->shape() << endl;
      //   name << "theModel.im";
      //   PagedImage<Float> tmp(image->shape(), image->coordinates(), name);
      //   Array<Complex> buf;
      //   Bool isRef = lattice->get(buf);
      //   cout << "The model max. = " << max(buf) << endl;
      //   LatticeExpr<Float> le(abs((*lattice)));
      //   tmp.copyData(le);
      // }
      //
```

- Now do the 2D FFT in place.

```
      //     {
      //       Array<Complex> buf;
      //       Bool isRef = lattice->get(buf);
      //     }
          LatticeFFT::cfft2d(*lattice);

          logIO() << LogIO::DEBUGGING << "Finished FFT" << LogIO::POST;
        }
```

- Initialize the FFT to the visibility plane using the image as a template. The image is loaded and Fourier transformed. This version returns the gridded visibilities. It should be used in conjunction with the version of 'get' that needs the gridded visdata.

```
void nPBWProjectFT::initializeToVis(ImageInterface<Complex>& iimage,
                                    const VisBuffer& vb,
                                    Array<Complex>& griddedVis,
                                    Vector<Double>& uvscale)
  {
    initializeToVis(iimage, vb);

    griddedVis.assign(griddedData); //using the copy for storage
    uvscale.assign(uvScale);
  }
```

- Finalize the FFT to the visibility plane: flushes the image cache and shows statistics if it is being used.
- Initializes the FFT to the Sky plane: initializes the image. Here we have to set up and initialize the grid.
- Initialize the maps for polarization and channel. These maps translate visibility indices into image indices.
- Initialize for in-memory or to disk gridding. lattice will point to the appropriate Lattice, either the ArrayLattice for in-memory gridding or to the image for disk gridding.
- Finalize FFT to the Sky plane: flushes the image cache and shows statistics if it is being used. DOES NOT DO THE FINAL TRANSFORM!
- Get the appropriate data pointer ….

- Fortran methods related to the underlying Fortran gridder fpbwproj.f. The arguments must all be pointers and the value of the GCF at the given (u,v) point is returned in the weight variable. Making this a function which returns a complex value (namely the weight) has problems when called in FORTRAN …

```
virtual void runFortranGet(Matrix<Double>& uvw,Vector<Double>&
dphase, Cube<Complex>& visdata,….

virtual void runFortranPut(Matrix<Double>& uvw,Vector<Double>&
dphase,const Complex& visdata_p, IPosition& s,….

void runFortranGetGrad(Matrix<Double>& uvw,Vector<Double>& dphase,
              Cube<Complex>& visdata,IPosition& s,
              Cube<Complex>& gradVisAzData,
              Cube<Complex>& gradVisElData, …..
```

- Get the uvws in a form that Fortran can use and do that necessary phase rotation…
- A lot of housekeeping things here.
- Eventually… Use FORTRAN to do the gridding. Remember to ensure that the shape and offsets of the tile are accounted for.

```
          Vector<Double> actualOffset(3);
          for (Int i=0;i<2;i++) actualOffset(i)=uvOffset(i)-Double(offsetLoc(i));
          actualOffset(2)=uvOffset(2);
          IPosition s(flags.shape());
```

- Now pass all the information down to a FORTRAN routine to do the work.

```
Int Conj=0,doPSF;
          Int ScanNo=0,doGrad=0;
          Double area=1.0;

          Int tmpPAI=PAIndex+1;
          if (dopsf) doPSF=1; else doPSF=0;
          runFortranPut(uvw,dphase,*datStorage,s,Conj,flags,rowFlags,
                        *imagingweight,rownr,actualOffset,
                        *dataPtr,aNx,aNy,npol,nchan,vb,NAnt,ScanNo,sigma,
                        l_offsets,m_offsets,sumWeight,area,doGrad,doPSF,tmpPAI);
        }
    }
     }
   else
     {//Non-tiled version
  IPosition s(flags.shape());

  Int Conj=0,doPSF=0;
  Int ScanNo=0,doGrad=0;Double area=1.0;

  if (dopsf) doPSF=1;

  Int tmpPAI=PAIndex+1;
  runFortranPut(uvw,dphase,*datStorage,s,Conj,flags,rowFlags,
              *imagingweight,
              row,uvOffset,
              griddedData,nx,ny,npol,nchan,vb,NAnt,ScanNo,sigma,
              l_offsets,m_offsets,sumWeight,area,doGrad,doPSF,tmpPAI);
    }

   data->freeStorage(datStorage, isCopy);
 }
```

- Predict the (model) visibilities as well as their derivatives w.r.t. the pointing offsets…
- visdata now references the Mout data structure rather than to the internal VB storeage.

```
    visdata.reference(Mout);

    if (doGrad)
      {
   gradVisAzData.reference(dMout1);
   gradVisElData.reference(dMout2);
      }
    // Begin the actual de-gridding.
    if(isTiled)
      {
   logIO() << "nPBWProjectFT::nget(): The sky model is tiled" << LogIO::NORMAL <<
LogIO::POST;
   Double invLambdaC=vb.frequency()(0)/C::c;
   Vector<Double> uvLambda(2);
   Vector<Int> centerLoc2D(2);
   centerLoc2D=0;
```

- Now use FORTRAN to do the gridding. Remember to ensure that the shape and offsets of the tiles are accounted for….
- Get the visibility from the grid, return it in **degrid** . This is used especially when scratch columns are not present in the MS.

```
void nPBWProjectFT::get(VisBuffer& vb,
                  VisBuffer& gradVBAz,
                  VisBuffer& gradVBEl,
                  Cube<Float>& pointingOffsets,
                  Int row,  // default row=-1
                  Type whichVBColumn, // default whichVBColumn = FTMachine::MODEL
                  Type whichGradVBColumn,// default whichGradVBColumn = FTMachine::MODEL
                  Int Conj, Int doGrad) // default Conj=0, doGrad=1
  {….
```

- More housekeeping stuff, then, again, use FORTRAN to do the gridding. Remember to ensure that the shape and offsets of the tile are accounted for.
- Get actual visibility from the grid by degridding.

```
void nPBWProjectFT::get(VisBuffer& vb, Int row)
  {
    // If row is -1 then we pass through all rows
    Int startRow, endRow, nRow;
    if (row==-1)
      {
   nRow=vb.nRow();
   startRow=0;
   endRow=nRow-1;
   vb.modelVisCube()=Complex(0.0,0.0);
      }
    else
      {
   nRow=1;
   startRow=row;
   endRow=row;
   vb.modelVisCube().xyPlane(row)=Complex(0.0,0.0);
      }

    findConvFunction(*image, vb);

    if (bandID_p == -1) bandID_p=getVisParams(vb);
    Int NAnt=0;
    if (doPointing)   NAnt = findPointingOffsets(vb,l_offsets,m_offsets,True);

    // Get the uvws in a form that Fortran can use
    Matrix<Double> uvw(3, vb.uvw().nelements());
    uvw=0.0;
```

etc…

- This cycle repeats … with more housekeeping issues.
- Eventually … Finalize the FFT to the Sky. Here we actually do the FFT and

return the resulting image, i.e., we get the final image: by doing the FFT and grid-correcting, then optionally normalizing by the summed weights.

```
ImageInterface<Complex>& nPBWProjectFT::getImage(Matrix<Float>& weights,
                                                 Bool normalize)
  {
    AlwaysAssert(image, AipsError);

    logIO() << "#########getimage########" << LogIO::DEBUGGING << LogIO::POST;

    logIO() << LogOrigin("nPBWProjectFT", "getImage") << LogIO::NORMAL;

    weights.resize(sumWeight.shape());

    convertArray(weights, sumWeight);
    //
     // If the weights are all zero then we cannot normalize otherwise
     // we don't care.
    //
    if(max(weights)==0.0)
      {
   if(normalize) logIO() << LogIO::SEVERE
                         << "No useful data in nPBWProjectFT: weights all zero"
                         << LogIO::POST;
   else logIO() << LogIO::WARN << "No useful data in nPBWProjectFT: weights all zero"
                << LogIO::POST;
      }
    else
      {
   const IPosition latticeShape = lattice->shape();

   logIO() << LogIO::DEBUGGING
           << "Starting FFT and scaling of image" << LogIO::POST;
```

// x and y transforms (lattice has the gridded visbilities. Make the
// dirty images)

```
LatticeFFT::cfft2d(*lattice,False);
```

```
   // Apply the gridding correction
   //
   {
     normalizeAvgPB();
     Int inx = lattice->shape()(0);
     Int iny = lattice->shape()(1);
     Vector<Complex> correction(inx);

     Vector<Float> sincConv(nx);
     Float centerX=nx/2;
     for (Int ix=0;ix<nx;ix++)
       {
         Float x=C::pi*Float(ix-centerX)/(Float(nx)*Float(convSampling));
         if(ix==centerX) sincConv(ix)=1.0;
         else         sincConv(ix)=sin(x)/x;
       }
```

etc….

- Get the final weight image

```
   void nPBWProjectFT::getWeightImage(ImageInterface<Float>& weightImage,
                                      Matrix<Float>& weights)
   {

     logIO() << LogOrigin("nPBWProjectFT", "getWeightImage") << LogIO::NORMAL;

     weights.resize(sumWeight.shape());
     convertArray(weights,sumWeight);

     const IPosition latticeShape = weightImage.shape();
```

```
    Int nx=latticeShape(0);
    Int ny=latticeShape(1);

    IPosition loc(2, 0);
    IPosition cursorShape(4, nx, ny, 1, 1);
    IPosition axisPath(4, 0, 1, 2, 3);
    LatticeStepper lsx(latticeShape, cursorShape, axisPath);
    LatticeIterator<Float> lix(weightImage, lsx);
    for(lix.reset();!lix.atEnd();lix++)
      {
       Int pol=lix.position()(2);
       Int chan=lix.position()(3);
       lix.rwCursor()=weights(pol,chan);
      }
  }
```

- Save and restore thecurrent nPBWProjectFT object to and from an output state record.
- Make the grid the correct shape and turn it into an array lattice. Check the section from the image BEFORE converting to a lattice.

```
IPosition gridShape(4, nx, ny, npol, nchan);
           griddedData.resize(gridShape);
           griddedData=Complex(0.0);
           IPosition blc(4, (nx-image->shape()(0)+(nx%2==0))/2,
                         (ny-image->shape()(1)+(ny%2==0))/2, 0, 0);
           IPosition start(4, 0);
           IPosition stride(4, 1);
           IPosition trc(blc+image->shape()-stride);
           griddedData(blc, trc) = image->getSlice(start, image->shape());

           arrayLattice = new ArrayLattice<Complex>(griddedData);
           lattice=arrayLattice;
         }

       AlwaysAssert(image, AipsError);
     };
   return retval;
 }
```

- Make the entire image. This returns a complex image, without conversion to Stokes. The polarization representation is that required for the visibilities.

```
void nPBWProjectFT::makeImage(FTMachine::Type type,
                              VisSet& vs,
                              ImageInterface<Complex>& theImage,
                              Matrix<Float>& weight)
{
 logIO() << LogOrigin("nPBWProjectFT", "makeImage") << LogIO::NORMAL;

 if(type==FTMachine::COVERAGE)
  logIO() << "Type COVERAGE not defined for Fourier transforms"
          << LogIO::EXCEPTION;
```

- Loop over all visibilities and pixels

```
    VisBuffer vb(vi);
```

- Initialize put (i.e. transform to Sky) for this model

```
  vi.origin();
      if(vb.polFrame()==MSIter::Linear)
StokesImageUtil::changeCStokesRep(theImage, SkyModel::LINEAR);
  else
    StokesImageUtil::changeCStokesRep(theImage, SkyModel::CIRCULAR);

  initializeToSky(theImage,weight,vb);
```

- Loop over the visibilities, putting VisBuffers

```
paChangeDetector.reset();

    for (vi.originChunks();vi.moreChunks();vi.nextChunk())
      {
        for (vi.origin(); vi.more(); vi++)
          {
            if (type==FTMachine::PSF) makingPSF=True;
            findConvFunction(theImage,vb);

            switch(type)
              {
              case FTMachine::RESIDUAL:
                vb.visCube()=vb.correctedVisCube();
                vb.visCube()-=vb.modelVisCube();
                put(vb, -1, False);
                break;
              case FTMachine::MODEL:
                vb.visCube()=vb.modelVisCube();
                put(vb, -1, False);
                break;
              case FTMachine::CORRECTED:
                vb.visCube()=vb.correctedVisCube();
                put(vb, -1, False);
                break;
              case FTMachine::PSF:
                vb.visCube()=Complex(1.0,0.0);
                makingPSF = True;
                put(vb, -1, True);
                break;
              case FTMachine::OBSERVED:
              default:
                put(vb, -1, False);
                break;
              }
          }
      }
    finalizeToSky();   // Finalize FFT to Sky plane: flushes the image cache and shows statistics if it is being used.
```

- Normalize by dividing out weights, etc.

```
getImage(weight, True);
```

- The functions are not azimuthally symmetric - so compute the support size carefully. Collect every PixInc pixel along a quarter circle of radius R (four-fold azimuthal symmetry is assumed), and check if any pixel in this list is above the threshold. If TRUE, then R is the support radius. Else decrease R and check again.

```
  Bool nPBWProjectFT::findSupport(Array<Complex>& func, Float& threshold,Int& origin,
Int& R)
  {
    Double NSteps;
    Int PixInc=1;
    Vector<Complex> vals;
    IPosition ndx(4,origin,0,0,0);
    Bool found=False;
    IPosition cfShape=func.shape();
    for(R=convSize/4;R>1;R--)
      {
       NSteps = 90*R/PixInc; //Check every PixInc pixel along a
                             //circle of radious R
       vals.resize((Int)(NSteps+0.5));
       vals=0;
       for(Int th=0;th<NSteps;th++)
         {
           ndx(0)=(int)(origin + R*sin(2.0*M_PI*th*PixInc/R));
           ndx(1)=(int)(origin + R*cos(2.0*M_PI*th*PixInc/R));
```

```
            if ((ndx(0) < cfShape(0)) && (ndx(1) < cfShape(1)))
              vals(th)=convFunc(ndx);
          }
        if (max(abs(vals)) > threshold)
          {found=True;break;}
          }
         return found;

          }
```

- uvScale/nx == Size of a pixel size in the image in radians. Lets call it dx. HPBW is the HPBW for the antenna at the centre freq. in use. HPBW/dx == Pixel where the PB will be ~0.5x its peak value. ((2*N*HPBW)/dx) == the pixel where the N th PB sidelobe will be (rougly speaking). When this value is equal to 3.0, the Spheroidal implemtation goes to zero.

```
void nPBWProjectFT::makeAntiAliasingOp(Vector<Complex>& op, const Int nx_l)
  {
    MathFunc<Float> sf(SPHEROIDAL);
    if (op.nelements() != (uInt)nx_l)
      {
        op.resize(nx_l);
        Int inner=nx_l/2, center=nx_l/2;

        Float dx=uvScale(0)*convSampling/nx;
        Float MaxSideLobeNum = 3.0;
        Float S=1.0*dx/(MaxSideLobeNum*2*HPBW),cfScale;

        cout << "UVSCALE = " << uvScale(0) << " " << convSampling << endl;
        cout << "HPBW = " << HPBW
             << " " << Diameter_p
             << " " << uvScale(0)/nx
             << " " << S
             << " " << dx
             << endl;


        cfScale=S=6.0/inner;
        for(Int ix=-inner;ix<inner;ix++)
op(ix+center)=sf.value(abs((ix)*cfScale));
```

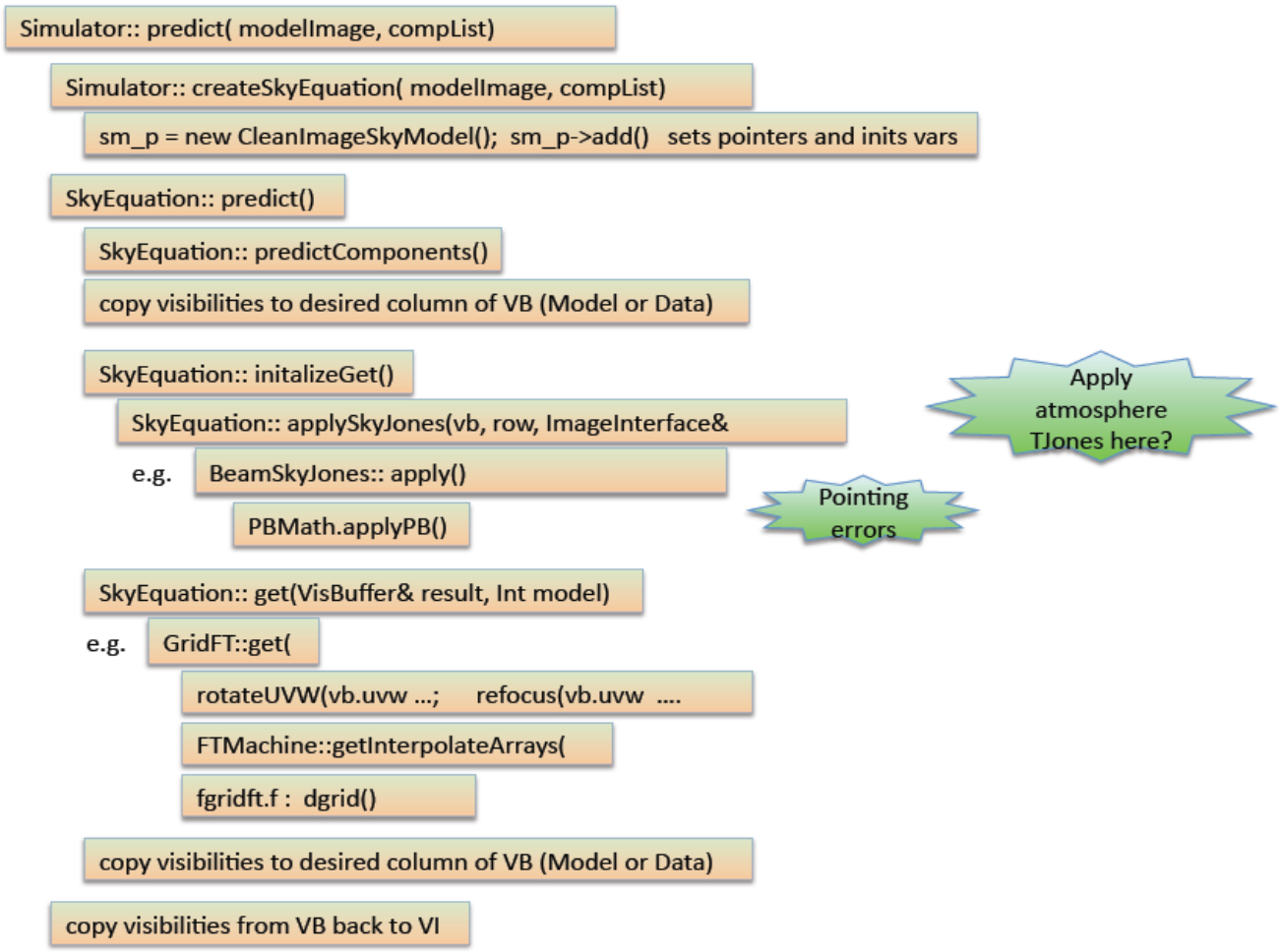
- Make an anti-aliasing operator and an anti-aliasing correction, and apply anti-aliasing operator…

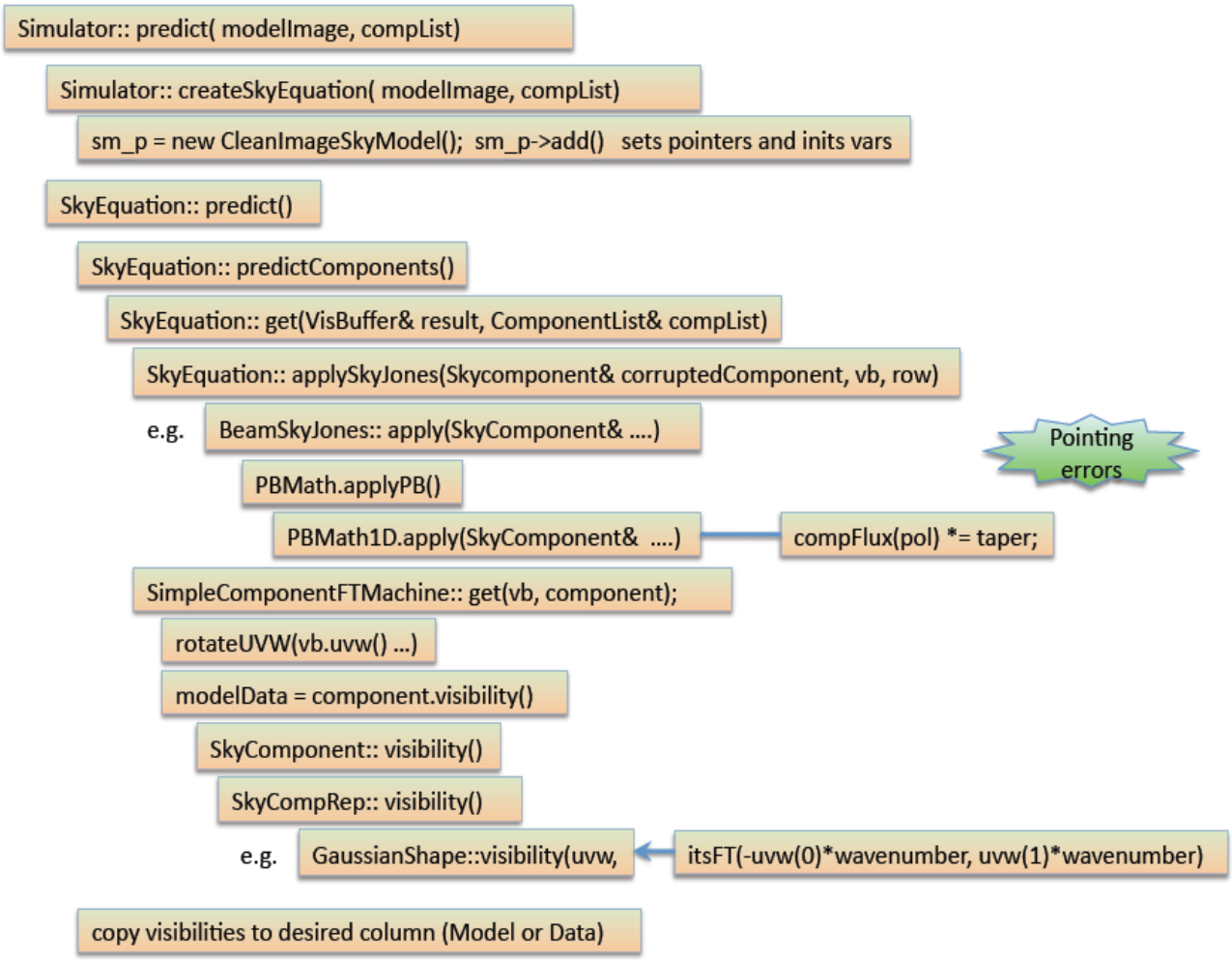
2.4.5 Overall CASA FTMachine Hieracrhy

The following url link is to a May 2011 class diagram of the **FTMachine**, **ConvolutionFunction**, **Resampler** classes and related classes (S. Bhatnagar).
http://www.aoc.nrao.edu/~rurvashi/ImagingAlgorithmsInCasa/FTMachine.png.

Appendix 1: Summary of Classes and Methods/Member Functions

Below (pages 44-45) we give a graphical representation of sorts of the different classes and different methods discussed in this document for predicting model visibilities (during simulation) and then corrupting the visibilities with a primary beam model. (Reference: https://safe.nrao.edu/wiki/pub/ALMA/Jan2010Wkshop/wkshp2010.pdf)


Simulator:: predict(modelImage, compList)
Simulator:: createSkyEquation(modelImage, compList)
sm_p = new CleanImageSkyModel(); sm_p->add() sets pointers and inits vars
SkyEquation:: predict()
SkyEquation:: predictComponents()
copy visibilities to desired column of VB (Model or Data)
SkyEquation:: initalizeGet()
SkyEquation:: applySkyJones(vb, row, ImageInterface&
e.g. BeamSkyJones:: apply()
PBMath.applyPB()
Apply atmosphere TJones here?
Pointing errors
SkyEquation:: get(VisBuffer& result, Int model)
e.g. GridFT::get(
rotateUVW(vb.uvw ...; refocus(vb.uvw ....
FTMachine::getInterpolateArrays(
fgridft.f : dgrid()
copy visibilities to desired column of VB (Model or Data)
copy visibilities from VB back to VI



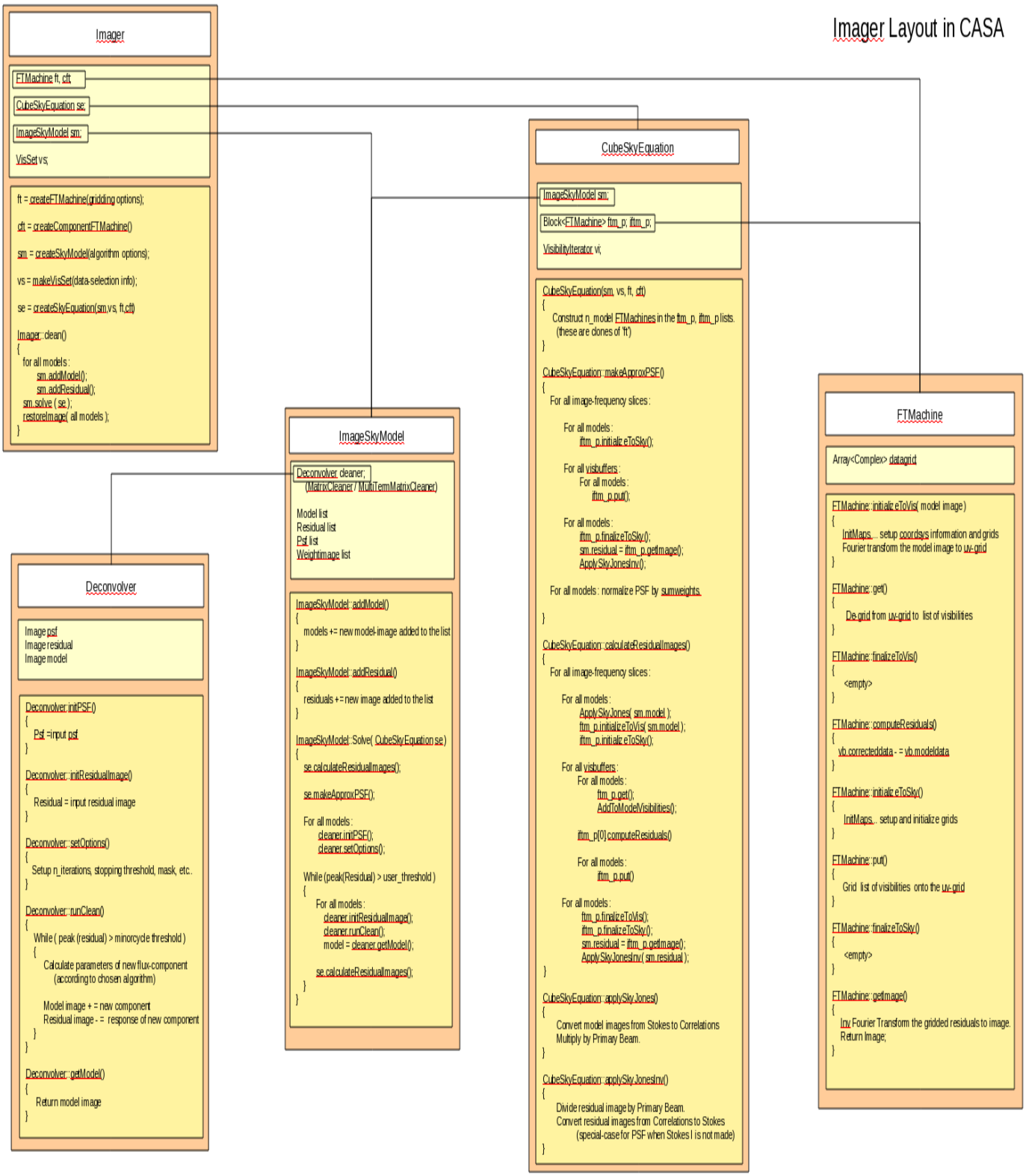
Appendix 2: Imager Layout and Control Flow

The diagram on page 47 has been pasted directly from R. V. Urvashi's web pages ("Imaging Algorithms in CASA": http://www.aoc.nrao.edu/~rurvashi/ImagingAlgorithmsInCasa/node6.html and http://www.aoc.nrao.edu/~rurvashi/ImagingAlgorithmsInCasa/ImagerLayout.png (May 2011)) and gives a summary of the basic flow of control within the CASA Imager classes for major/minor cycles and gridding/degridding (written as pseudo-code). **Note that this diagram does not show the implementation of the A-Projection algorithm (via the nPBWProjectFT class). I assume that this image layout / control flow diagram is for versions of CASA before the implementation of A-Projection.**

"Note : This diagram shows the main calls that are common to all derivatives of **ImageSkyModels** and **FTMachines**. The actual code tree has much more code for many special-cases controlled by various

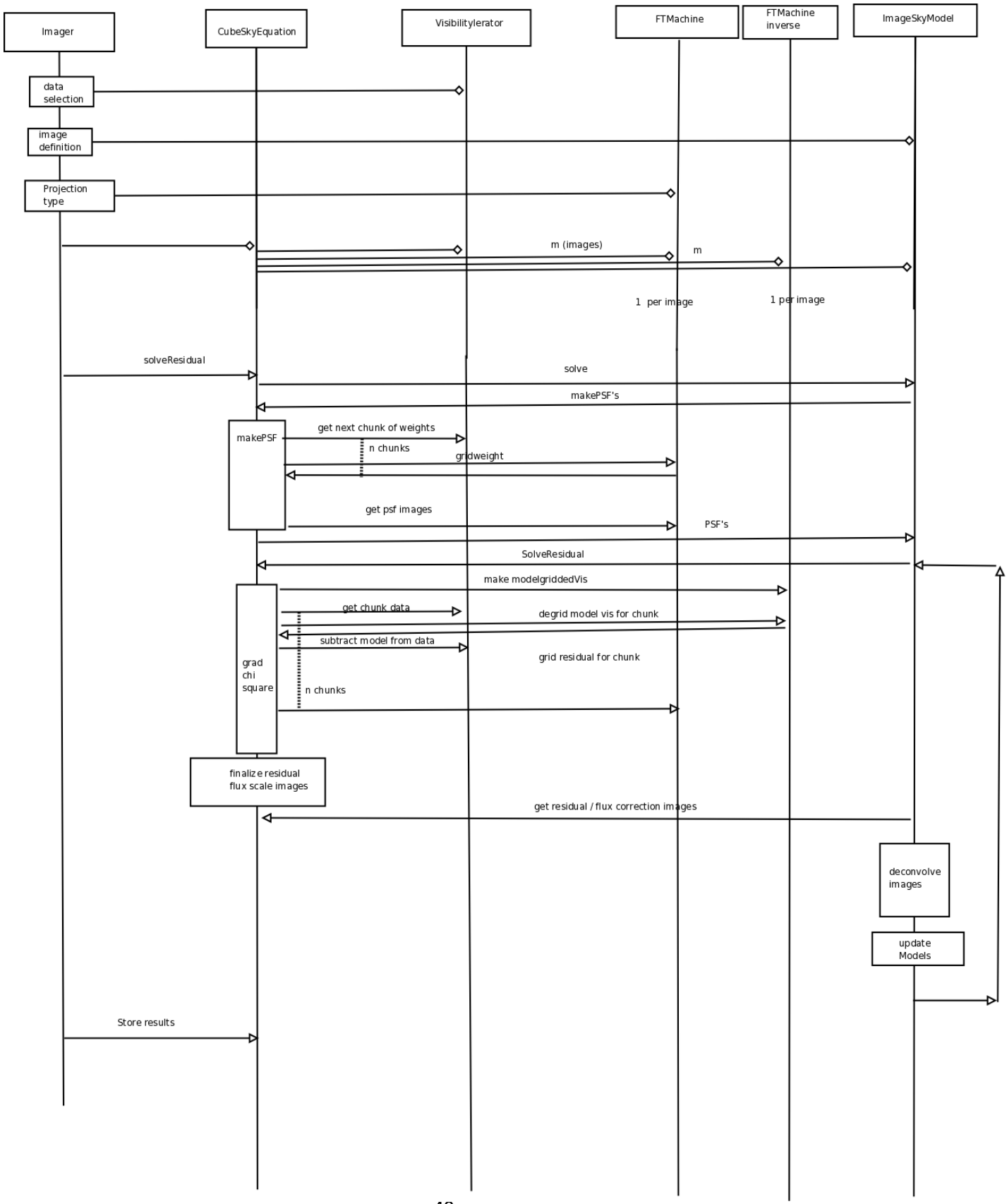
state variables. Various set-up functions are not listed. There are also some naming differences between the diagram and code : The function CubeSkyEquation::calculateResidualImages() is actually **CubeSkyEquation::gradientsChiSquared()** in the code. Also, **ImageSkyModel::solve()** is in some cases, **ImageSkyModel::solveResiduals()**. The Deconvolver interface is not consistent across all minor-cycle algorithms." (Reference: http://www.aoc.nrao.edu/~rurvashi/ImagingAlgorithmsInCasa/node6.html).

Imager Layout in CASA



Appendix 3: Imager Control Flow

The diagram on page 49 is a summary of the Imager/ImageSkyModel/FTMachine class hierarchy and function-call from K. Golap (May 2010). A much clearer view of this can be found on-line at: http://www.aoc.nrao.edu/~rurvashi/ImagingAlgorithmsInCasa/KG_Imaging_seq.png


Imager
CubeSkyEquation
Visibilitylerator
FTMachine
FTMachine inverse
ImageSkyModel
data selection
image definition
Projection type
m (images)
m
1 per image
1 per image
solveResidual
solve
makePSF's
makePSF
get next chunk of weights
n chunks
gridweight
get psf images
PSF's
SolveResidual
make modelgriddedVis
get chunk data
degrid model vis for chunk
subtract model from data
grid residual for chunk
grad chi square
n chunks
finalize residual flux scale images
get residual / flux correction images
deconvolve images
update Models
Store results

References

- CASA class hierarchy documentation, http://casa.nrao.edu/active/docs/doxygen/html/hierarchy.html.
- Rau, U., "Imaging Algorithms in CASA", 2011, http://www.aoc.nrao.edu/~rurvashi/ImagingAlgorithmsInCasa/.
- Rau, U., "Parameterized Deconvolution for Wide-Band Radio Synthesis Imaging" (PhD Thesis), 2010, http://www.aoc.nrao.edu/~rurvashi/DataFiles/UrvashiRV_PhdThesis.pdf.
- Cornwell, T. & Wieringa, M. , "The Generic Instrument: III Design of Calibration and Imaging", AIPS++ Note #189, 1996, http://casa.nrao.edu/aips2_docs/notes/189/.
- Bhatnagar, S. et al., A & A, 487, 419, 2008.
- Bhatnagar, S. et al., "Correction of errors due to antenna power patterns during imaging", EVLA Memo #100, 2006, http://www.aoc.nrao.edu/evla/geninfo/memoseries/evlamemo100.pdf.
- Rau, U. et. al. , Proc. IEEE, Vol. 97, No. 8, 2009.
- Golap, K. , "Clean and Imager", Presentation at CASA Developers Meeting, 2010, https://safe.nrao.edu/wiki/pub/Software/CASADevelopersMeeting/imager_dev_2010.pdf
- Slides/Presentations from NRAO/NAASC (North American Alma Science Center) Workshop (Charlotsville), January 2010, https://safe.nrao.edu/wiki/pub/ALMA/Jan2010Wkshop/wkshp2010.pdf, https://safe.nrao.edu/wiki/bin/view/ALMA/Jan2010Wkshop