Hello World

Today's Menu

- Introductions
- What's a MeqTree and what's a Timba?
- Building simple trees
- Exploring the system
- Working with Measurement Sets
- Generally messing about
- Dinner!

NOT On Today's Menu

- Measurement Equations
- Simulation or calibration
- The Meaning Of Life

System Issues? What System Issues?

5. Linux/Unix...
   [1] heard of it, use Windows
   [17] is my primary working environment
   [3] I'm a [former] sysadmin
   [0] I'm Linus Torvalds

- If you have any problems, just ask any of the 3 guys up there for help...

System Issues

```
$ slogin -Y
   username@host
...
$ xclock  # just to test X
$ cd Workshop2007
$ svn up
$ cd Intro1
$ meqbrowser.py
```

Your host & username is:
(note temporary pairings for first 2days)

birch: oosterlo/boomsma kemper/butcher rnijboer/hamaker bemmel/jmiller
lofar9: omar/strom brentjens/vogt
lofar10: bernardi/pandey panos/thomas
cedar: vjelic/mohan kieviet/usov ger/saleem
jop01 (10.87.10.1): reynolds/sundaram
jop03 (10.87.10.3): loose/zwieten

Please change password both on your host and on lofar9!
(“`$ passwd ; ssh lofar9 passwd`”)

General Structure

9:00 ~ 10:30	**session 1**
10:30 ~ 11:00	coffee
11:00 ~ 12:30	**session 2**
12:30 ~ 13:30	lunch
13:30 ~ 15:30	**session 3**
15:30 ~ 16:00	coffee
16:00 ~ 17:30	**session 4**
17:30 ~ 9:30	beer & homework

Special Events, Week 1

- **Tuesday**
  Workshop dinner in Dwingeloo
- **Wednesday** 20:00
  Bridge, world politics & cross-cultural alcohol tasting
  - no prior skills required
- **Friday** 17:00
  Indoor football
  - prior skills not encouraged
  - indoor (non-marking) shoes required

General Structure II

- During each session, presentations will be interspersed with demos, which you run via your laptops...
  - PLEASE only **ONE** running demo per team (use either login)
- ...and occasional exercises, which teams do on their own
- Please interrupt with questions at any time
  - no questions implies a lack of understanding, therefore the material will be repeated in an infinite loop until you DO ask questions.

What's MeqTr

- “...a flexible sy
  arbitrary M
  to sol
- ge...”
  d of the tunnel...”
- verts

RESISTANCE IS FUTILE!
RESISTANCE IS FUTILE!
RESISTANCE IS FUTILE!

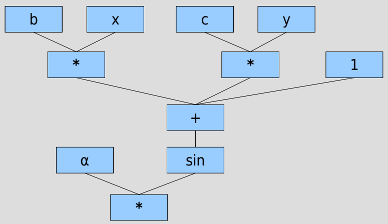
MathLab For Perverts

- In a nutshell, you build **MeqTrees** to evaluate mathematical expressions.
- Today, we're going to keep it simple and play with basic things.
- The things we do today you could probably do much faster in something like MathLab.
- So lots of what you see will seem to be an overly elaborate way to achieve rather simple results.
- ...but be patient, by tomorrow these will turn into complex radioastronomical simulations.

A Basic Tree

- Any mathematical expression can be represented by a tree.

$$f = \alpha * \sin(b * x + c * y + 1)$$



The World Of Nodes

- A single tree element is a ***node***.
- A ***parent*** node operates on its ***child*** nodes.
- A node belongs to a ***class.*** This determines the kind of operation it performs
  - e.g.: **Add**, **Multiply**, **Sin**.
- A ***leaf*** node has no children (“top” of tree)
- A ***root*** node has no parents (“bottom” of tree)
- A ***subtree*** is any complete part of a tree, rooted at a specific node.
  - a subtree evaluates some compound function of its leaf nodes.
- A ***forest*** has many (1+) trees.

Our First Tree...

cd Workshop2007/Intro1

- Run **meqbrowser.py**, press “OK” to “connect to a kernel”.
- From the menu, choose “TDL | Load & compile TDL script” (or just press Ctrl+T).
- Select **demo1-first-tree.py**
- Observe....

MeqBrowser and MeqKernel

- The GUI you're looking at is called the meqbrowser
- There's also something called a “meqkernel”
  - which runs in a separate process and talks to the browser
- Trees are built inside the kernel...
  - ...according to commands from a **TDL** script, which is loaded by the browser.
  - a tree in the kernel is like an “evaluation machine”
- The browser provides a graphical representation of the trees in the kernel.

TDL

- **T**ree **D**efinition **L**anguage
- TDL is Python
- ...with some “magic” objects for **declaring** trees.
- ns.xxx ***declares*** a node named “xxx”
- all nodes must have unique names
  - this is not the same thing as their ***class***!
- (ns.xxx << ...) ***binds*** the named node to a definition of what that node is and does.
- TDL provides shortcuts for implicit naming and binding of intermediate nodes.

Running The Tree

- From the “TDL Exec” drop-down, choose “test forest”
- Observe...
- From the “Bookmarks” menu, select “result of 'f'”
- Observe...
- A long way to go for a simple answer...

A More Interesting Tree

- Load demo2-improved-tree.py.
- Execute “test forest”
- Look at Bookmarks | Result of 'f'.
- And Bookmarks | Result of 'f1'
- Observe the difference...
- The crucial difference are the **Meq.Time** and **Meq.Freq** nodes.

Dealing In Functions

- MeqTrees are designed for evaluating *functions*, not just single scalars.

Start with the following expression:

$$f=\alpha\sin(x cos(2y))$$

If x and y are functions of time t and frequency ν, then

$$f(t,\nu)=\alpha\sin(x(t,\nu)\cos(y(t,\nu)))$$

In our tree, we used $x(t,\nu)=t$ and $y(t,\nu)=\nu$,

thus ending up with

$$f(t,\nu)=\alpha\sin(t*\cos\nu)$$

Functions Everywhere

- Most concepts we deal with are a function of something.
  - the visibility observed by an interferometer is a function of frequency and time.
  - images are functions of *l,m* (or RA/Dec, ...)
  - the Fourier Transform turns a function of *l,m* into a function of *u,v*.
- Numerically, we represent a function by a set of values on some grid

e.g. for a set of $\{t_1 \ldots t_n\}$ and $\{\nu_1 \ldots \nu_m\}$, we represent f as an $n \times m$ array $\{f_{ij} = f(t_i, \nu_j)\}$

Grids

- Numerical code is often filled with **for** statements iterating over grids...
- With MeqTrees, you instead build up a compound function (*f*) from its constituent parts...
- ...then you give it a grid, and get back the values of the function on that grid.
- This happens at every level of the tree – every subtree can be considered to represent some function.

Supplying a Grid

- Look at **_test_forest**
- First we make a ***domain*** object.
  - here we use t and ν from 1 to 10
- Then, we make a ***cells*** object. This represents our grid.
  - in this case, we specify a regular 100x100 grid over the given domain
- Then, we put the cells into a ***request***.
- We ***execute*** the root node with the given request.
- Try changing the grid...

Exercise 1: Basic Trees

$$r = \sqrt{(t^2 + \nu^2)}$$

$$C(t,\nu) = \cos(r) e^{-\frac{|r|}{30}}$$

$$f(t,\nu) = C(t,\nu) + |C(t,\nu)|$$

- Implement tree for *f* above (use demo2 as starting point)
- Evaluate over a 100x100 grid, with time/freq from [-30,30]
- Plot results & cross-sections

A Node's Life

- A node just sits there, until its parent gives it a request
  - in the case of a root node, the request is supplied from somewhere else, via execute()
- If it has children, it sends the request up to its children (by calling *their* execute())
- A leaf node (Freq, Time, Constant) can *evaluate* a request directly and return a ***result***.
- Once a parent has collected results from its children, it performs some operation on them – according to its class -- and returns a result of its own.

Botany For Beginners

- Each node has a *state record*. You can see it by clicking on a node. This tells you way more than you wanted to know about that node.
- The **request** field contains the most recently executed request.
- Note that just about data object in the system is a record
  - or at least can be viewed as a record
- **request.cells** contains the grid of the request.
  - also the domain, cell sizes, and other stuff...

The Result Cache

cache.result contains the last result returned by the node.
Important for two reasons:
lets you see what's been going on in the tree;
speeds things up, because often intermediate results can be reused.
- In real life, a tree deals in millions of values, so caching everything would use too much memory;
  - nodes are smart enough to cache only those results that are actually reused.
- For small demos, having everything cached is instructive, so we change the cache policy...

Dissecting a Result

- A **Result** object represents a real or complex-valued function on an *N*-dimensional real grid.
- It contains a copy of the **Cells**, giving the grid.
- The function value is found inside a **VellSet:**
  - as **result.vellsets[0].value**
  - the value is a **Vells** object -- a glorified array
  - VellSets can also contain optional flags and derivatives, but we won't be meeting them soon.
- A Result with one VellSet ([0]) represents a scalar function, but later on we'll meet vector and tensor functions.

e.g. $\vec{f}(t,\nu)=(f_1(t,\nu),f_2(t,\nu),f_3(t,\nu))$

The Priciple of Informational Greediness

- SKA source subtraction requires 47 million laptops (J. Bregman):
  - SKA a contributor to global warming?
- Redundant computations and redundant information should therefore be avoided as much as possible
- MeqTrees does a lot of this for you
  - environmentally-minded people use MeqTrees

P.I.G. In Essense

- A **Vells** may contain only as many axes as actually needed to represent the function.
- E.g. for an NxM grid, a node may return a result with an NxM Vells, or an Nx1 Vells, or an 1xM Vells, or even a 1x1 Vells
  - Nx1 is the same thing as N
  - but 1xM is not the same thing as M – the order of the axes is *fixed*.
- We refer to the missing axes as ***collapsed***. A collapsed axis simply means that the function is NOT variable over that dimension, i.e. not variable in freq or time or whatever.

P.I.G. In Action

- The node knows its function best...
  - some nodes return constant values
  - some nodes return things variable in time only
  - some nodes return things variable in frequency
- When you combine an Nx1 (i.e. time-variable only) value with an 1xM (freq-variable only) value in, e.g., a **Multiply** node, the result is NxM.
- The tree does the “right” thing regardless.
- So in fact you don't need to worry about this at all...
  - unless you're Tony;
  - unless you're optimizing for calculations.

TDL Is Python

4. Python...
[1] never heard of it
[14] vaguely familiar
[5] I write Python scripts regularly
[1] I develop Python packages
[0] I'm Guido van Rossum

Node Names & Qualifiers

Let's make a Fourier series:

$$f(x,y)=\sum_{k=-n}^{n}\sum_{l=-n}^{n} f_{kl}(x,y)=\sum_{k=-n}^{n}\sum_{l=-n}^{n} e^{-2\pi i(kx+ly)}$$

- Load `Intro1/demo3-quals.py`
- This makes a tree to sum the series above.
  - note how we create nodes "**f:***k***:***l*" to represent f_{kl}
  - note how *n* became a GUI option
  - We use Python list comprehension and the *-syntax to specify a large number of children with one compact statement.

Exercise 2: A Fourier Series

Let's add an extra term:

$$f(x,y)=\sum_{k=-n}^{n}\sum_{l=-n}^{n} e^{i(ak+bl)} e^{-2\pi i(ix+jy)}$$

- Start with `Intro1/demo3-quals.py`
- Modify it to compute the series above.
- Make ***a*** and ***b*** GUI options, with possible values of, e.g., [0, 5, 10].
- What have we demonstrated?

Passing Nodes Around

Let's make a slightly different series:

$$f(x,y)=\sum_{k=-n}^{n}\sum_{l=-n}^{n} f_{kl}^{2}(x,y)=\sum_{k=-n}^{n}\sum_{l=-n}^{n} (e^{-2\pi i(kx+ly)})^{2}$$

- Load `Intro1/demo4-more_quals.py`
- This has functions sum_series() and sum_sq_series() to sum a series and the square of a series. Check both bookmarks.
- Note how "collections" of nodes, and "undefined" nodes, may be passed around.

A Menagerie Of Nodes

- At the moment, there's ~100 node classes available in the system.
- See wiki:
  http://lofar9.astron.nl/meqwiki
  for not-quite-complete documentation.
- Most current needs are catered for, but it is always possible to add new ones (yourself, too)
- We're now going to go in random directions with various demos and exercises, so I'll try to introduce the new nodes as they come up.

Building An Ionosphere

- Let's make a tree to model an ionosphere.
- Assume a “flat world” model, and a two-dimensional ionospheric “blanket”, i.e. Total Electron Content is then just TEC(*x*,*y*,*t*).
- A number of sine waves to model Travelling Ionospheric Distrubances (TIDs).

Ionosphere 1

- Load up `Intro1/example5_iono.py`
- Run “test forest” and look at bookmarks.
  - this gives us TECs at two *x*,*y* points
- Lots to be learned from this script, so take it home to study:
  - passing nodes in and out of functions
  - using qualifiers (“rate”) on top of existing nodes to avoid naming conflicts
  - making vectors, using matrix multiplication
  - ...and generally making your code assumption-free
  - Hence, **example5** and not **demo5**.

Ionosphere 2

- Let's get a more elaborate picture...
- Load up `Intro1/example6-iono2.py`
- Run “test forest” and look at “tec:2”
- More to be learned from this script:
  - we import the TID function from demo5
  - the “xy” nodes can be functions of anything, our tree doesn't care
  - so we can form up a request in *t*,*l*,*m*
- Bonus question: why is “tec:2” only two-dimensional?

Ionosphere 3

- OK, let's make a truly 3D picture...
- Load up `Intro1/example7-iono3.py`
  - this adds a second sine wave moving in a different direction
  - note the use of another qualifier
- Run “test forest” and look at “tec:2”
- If you're lucky, you can access the “Warped surface” and “3D display” options via right-click.

Exercise 3:
Ionospheric Phase

- Start with `Intro1/example7-iono3.py`
- Add nodes to compute ionospheric phase delays (see below)
- Add a frequency axis of 30-200MHz to the domain.
  - use 30 points per each axis, or you **WILL** run us out of memory!!!
- Display the "phase screen" in *l,m,* as a function of frequency.
- Ponder on the joys of LOFAR calibration

Ionospheric phase delay is $\zeta = 2\pi \cdot 25 \cdot \frac{c}{\nu} \cdot \mathrm{TEC}$