The Generic Instrument: III Design of Calibration and Imaging

T.J. Cornwell, NRAO
M.H. Wieringa, ATNF

1996 April 15

Contents

1 Introduction

A previous memo described calibration and imaging for the generic Interferometer (Cornwell, 1995) using the measurement equation developed by Hamaker, Bregman and Sault (1995) (see also Noordam, 1995). The purpose of this memo is to sketch the form of the design adopted for calibration and imaging. This is a sketch only and, in keeping with AIPS++ philosophy, the reader is referred to the AIPS++ header files for more information.

We assume that the reader is familiar with the terminology, notation and results from the above mentioned references.

2 Goals

The goals for the design were:

- Implement the `MeasurementEquation`, thus allowing straightforward polarization calibration and imaging in AIPS++, but with special cases corresponding to single polarization or just parallel hands,
- Allow all important and hopefully most extant calibration and imaging schemes to be coded in AIPS++,
- Allow observatories to develop custom calibration and imaging packages,
- Allow a programmer to add new calibration and deconvolution algorithms as easily and naturally as possible,
- Allow future instruments (such as the MMA and the SKAI) to be supported in AIPS++ with minimal changes to existing code,
- Support as standard observing: mosaiced, spectral-line, multi-feed, polarization observations with synthesis arrays and single dishs,
- Allow calibration and imaging algorithms to be used as simply part of higher level algorithms,
- Acheive performance goals that will allow the calibration and imaging system to be used for standard production processing.

The design has been checked against the user specifications (Cornwell, 1995).

3 The Measurement Equation

The measurement equation is:

$$\vec{V}_{\mathsf{ij}} = X_{\mathsf{ij}} \left(\mathsf{M}_{\mathsf{ij}} \left[{\mathsf{J}^{\mathsf{vis}}}_{\mathsf{i}} \otimes {{\mathsf{J}^{\mathsf{vis}}}_{\mathsf{j}}}^{*} \right] \sum_{k} \left[{\mathsf{J}^{\mathsf{sky}}}_{\mathsf{i}} \left(\vec{\rho}_k\right) \otimes {\mathsf{J}^{\mathsf{sky}}}_{\mathsf{j}} \left(\vec{\rho}_k\right)^{*} \right] \mathsf{S}\ \vec{I}_k + \vec{A}_{\mathsf{ij}} \right) \tag{1}$$

We need to be able to do the following things:

- Evaluate predicted coherences in a MeasurementSet $\vec{V}_{\mathsf{ij}}$ if given all terms on right hand side,
- Given a MeasurementSet, construct an image of the Sky Brightness,
- Solve for other terms on right hand side given a Measurement set and a model for the sky brightness.

In previous memos, we described how these are to be accomplished. Here we recapitulate that description, with some minor changes.

3.1 Prediction of coherences

This is quite straightforward, except that efficiency requires that FFTs be used whereever possible.

3.2 Correction of coherences

The Jones Matrices can be usefully split into two classes: those that are sky-position dependent, $\mathsf{J}^{\mathsf{sky}}$, and those that are not, $\mathsf{J}^{\mathsf{vis}}$. The latter can be corrected trivially in the visibility domain whereas the former cannot be corrected without going through an imaging process.

To ensure compatibility, we have chosen to impose a standard set and ordering of matrices. These split as follows. The `SkyJones` matrices are:

$\mathsf{F}(\vec{r}_{\mathsf{i}}, \vec{\rho})$	ionospheric Faraday rotation
$\mathsf{T}(\vec{r}_{\mathsf{i}}, \vec{\rho})$	atmospheric gain
$\mathsf{E}_{\mathsf{i}}(\vec{\rho}')$	primary beam

The `VisJones` matrices are:

P_{i}	receptor position angle
C_{i}	configuration of receptors
D_{i}	receptor cross-leakage
G_{i}	electronic gain (antenna-based only)
$\mathsf{K}_{\mathsf{i}}(\vec{\rho}_0 . \vec{r}_{\mathsf{i}})$	factorised ‘FT’ phase for phase center

The order is:

$$\mathsf{J}^{\mathsf{vis}} = \mathsf{K}\ \mathsf{G}\ \mathsf{D}\ \mathsf{C}\ \mathsf{P} \tag{2}$$

$$\mathsf{J}^{\mathsf{sky}} = \mathsf{E}\ \mathsf{T}\ \mathsf{F} \tag{3}$$

The conventional meaning of these matrices is given by Noordam (1995) with one important exception: we have changed the K-matrix to be a single phase rotation for a coherence rather than something that is position-dependent.

Let us define the *sky* coherence to be:

$$\vec{V}_{\mathsf{ij}}^{\mathsf{sky}} = \sum_k \left[\mathsf{J}^{\mathsf{sky}}{}_{\mathsf{i}}\left(\vec{\rho}_k\right) \otimes \mathsf{J}^{\mathsf{sky}}{}_{\mathsf{j}}\left(\vec{\rho}_k\right)^*\right] \mathsf{S}\ \vec{I}_k \tag{4}$$

We then have that:

$$\vec{V}_{\mathsf{ij}} = X_{\mathsf{ij}}\left(\mathsf{M}_{\mathsf{ij}}\left[\mathsf{J}^{\mathsf{vis}}{}_{\mathsf{i}} \otimes \mathsf{J}^{\mathsf{vis}}{}_{\mathsf{j}}{}^*\right]\vec{V}_{\mathsf{ij}}^{\mathsf{sky}} + \vec{A}_{\mathsf{ij}}\right) \tag{5}$$

And then a natural definition of a corrected (visibility) coherence would be:

$$\vec{V}_{\mathsf{ij}}^{\mathsf{cal}} = \left[\mathsf{J}^{\mathsf{vis}}{}_{\mathsf{i}} \otimes \mathsf{J}^{\mathsf{vis}}{}_{\mathsf{j}}{}^*\right]^{-1} \mathsf{M}_{\mathsf{ij}}^{-1}\left(X_{\mathsf{ij}}^{-1}\left(\vec{V}_{\mathsf{ij}}\right) - \vec{A}_{\mathsf{ij}}\right) \tag{6}$$

Full correction for sky-plane-based calibration terms is not possible without constructing an image. This we consider next.

3.3 Image estimation

In Cornwell (1995), we described how to solve for the sky brightness. A *generalized* dirty image can be defined as:

$$\vec{I}_k^D = -\left[\frac{\partial^2\chi^2}{\partial\vec{I}_k\partial\vec{I}_k^T}\right]^{-1} \frac{\partial\chi^2}{\partial\vec{I}_k}\ |_{\vec{I}_k=0} \tag{7}$$

where χ^2 is given by:

$$\chi^2 = \sum_{\mathsf{ij}} \Delta\vec{V}_{\mathsf{ij}}^{*T}\Lambda_{\mathsf{ij}}\Delta\vec{V}_{\mathsf{ij}} \tag{8}$$

where the residual is given by the difference between that observed and predicted using the Measurement Equation:

$$\Delta\vec{V} = \vec{V} - \widehat{\vec{V}} \tag{9}$$

The derivatives are given by:

$$\frac{\partial \chi^2}{\partial \vec{I}_k} = -2\ \Re \sum_{\mathsf{ij}} \left[\mathsf{J_i}\left(\vec{\rho}_k\right) \otimes \mathsf{J_j}\left(\vec{\rho}_k\right)^*\right]^{*T} \Lambda_{\mathsf{ij}}\ \Delta\vec{V}_{\mathsf{ij}} \tag{10}$$

$$\frac{\partial^2 \chi^2}{\partial \vec{I}_k \partial \vec{I}_k^T} = 2\ \Re \sum_{\mathsf{ij}} \mathsf{S}^{*T} \left[\mathsf{J_i}\left(\vec{\rho}_k\right) \otimes \mathsf{J_j}\left(\vec{\rho}_k\right)^*\right]^{*T}\ \Lambda_{\mathsf{ij}}\ \left[\mathsf{J_i}\left(\vec{\rho}_k\right) \otimes \mathsf{J_j}\left(\vec{\rho}_k\right)^*\right]\ \mathsf{S} \tag{11}$$

where Λ is a weighting array.

The misfit, χ^2, can be evaluated with respect to the corrected values, thus allowing imaging in the traditional two-step process. One caveat is that the Weight matrix must then be adjusted for the correction.

3.4 Calibration estimation

We allow least squares estimation of calibration effects by providing derivatives of χ^2 with respect to the various gain matrices. Differentiating χ^2 with respect to each element of a gain matrix in turn we can build up the gradient with respect to the overall matrix. For clarity, consider the case where only G is active in $\mathsf{J^{vis}}$. We then have that:

$$\frac{\partial \chi^2}{\partial \mathsf{G}_{\mathsf{i},p,q}} = -2\ \sum_{\mathsf{j}}\ \left[U_{p,q} \otimes \mathsf{G_j}^*\right]^{*T}\ \ \Lambda_{\mathsf{ij}}\ \Delta\vec{V}_{\mathsf{ij}} \tag{12}$$

$$\frac{\partial^2 \chi^2}{\partial \mathsf{G}_{\mathsf{i},p,q} \partial {\mathsf{G}_{\mathsf{i},p,q}}^{*T}} = 2\ \sum_{\mathsf{j}}\ \left[U_{p,q} \otimes \mathsf{G_j}^*\right]^{*T}\ \ \Lambda_{\mathsf{ij}}\ \left[U_{p,q} \otimes \mathsf{G_j}^*\right] \tag{13}$$

where the matrix $U_{p,q}$ is unity for element p, q and zero otherwise. This equation simplifies to the well-known scalar version and corresponds physically to just referencing errors to one antenna and summing over all baselines to that antenna (see Cornwell and Wilkinson, 1981 for a heuristic derivation of the scalar version).

4 The Design

We introduce the following classes to model the measurement equation:

`SkyEquation` and `VisEquation` are responsible for evaluating this equation, χ^2, and the gradients of χ^2.

`VisSet` is responsible for providing coherence data to the `SkyEquation` and `VisEquation` and storing the results of predictions. It is thus simply a convenient interface to a Measurement Set. Iteration through the `VisSet` is accomplished by Iterators `VisIter` and `ROVisIter`.

`SkyModel` supplies a set of images such as $\vec{I}$ to the `SkyEquation`.

`SkyJones` supplies sky-plane-based calibration effects to the `SkyEquation` by multiplying a given image by matrices such as $\mathsf{J}^{\mathsf{sky}}$. Indexing is via a `VisIter` object and image coordinates.

`VisJones` supplies visibility-plane-based calibration effects to the `VisEquation` via either 2 by 2 or 4 by 4 matrices such as $\mathsf{J}^{\mathsf{vis}}$. Indexing is via a `VisIter` object.

`MJones` supplies interferometer-based gain effects to the `VisEquation` via a 4 by 4 matrix M. Indexing is via a `VisIter` object.

`ACoh` supplies interferometer-based offsets via a 4 vector $\vec{A}$. Indexing is via a `VisIter` object.

`XCorr` applies a non-linear correlator function via a function X to a 4 vector. Indexing is via a `VisIter` object.

`FTCoh` performs the Fourier transform. It is initialized and then coherences are fetched one by one from it, indexed via a `VisIter` object.

`IFTCoh` performs the inverse Fourier transform. It is initialized and then fed coherences one by one. Finally an image is constructed.

`WTCoh` performs reweighting. It is initialized and then weight are fetched one by one from it, indexed via a `VisIter` object.

All the objects `SkyModel`, `SkyJones`, `VisJones`, `MJones`, `ACoh`, and `XCorr` can potentially solve for themselves given a specific `SkyEquation` or `VisEquation` and `VisSet`. To do this, gradient information is accumulated into each object and then a method `solve` is used to estimate or update the object. For convenience, we actually provide an overloaded method `solve` for `SkyEquation` or `VisEquation` that simply calls the appropriate `solve`.

5 Elucidation

The classes `SkyModel`, `SkyJones`, `VisJones`, `MJones`, `ACoh`, `XCorr` are base classes from which concrete classes must be derived. For example, the most useful form of `SkyModel` is `ImageSkyModel` which can be constructed from an image or set of images. Similarly, we envisage a `ComponentSkyModel` which implements Sky Brightness modeling via a set of discrete components such as Gaussians. Note that the interface to `SkyEquation` is via images, but that the internal workings of a `SkyModel` are not prescribed. Similarly, the two types of Jones classes, `SkyJones` and `VisJones`, interface to the `VisEquation` via actual matrices but may be implemented internally in any way desired.

Fourier transforms are implemented by the `FTCoh` and `IFTCoh` objects.

None of the matrices is required. For single dish processing, E must be present to implement summation over a primary beam. We have chosen a specific number and set of slots to make integration of different matrices part of the services supplied by the `SkyEquation`and `VisEquation`.

The use of `SkyModel` is subtle and deserves some description.

Prediction The `SkyEquation` calls the `get` method of `SkyModel` to get a set of `Image`s for which it can do a prediction of the coherences in a `VisSet`. Note that because of the form of the measurement equation, `Image`s are required and, in particular, a collection of *e.g.* Gaussian components cannot be used directly. To obtain the predicted coherences, one uses the `predict` method of `SkyEquation`.

Solution To solve for a `SkyModel`, one uses the `solve` of the `SkyEquation`. This calls the `solve` method of the `SkyModel`. This will usually be an iterative procedure that in turn calls the `gradientsChiSquared` method of `SkyEquation` which accumulates χ^2 and first and second gradients into the `SkyModel`. The gradients are with respect to the set of `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`. Thus, 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.

Thus although `SkyModel` is very general, the interface between the `SkyEquation` and `SkyModel` is quite simple and is based upon `Images`. The `apply` method can apply a correction to an `Image`.

A similar scheme works for Jones classes. Here the interface is in terms of the coherence vector.

Prediction The `apply` method of a Jones class applies the interferometer gain term (i.e. the direct product of the Jones Matrices).

Solution The gradients are with respect to these same matrices. These are accumulated into a Jones classes by the `gradientsChiSquared` method of the `VisEquation`. The `solve` method of the Jones class is responsible for calling `gradientsChiSquared`. The particular Jones class, *e.g.* `PhaseScreenVisJones` is responsible for converting the gradients of χ^2 with respect to the matrices $\mathsf{J}^{\mathsf{vis}}$ into gradients with respect to the internal variables of the Jones class *e.g.* parameters of a phase screen across a synthesis array.

Correction The `applyInv` of a Jones class applies the inverse of the interferometer gain term (i.e. the direct product of the Jones Matrices).

6 Examples

Here we give some examples. These should be considered illustrative rather than production quality code. However the general framework will stand.

6.1 Top level use of `VisEquation`and `SkyEquation`

The following reads a MeasurementSet and an image model from disk and calibrates the data, first for a simple G-term every 5 minutes and then for a D-term every 12 hours.

```
// Make a SkyEquation
SkyEquation se;

// Read the VisSet from disk
VisSet vs("3c84.MS");
se.setVisSet(vs);

// Create an ImageSkyModel from an image on disk
ImageSkyModel ism(Image<StokesVector>("3c84"));
se.setSkyModel(ism);

// Use DFT for forward transform, FFT for reverse
IFFTCoh ift;
DFTCoh ft;
se.setFTCoh(ft);
se.setIFTCoh(ift);

// Predict the visibility set
se.predict();

// Make a Vis equation
VisEquation se();

// Solve for calibration of G matrix every 5 minutes
GJones gj(vs, 5*60);
ve.solve(gj);
ve.setVisJones(gj);

// Solve for calibration of D matrix every 12 hours
DJones dj(vs, 12*60*60);
ve.solve(dj);
ve.setVisJones(dj);
```

6.2 `solve` for a simple Clean algorithm

Next we give a simple two-dimensional Clean algorithm that finds peaks (actually the peak maximum eigenvalue of the coherence matrix) and subtracts each longwindedly.

```
// Clean solver
Bool HogbomCleanImageSkyModel::solve(SkyEquation& se) {

  // Make a local copy of the Sky Equation so we can change
  // some of the entries
  SkyEquation lse(se);

  // Tell the SkyEquation to use this SkyModel
  lse.setSkyModel(*this);

  // Predict visibility for this Sky Model
  lse.predict ();

  // Find the gradients
  lse.gradientsChiSquared(*this);

  // Make the residual image
  makeNewtonRaphsonStep(1.0);

  Int nx=image_.shape()(0);
  Int ny=image_.shape()(1);

  Matrix<StokesVector> limageStep(imageStep_.shape());
  Matrix<StokesVector> limage(image_.shape());
  Matrix<Float> lpsf(psf_.shape());
  IPosition ipStart(image_.ndim(), 0);
  IPosition ipStride(image_.ndim(), 1);

  imageStep_.getSlice(limageStep, ipStart, imageStep_.shape(), ipStride);
  image_.getSlice(limage, ipStart, image_.shape(), ipStride);
  psf_.getSlice(lpsf, IPosition(psf_.ndim(),0), psf_.shape(),
IPosition(psf_.ndim(),1));

  cout<<"Center of PSF = "<<lpsf(nx/2,ny/2)<<endl;

  // Iterate
  Float max=0.0;
  for (uInt iter=0;iter<numberIterations();iter++) {
    // Now find peak in residual image
    StokesVector maxVal;
    Int ix, iy;
    Int px=nx/4;
    Int py=ny/4;
    max=0.0;
    for (iy=ny/4;iy<3*ny/4-1;iy++) {
      for (ix=nx/4;ix<3*nx/4-1;ix++) {
if(abs(limageStep(ix,iy).maxEigenValue())>max) {
  px=ix;
  py=iy;
  maxVal=limageStep(ix,iy);
  max=abs(limageStep(ix,iy).maxEigenValue());
}
      }
    }

    AlwaysAssert(max>0.0, AipsError);
    AlwaysAssert(px>=nx/4&&px<3*nx/4-1,AipsError);
    AlwaysAssert(py>=ny/4&&py<3*ny/4-1,AipsError);

    // Output ten lines of information if run to the end
    Int cycle;
    cycle=numberIterations()/10;
    if(iter==0||(iter%cycle)==0) {
      cout<<"Iteration "<<iter<<" peak is "<<maxVal<<" at "<<px<<","<<py<<endl;
    }
    if(max<threshold()) {
       cout<<"Converged"<<endl;
       cout<<"Final iteration "<<iter<<" peak is "<<maxVal<<" at "<<px<<","<<py<<endl;
       break;
    };

    // Add the scaled peak to the current image
    StokesVector pv=gain()*maxVal;
    limage(px,py)+=pv;

    // Subtract the scaled PSF from the residual image
    for (iy=ny/4;iy<3*ny/4-1;iy++) {
      for (ix=nx/4;ix<3*nx/4-1;ix++) {
limageStep(ix,iy)-=pv*lpsf(nx/2+ix-px,ny/2+iy-py);
      }
    }
  };

  imageStep_.putSlice(limageStep, ipStart, ipStride);
  image_.putSlice(limage, ipStart, ipStride);

  if(max>threshold()) {
    cout<<"Did not converge in "<<numberIterations()<<" iterations"<<endl;
    return(False);
  }
  else {
    return(True);
  };
};
```

Notes:

1. A pixel is a `StokesVector` which is actually a `RigidVector` having 4 elements, the Stokes parameters I, Q, U, V. The method `maxEigenvalue` returns the maximum eigenvalue of the coherence matrix: $I+\sqrt{Q^2+U^2+V^2}$.
2. `numberIterations`, `gain` and `tolerance` are set via methods such as `setNumberIterations` of the class `Iterate` from which `SkyModel` is derived.
3. `makeNewtonRaphsonStep` is a `protected` method of `ImageSkyModel` that takes the accumulated gradients and forms the optimum Newton-Raphson step (*i.e.* residual image.
4. `chisq_` is calculated along with the gradients with respect to the image in `me.gradientsChisqSquared(*this)`.

6.3 `solve` for `VisJones` terms

Next we give an example of the solution for a single fixed "D" term per antenna. This actually solves for a general matrix with no assumed symmetries.

```
// Solve for the  Jones matrix. Updates the VisJones thus found.
// Also inserts it into the MeasurementEquation thus it is not const.
Bool TimeVarVisJones::solve (VisEquation& ve)
{

  // Make a local copy of the Vis Equation
  VisEquation lve(ve);

  AlwaysAssert(gain()>0.0,AipsError);
  AlwaysAssert(numberIterations()>0,AipsError);
  AlwaysAssert(tolerance()>0.0,AipsError);

  // Mask for gradient computations
  Matrix<Bool> required(2,2);
  setMask(required);

  // Set the Jones matrix in the Measurement Equation
  lve.setVisJones(*this);

  // Count number of failed solutions
  Int failed=0;

  // Tell the MeasurementEquation to use the internal (chunked) VisSet
  lve.setVisSet(intvs_);

  // Reset (chunked) iterators to start of data
  intvs_.observedCoherence().originChunks();
  intvs_.correctedCoherence().originChunks();
  intvs_.modelCoherence().originChunks();

  // Now iterate over the solution intervals
  for (currentSlot_=0; currentSlot_<nInterval_; currentSlot_++) {

    cout<<"Finding solution for slot "<<currentSlot_<<endl;

    // Find gradient and Hessian
    lve.gradientsChiSquared(required,*this);

    // Assess fit
    Float sumwt=0.0;
    Float currentFit=0.0;
    currentFit=chisq_;
    cout<<"Initial fit = "<<currentFit<<endl;
    Float originalFit=currentFit;
    Float previousFit=currentFit;

    // Iterate
    for (uInt iter=0;iter<numberIterations();iter++) {

      // update antenna gains from gradients
      updateAntGain();

      updateIntGain();
```

```
      // Find gradient and Hessian
      lve.gradientsChiSquared(required,*this);

      // Assess fit
      previousFit=currentFit;
      currentFit=chisq_;
      cout<<"Iteration "<<iter<<" fit = "<<currentFit<<endl;

      // Stop?
      if(abs(currentFit-previousFit)<tolerance()*originalFit) break;
    }
    if(abs(currentFit-previousFit)>tolerance()*originalFit) failed++;
    // Advance the iterators to the next interval
    intvs_.observedCoherence().nextChunk();
    intvs_.modelCoherence().nextChunk();
    intvs_.correctedCoherence().nextChunk();
  }

  if(failed>0) {
    cout<<"Did not converge in "<<numberIterations()<<" iterations"<<
      "for "<<failed<<" time intervals"<<endl;
    return(False);
  }
  else {
    return(True);
  }

}
```

Notes:

1. To avoid disturbing the existing `VisEquation` we work with a copy.
2. To store and manipulate matrices, we have implemented a class that encapsulates a 2 by 2 matrix of complex numbers with various possible symmetries: ScalarIdentity, Diagonal, General. By exploiting and remembering symmetries, we reduce the amount of unnecessary mathematics.
3. The class `TimeVarVisJones` implements one gain per antenna per interval of time.
4. `updateAntGain` is a `protected` method that actually changes the gains according to the gradients found. This method is the only place that needs to know about the symmetry.
5. `updateIntGain` is a `protected` method of `TimeVarVisJones` that updates caches of direct products, inverses thereof and gradients.

7 Discussion

7.1 Scalar

A scalar version of the formalism is needed to handle calibration and imaging of *e.g.* RR alone, or RR, LL to I, V.

7.2 Weights

We need to introduce true weights and keep track of the covariance structure, at least in the visibility domain.

7.3 Persistence

This design is based upon the observation that calibration and imaging are best viewed as activities involving three separate categories of things:

1. A source of data (the MeasurementSet in AIPS++),
2. A method of interpretation (the `SkyEquation` and `VisEquation`),
3. Tools used in the interpretation (methods and data in the Jones matrices and `SkyModel`.)

All these things are best modelled explicitly as classes. What then does one store? And where? And how is the connection between the various elements to be remembered? We can legitimately regard these as questions for higher level design. However, it is probably worth a few words. We can assume that each of the components will be persistent (through AIPS++ tables - see next item). Thus the major thing to be remembered is the connection between the `VisSet` the `SkyModel` the `VisJones` the `SkyJones` etc. We think that since this association is one major role for the `SkyEquation` and `VisEquation` , it makes sense to make these persistent.

7.4 Higher level organization

We have concentrated on how these various classes inter-relate and we have omitted discussion of how the classes relate to other parts of AIPS++ and to the User Interface. By longstanding policy, the data in the calibration classes should reside in AIPS++ tables. Each such AIPS++ table should be made available for manipulation and plotting by the user in whatever way is deemed appropriate. Furthermore, the classes described here are sufficiently high level that the user will want to use the methods directly as envisaged in the Tasking design. One can then envisage high level algorithms being possible at, for example, the Glish script level, just as spelled out in the User requirements.

The design of overall calibration and imaging is described in a subsequent memo in this series.

7.5 Iteration order

Different algorithms may require different sort orders. In AIPS++, sort order of the data is regarded as responsibility of the Table system. Iteration order will therefore be specified by the `solve` method via construction of the appropriate `VisIter`. We also envisage formation of transient tables by a `solve` method to allow selection of the data, by, for example, calibration algorithms that solve for the gain every 5 minutes in a 8 hour observing run.

7.6 Different telescopes

The historically thorny question of whether a MS contains data from more than one telescope has been finessed in a particularly sweet way. As long as the data obeys the `SkyEquation` and `VisEquation` it belongs in a single MS, and it makes no sense to require that, for example, VLA and WSRT data reside in different MSes. Having said that, we should recognize that each observatory is likely to require different calibration strategies for a given telescope. This level of observatory dependence could be quite thin, and implemented via Glish scripts, for example. Another important level of observatory dependence can occur in some of the classes themselves. Redundant spacing calibration would presumably be implemented by WSRT internal to a classes similar to `GJones` shown above. Similarly, the VLA would be interested in implementing some of the wide-field imaging algorithms needed for low-frequency imaging with the VLA. These would be implemented in special versions of the `FTCoh` and `IFTCoh` and a `SkyModel`.

7.7 Composition of Jones matrices

A telescope formed by adding the voltages of the elements of an array is called a *tied array.* Support for tied arrays is possible in the `SkyEquation` and `VisEquation` if a Jones matrix object (most naturally the E matrix) actually just sums the relevant Jones matrices of the array elements.

7.8 Further optimization for speed

The FFT and gridding classes limit the speed available. Since these are encapsulated in the classes `FFTCoh` and `IFFTCoh`, re-implementing these is a clearly separable task. It would be worth investigating the following (perhaps disjoint) possibilities:

- Implement gridding via `FORTRAN` code.
- Allow gridding and FFT of any object that can be scaled with complex numbers $ax + b$. This would allow FFT of a frequency spectrum of `StokesVectors`.

An explicit performance goal should be set soon to test against. To be meaningful, this probably requires a scalar version.

7.9 Simulation

Simulation of data is vital but is quite straightforward. We need a tool to construct a template MeasurementSet. If a test image is then generated, the `predict` service of the `SkyEquation` and the `apply` service of the `VisEquation` can be used, togther with whatever calibration effects are desired, to form a set of predicted coherences. Another `SkyEquation` or `VisEquation` can then be constructed, with the usual calibration objects, and a reconstruction attempted. This is available in the program `simulator`.

7.10 Completion of the classes

We have not yet implemented significant versions of `MJones ACoh` or `XCorr`. These seem quite straightforward.

7.11 Inter-relatedness of classes

The calibration classes can be inter-related. For example, the correction for channel averaging decorrelation in the VLBA FX correlator depends upon the delay found in a global fringe fitting procedure (via a `VisJones solve`). There are two ways of addressing this: either implement the correction via an `MJones` and allow it to talk to the global fringe fitting `VisJones` or allow `VisJones` to report to `VisEquation` via a 4 by 4 matrix for a given baseline. It is not clear which one of these should be preferred.

7.12 Versioning and related issues

One presumably wants to be able to do various things with calibration objects *e.g.* copy, edit, delete, attach version numbers, etc. We regard these as higher level operations. We will discuss these in a subsequent memo.

8 Comments on the design process

It is useful to record how the design process worked.

We did not use any formal methodology or even diagramming tools, except at the end to record the design. We did a detailed analysis of the `MeasurementEquation` before starting the design. We did lots of prototype coding, all in c++. The Sun native compiler (in various beta releases) was mainly used. Compile and link times ranged from a few minutes to more than ten once we increased the use of templates. This is bearable but far from ideal.

We were able to work together effectively by splitting responsibilities for development in different areas and reconciling, by hand, code divergences every few days. That this is possible is a good sign for close-knit collaborations in AIPS++ and is in line with experience in other parts of the project. One of us (TJC) worked principally on design, and the other (MHW) on efficiency issues. We spent a considerable amount of time writing special classes for matrices with few elements and special symmetries (SquareMatrix) and vectors with few elements (RigidVector).

We started with toy classes representing the MeasurementSet and the Image and only later changed to use the actual AIPS++ classes. This was an excellent idea which enabled more rapid debugging than would have been possible otherwise.

The overall breakup into classes was settled fairly early on and changed relatively little. The assignment of responsibilities to classes changed a lot as we looked for and found a natural split of the evaluation of gradients of χ^2. This probably represented most of the experimentation that was performed. We also flirted with global functions for the the `solve` step. That `solve` is not polymorphic is conventional wisdom of long-standing in the project. While this is obviously true in general, we believe that we have chosen an approach that is flexible enough to allow most calibration and imaging schemes now known.

Symmetry in the design between the Jones matrices and the `SkyModel` was the most powerful organizing principle that we came upon.

Enforcing uniformity of interface across objects was also important.

Finally, we came to the split between `SkyEquation` and `VisEquation` quite late on. While the split was obvious, the reasons for maing the design split became more pressing as we got deeper into the details and also began to think in more detail about e.g. Single Dish processing.

Acknowledgement

We should thank many people for useful comments, but we wish to particularly thank Jan Noordam and Brian Glendenning for making numerous useful criticisms and suggestions.

References

Cornwell, T.J., 1995, AIPS++ Implementation Note 183.
Cornwell, T.J. and Wilkinson, P.N., 1981, MNRAS, **196**, 1067.
Hamaker, J.P., Bregman, J.D., and Sault, R.J., 1995, *Understanding radio polarimetry: I Mathematical foundations*, submitted to A&A.
Noordam, J., 1995, AIPS++ Implementation Note 185.