Prototype/Pilot Selfcal System (PSS) Version 5 (PSS5)

J.E. Noordam

(jnoordam@astron.nl)

Dwingeloo

Version 1.0: 25 November 2004

Verified:			
Name	Signature	Date	Rev.nr.
	o.p.v.	15th November 2005	0.9.5

Accepted:		
Work Package Manager	System Engineering Manager	Program Manager
J.E. Noordam	C.M. de Vos	J. Reitsma
.....	.....	.....
date:	*date:*	*date:*

Distribution list:

Group:	For Information:
ASTRON	dOList
O.M. Smirnov (OMS)	C.M. de Vos
R. J. Nijboer (RJN)	K. van der Schaaf (KvdS)
A. G. Willis (AGW)	H. Holties
R. Assendorp (RXO)	A.G. de Bruyn
T. A. Oosterloo (TAO)	M. A. van Haarlem
G. van Diepen (GVD)	
M. Brentjens (MAB)	

Document revision:

Revision	Date	Section	Page(s)	Modification
0.5	2004-Nov-25	-	-	Creation
2.0	Version 1.0: 25 November 2004	??	??-??	Updated class description

Abstract

PSS5 is the version of PSS that will be able to deal with WSRT data, including LFFE data with a reasonably behaved ionosphere.

1 Introduction

***Disclaimer**: Due to severe time pressure, this document is a little unfinished. A more complete version will be produced after the meeting of Nov 25th. However, even in its present form it should satisfy many questions.*

The **bottom line** is that the new generation of radio telescopes (and also the existing telescopes) need software that applies more complex Measurement Equations to the data reduction. The Pilot Selfcal System (PSS) that is being developed for LOFAR is an attempt to provide such software. It is a stand-alone module in which the necessary algorithms will be developed and demonstrated. At suitable points in time, a sleek version of these algorithms will be implemented on LOFAR hardware by the LOFAR BBS team, using LOFAR software technology.

Part of the PSS development strategy is to exercise the system on WSRT data, especially on observations made with the new Low Frequency Frontends (LFFE). These operate in the upper LOFAR frequency band (100-200 MHz), and will have many of the ionospheric problems that are expected for LOFAR itself.

An added, and very important advantage of exercising the new software on real data is the close involvement of experienced astronomers, working on their own projects. The two basic requirements for attracting real users are processing speed, and real improvement of the result, in that order. Both will have to be demonstrated. A staged program with clear milestones is outlined below. The **proof of success** is when active users start exploiting the many possibilities to interact with the PSS system themselves (Freedom Layer). These range from access to data and parameter values (all in AIPS++ tables) to generating their own trees, or even writing their own specialised C++ node classes for insertion in existing trees.

In forging a real partnership with active astronomers, ease of use is an important consideration. Therefore, we have developed the concept of 'canned' forests, i.e. copies of the C++ node repository of the kernel that are stored in a binary file. These can be rapidly loaded into the kernel again, and applied to arbitrary observations. A range of such canned forests for different applications will be made available, together with a set of suitable uv-data sets, for playing and learning. This is called the 'water-hole', after the place in nature where all animals come to drink and interact.

PSS6 should eventually run on a cluster computer, albeit with only *functional* parallellization. Ideally, it should be exportable to the Groningen machine, for use by the WSRT group. It is envisaged that the PSS-series will continue to evolve, and may be exported for use with other telescopes (WSRT, JIVE, GMRT, ATCA, VLA, etc).

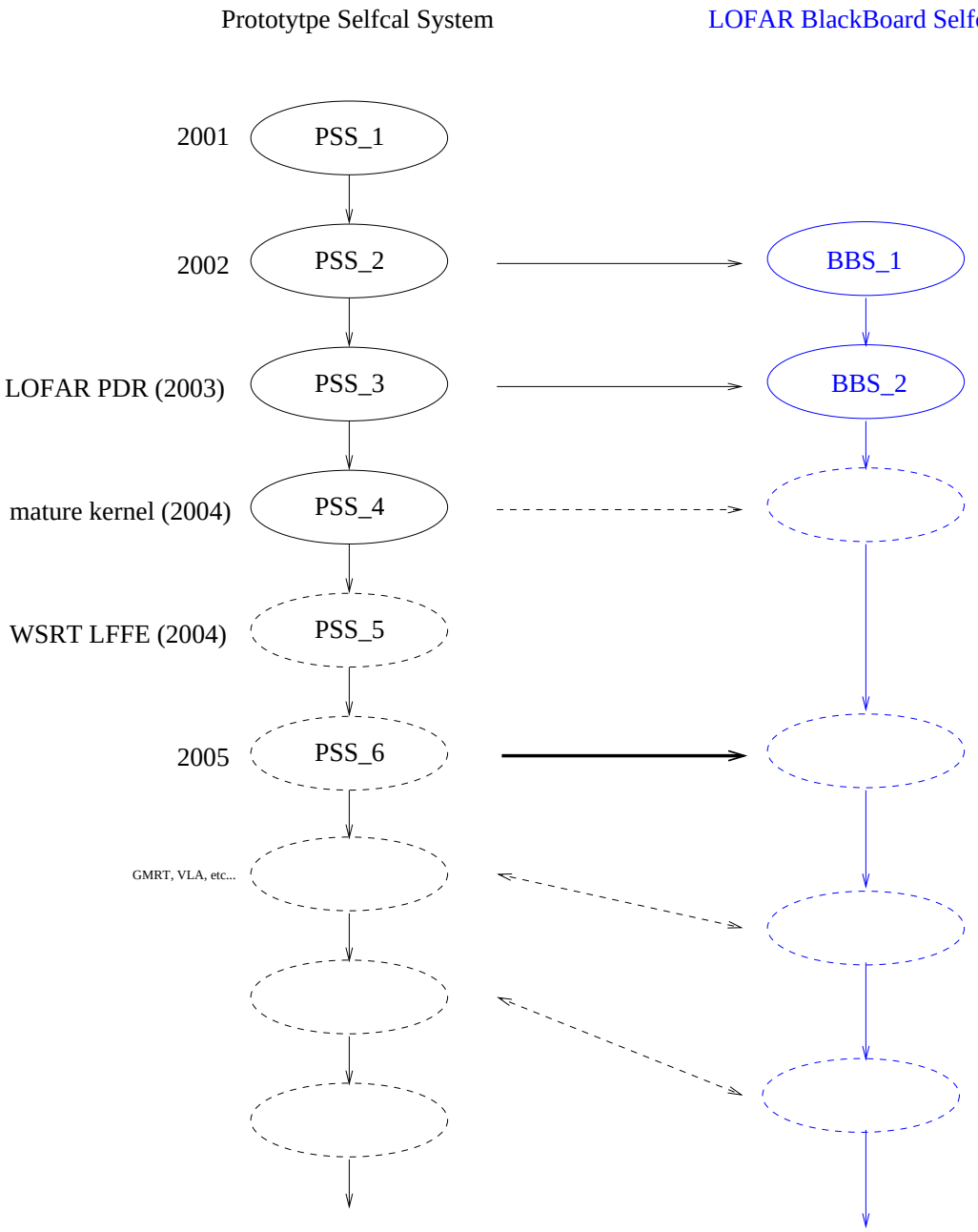


Figure 1: *Parallel development. PSS is very much a Pilot Selfcal System, in which the new algorithms that are needed for calibrating LOFAR are developed and tested. This is an experimental process, which requires a system that is different from the highly optimised one that will eventually have to deal with the huge LOFAR data volumes. Therefore, PSS will be developed as a standalone module, which will be applied to various telescopes. At suitable moments, the LOFAR BBS team implements the PSS algorithms on LOFAR hardware, using LOFAR software technology.*

2 MeqTrees and Measurement Equations (M.E.)

A Measurement Equation (M.E.) is a mathematical model of an observing instrument like a radio aperture synthesis telescope. If it includes a model of the observed object(s), i.e. a Sky Model in our case, an M.E. can *predict* values for measured data, and subtract them. The residuals can be used to make an image from which the sky model is subtracted, or to solve for better values of the M.E. parameters. The latter include instrumental parameters like phase and gain, but also parameters of the Sky Model.

The existing radio-astronomical reduction packages (AIPS, NEWSTAR, MIRIAD, AIPS++, DIFMAP, etc) all have an implicit M.E., which is usually optimised for a particular observing instrument. Although these have produced quite satisfactory results over the years, it is clear that the new generation of telescopes will require a more complex M.E. to achieve the dynamic range to exploit their greater sensitivity. Also the existing telescopes will give better results with a more accurate M.E.

Over the last decade, the Measurement Equation of a generic radio telescope has finally been written down in a closed mathematical form (see Hamaker, Bregman, Sault...). It is a matrix equation, which properly describes the polarisation. The formalism appears to be complete, in the sense that it can model all the existing and planned radio-telescopes, including notoriously difficult cases like parabolic cylinders. In a simple fom of the M.E., the 4-element visibility vector $\vec{V_{ij}}$ of the inteferometer ij between stations i and j can be written as:

$$\vec{V_{ij}}(u,v) = \sum_k (J_{ik} \otimes J_{jk})\, S\, \vec{I_k}$$

where all instrumental effects are modelled in the 2x2 Jones matrices J, whose elements will be different mathematical expressions for different telescopes. The sum is taken over the contributions (4-element vectors) $\vec{I_k}$ of the k relevant sources/patches in the Local Sky Model. The operator $\otimes$ is the Kronecker product, or direct matrix product, and the 4x4 Stokes matrix S converts between (I,Q,U,V) and (XX,XY,YX,YY) or (RR,RL,LR,LL) polarisation representations.

The MeqTree system is just an elegant way to implement a M.E. of arbitrary complexity, and to solve for (subsets of) its parameters. It does this by distributing the formalism over a tree (a graph, really) ofsoftware nodes, each of which performs a specific mathematical operation on its children (other nodes). The 'leaf' nodes at the end represent M.E. parameters (MeqParm nodes), constants (MeqConstant nodes) or data-interface nodes (MeqSpigots and MeqSink nodes). MeqSolver nodes may solve for improved values of subsets of MeqParms. Their condition equations are formed by MeqCondeq nodes, which compare the results of their two children, which are themselves the root nodes of subtrees that calculate measured and predicted data[1].

It is important to realise that a given M.E. is enshrined in the structure of the tree(s), and that **the**

[1] The selfcal process has converged if the difference between measured and predicted data values is purely noise-like. Therefore, the MeqCondeq nodes are ideal for policy-free monitoring.

kernel is entirely policy-free, i.e. it knows nothing about radio telescopes or any other application[2]. It merely executes the trees by transporting requests and results in opposite directions, and by offering services for solving for MeqParm values, and for monitoring what is going on.

3 PSS block diagram

4 Some milestones

The PSS development strategy has a number of milestones:

1. **nov 2004:** Exercition of the basic tree operations, as described in the rest of this document[3]. This includes the use of canned forests, pre-defined views and solvers, and streamlining the use of the tree browser.
2. **nov 2004:** Simuated data from Haystack: Central Point Source, but more than one freq channel and time slot.
3. **dec 2004:** Demonstration of peeling on 3c343/3c343.1 @21cm: two bright point sources of roughly equal flux, 40 arcmin apart, i.e. about the FWHM of the WSRT primary beam. This requires more sophisticated MeqParm behaviour, both when solvable and otherwise. Also the use of a simple LSM, with two Cat I sources. Simple update from residual images.
4. **jan 2005:** Something intermediate before 3c84....
5. **jan 2005:** Demonstration of Cat II prediction on 3c84. This includes the application (and solving?) of image-plane effects over extended sources or patches, using the ideas we developed at the last ADASS. Also the use of a more complicated LSM, with Cat II sources in the form of parametrised components, and in the form of images of CLEAN components. Update from residual images.
6. **feb 2005:** Demonstration of LFFE reduction, especially those cases where there is no bright source in the field. Since the latter do not have enough S/N per time-slot to solve for individual telescope errors, they require the transfer of MeqParm values derived from calibrator observations, and the solution for phase gradients only.
7. **mar 2005:** Inclusion of the WHAT station in WSRT/LFFE reduction.
8. etc...

[2] In fact, the MeqTree kernel can be used for any instrument for which a Measurement Equation can be written down. The plan is to try this eventually for the next generation of non-rigid optical instruments.

[3] One of the greatest errors of AIPS++ was the neglect of humble uv-data operations, like inspection and editing, on which astronomers have to spend most of their time.

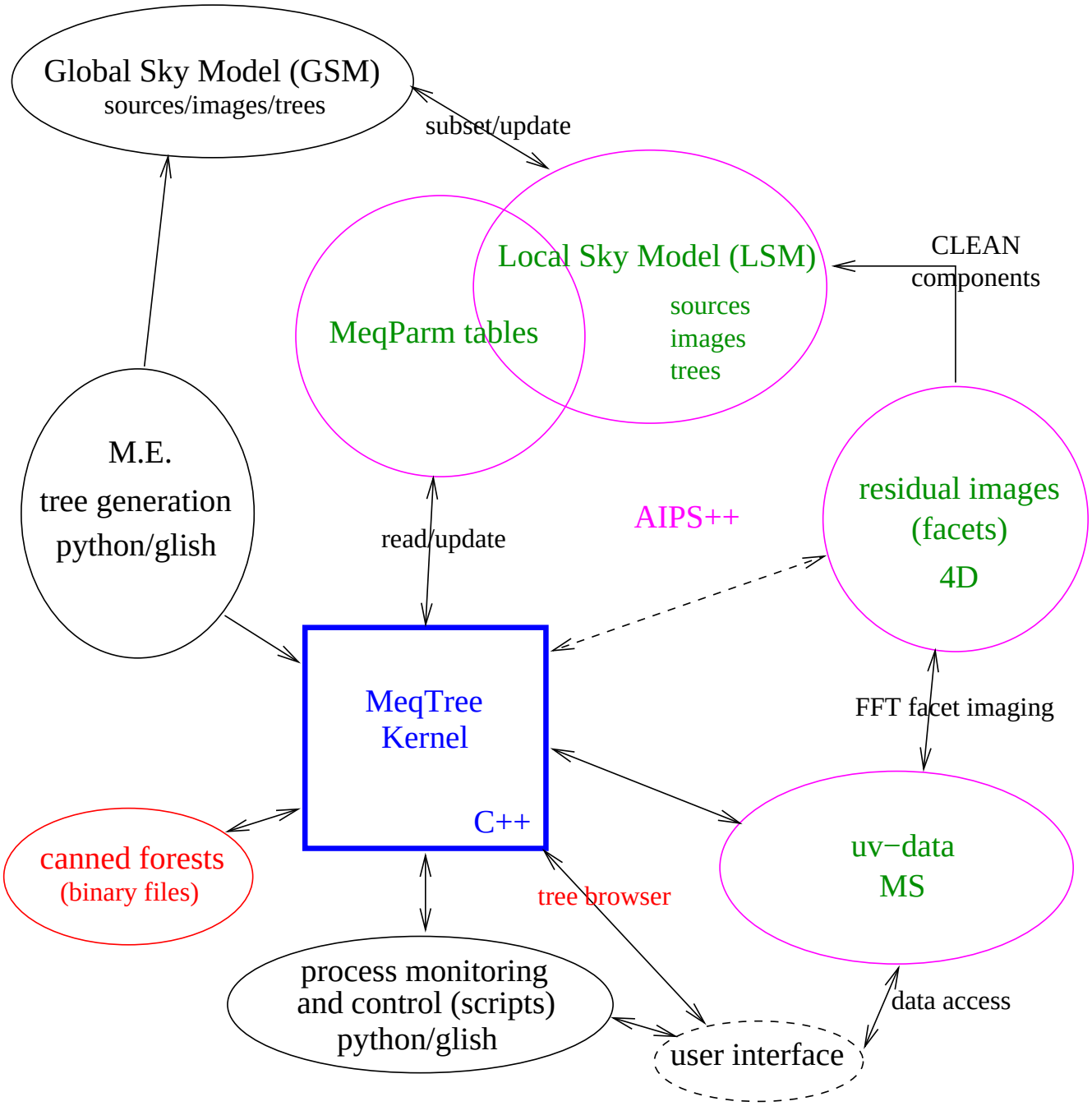


Figure 2: *PSS block diagram. AIPS++ modules (tables, imaging, images, measures, fitting, Glish) are gratefully used, but the MeqTree kernel takes care of all uv-data processing and calibration.*

5 Some important features of PSS

The following is a semi-random reminder of some PSS features that are not offered by many existing packages. It will be prettified later.

- Arbitrary Measurement Equation (bottom line!)
- Built-in support for peeling (position-dependent errors)
- Continuous development (not frozen like the rest)
- Freedom Layer around a stable kernel
  - Access to uv-data (AIPS++ tables)
  - Access to MeqParm values (AIPS++ tables)
  - Possibility to write own MeqParm tools
  - Possiblity to write/cannabilise specialised C++ node classes
  - Possibility to generate own trees
  - Possibility to write/cannibalise own processing scripts
  - Possibility to provide own user interface
- A Python tree-browser which is a powerful user interface in itself
- Visualisation (each node, and pre-defined views)
- Pre-defined solvers may be (de-)activated with the browser
- Debugger functionality (halt, resume, step, breakpoints)
- Speed (self-optimisation, peeling, parallelisation)
- The paradigm allows advanced processing features:
  - Fringe fitting (VLBI)
  - 17 MHz standing-wave pattern (WSRT)
  - Spectral-line processing for ALMA
  - RM synthesis
- Continuity processing (each MeqParm is (f,t), solve for coeff): Phase tracking
- Solving for source parameters (from uv-data and from residual images)

- MeqTrees in LSM/GSM (allows arbitrary source parametrisation)
- Solving for derived parameters (like phase gradients, beamshape parameters, etc)
- Solving for longer integration times
- Have arbitrary succession of specialised solvers, each with its own set of parameters, and its own set of condeqs. NB: Solvers do not take a lot of space. May be de-activated.

6 The MeqTree Kernel

The MeqTree kernel is the heart of the system.

6.1 Trees (graphs, really) of nodes

See fig 3.

6.2 Requests and results (and riders)

See fig 3.

6.3 MeqParms

See fig 3 and 4 and 5 and .

6.4 Solving for subsets of M.E. parameters

See fig 6.

6.5 Interface with uv-data: From Spigot to Sink

The PSS kernel is a standalone module, which can be made to interface with any dataset. This is the task of two specialised nodes: The MeqSpigot[8] finds and reads the data for a requested domain, and the

[8] A spigot is common English word, meaning a small tap, e.g. for water.

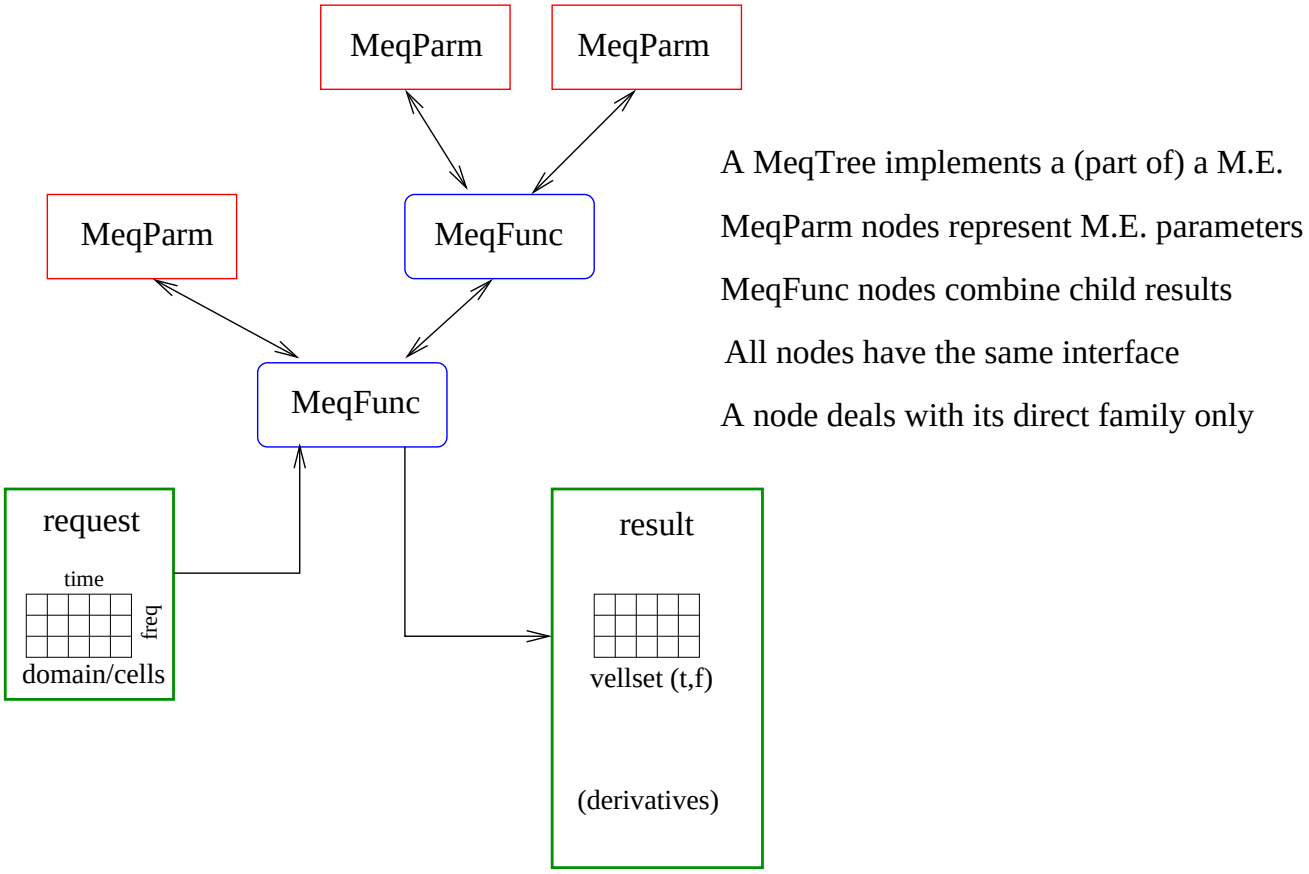


Figure 3: *A basic MeqTree. The nodes are software objects that implement parts of the M.E. Their function is to return a 'result' whenever they get a 'request'. The latter has a 'domain', which usually is just a rectangle in time-freq (t,f) plane*[6]*. A domain is subdivided in 'cells', which often have the same size*[7]*. If a node has any children, it first passes the request to them, and waits for their results. The latter are arrays of values (vellsets) for the cells of the requested domain. The node then combines the results of its children into a result of its own, and returns it.*
Nodes that do not have children are 'leaf' nodes. They have access to information that allows them to return a result for a given request. Examples are MeqParm nodes, which represent M.E. parameter values.

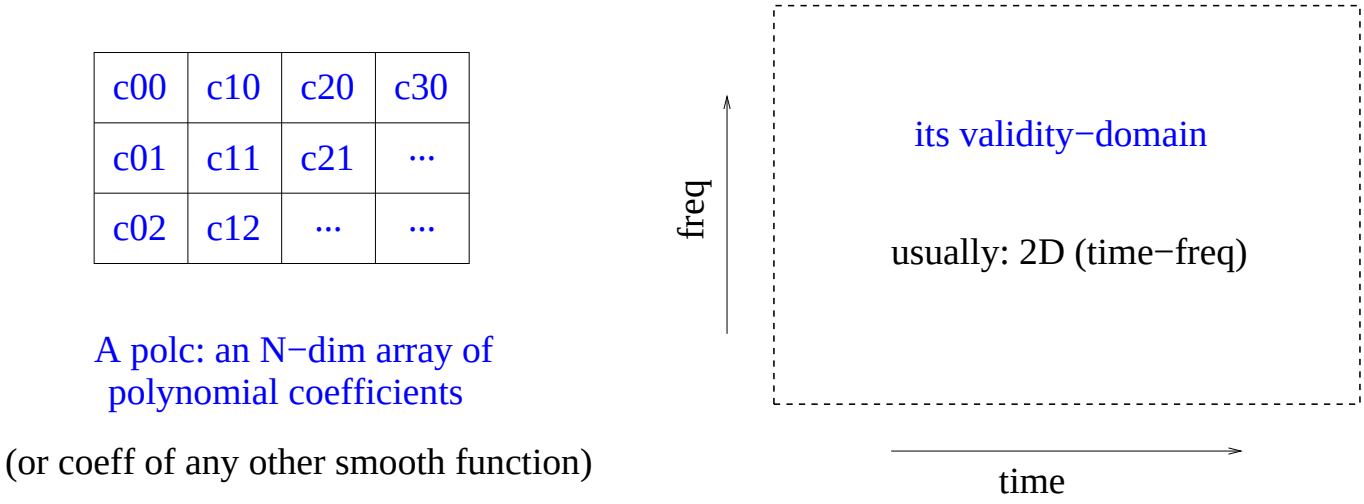


Figure 4: *A 'polc' is an N-dimensional array of coefficients for a smooth N-dimensional function, e.g. a polynomial. Each polc has an associated validity domain. The polc coefficients are the* actual *parameters of the M.E., since this is what the solvers solve for. Obviously, physical knowledge about individual parameters can be inserted by the degree of the polynomials. This is a powerful way to constrain a selfcal solution, and to increase the S/N by reducing the number of independent parameters.*

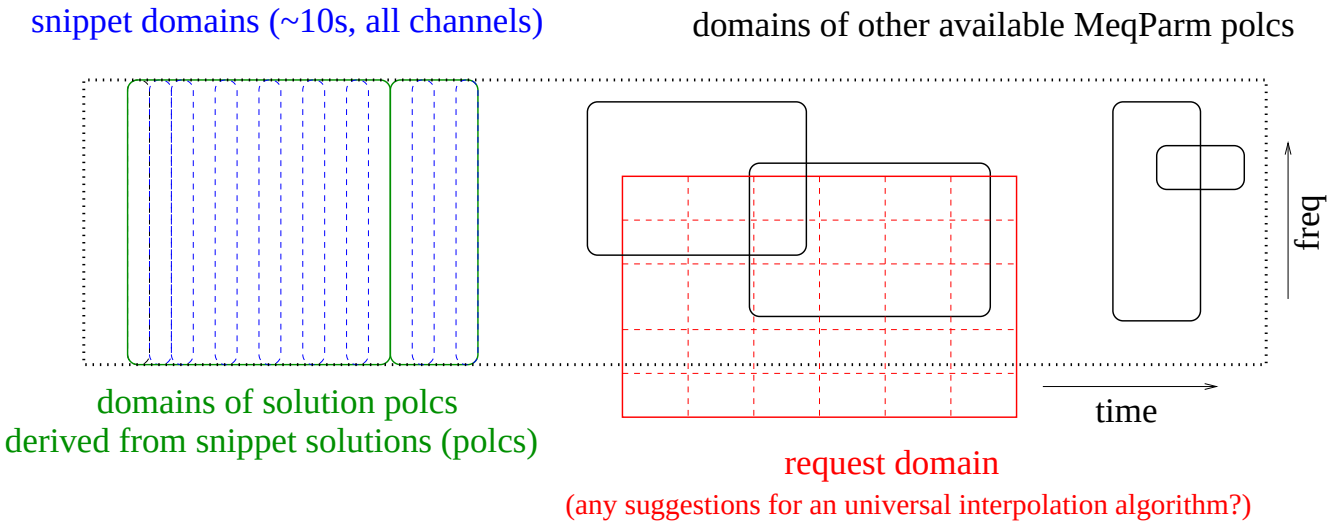


Figure 5: *Whenever a MeqParm gets a request for a particular domain, its calculates the best possible values from its collection of zero or more polcs that are stored in a MeqParm table. Each of them has its own validity domain, as indicated here. The polcs may be the result of a selfcal process, but they may also be copied from an external source, e.g. a calibrator observation. Tools are needed to visualise the available polcs of a specific MeqParm, to smooth edge discontinuities between them, or even to re-organise them when they have become too fragmented. Etc. Fortunately, such tools can be developed by any user that can operate on an AIPS++ table.*

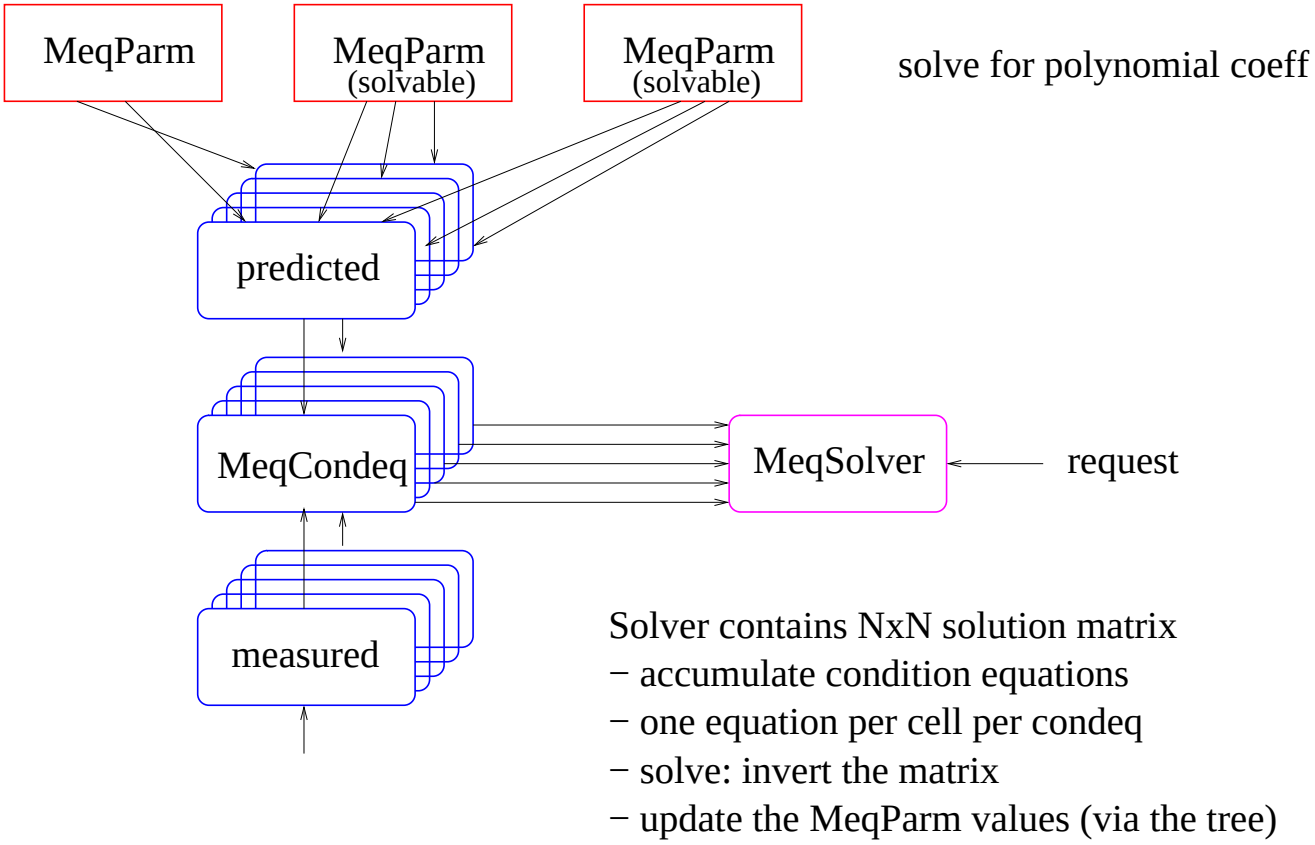
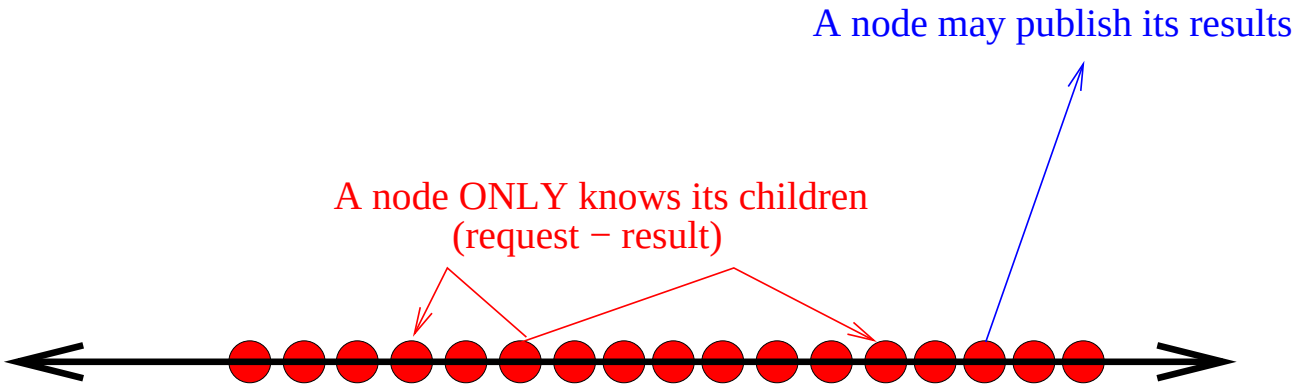


Figure 6: *Solving for M.E. parameters. Whenever a MeqSolver node gets a request to solve for a specific set of M.E. Parameters (represented by MeqParm nodes) over a specific domain(f,t), it passes the request to its children, which are always MeqCondeq nodes. These ask their two children to calculate values for the cells of the requested domain, and then generate 'condition equations' that are accumulated in the solution matrix of the MeqSolver.*



C++ node repository of the kernel. Each node
- may be addressed by name or index number (e.g. to get_state, or execute)
- may publish its results (picked up by subscribers)
- can ONLY communicate with its children (request−result)
- may keep its last result in its cache (efficiency)

Figure 7: *In a naive sense, the C++ node repository of the kernel can be regarded as a one-dimensional collection of nodes, each of which can be addressed individually by its name or by its index-number. In practice, the nodes may be distributed over different processors, while the index-numbers are just integer labels.*

MeqSink writes it back again. At the moment, only the AIPS++ Measurement Set (MS) is supported. A separate spigot/sink pair is used for each correlation.

During processing, a sequence of requests is generated by the MeqSinks for successive 'snippets': smallish domains, with all the frequency channels of a spectral window, but only 10-100s in time. Obviously the MeqSink will request cells with the (t,f) resolution of the available uv-data, so that the MeqSpigot does not have to resample. Moreover, the MeqSink will issue hints to the MeqSpigot about the snippet(s) that will be needed next.

6.6 Kernel organisation

See fig 7.

7 Spigot2Sink forest(s) for one source

This is the easiest form of selfcal, since it does not involve a Local Sky Model. Since a calibrator is assumed to be a dominating point source in the centre of the field, the model is simply that all the phases should be zero, and the amplitudes equal to 1 Jy. As a next step, different forests can be generated for the most popular calibrator sources, with their actual I,Q,U,V flux values in the MeqParm table of a first version of the WSRT Global Sky Model (GSM).

Test case: WSRT calibration (incl polarization), see fig 9.

8 Classical selfcal with multiple sources

This is a extension of the case of the central (calibrator) source treated above. The prediction is for the sum of the contributions of the various sources in the field. Since these may be far from the phase centre, they have to be predicted at high (t,f) resolution. The solver solves for uv-plane effects, i.e. only one phase and gain parameter per voltage beam that is supposed to be valid for the entire field.

9 Peeling

See fig 10.

10 MeqParm tables

See figs 4,5,14 etc.

11 Local Sky Model (LSM)

See figs 11 and 12.

12 Global Sky Model (GSM)

This is one of the main deliverables of the radio telescope of the future.

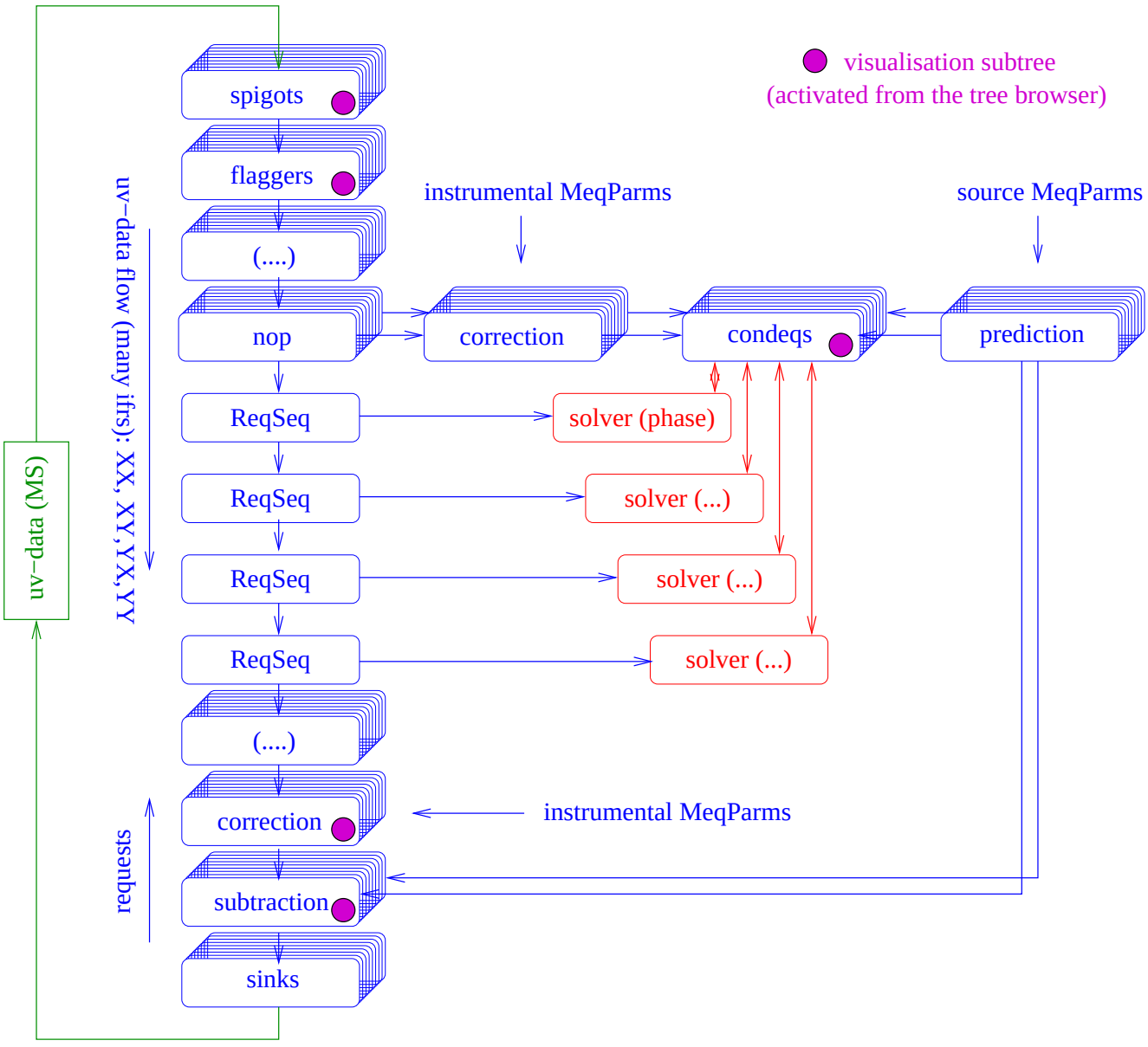
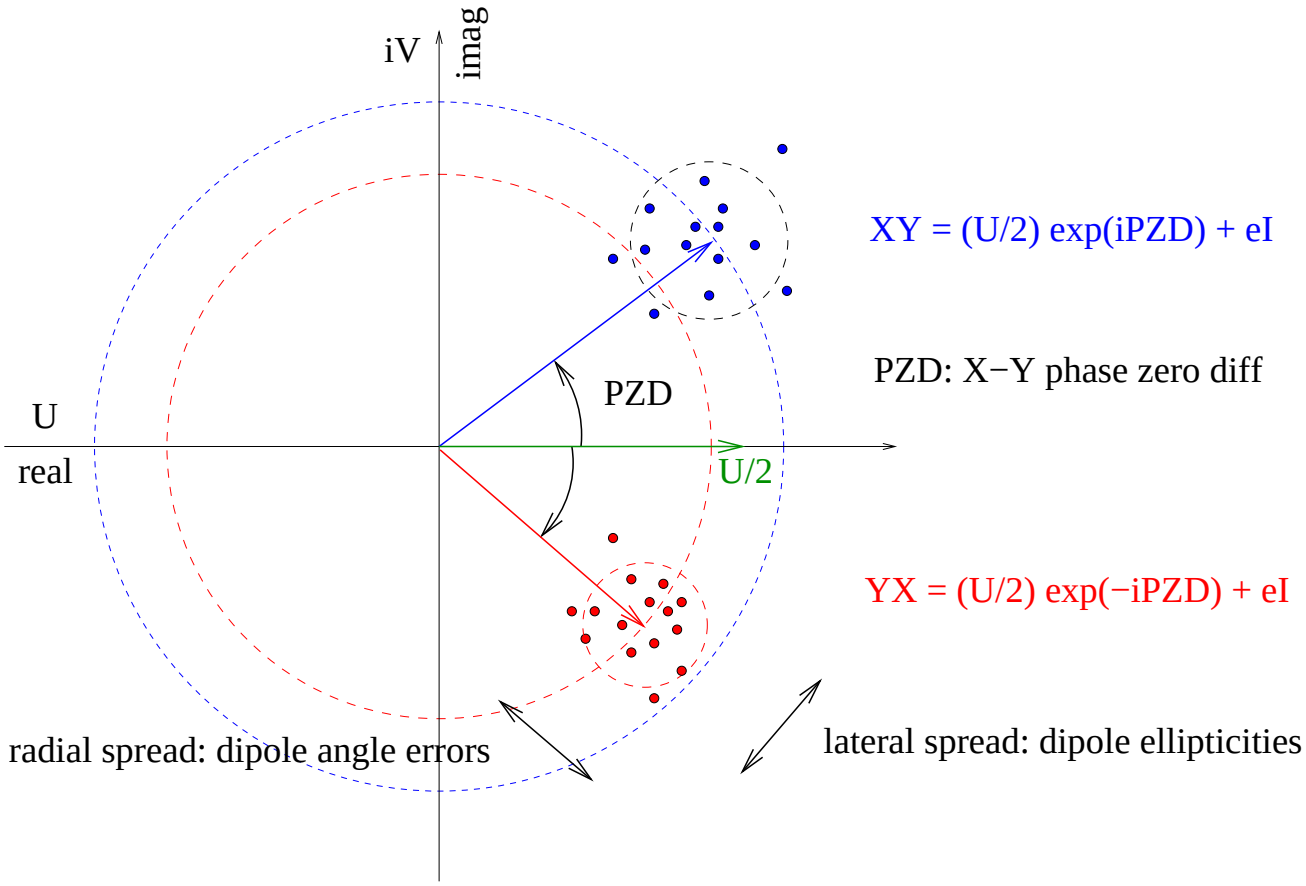


Figure 8: *A spigot2sink forest for general uv-data processing, including classic selfcal. For each correlation, the uv-data travels from spigot to sink, from the MS DATA colum to another MS column (e.g. RESIDUALS). The spigots issue a sequence of requests for successive smallish 'snippet' domains, which travel up the trees and trigger activity in the nodes. Arbitrary sequences of processing modules (subtrees) may be linked into trees for different applications. With suitable tools, a user may rapidly assemble his own MeqForest, much like LEGO. In addition, a range of 'canned' MeqForests will be made available for standard applications like data-inspection, flagging, calibrator reduction, etc.*



WSRT XY and YX data from a polarised (but: V=0) central point source, e.g. 3c286

Figure 9: *Polarisation calibration of the WSRT. This particular visualiser subtree shows the average visibility per correlation for a particular time. The source is a polarised (U) point source calibrator (e.g. 3c286@21cm) in the centre of the field. It is assumed that the dipole phases and gains have been corrected, but not the dipole angle errors and ellipticities. The latter manifest themselves as lateral and radial spread of the clouds of XY and YX data. These are rotated in opposite directions away from the real axis by the X/Y Phase-Zero-Difference (PZD). This is an artifact of the two separate selfcal solutions for the phases of the X and Y dipoles. After solution for the remaining dipole errors, the two data-clouds should be much smaller, and on top of each other on the real axis, at an amplitude of U/2.*
The prevailing view is that an unpolarised calibrator is needed to solve for phase and gain, while a polarised calibrator with appreciable U is needed to solve for the remaining errors. This is laborious, and also unfortunate since suitably polarised calibrators are rare, and the U may be converted into Q by ionospheric Faraday rotation. The proposition is that a single unpolarised calibrator is sufficient to solve for all errors. This technique can be verified and tuned by the unprecedented visualisation and MeqParm fiddling that is offered by PSS.

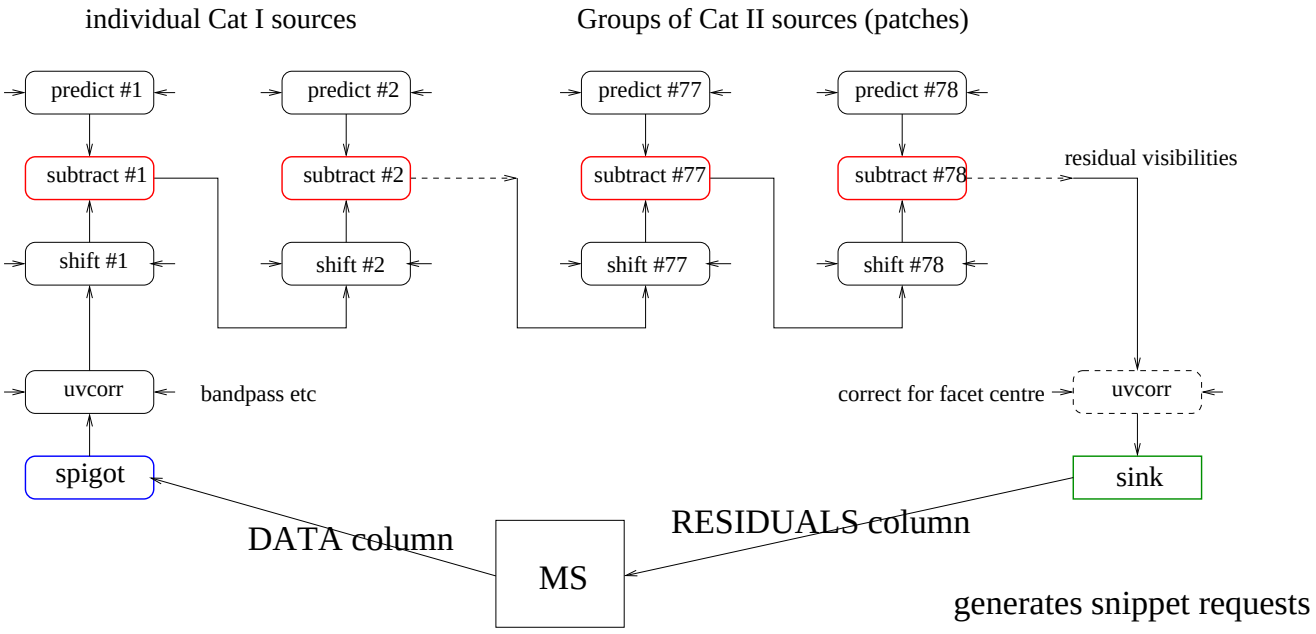
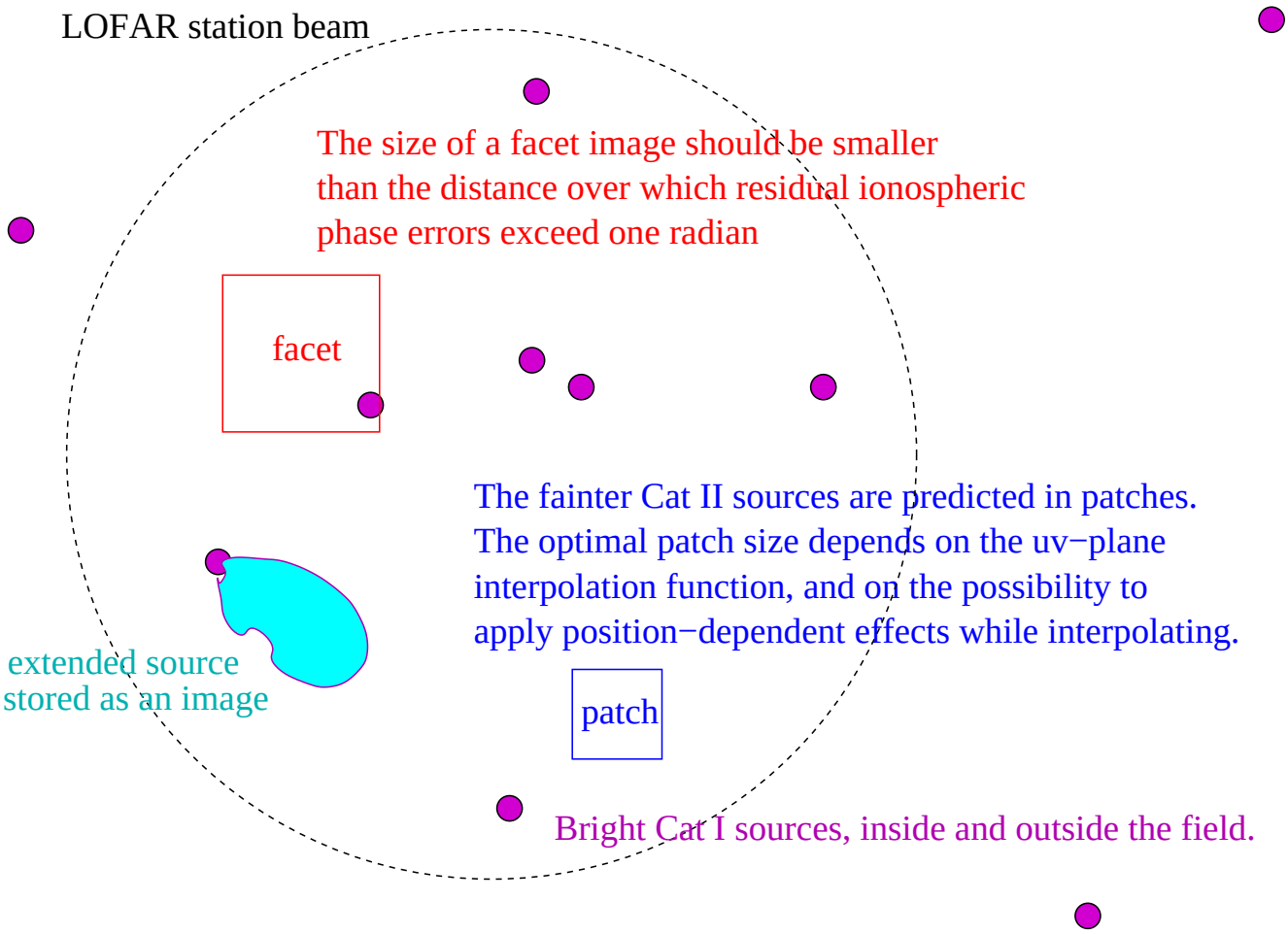


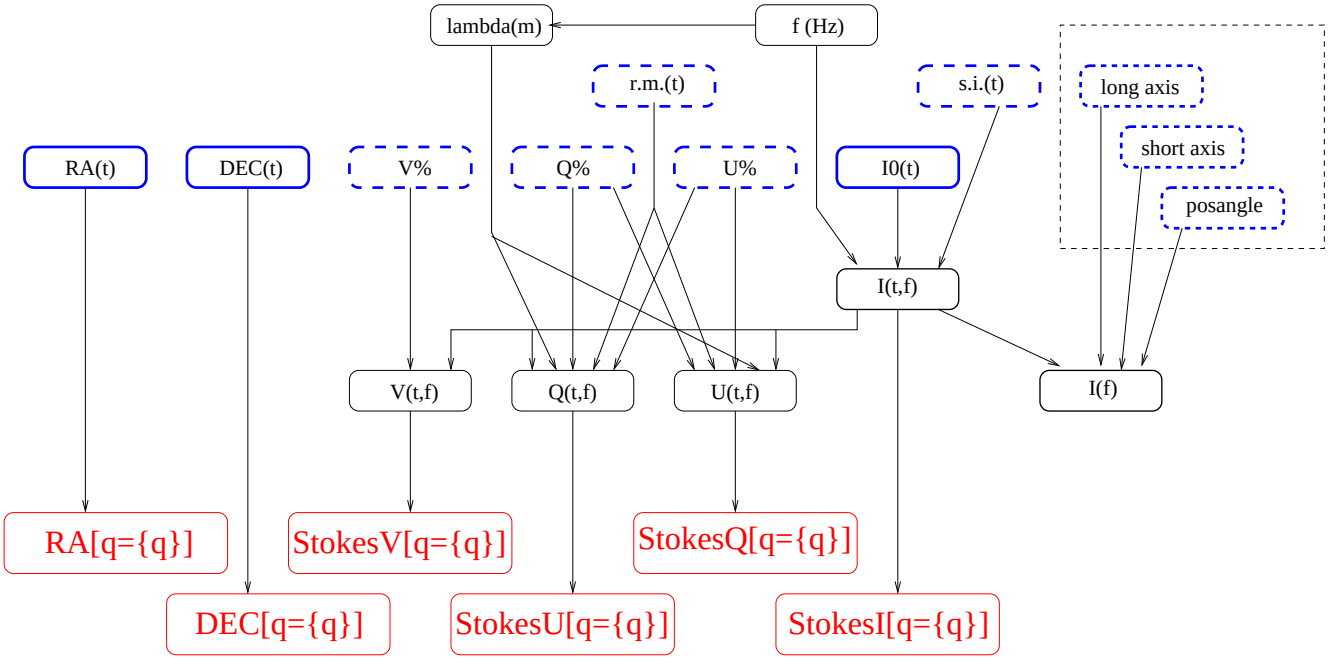
Figure 10: *The peeling process solves for position-dependent instrumental effects by doing selfcal on individual sources in order of brightness. The phase centre of the uv-data is shifted to the (apparent) position of the brightest source, so that its visibility function is approximately flat over a snippet domain (t,f). This causes a large reduction in processing, since the source only needs to be predicted for a small number of relatively large cells.*
After solving for instrumental parameters in the direction of this source, it is subtracted (peeled off) and the next brightest source becomes the peeling source. This is repeated until there are no more Cat I sources, i.e. sources that are bright enough to give a S/N>3 for a snippet. Obviously, the presence of fainter sources will influence the selfcal solution for the peeling source. This contamination can be reduced to abirarily low levels by taking 1-10 such sources into account.
The position-dependent instrumental errors are used to determine the shape of individual station voltage beams, or the ionospheric phase screen across the field of view. Images are made of residual data, i.e. data from which all the sources in the LSM have been subtracted. Peeling is used again, first to subtract the Cat I sources with their own corrections (so maximum accuracy), and then to subtract the remaining sources in groups (patches), using interpolated instrumental corrections.



Contents of a Local Sky Model

Figure 11: *The Local Sky Model (LSM). This is a subset of the Global Sky Model (GSM). The LSM contains all the known sources in the main lobe of the station beam around the pointing centre, and as many of the contaminating bright sources all over the sky as cause trouble via the relatively high sidelobes of the LOFAR station beams. The LSM contains a (large) list of parametrised sources, whose parameters are in MeqParm tables. Each source is associated with 6 MeqTrees that calculate RA, DEC, StokesI,Q,U,V for use by the MeqTree kernel. These are incorporated in the M.E. forest, and used to predict the contributions of brightest (Cat I) sources, and to solve for their parameters. They are also used to improvethe parameters of the fainter (Cat II) sources in the LSM from the pits and bumps in the residual images at the position where they have been subtracted. Thus, the LSM MeqTrees are a universal interface between the image plane, and ANY source parametrisation scheme. At this moment, only the NEWSTAR parametrisation scheme is used. But in the future we might think of other schemes like shapelets or pixons.*
The LSM also contains images (of CLEAN components) of extended sources that are not easy to model as parametrised components. These can be predicted and subtracted from the uv-data. It has recently been shown by Bhatnagar et al that position-dependent instrumental effects may be applied to extended sources as part of the uv-plane interpolation scheme. This can be used in a general scheme where patches of Cat II sources are collected into a temporary image, and predicted as a group using an FFT and uv-plane interpolation.

Example: NEWSTAR source parameters



6 standard 'connection' nodes (independent of source parametrisation!)

Figure 12: *A typical clump of subtrees that enshrine the relationship between the parameters of a source (stored in a MeqParm table of the LSM), and the image quantities used in processing. It appears to be sufficient that, for each Cat I source or Cat II patch, the LSM can produce a small set of only 6 connection nodes* ***with standard names****, which replace place-holder nodes with the same names in other trees, e.g. for peeling. For each source, the parametrised qualifier [q={q}] is 'qualified' to [q=2c84] or q=[patch34]. Each connection node is the root of a subtree that can be quite complex, but is mercifully hidden from view. Note that sources may be parametrised in many different ways, ranging from the simple set that is used in the NEWSTAR package, to shapelets or pixons. The important thing from an operational standpoint is that such details are hidden behind the standard connection nodes.*

See 2.

13 Residual Images

Together with the GSM, residual images represent the main deliverables of the radio telescope of the future. Since uv-data can only be corrected for a single point in the sky[9], as many sources as possible must be subtracted from the uv-data before transforming them into a (facet) residual image. Before the FFT, the uv-residuals are corrected for the centre of the facet (sub-image). Due to position-dependent instrumental errors, the image quality of the remaining (Cat III) sources will decrease towards the edge of the facet[10]. Therefore, the maximum size of a facet depends on the observing conditions, e.g. the behaviour of the ionsphere.

Residual images may be used to update the parameters of LSM sources. This is done by using the residual values at the positions where these sources have been subtracted (from the uv-data) together with the source prediction subtrees in the LSM (see fig 12). The latter are used in simple solving trees in the MeqTree kernel, which solve for improved values of the source parameter in the MeqParm table of the LSM (see fig 11).

See 3

14 Generating trees

In principle, this is very simple. The kernel interface provides a number of (Glish, later also Python) simple commands that make it possible to define nodes in the form of a definition-record (defrec). As a minimum, a defrec specifies a (unique) nodename, a node class (e.g. MeqParm) and zero or more children. The latter may be in the form of node-names, or in the form of defrecs. The latter possibility thus allows hierarchical defrecs that represent complex subtrees. The command that creates a C++ node looks like:

MQSV.MEQ('CREATE.NODE', [CLASS, NAME, CHILDREN, EXTRA])

where MQSV is a MeqServer object that interfaces with the kernel. After creating any number of nodes, they are instructed to locate their children via the command:

MQSV.MEQ('RESOLVE.CHILDREN', [NAME])

where NAME is the name of a starting node. That is basically all! It is now possible to inspect the

[9] This implies that there is no such thing as 'calibrated' uv-data.

[10] Since the PSF seems to degrade rather benignly towards the edge of a facet, it may still be possible to deconvolve Cat III sources in residual images.

structure of the new forest with the (Python) tree browser, and execute it in various ways.

In practice, software tools will often be needed to generate forests, which can be very complex.

14.1 Example: the JEN scheme

With the simple PSS kernel interface, many different schemes for tree generation are possible, and we hope that users of PSS will develop such schemes, with many lovely tools to allow users to cobble together trees of mind-boggling complexity. Of course the PSS team has developed its own scheme, which may serve as a starting point. An important feature is the use of node-names with parametrised qualifiers.

XXX_YYY_ZZZ[S1={S1}][S2={S2}][Q={Q}]

A tree is built up from predefined subtree modules that may reside in files, or are obtained from the Local Sky Model (e.g. StokesI[q={q}], see fig 12).

NB: The rest will come later. I have to deliver this document for dissemination.

15 Executing forests of trees

15.1 Each node can be executed by name

Every node can be accessed by name or index number, via the kernel command interface, or via the browser. In this way, its internal state can be read (and modified!), and it can be executed with a new request or with the one it already has. A node may be activated or de-activated, or be made to publish its result whenever it produces one. Finally, break-points may be set in each node, and the process can be halted, stepped and resumed via the browser.

15.2 MeqSinks and other root nodes

MeqSinks generate a succession of 'snippet' requests, i.e. requests with a certain domain size, which iterate through the available uv-data.

15.3 Visualisation: pre-defined views

Each node can be inspected individually, and its cache-result can be plotted. In addition, a well-designed tree has one or more 'data-collection' and 'data-concatenation' nodes, which offer pre-defined views of the (processed) results of their child nodes. When activated from the browser, these nodes display their contents whenever a new result is calculated. When inactive, these nodes do not slow down the system, and take up little memory.

15.4 Solving: pre-defined solvers

A MeqSolver solves for a particular subset of MeqParm values (polc coefficients, really). Although the user may certainly change this subset, it is much more convenient to have a number of pre-defined solvers that each solve for a particular MeqParm subset. Since they take up very little space, a large number of pre-defined solvers may be designed into the tree, and activated on demand.

15.5 Canned C++ Forests

Generating complex trees will be an advanced topic for some time to come. Moreover, the generation of a large forest from Glish or Python will be slow. Therefore, we will offer the user a range of 'canned' C++ forests for specific applications, which can be quickly loaded from a file. This has the added advantage that a canned forest has the following properties:

- It will adapt itself to a given MS. In the case of the WSRT, it will be designed for the nominal number of stations (14) and correlations (4, linear):
- Check whether the forest represents a suitable M.E. for the observation?
- Non-available ifrs: If a spigot cannot find its data, it returns a FAIL, wich should be handled gracefully, e.g. by a solver.
- If a solver is asked to solve for MeqParm(s) for which there is not enough information (e.g. in case of a missing antenna), it should handle this gracefully by solving for the others (e.g. by SVD), and not updating the relevant MeqParms. GVD: A solvable parameter coefficient will only be added to the solver if the solver gets perturbed data for it. So it will never appear in the solver if there is no data for it (e.g. if a station is not part of the MS).
- Connection to MS info, e.g. station positions (for DFT and redundancy calibration) etc. Such info should be made available to the tree in regular nodes. How is that done? OMS suggestion: send up the data-stream header in the request rider via the sink, and let the relevant nodes look for their own information, using a field-specification (defrec.header_field_id = hiid). This is elegant, because it allows for time-variable info.

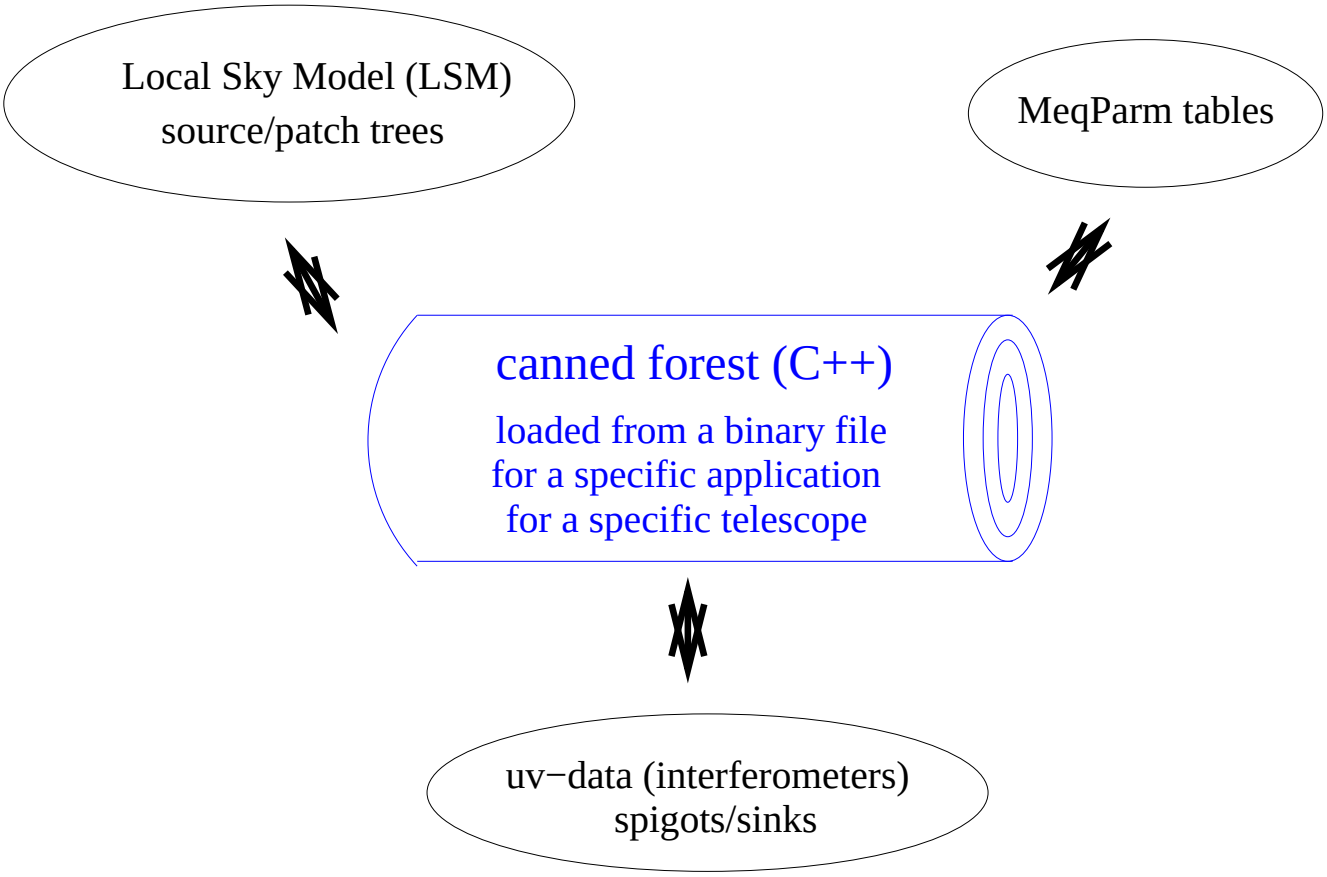


Figure 13: *Canned forests will probably play an important role in the routine use of PSS for data reduction. The collection of C++ nodes of a particular forest may always be stored in a binary file, and rapidly loaded back into the kernel for use with another observation. This is much faster, and much more convenient, than creating new forests from the scripting side each time. It is also much easier to exchange canned forests between users in different places. Canned forests of arbitrary complexity may be created by specialists for a wide range of applications and telescopes, and made available for more pedestrian users. However, canned trees can only be used for different observations if they can automatically interface with their environment, as indicated above.*

- The generation/filling of the parm table needed for the given MS and tree has to become part of the canned forest adaptation.
- It will adapt itself to a given LSM. For the moment this means that it will have a sufficient number of 'connection' nodes that serve as placeholders for prediction trees for either Cat I sources or Cat II patches from the LSM.
- Check whether the LSM and the MS overlap in the sky....
- Attach source prediction trees from the LSM to the successive peeling units. These are different for Cat I/II. Unused peeling units are shunted out, or even deleted. Qualifiers [q={q}] in the node-names of the peeling-units must be modified [q=xyz].
- Something similar must be done for non-peeling trees.
- It is possible to replace specific nodes/subtrees of a canned forest with other nodes/subtrees. This allows people like Sanjay to experiment with their own Cat II prediction node, without having to generate a complex tree himself.

16 Meeting at the water hole

In nature, a water-hole is where animals of all kinds come to drink and interact. This is the metaphore we have chosen for a repository of selected MS's, a growing number of canned forests, and some simple processing scripts. The hope is that this will be intersting enough, at a low enough threshold, to involve active astronomers in the exercise and development of PSS. We can but try.

17 Freedom layer (at last!)

The set of official node classes has been defined in such a way that a tree-literate user may build his own functions (subtrees) as much as possible. Examples are visualisation and flagging (see elsewhere uin this document).

Exchange with other users:

- Canned forests

- Forest templates

- Processing scripts

- C++ node classes

- All kinds of tools

Writing your own:

- Forests

- C++ node classes (various levels)

- LSM

- Scripts (Python and Glish)

Full access to:

- MeqParm tables (AIPS++)

- LSM ...

- residual images

- uv-data (AIPS++ MS)

LSM/GSM

Scripts (Python and Glish)

Tools

Tree browser and its many features

Look-and-feel (user interface)

18 PSS status

Lots of bits and pieces exist, are coming together now.

Solutions exist for many problems

18.1 Things that exist (and work)

OMS: A working kernel (creation and execution of nodes), including documentation

- Scripting interfaces with Glish and Python

- Interface with AIPS++ MS (fast, forward-looking)

- Interface with MeqParm tables (AIPS++)

- A powerful tree browser (Python)

- A range of 'official' nodes (for math, solving, data, visualisation etc)

RXA: Organisation of the water-hole

OMS: An interface for designing and creating simple trees

JEN: A system for designing highly complex trees, including symbolic calculator for generating very complex mathematical expressions.

AGW/OMS: Visualisation (Python)

JEN: A range of canned forests (with in-built request) for playing/learning

18.2 The short term (weeks)

MeqParm behaviour (solvable and otherwise)

Cat II prediction

Remodulate the phaser banks

18.3 Slightly longer term (months)

Fringe fitting (VLBI)

RM-synthesis

Acknowledgements

It is a privilege to work with the highly diverse PSS team, consisting of Oleg Smirnov, Tony Willis, Ronald Nijboer, Rob Assendorp, Ger van Diepen and Tom Oosterloo. Special mention must go to Oleg who, in his inimitable way, keeps his MeqKernel in the air, and also his lovely tree browser.

References

[1] MeqTree Kernel (OMS)

[2] MeqTrees for dummies (AGW)

[3] HBS formalism (Hamaker et al)

[4] Measurement Equation, AIPS++ note 185 (JEN)

[5] LOFAR calibration, peeling, etc (JEN)

19 Appendix: Current focus program

Our current focus program is to develop a series of Spigot2sink trees of increasing compexity. The following list of (canned) spigot2sink forests is just a start. Their available number and complexity will rapidly increase. We need a working example of a simple python interface (with explanation) with which to execute a spigot2sink forest with a given MS. At the suggestion of GVD, I hereby include performance testing as an explicit goal, both in speed and memory use. Speedwise, this includes data-access (getting in and out of an MS), MeqParm behaviour, and the identification of bottlenecks in specific nodes (like Cat II prediction, or the full-resolution shift of the phase-centre for peeling). Memorywise, it includes caching strategy, and the minimisation of the number of nodes.

19.1 spigot2sink 1: Just uv-data visualisation

We already have a suitable spigot2sink forest, with built-in views (visu_nodes) that are visible, and can be activated, from the top of the browser. We also have node classes for data collection node and concatenation. The missing link is a python plugin for the browser that receives the result from such a node, and displays it. For the moment, we are concentrating on the following views:

- Visibility data from groups of ifrs (e.g all XX) in the complex plane (2D). Rather than tranporting the values of all the cells, it is often sufficient to transmit only the rms/mean over the entire vells (ignoring flags, of course). This can be done by the existing math nodes.
- Spectra (3D), real and imaginary parts. To increase speed, we might limit the number of ifrs per view (e.g. all combinations with a particular station/telescope).

Hopefully, this will be ready for demonstration at the Nov 25th gathering. Once the groundwork has been done for these these two items, it will be straightforward to generate many other kinds of views. Most importantly, this can be done by users themselves, as part of the freedom layer.

19.2 spigot2sink 2: Flagging

This includes its visualisation (before and after) and flag-statistics. We already have the node classes that are needed for flagging, and flags are read from the MS and written back. However, some more work is needed in the bowels of the kernel to ensure the proper treatment of flags in mathematical operations.

The flagging node classes have been defined in such a way that the user himself may construct flaggers of various kinds (using various flagging criteria and thresholds). Flagger subtrees may of course be cascaded.

19.3 spigot2sink 3: Bandpass estimation

Just solve for a 5th-degree polynomial in the frequency direction. Spectrum visualisation before and after.

19.4 spigot2sink 4: Gain and phase selfcal

On a bright calibrator, i.e. without LSM. The 'model' is a 1 Jy point source in the centre of the field, with a spectral index of -0.7. We already have a suitable spigot2sink forest, with multiple solvers, and phase constraints. For the moment, only standard math nodes have been used.

- Straightforward solution for XX/YY phase and gain per telescope.
- Redundant-baseline solution, followed by a solution for absolute gain and phase gradients (align).
- Solvers may be (de-)activated via the browser.

The MeqParm in its current state is sufficient for calibrator selfcal (but for real reduction we need the new behaviour that we recently figured out).

- Solve for one time-slot (1-2 min) at a time, save a separate polc per timeslot in the MeqParm table.
- When asked for values, it uses its available polcs somehow.

We will offer condeq visualisation for all solvers. The simplest way is to have fixed views that show the XX and YY condeqs in the complex plane. This uses the same 2D visualisation nodes as above. More advanced viewers that show the convergence can be developed later. It would be nice to show the solver metrics, especially in the case of problems. We will then subtract the 'model' from the corrected uv-data (NB: This is a demo of peeling, with one peeling source).

19.5 spigot2sink 5: Polarisation selfcal

See fig 9.

On a bright polarized calibrator (3c286 has 10% U at 21cm).

- JEN: spigot2sink forest, with multiple solvers (ReqSeq)
- JEN: XX*YY - XY*YX = I^2 - (Q^2 + U^2 + V^2)

19.6 spigot2sink 6: Manual MeqParm fiddling

Changing the value(s) of MeqParm(s), while executing and visualising. Some examples of its use:

- Fiddling the X/Y phase-zero difference and/or Faraday rotation, while visualising XY/YX, or solving for dipole errors.
- Fiddling StokesQ, while visualising XX/YX, or solving for dipole errors.

Start/stop an endless loop (from the browser), in which a selected node calculates a result for the same (current) request. Obviously, this should only be possible when the system is halted, and be de-activated automatically when oparations are resumed. We need an elementary MeqParm fiddling widget, to be activated from the browser (plugin). Uses getState and setState funcions. For the moment, only allow one coefficient (c00) to be modified. A byproduct is a phased plan for MeqParm visualisation (eventually this must include the available polcs in the MeqParm table). To be activated from the browser (plugin).

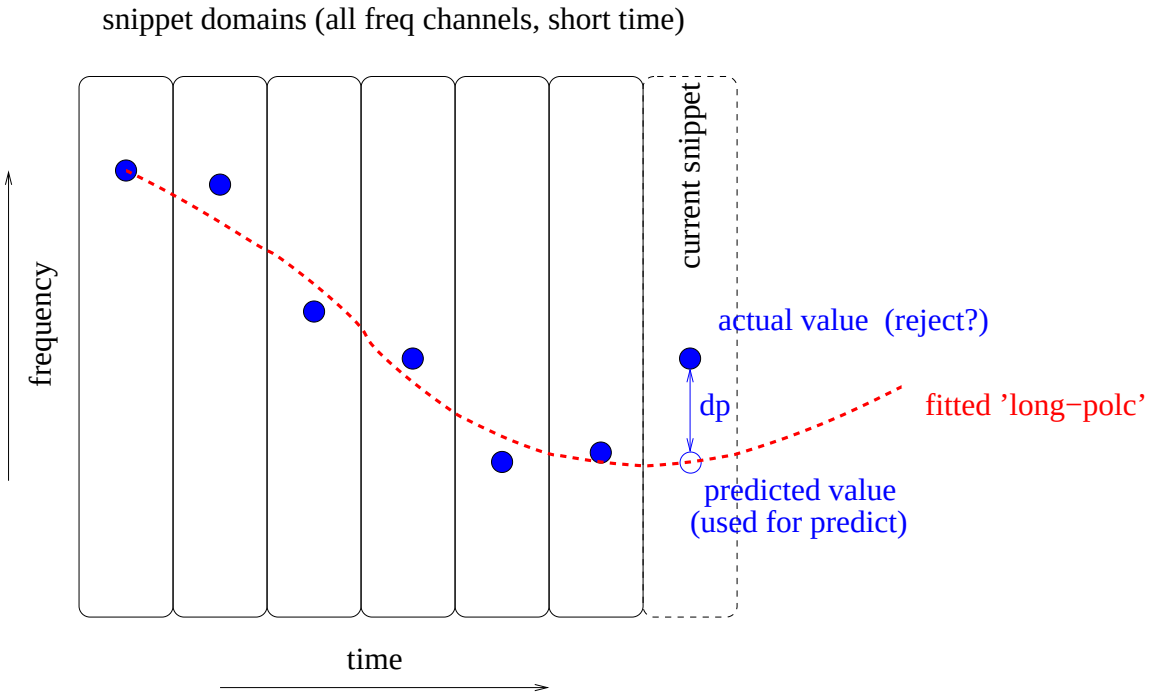
20 Appendix: Some more detail

20.1 MeqParm behaviour while solving

20.2 Cat II prediction

Position-dependent instrumental effects may be applied via a suitable interpolation function (Cornwell, Bhatnagar, Golap, 2004).

20.3 Other?



An elegant scheme for MeqParm solution behaviour:

- The blue dots (●) are the results of snippet solutions (c0f only)
- Each provides an equation that is added to the 'long−polc' solution matrix
- The latter is inverted each time, to give better long−polc coefficients
- The validity domain of the current long−polc is increased with each snippet
- The updated long−polc is used to predict the value for the current (next) snippet
- The differences (dp) between predicted and actual snippet values are kept
- These can be used to accept/reject a snippet solution in a policy−free manner
- They can also be used to decide to start a new long−polc

Question: Is this scheme equivalent to a single long−snippet solution for a long−polc?

Figure 14: *A solvable MeqParm.*

Contents