ME2: The Full-Sky Measurement Equation

Objectives:

- Extending the M.E. to a full sky
- Mapping this onto conventional implicit assumptions, and understanding their limitations
- Simulating multiple point sources with image-plane effects (e.g. primary beam)

svn up Workshop2007 please!

The Full-Sky ME

ME of a single point:

$$\boldsymbol{V}_{pq} = \boldsymbol{J}_p \boldsymbol{B} \boldsymbol{J}_q^{\dagger}$$

The sky has a brightness density: $\boldsymbol{B}(\vec{\sigma})$

(where $\vec{\sigma}$ is a unit direction vector)

So the total visibility is obtained by integrating over a sphere:

$$\boldsymbol{V}_{pq} = \int_{\text{sky}} \boldsymbol{J}_p(\vec{\sigma}) \boldsymbol{B}(\vec{\sigma}) \boldsymbol{J}_q^{\dagger}(\vec{\sigma})\, d\,\Omega$$

This is not very useful, so we project $\boldsymbol{B}$ onto the lm plane, tangential at the phase centre...

The Full-Sky ME

...and since $d\,\Omega = \frac{d\,l\,d\,m}{\sqrt{1-l^2-m^2}} = \frac{d\,l\,d\,m}{n}$,

in the lm plane we get:

$$\boldsymbol{V}_{pq} = \iint_{lm} \boldsymbol{J}_p(l,m) \frac{\boldsymbol{B}(l,m)}{n(l,m)} \boldsymbol{J}_q^{\dagger}(l,m)\, d\,l\,d\,m$$

Image-plane vs. *uv*-plane

$\boldsymbol{J}_p$ is composed of multiple effects: $\boldsymbol{J}_p = \boldsymbol{J}_{pn} \boldsymbol{J}_{pn-1} \cdots \boldsymbol{J}_{p1}$

($\boldsymbol{J}_{pn}$ is "in the receiver", $\boldsymbol{J}_{p1}$ is "in the sky".)

Some $\boldsymbol{J}$'s do not vary with l,m -- call them *uv-plane effects.*

e.g. receiver gain, leakage.

Some $\boldsymbol{J}$'s do vary with l,m -- call them *image-plane effects.*

e.g. $\boldsymbol{K}$, beam gain, ionosphere

Let's rewrite the $\boldsymbol{J}_p$ product as:

$$\boldsymbol{J}_p = \underbrace{\overbrace{\boldsymbol{J}_{pn} \cdots \boldsymbol{J}_{pk+1}}^{\text{uv-plane only}}}_{\boldsymbol{G}_p} \boldsymbol{K}_p \underbrace{\overbrace{\boldsymbol{J}_{pk-1} \cdots \boldsymbol{J}_{p1}}^{\text{uv- \& image-plane}}}_{\boldsymbol{E}_p(l,m)}$$

Or in other words, $\boldsymbol{J}_p(l,m) = \boldsymbol{G}_p \boldsymbol{K}_p(l,m) \boldsymbol{E}_p(l,m)$

(and depending on our particular M.E., $\boldsymbol{G}$ or $\boldsymbol{E}$ may be $\equiv \boldsymbol{1}$)

And Back To The ME....

$$\mathbf{V}_{pq}=\iint_{lm}\mathbf{J}_p(l,m)\frac{\mathbf{B}(l,m)}{n(l,m)}\mathbf{J}_q^\dagger(l,m)\,dl\,dm$$

then becomes:

$$\mathbf{V}_{pq}=\mathbf{G}_p\left(\iint_{lm}\mathbf{K}_p\mathbf{E}_p\frac{\mathbf{B}}{n}\mathbf{E}_q^\dagger\mathbf{K}_q^\dagger\,dl\,dm\right)\mathbf{G}_q^\dagger$$

(with everything under the $\iint$ being a function of l,m)

The Fourier Transform

and now expanding the $\mathbf{K}$ terms:

$$\mathbf{V}_{pq}=\mathbf{G}_p\left(\iint_{lm}\left(\mathbf{E}_p\frac{\mathbf{B}}{n}\mathbf{E}_q^\dagger\right)e^{-2\pi i(u_{pq}l+v_{pq}m+w_{pq}(n-1))}\,dl\,dm\right)\mathbf{G}_q^\dagger$$

for narrow fields $n\to 1$ (and for coplanar arrays $w=0$), so:

$$\mathbf{V}_{pq}=\mathbf{G}_p\left(\iint_{lm}\Big(\underbrace{\mathbf{E}_p\mathbf{B}\mathbf{E}_q^\dagger}_{\text{"apparent sky"}}\Big)\underbrace{e^{-2\pi i(u_{pq}l+v_{pq}m)}}_{\text{F.T. kernel}}\,dl\,dm\right)\mathbf{G}_q^\dagger$$

- The integral then becomes a 2D Fourier transform of the “apparent sky”.

The Fly In The Ointment

$$\mathbf{V}_{pq}=\mathbf{G}_p\left(\iint_{lm}\left(\mathbf{E}_p\frac{\mathbf{B}}{n}\mathbf{E}_q^\dagger\right)e^{-2\pi i(u_{pq}l+v_{pq}m+w_{pq}(n-1))}\,dl\,dm\right)\mathbf{G}_q^\dagger$$

- In the general case, the exponent is **not quite** an F.T. kernel.
- Let's collect the *n*-terms into an “***N***-Jones”, and define an “*apparent projected sky*”:

$$\mathbf{B}_{pq}=\mathbf{N}_p\mathbf{E}_p\mathbf{B}\mathbf{E}_q^\dagger\mathbf{N}_q^\dagger=\mathbf{N}_{pq}\mathbf{E}_p\mathbf{B}\mathbf{E}_q^\dagger$$

where $\mathbf{N}_p=\frac{1}{\sqrt{n}}e^{-2\pi i w_p(n-1)}$, $\mathbf{N}_{pq}=\mathbf{N}_p\mathbf{N}_q^\dagger$

(for narrow fields and/or coplanar arrays, $\mathbf{N}_p\to 1$)

Apparent Skies & Apparent Cohaerencies

We now have:

$$\mathbf{V}_{pq}=\mathbf{G}_p\left(\iint_{lm}\mathbf{B}_{pq}e^{-2\pi i(u_{pq}l+v_{pq}m)}\,dl\,dm\right)\mathbf{G}_q^\dagger=\mathbf{G}_p\mathbf{X}_{pq}\mathbf{G}_q^\dagger,$$

where $\mathbf{X}_{pq}=\mathcal{F}(\mathbf{B}_{pq})=\mathcal{F}(N_p\mathbf{E}_p\mathbf{B}\mathbf{E}_q^\dagger N_q^\dagger)$

- In other words, each antenna pair p-q measures an *apparent cohaerency distribution* $\mathbf{X}_{pq}(u,v)$ that corresponds to a 2D Fourier Transform of its own *apparent projected sky* $\mathbf{B}_{pq}$.
- ...at a single point in time!

Time Is ***Not*** On Our Side

- Cohaerencies are sampled along a "uv track" over some period of time:

$$V_{pq}(t) = G_p(t)\, X_{pq}(t, u(t), v(t))\, G_q^{\dagger}(t)$$

- The true sky B is probably constant(?) in time
- Image-plane effects (beam shapes, ionosphere) may vary in time.
- For wide fields, the N term is non-negligible. It varies with w which varies with time.
- All this is especially relevant with new telescope designs.

The "Classic" Assumptions

The full-sky ME: $V_{pq} = G_p X_{pq} G_q^{\dagger}$,
where $X_{pq} = \mathcal{F}(B_{pq})$, $B_{pq} = N_p E_p B E_q^{\dagger} N_p^{\dagger}$

If we assume that $B(t) \equiv B$, and $E_p(t) \equiv E_p \equiv E$, and $N_p \rightarrow 1$,
then all baselines will see the same, constant apparent sky:

$$B_{pq}(t) = E\, B\, E^{\dagger} \equiv \tilde{B}$$

and the array will sample one apparent cohaerency plane:

$$X_{pq}(t, u, v) \equiv X(u, v)$$

- Only under these assumptions is a *single* F.T. of the sky sufficient to simulate the entire observation!

Conclusions

- Under the "classic" assumptions, the visibilities measured by an array correspond to ONE cohaerency distribution X that is in an F.T. relationship with ONE apparent sky.
- In the presence of non-trivial image plane effects, or with wide fields and non-coplanar arrays, each interferometer *p-q* measures its "own" cohaerency $X_{pq}(t)$ corresponding to its "own" apparent sky $B_{pq}(t)$ -- variable in time!
- The K term becomes an F.T. kernel with narrow FOVs/coplanar arrays, but is "not quite" an F.T. otherwise.

Divide And Conquer

$$V_{pq} = G_p \left(\iint_{lm} K_p E_p \frac{B}{n} E_q^{\dagger} K_q^{\dagger}\, dl\, dm \right) G_q^{\dagger}$$

This is linear over B, so if the sky is a sum of sources:

$$B(l, m) = \sum_s B_s(l, m),$$

then

$$V_{pq} = G_p \left(\sum_s \iint_{lm} K_p E_p \frac{B_s}{n} E_q^{\dagger} K_q^{\dagger}\, dl\, dm \right) G_q^{\dagger}$$

And for some sources we can work out the integral exactly.

A Sky Of Point Sources

For a point source s of flux $\boldsymbol{B}_{0s} = \frac{1}{2}\begin{pmatrix} I_s + Q_s & U_s + iV_s \\ U_s - iV_s & I_s - Q_s \end{pmatrix}$

the $\boldsymbol{B}$ distribution is a delta-function:

$\boldsymbol{B}_s(\vec{\sigma}) = \boldsymbol{B}_{0s}\delta(\vec{\sigma} - \vec{\sigma}_s)$, or

$\boldsymbol{B}_s(l,m) = \boldsymbol{B}_{0s} n_s \delta(l - l_s, m - m_s)$ (do note the n)

So for a sky of point sources:

$$\boldsymbol{V}_{pq} = \boldsymbol{G}_p \left(\sum_s \boldsymbol{K}_{ps} \boldsymbol{E}_{ps} \boldsymbol{B}_{0s} \boldsymbol{E}^\dagger_{qs} \boldsymbol{K}^\dagger_{qs} \right) \boldsymbol{G}^\dagger_q,$$

where $\boldsymbol{K}_{ps} = \boldsymbol{K}_p(l_s, m_s)$, and $\boldsymbol{E}_{ps} = \boldsymbol{E}_p(l_s, m_s)$.

Let's Build a Tree

Assuming a perfect instrument again:

$$\boldsymbol{V}_{pq} = \sum_s \boldsymbol{K}_{ps} \boldsymbol{B}_{0s} \boldsymbol{K}^\dagger_{qs}$$

- See `ME2/demo1-predict-nps.py`
- We can already do one point source, adding more is just some **for** loops...
- ...and a **Meq.Add** node to sum the visibilities.
- We'll put the sources on a grid.
- Run the tree and make an MFS map.

Exercise 1: Complex Gains

Let's add some gain terms:

$$\boldsymbol{V}_{pq} = \boldsymbol{G}_p \left(\sum_s \boldsymbol{K}_{ps} \boldsymbol{B}_{0s} \boldsymbol{K}^\dagger_{qs} \right) \boldsymbol{G}^\dagger_q$$

- Use `ME2/demo1-predict-nps.py` as a starting point.
- Add the gain terms from `ME1/demo3-predict-ps-gain.py`.
- Make an MFS map.

Exercise 2: Adding Beams

Let's add a primary beam:

$$\boldsymbol{V}_{pq} = \boldsymbol{G}_p \left(\sum_s \boldsymbol{E}_{ps} \boldsymbol{K}_{ps} \boldsymbol{B}_{0s} \boldsymbol{K}^\dagger_{qs} \boldsymbol{E}^\dagger_{qs} \right) \boldsymbol{G}^\dagger_q$$

use $\boldsymbol{E}(l,m) = \cos^3(2 \cdot 10^{-6} \nu \sqrt{l^2 + m^2})$

(i.e. same for all antennas)

- Use the previous script as a starting point.
  - reset Gs to 1
- Make a per-channel map.

Code Reuse?

- By now our scripts are getting rather complex.
- On the other hand, we're reusing the same building blocks, e.g.:
  - point sources
  - Jones matrices
- Good programming strives for maximum code reuse; good languages simplify this via modules, libraries, objects, etc.
- TDL is Python, and Python is an excellent programming language.

Frameworks!

- A tree is like assembly language – the nuts'n'bolts view of what's going on.
- Pure TDL is like C – a higher level language, but still very close to what the “tree machine” is doing.
- An OO framework provides abstraction, so you talk in terms of your “domain language”:
  - I have an interferometer array of *N* antennas
  - make me a point source here
  - make me the nodes to compute visibilities at each baseline
  - apply this Jones matrix and give me the corrupted visibilities

OOP

6. Object-Oriented Programming:
[0] wasn't that in FORTRAN77?
[8] heard of it
[10] have used OOP concepts in my programs
[2] can't imagine writing a non-trivial program without it
[1] I use multiple inheritance to design my breakfast

Meow

- (**M**easurement **E**quation **O**bject Frame**w**ork)
- See `ME2/example2-nps-meow.py`.
- This is the equivalent of demo1.
- Highlights:
  - we deal in “sky components”, “arrays”, “observations”
  - details of sources are hidden, it just gives us the visibilities
- also provides convenient utilities for
  - GUI options
  - I/O records
  - imaging, bookmarks, etc.

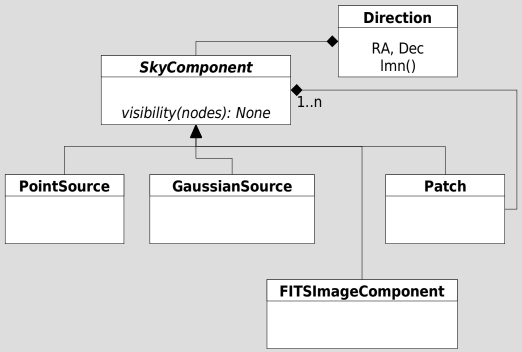
Meow With Jones

- Now let's add **E** and **G** terms
- See `ME2/example3-nps-corrupt-meow.py.`
- We make CorruptComponents from components by adding a Jones corruption term.
- Corrupt Components are also sky components, so they can be treated the same way.
- Modularity:
  - sky models defined in one place...
  - Jones terms defined in another place...
  - main sim script just puts them together

...Meooooooow

- Extended sources?
- See `ME2/example4-nps-ext-meow.py.`
- Run & make per-channel map, observe frequency behaviour.
- Small change here: we use GaussianSource in place of some PointSources.
- Don't need to know the details of a Gaussian implementation, since we can just get the visibility nodes from the source.

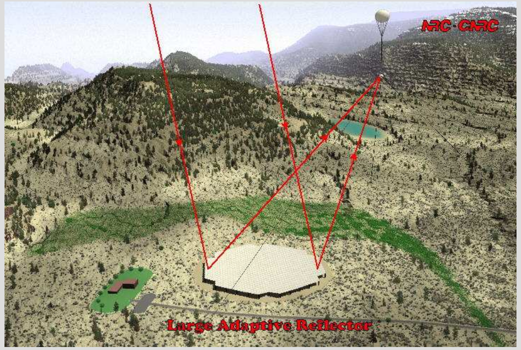
Meow Inheritance...



“Well I'm liberal, but to a degree...”
-- Bob Dylan

- Meow is just “a” framework, we can have others
- We believe in creative pragmatic anarchy...
- ...so don't take any framework as gospel.
  - especially as it's a work in progress
  - your feedback will drive that progress!
- Use them, or make your own, but be pragmatic:
  - if you feel if something is missing, think of how to improve it, and talk to the author
  - ideally, we want to fold your improvements back into the mainstream, for the benefit of others.

A “Real”-Life Example: The CLAR



A Simple CLAR Sim

- The CLAR primary beam is elevation-dependent
  - symmetric at zenith
  - broadened vertically as we track towards the horizon
- Let's simulate 10 point sources:

$$\boldsymbol{V}_{pq}=\sum_s \boldsymbol{K}_{ps}\boldsymbol{E}_p(l_s,m_s)\boldsymbol{B}_{0s}\boldsymbol{E}_q^\dagger(l_s,m_s)\boldsymbol{K}_{qs}^\dagger$$

$$\boldsymbol{E}_p(l,m)=\boldsymbol{E}_{CLAR}(l,m;El) \qquad (El=\text{elevation})$$

A Simple CLAR Sim

- See `ME2/example5-clar.py`
- This defines a per-station, per-source **E** Jones term.
- E Jones details are in `clar_model.py`.
- We use pre-computed beam gains:
  - The vgain parameter is a **Meq.Parm** node
  - There's a ParmTable (*.mep) supplying vgain values (as a function of time)
  - These are precomputed by another script (`clar_fit_dq.py`)
  - ...but in principle could have come from anywhere.

A Simple CLAR Sim, continued

- Note also the “source model” option in the GUI.
- This selects a function, which the script then calls to obtain a source model.
- This a “compile-time” option
  - determines the kind of tree that is built
  - ...as opposed to run-time options, which determine what kind of request to give the tree.
- Python makes this sort of thing easy, and it gives us a further degree of abstraction.

An Ionospheric Sim

- See ME2/example6-iono.py
- This is adaptation of our previous ionospheres:
  - multiple sources laid out in a grid (size and grid step configurable)
  - we compute proper piercing points per source and per station
  - code to compute ***Z***-Jones resides in iono_model.py and iono_geometry.py
  - this returns the Z nodes as a series, individual matrices are Z(src.name,p)

$$\boldsymbol{V}_{pq} = \sum_s \boldsymbol{Z}_{ps} \boldsymbol{K}_{ps} \boldsymbol{B}_s \boldsymbol{K}_{qs}^{\dagger} \boldsymbol{Z}_{qs}^{\dagger}$$

First, A Different MS...

- We'll use a different MS:
  - 30-190 MHz in steps of 5 MHz
  - more LOFAResque
- Make demo-30-190.MS by running:
  `glish -l demo_sim_30-190.g`

An Ionospheric Sim, continued

- Set compile-time options as follows:
  - Rotate ionosphere with sky: True
  - TID X amplitudes: 0.01 at t=0 and t=1hr
  - Size 50km, speed 200 km/h
  - TID Y amplitudes: 0
  - Grid size 3, grid step 5'
  - Noise: 0 Jy
- Run tree and make a per-channel map
- Make a time-slice movie:
  `glish -l make_movie.g DATA ms=demo.MS channel=32 npix=300 cell=3arcsec`
  (or whatever output column you used)

And Now For Something Completely Different:
Time & Bandwidth Smearing

Way back, we assumed: $\langle \boldsymbol{J}_p (\vec{e}\,\vec{e}^{\dagger}) \boldsymbol{J}_q^{\dagger} \rangle = \boldsymbol{J}_p \langle \vec{e}\,\vec{e}^{\dagger} \rangle \boldsymbol{J}_q^{\dagger}$

In effect, we've been computing $\boldsymbol{V}_{pq}(t_0, \nu_0)$, and assuming that this close enough to the *vector average* over $\Delta t, \Delta \nu$.

This is OK as long as $\boldsymbol{J}_p$s are sufficiently constant over $\Delta t, \Delta \nu$.

But as a minumum, $\boldsymbol{J}_p$ contains $\boldsymbol{K}_p$, and :

$$\boldsymbol{K}_p \boldsymbol{K}_q^{\dagger} = \exp\left(\frac{2\pi i \nu}{c}(u\,l + v\,m + w(n-1))\right)$$

(uvw's change with t, faster for longer baselines)

So even in the absense of any additional effects,

$$\langle \boldsymbol{K}_p \boldsymbol{B} \boldsymbol{K}_q^{\dagger} \rangle_{\Delta t, \Delta \nu} \neq \boldsymbol{K}_p(t_0, \nu_0) \boldsymbol{B} \boldsymbol{K}_q^{\dagger}(t_0, \nu_0)$$

This is usually known as time and bandwidth smearing.

The effect goes up with $\Delta t, \Delta \nu, l, m$, and baseline length.

Simulating Smearing

- The same effect occurs with other Jones terms, such as ionospheric or tropospheric phase, etc. (hence, *decohaerency time*).
- How to simulate?
- The brute force approach:
  - Divide each $\Delta t, \Delta \nu$ into $N \times M$ sub-intervals, and use

$$\langle \boldsymbol{V}_{pq} \rangle_{\Delta t, \Delta \nu} \approx \frac{1}{NM} \sum_{i,j} \boldsymbol{V}_{pq}(t_i, \nu_j)$$

- See `ME2/example7-smear.py` (and compare to example2...)
  - **Meq.ModRes** changes resolution
  - **Meq.Resampler** averages back

So What's The Difference?

- Run the tree and make an per-channel map
  - 5x5=25 times more visibilities to compute, so it takes longer...
- Hard to see all that much in the map (although you could make another map without smearing, and subtract it...)
- So let's build a *differential tree* instead, to compute

$$\boldsymbol{\Delta V}_{pq} = \boldsymbol{V}_{pq}(t_0, \nu_0) - \langle \boldsymbol{V}_{pq} \rangle_{\Delta t, \Delta \nu}$$

...and write the delta-visibilities to the MS.

Differential Trees
(Or Simulations About Simulations)

- Given infinite CPUs, we can implement m.e.'s of arbitrary precision.
- In real life, we have to take shortcuts (e.g. choosing time/freq intervals here).
- The main question: how much error does a particular shortcut introduce?
  - given infinite mathematical skill, we could work it out analytically...
  - ...but given MeqTrees, we don't have to.

Interlude:
How To Make Your Tree Run Very Slow

- There's a naïve way to compute the deltas:
  - subtract “predict” from “resampled”, and connect that to the sink.
- Why is this so slow??
  - each “predict:p:q” subtree is called twice, once at low res, once at high res.
  - ...so we're not using the node caches.
- The right way to do it: parallel trees
- See `ME2/example8-smear-diff.py` (run the tree and make a per-channel map)
- Moral: reuse *values*, not nodes.
  - normally, this only occurs with resampling

The Lowly Point Source
as a probe of the simulations universe

- For single point sources, we can implement a very precise form of the ME.
- For large-scale simulations, we're forced to implement an approximate m.e.
- We can cheaply predict a grid of point sources:
  - with a precise m.e.
  - with an approximate m.e.
- The difference tells you the error you make when using the a.m.e.

A CLAR Shortcut?

- Do we need per-station beams?
  - the beam depends on elevation
  - all antennas track the same point on the sky
  - ...so will have slightly different elevations
  - ...very slightly (max separation is ~30km)
- Can't we just use an average beam?

i.e. $\boldsymbol{E}\boldsymbol{B}\boldsymbol{E}^{\dagger}$ instead of $\boldsymbol{E}_p\boldsymbol{B}\boldsymbol{E}_q^{\dagger}$, where $\boldsymbol{E}=\frac{1}{N}\sum_p \boldsymbol{E}_p$

- Let's make a tree to compute the delta-visibilities between the "precise" CLAR sim with per-station beams, and an "approximate" sim with an averaged beam.

A CLAR Shortcut, implementation

- See `example9-clar-shortcut.py`.
- Here we put sources in a "star8" pattern.
- Since we don't have pre-computed beams for the test pattern, we use another function to compute beams:

Ej = clar_model.EJones(ns,array,observation,source_list);

- We then use a **Meq.Mean** to compute the average beam (**Eavg**) per source, across all stations.
- We make a separate patch containing sources corrupted by the average beam Eavg, and write out the differences.
- Run the script and make a per-channel map.

Exercise 3: Ionospheric Phase Diffs

- The iono demo was all good, but it would be nice to see if there's any *differential* movement.
- Start with ME2/example6-iono.py
- Make a tree to compute the following modified m.e., and make images and time-slice movies:

$$\Delta\boldsymbol{V}_{pq}=\sum_s \Delta\boldsymbol{Z}_{ps}\boldsymbol{K}_{ps}\boldsymbol{B}_s\boldsymbol{K}_{qs}^{\dagger}\Delta\boldsymbol{Z}_{qs}^{\dagger}$$

$$\Delta\boldsymbol{Z}_{ps}=\boldsymbol{Z}_{ps}/\boldsymbol{Z}_{p0}$$

(i.e difference w.r.t $\boldsymbol{Z}$ of central source)

Tracking Errors

- Let's make a tree to simulate tracking errors:

Assume each antenna has the same beam pattern $E(l,m)$, but a different pointing error of $\Delta l_p, \Delta m_p$.
For source s at position l_s, m_s, the beam gain $\boldsymbol{E}_{ps}$ is then:

$$\boldsymbol{E}_{ps} = E(l_s + \Delta l_p, m_s + \Delta m_p)$$

- Let's make a tree to simulate tracking errors:
- See ME2/example10-tracking.py

Tracking Errors, continued

- We generate a random set of tracking offsets per each antenna
  - slowly variable in time
- This gives us "apparent" *l'*,*m'* coordinates per source, per antenna:
  ns.lm1(src.name,p))
- We then use *l'*,*m'* to compute the beam gain per source, per antenna.

Exercise 4: Differential Tracking Errors

- It's hard to see anything meaningful in the previous images.
- ...so let's make a differential tree to examine the errors closely.
- Start with ME2/example10-tracking.py
- Make a differential tree and examine the difference between a sim with tracking errors, and a sim with perfect tracking.

Exercise 5: Patches & Beams

- Use the pseudo-WSRT beam model of Exercise 2.
- Create three patches
  - at l_0,m_0 = 0,0; 2',2' and 4',4'
  - each patch to contain 9 sources of 1Jy each arranged in a cross, at 0,0; ±.5' and ±1' in each direction, relative to the center of the patch.
- Apply $\boldsymbol{E}_p(l_0,m_0)$ to each patch as a whole.
- Make a differential tree to compute the delta-Vs between this approximation, and a "precise" model where each source has its own $\boldsymbol{E}_p$.
- Make MFS and per-channel maps.
- You should be able to do it within 35000 nodes.