Some Announcements

- Wednesday bridge and Friday football is on, just like the previous week
  - think about it, we'll do a head count later
- Previous days' exercise solutions now linked on the wiki.

ME3: Calibration & Correction

Objectives:

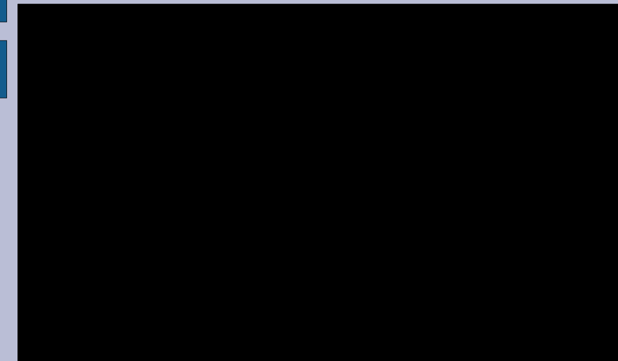
- Figuring out what calibration is!
- Framing the calibration problem in Measurement Equation terms.
- Implementing some calibration trees.

svn up Workshop2007 please!

What Is Calibration? ...according to Google

- # The process whereby the magnitude of the output of a measuring instrument is related to the magnitude of the input force driving the instrument (ie Adjusting a weight scale to zero when there is nothing on it). (Course Material/Ultrasonics/CalibrationMeth/calibrationmethods.htm)
- www.ndt-ed.org/GeneralResources/Glossary/letter/c.htm
- # The process of adjusting an instrument or compiling a deviation chart so that its reading can be correlated to the actual value being measured.
- www.omega.com/literature/transactions/volume1/glossary.html
- # The process of choosing attribute values and computational parameters so that a model properly represents the real-world situation being analyzed. For example, in pathfinding and allocation, calibration generally refers to assigning or calculating appropriate values to be entered in impedance and demand items.
- www.geog.leeds.ac.uk/staff/m.blake/magis/glossary/esriglos.htm
- # (cal·i·bra·tion) (kal^{2}[ibreve]-bra¢sh[schwa]n) 1. determination of the accuracy of an instrument, usually by measurement of its variation from a standard, to ascertain necessary correction factors. 2. measurement of the caliber of a tube.
- www.mercksource.com/pp/us/cns/cns_hl_dorlands.jspzQzpgzEzzSzppdocszSzuszSzcommonzSzdorlandszSzdorlandzSzdmd_c_03zPzhtm
- # Checking, adjusting and systematically standardizing the graduations of a device.
- www.inkcartridgesworld.com/_glossary.html
- # The process of establishing a set of values for correct operation.
- www.blazepoint.co.uk/faq_label_printers_glossary.html
- # A process that establishes, under specified conditions, the relationship between the values indicated by the measuring system, and the corresponding values of a quantity realised by a reference standard or working standard.
- www.nzelectricity.co.nz/H9glossary.htm
- # process of comparing an instrument's accuracy to known standards
- www.tsgc.utexas.edu/stars/glossary1.html
- # is defined as the process of quantitatively defining the system response to known, controlled signal inputs.
- www.eumetsat.int/en/dps/helpdesk/glossary.html

...according to Noordam



...according to the EoR KSG

EoR Signature???

Our Definition For Today...

- Determining the properties of the sky and the instrument with sufficient accuracy.
- “Taking out” (as much as possible) instrumental corruptions.
- Subtracting known sources
- ...to find the noise (or at least to achieve our scientific objectives)

“Classic” Calibration (for phases/gains)

Assume this m.e.: $v_{pq} = g_p \mathcal{F}(b) g_q^*$

1. Start with a model for the sky brightness, $M(l,m)$
2. F.T. that into model coherencies: $x(u,v) = \mathcal{F}(M)$
3. Predict 'corrupted' model: $x'_{pq} = g_p x_{pq} g_q^*$
4. Find g_ps by fitting x'_{pq} to observed v_{pq}
5. Compute corrected visibilities: $v'_{pq} = g_p^{-1} v_{pq} (g_q^{-1})^*$

The "corrected" visibilities are then in an F.T. relationship with the true sky b: $v'_{pq} = \mathcal{F}(b)$

The M.E. Analogue

Assume this m.e.: $\boldsymbol{V}_{pq} = \boldsymbol{G}_p \mathcal{F}(\boldsymbol{B}) \boldsymbol{G}_q^\dagger$

1. Start with a model for the sky brightness, $\boldsymbol{M}(l,m)$
2. F.T. that into model visibilities: $\boldsymbol{X} = \mathcal{F}(\boldsymbol{M})$
3. Predict 'corrupted' model: $\boldsymbol{X}'_{pq} = \boldsymbol{G}_p \boldsymbol{X}_{pq} \boldsymbol{G}_q^\dagger$
4. Find $\boldsymbol{G}_p$s by fitting $\boldsymbol{X}'_{pq}$ to observed $\boldsymbol{V}_{pq}$
5. Compute corrected visibilities: $\boldsymbol{V}'_{pq} = \boldsymbol{G}_p^{-1} \boldsymbol{V}_{pq} (\boldsymbol{G}_q^{-1})^\dagger$
   (note that $(\boldsymbol{G}^\dagger)^{-1} = (\boldsymbol{G}^{-1})^\dagger$)

We then again have $\boldsymbol{V}'_{pq} = \mathcal{F}(\boldsymbol{B})$

Or In Broad Terms...

1. Predict corrupted visibilities
   - we covered this last week
2. Fit to observed visibilities
   - solving for parameters of the sky and/or the instrument
3. (Optional: subtract bright sources)
4. **Correct**
5. Rinse & repeat
   - aka the “major loop”: source extraction, updating sky model, etc.

Applying Corrections With MeqTrees

- The **Meq.MatrixInvert22** node inverts 2x2 matrices.
  - (generalized inversion not yet available)
- A **Meq.Spigot** reads data from the MS.
- See `ME3/demo1-correct-gains.py`.
- Re-run ME1 exercise 1 to simulate an MS with instrumental polarization

  (but fix a_p=*p1e-10)**
- Run `ME3/demo1` on this MS, write to the CORRECTED_DATA column, make an *IQUV* channel map of this column.

Exercise 1: Correcting For Parallactic Angle

- Re-run ME1 exercise 2 to produce an MS with instrumental polarization and alt-az mounts

  (fix a_p=*p1e-10)**
- Make an per-channel IQUV map of the DATA column
- Modify `ME3/demo1` to correct for P.A. as well
- Write corrections to CORRECTED_DATA
- Make an *IQUV* channel map of the CORRECTED_DATA column
- Did you get the original, uncorrupted point source back? (*I*=1 Jy, *Q*=.2 Jy, *U*=*V*=0)

Correcting For Multiple Jones Terms

Given an m.e. of the form:

$$\boldsymbol{V}_{pq} = \boldsymbol{J}_{pn} \cdots \boldsymbol{J}_{p1} \boldsymbol{X}_{pq} \boldsymbol{J}^{\dagger}_{q1} \cdots \boldsymbol{J}^{\dagger}_{qn}$$

the corrections need to be applied in reverse order:

$$\boldsymbol{V}'_{pq} = \boldsymbol{J}^{-1}_{p1} \cdots \boldsymbol{J}^{-1}_{pn} \boldsymbol{V}_{pq} (\boldsymbol{J}^{-1}_{qn})^{\dagger} \cdots (\boldsymbol{J}^{-1}_{q1})^{\dagger} =$$

$$= \boldsymbol{J}^{-1}_{p1} \cdots \underbrace{\boldsymbol{J}^{-1}_{pn} \boldsymbol{J}_{pn}}_{=1} \cdots \boldsymbol{J}_{p1} \boldsymbol{X}_{pq} \boldsymbol{J}^{\dagger}_{q1} \cdots \underbrace{\boldsymbol{J}^{\dagger}_{qn} (\boldsymbol{J}^{-1}_{qn})^{\dagger}}_{=1} \cdots (\boldsymbol{J}^{-1}_{q1})^{\dagger} =$$

$$= \boldsymbol{X}_{pq} = \mathcal{F}(\boldsymbol{B})$$

...and all matrix (non-)commutation rules apply.

So, Is There Always Such A Beast As "Corrected" *uv*-Data?

Say we now have some image-plane effects:

$$\boldsymbol{V}_{pq} = \boldsymbol{G}_p \mathcal{F}(N_p \boldsymbol{E}_p \boldsymbol{B} \boldsymbol{E}_q^\dagger N_q^\dagger) \boldsymbol{G}_q^\dagger$$

... and we know all of the $\boldsymbol{G}_p, \boldsymbol{E}_p$, and (of course) N_p's;
then is there a way to obtain "corrected" visibilities $\boldsymbol{V}'$

such that $\boldsymbol{V}'_{pq} = \mathcal{F}(\boldsymbol{B})$???

(or at the very least $\boldsymbol{V}'_{pq} = \mathcal{F}(N_p \boldsymbol{B} N_q^\dagger)$) ???

Exercise 2: Ionospheric Corrections

- Take our old `ME2/demo-30-190.MS`
  - re-run ME2/example6-iono.py to corrupt for ionosphere
  - make an image to verify corruptions
- Make a script to take the ionosphere back out, write results to the CORRECTED_DATA column.
- Make a per-channel map, then a movie.

And The Answer Is...

- In general, **NO!**
- *uv*-plane effects (the $\boldsymbol{G}$s) can be taken out.
- Image-plane effects correspond to convolution in the *uv*-plane:

$$\boldsymbol{V}_{pq} = \mathcal{F}(N_p \boldsymbol{E}_p \boldsymbol{B} \boldsymbol{E}_q^\dagger N_q^\dagger) = \mathcal{F}(\boldsymbol{E}_p) \circ \mathcal{F}(N_p \boldsymbol{B} N_q^\dagger) \circ \mathcal{F}(\boldsymbol{E}_q^\dagger)$$

  - ...with time-variable kernels
  - ...and with each baseline's *uv*-plane sampled along just a single track

(Note: Bhatnagar et al. (EVLA Memo 100) suggest a method for <u>approximate</u> correction during the imaging step. We'll return to this later.)

Correcting At A Single Point

But we can correct for a single point l_0, m_0:

$$\boldsymbol{V}' = \boldsymbol{E}_p(l_0, m_0)^{-1} \boldsymbol{V} \boldsymbol{E}_q^{-1}(l_0, m_0)^\dagger =$$

$$= \mathcal{F}\left((\boldsymbol{E}_p(l_0, m_0))^{-1} \boldsymbol{E}_p N_p \boldsymbol{B} N_q^\dagger \boldsymbol{E}_q^\dagger (\boldsymbol{E}_q^\dagger(l_0, m_0))^{-1} \right)$$

$$= \mathcal{F}(N_p \boldsymbol{B}' N_q^\dagger)$$

where $\boldsymbol{B}'(l_0, m_0) = \boldsymbol{B}(l_0, m_0)$, but diverges further away.

- In general, *uv*-data can only be "corrected" for a single point on the sky.
- This is the motivation for facet imaging.

Calibration, Revisited

1. Predict corrupted visibilities
   - we know this, this is simulation
2. **Fit to observed visibilities**
   - solving for parameters of sky and/or instrument
3. (Optional: subtract bright sources)
4. Correct
5. Rinse & repeat
   - aka the "major loop": source extraction, updating sky model, etc.

A General Approach To Fitting & Solving

- A tree evaluating any $m(t,\nu)$ also depends on values of parameters up in the tree. We write this as:
  $m(t,\nu;a,b,...)$
- Imagine a "magic" constant node ***a*** that returns not one, but two values: a and $a+\delta a$.
- Its parent, *f*, then returns $f(a)$ and $f(a+\delta a)$...
- ...and at the bottom we get $m(t,\nu;a)$ and $m(t,\nu;a+\delta a)$

From this we can estimate:

$$\frac{\partial m}{\partial a} \approx \frac{m(a+\delta a)-m(a)}{\delta a}, \quad \frac{\partial m}{\partial b} \approx \frac{m(b+\delta b)-m(b)}{\delta b}$$

And then try to minimize or maximize *m*...

(Which even a salmon can do.)

Maaijke's Turn: Introduction To Solving

- cd Workshop2007/Solvers
- See separate slides in Solving.pdf

Exercise 3: Fitting The Ionosphere

- Start with `Intro1/example7-iono3.py`
- Take the tec:2 node, which returns TEC as a function of *x,y,t*
- Make a solver tree:
  - TEC(t;xy) on one side
  - MIM(t;xy) on other side: $MIM(t;xy)=\sum_{k,l} c_{kl}(t)\, x^k y^l$
  - each $c_{kl}(t)$ should be a polc in time, you solve for its coefficients
  - polynomial order (in time and *xy*) should be a compile-time option
- Play with various polynomials to see how well we can fit the TEC.

Calibration Of An MS

- A model tree computes corrupted visibilities $\boldsymbol{X}_{pq}(t,\nu)$
- Spigots return observed data $\boldsymbol{V}_{pq}(t,\nu)$
- We can take the difference and form up a χ^2 sum...
- ...and try to minimize it w.r.t. the solvable parameters.
- Which is the same as fitting the model to the data, in a least-squares sense.
- We can thus solve for any (reasonable) subset of parameters of a measurement equation.

A Real-Life Example: 3C343

- Field is dominated by two bright sources:
  - 3C343.1 (~6 Jy) at phase center
  - 3C343 (~1.8 Jy) off-center
- Significant polarization
- 12-hour WSRT observation, 03/08/2000
- 64 channels ~ 1.2 Ghz (we use 26)
- `3C343.MS` pre-processed by Michiel Brentjens
- Pristine copy:

```
cp -a (/net/birch)/data/oms/Workshop2007/3C343.MS .
(/apps/Timba/data on jop0x)
```


Viewing MSs

- See `ME3/demo2-view343.py`
- This is just like our old spigot-sink script from Intro2, but rewritten with Meow
- The simplest/quickest kind of MS inspector you can make with MeqTrees, you can use it for any MS...
- Load the inspect_spigots bookmark and run the tree.
- Make an image of the DATA column.

Step 1: Solving For Source Fluxes

- See `ME3/demo4-cal343.py`, build the tree but do not run it yet
- This uses Meow to construct a model with two point sources
- Simultaneous solution for two sources
  - note how this is different from peeling
- I and Q fluxes are **Meq.Parm**s: i.e. potentially solvable parameters
  - polynomial in frequency

Polynomial Fluxes?

- I and Q fluxes are set up with a *shape,* to make them polynomials of frequency
- This accounts for spectral indices, and also beams and instrumental polarization

This is the m.e. we end up with:

$$\boldsymbol{V}_{pq}=\boldsymbol{K}_{p1}\boldsymbol{B}_1\boldsymbol{K}^{\dagger}_{q1}+\boldsymbol{K}_{p2}\boldsymbol{B}_2\boldsymbol{K}^{\dagger}_{q2}; \quad \boldsymbol{B}_s=\begin{pmatrix} I_s+Q_s & 0 \\ 0 & I_s-Q_s \end{pmatrix}$$

$$I_s, Q_s=\sum_k c_k \nu^k$$

CondEqs and Solvers

- **Meq.CondEq**s form up the difference between two branches
  - predicted and measured
- ...and estimate derivatives.
- A **Meq.Solver** uses these to run an iterative solution
- A separate solution is run for each tile.
- The previous tile's solution is used as the starting point for the next tile.

Request Sequencing

- Once we have a solution, we want to *apply* it to the data to generate, e.g., corrected data, or residuals
- This means we want to execute two branches in strict sequence:
  - first the Solver branch
  - then the correct/subtract/etc. branch
- A **Meq.ReqSeq** executes its two children in sequence, then returns the result of one of the children.

Running The Tree...

- Set Solver options in TDL Exec menu:
  - Convergence threshold: 0.001
  - Assume balanced equations: false
- Load up all bookmarks, set output column to CORRECTED_DATA, and run “test forest” with a tile size of 100.
- Observe plots in bookmarks.
- We should get *I* fluxes of 5.5~6 Jy and 1.6~1.8 Jy

Solving For Phases

- Flux solutions have been written out to a table (`3C343.MS/sources.mep`)
- The next time we use the I and Q Meq.Parms in a tree, they will be initialized from this table.
- So now we can try to solve for gain-phases, while keeping fluxes fixed

Exercise 4: Solving For Phases

Start with the previous demo, and add some solvable phase terms.

The following m.e. should be implemented:

$$\boldsymbol{V}_{pq}^{(\mathrm{predict})} = \boldsymbol{G}_p (\boldsymbol{K}_{p1} \boldsymbol{B}_1 \boldsymbol{K}_{q1}^\dagger + \boldsymbol{K}_{p2} \boldsymbol{B}_2 \boldsymbol{K}_{q2}^\dagger) \boldsymbol{G}_q^\dagger$$

$$\boldsymbol{G}_p = \begin{pmatrix} e^{i\phi_{px}} & 0 \\ 0 & e^{i\phi_{py}} \end{pmatrix}$$

ϕ_{px}, ϕ_{py} are solvable Meq.Parms of 0-order (i.e. non-polc),

(use 0 for a starting value)

The following correction should be implemented:

$$\boldsymbol{V}_{pq}^{(\mathrm{corr})} = \boldsymbol{G}_p^{-1} (\boldsymbol{V}_{pq}^{(\mathrm{obs})} - \boldsymbol{V}_{pq}^{(\mathrm{predict})}) (\boldsymbol{G}_q^{-1})^\dagger$$

Exercise 4, Continued

- Set tile size to 1 (i.e. a separate phase solution for every timeslot)
- Adjust some Solver options:
  - Convergence threshold: 1e-6
  - Assume balanced equations: true
- Run the tree
- Complete MS would take too long, but you can verify correctness by tracking χ^2, which should get smaller.
- Look also at the residual inspector.
- If you're ambitious, add an “inspector” for phases.

Flagging

- Real data has RFI and stuff, always needs flagging
- You can flag an MS using your favourite flagger...
  - or MeqTrees itself
- 3C343.MS already contains some coarse preliminary flags, but from looking at the residuals, they are obviously insufficient

Flags On Trees

- Data flags are represented by a **flags** field in the vellset of a result
- Load `ME3/demo2-view343.py:`
  - publish, e.g., spigot:0:1
  - run with a tile size of 100
  - right-click in inspector, select “Show” or “Hide flagged data”.
  - look at snapshot for spigot tile #6
- **flags** is an integer array of (usually) the same shape as value, non-zero indicates “flagged”.

Flag Propagation In MeqTrees

- ...only *usually* same shape as **value**
  - collapsed axes are possible (i.e. 100x1 flags for a 100x32 value, a.k.a. “row flags”)
- **flags** is a bitmask, each bit represents a *flag category*.
- Nodes ignore flagged values by default
  - controlled by flag_mask option, so you can selectively ignore flag categories
- Flags automatically propagate from child to parent

Flagging On The Fly

- Let's implement two simple flagging algorithms:

1. Absolute-value clipping:

flag if $|v(t,\nu)| \geq v_0$ (for v in XX,XY,YX,YY)

2. RMS clipping:

$$v_{\mathrm{abs}} := |v| \quad v_{\mathrm{mean}} := \langle v_{\mathrm{abs}} \rangle_{t,\nu} \quad \Delta v := v_{\mathrm{abs}} - v_{\mathrm{mean}}$$

flag if $|\Delta v| \geq n \cdot \mathrm{rms}(\Delta v)$

- Obviously, they ought to be applied to residuals...

Making New Flags

- All you need is two special nodes:
  - **Meq.ZeroFlagger** flags its child's value based on a comparison to 0.
  - **Meq.MergeFlags** merges flags across children.
- The child of the ZeroFlagger is called the flag condition. You can make any tree you like for the flag condition!
- Flagging becomes a *side branch* of sorts.

Flagging In Action

- Load up `ME3/demo4-flag343.py`
- This makes flagging trees for our two algorithms
  - conditional on compile-time options
- First make sure "Write flags to output" is **not checked.**
- If "Ignore MS flags" is set, the spigot is created with a flag_mask of 0, thus ignoring initial flags from the MS

Flagging In Action, Continued

- The two flaggers work in sequence
  - this is usually a good idea, as absolute clipping makes the rms estimate more accurate
  - **Meq.StdDev** computes the rms w.r.t. the mean value
- Use the CORRECTED_DATA column
- Experiment with various flag settings and see the effect via the inspectors.
- Flags won't be written until "Write flags to output" is checked.

Exercise 5: Flagging The Solution

- After a solve, we have residuals in the tree, so we can flag based on residuals on the fly.
- Start with the previous calibration script, and insert our two flaggers from demo4 between the residuals and the sinks.
  - also insert inspector, so we can see residuals before and after flagging

More Meowing

- Most solve trees look similar, and involve a lot of housekeeping.
- ...which is usually all the same.
- Sounds like a job for a framework.
- **Meow.StdTrees** implements a standard solver tree (among other things)
- See `ME3/example5-meow343.py` for a complete example

DO Try This At Home

- We'll try to do a “complete” calibration
- This is a hefty demo, we don't have enough CPU or RAM for all of you to run it simultaneously
- I'll run it myself
- Please load up and study the tree and script, but don't execute anything.
- You can try running it yourself off-line (as long as you don't do it all together...)

Using Meow.StdTrees

- We form up a predict tree as before
- We then create a standard **SolveTree** based on our predict tree
- We give it inputs (the spigots), and we get back outputs (the residuals)
- We correct the residuals
- ...add a few visualizers
- And feed the residuals to sinks
- Finally, we define some “solve jobs”

Managing Parameters, The Problem

- The problem: we use something like Meow to form up trees
- These trees will have Meq.Parm nodes in them somewhere, but we don't know what they're called
  - and we shouldn't know – these are implementation details, and they can change
- ...yet we must pass a list of parameters to the solver so that we can solve for them

Managing Parameters, The Solution

- *node*.search() searches all subtrees above the designated node, and returns a list of nodes matching some criteria.
- In this case matching the given **tags**.
- When Meow creates Meq.Parms, it tags them, following a certain convention
- **`predict.search(tags=”flux solvable”)`** then returns all solvables related to flux.

Why Tags Are Good

- At the top level, we don't need to know any details about which Meq.Parms our tree has
- ...we just need to know the tagging convention
- We then have a generic mechanism for finding “interesting” sets of nodes
- Useful for other things, too:
  - e.g. generating bookmarks

Step 1: Solving For Fluxes (Again)

- Note how the “TDL Exec” menu now has separate sub-menus for different kinds of solutions
- These are set up by SolveTree.define_solve_job()
- Each kind of solution can have its own set of solver options, tiling, etc.
- We now clear out the old solutions, and solve for fluxes anew
- ...over the entire 12 hours (just because we can!)

Step 2: Phases

- Fluxes are underestimated (5.3, 1.6 Jy) because phases are unaligned
- We now solve for phases, using the current flux solution
- We solve with a tile size of 15, while the phase parameters are subtiled with a size of 1. This makes the solution go faster (and more parallel)
- Observe residuals and G inspectors as we go along
- Observe map

Step 3: Fluxes, redux

- We now repeat the flux solution. This time, the tree will pick up the phase solutions obtained in the previous step.
- Note that at no stage is the *input* data corrected. We don't take the phases “out” of the data, we just put them into the predict model.
- This time we get higher flux solutions.
- Observe map, background is showing up, but there's clear artifacts around our two sources.

Step 4: Gains

- We can now do a solution for gain-amplitudes, using the current estimates for fluxes and phases.
- Observe G inspector.
- Observe map – the central source is gone, but there's something left at the position of the off-center source.
- Conjecture: the off-center gain varies differently from the on-center gain. Pointing errors?

Exercise 6: Differential Gains

Let's implement this m.e.:

$$\boldsymbol{V}_{pq}^{(\mathrm{predict})} = \boldsymbol{G}_p (\boldsymbol{K}_{p1} \boldsymbol{B}_1 \boldsymbol{K}_{q1}^{\dagger} + \boldsymbol{E}_p \boldsymbol{K}_{p2} \boldsymbol{B}_2 \boldsymbol{K}_{q2}^{\dagger} \boldsymbol{E}_q^{\dagger}) \boldsymbol{G}_q^{\dagger}$$

- Start with `ME3/example5-meow343.py`
- Add an E-Jones term for off-center (differential) gain
- Solve for G and E amplitudes simultaneously, or for E separately
- Observe the E inspector
- Try to get rid of 3C343 in the residuals

Goodbye 3C343...

- By rerunning our flux and G phase solutions, we can completely eliminate both sources
- If we want to really get to the noise, we should add the faint background sources to our model
- This is essentially the “major cycle”
- Sarod has a script for CLEANing an image, and converting the clean components into a model

Slightly More Exotic Calibration

- In principle anything in the tree can be a solvable parameter
- ...and can be attempted to be solved for (given enough data)
- Instead of calibrating individual Jones matrix elements, we can make them functions of something else, and calibrate that “something else”
- E.g., a Minimum Ionospheric Model

Calibrating The Ionosphere

- Let's try to calibrate for the simulated ionosphere we produced before
- We'll pretend we know the source fluxes and positions
  - this what the LOFAR GSM is for...
- We'll pretend we know nothing about the ionosphere, and model it by a flat blanket, with a polynomial TEC distribution
- We'll then solve for the coefficients

The Gory Details...

Ionospheric phase delay is $\boldsymbol{Z}(T)=e^{-i50\pi T\frac{c}{\nu}}$

For TEC, we'll use $T(x,y)=\sum_{k,l} x^k y^l$

and implement the following m.e.:

$$\boldsymbol{V}_{pq}=\sum_s \boldsymbol{Z}_{ps}\boldsymbol{V}_{pq}^{(s)}\boldsymbol{Z}_{qs}^{\dagger}$$

where the source visibility $\boldsymbol{V}_{pq}^{(s)}$ comes from the Meow model,

and $\boldsymbol{Z}_{ps}=\boldsymbol{Z}(T(x_{ps},y_{ps}))$,

x_{ps},y_{ps} being the piercing point from station p to source s.

First, Simulate...

- We'll use a different MS:
  - 100-103.1 MHz, 10 kHz channels
  - more LOFAResque, and easier to fit phase when it doesn't wrap channel to channel!
- We'll simulate TIDs in *x* and *y*
- Starting with zero amplitude at t=0, and gradually increasing
  - This is because we need “phase lock”, which least-squares (usually) struggles with

Running The Simulation

- You can just copy a pre-fabbed MS from here:
  cp -a (/net/birch)/data/oms/ Workshop2007/demo-lofar.MS .
  (**/apps/Timba/data** on jop0x)
- I'll demo the simulation step to show what ionosphere we're putting in

Running The Solution

- Load up `ME3/example6-iono-cal.py` (use “Load TDL script” or Ctrl+L)
- Set the following options:
  - Grid size: 1, step: 5'
  - Ionospheric model: mim_poly
  - Subtract sources in output
  - MIM options | Polc degree in X/Y: 2
  - MIM options | Polc degree in time: 1
  - MIM options | Base TEC value: 10
- Compile and run with a tile size of 2, watch the bookmarks

Comments On The Tree

- Note how the script is extremely similar to the 3C343 script
  - just a different Jones term
  - that's the power of frameworks
- Details of the MIM are hidden inside mims.py, we could in principle add other models there
- MIM parameters are found through node.search()

Ionospheres Are Hard...

- This is obviously a hard problem for a least-squares solver
  - and our model is not the best
  - although it fits better if we bump up the polynomial order
  - though not always...
- Other approaches needed...
  - orthogonal polynomials
  - non-parametric models, subspace decomposition?
  - Solvers based on Kalman-type filters