# MeqTrees Beginners Guide

How To Find Your Way Among The Trees

J.E. Noordam
noordam@astron.nl

March 5, 2009

# Contents

# Chapter 1

# Introduction

If all else fails, consult the manual *(Ancient wisdom)*

First of all, a definition: *Meqtrees is designed for implementing an arbitrary Measurement Equation, and to solve for (arbitrary subsets of) its parameters.* A Measurement Equation (M.E.) is a mathematical model of an instrument and the thing it observes. It is used to predict the values of measured data, which may be used to produce simulated data-sets. In addition, the differences between measured and predicted values can be used to solve for M.E. parameters, which is closely related to calibration. MeqTrees can be used for any problem that has an M.E., but it was developed originally for radio astronomy. For a general overview of MeqTrees, and is place in the Grand Scheme of things, see [2].

To keep up the pace a bit, this Beginners Guide starts with a discussion of its graphical user interface, the MeqBrowser (section 1.1). After that we discuss TDL scripts (Python modules), the only things you will ever use in a MeqTrees project. They contain your instructions to build complicated trees, which ultimately boil down to a list of MeqNode specifications in Python. These are then dispatched to the MeqTrees back-end, the meqserver: there the trees are decoded and turned into instructions to be executed by the meqserver (written in C++). It is there that all the heavy numerical computation takes place. Chapter 3 contains a rather minimal description of Forests, Trees, Nodes, Requests and Results, and is followed by an introduction to the all-important Tree Definition Language (TDL, see also the TDL Bedside Companion [3]).

The building and execution of trees is discussed by means of a series of examples: myFirstTree, mySecondTree, myThirdTree, solving_11 etc. Just to whet your appetite for things to come, chapter 8 treats a variety of more advanced topics like user-defined nodes, matrix operations, avoiding node-name clashes, and the use of domains with other than the 2 default dimensions (freq and time). In the same spirit, chapter 9 sketches the treescape beyond this Beginners Manual, when you are no longer a Beginner. In many ways, these two chapters are what MeqTrees is all about.


Figure 1.1: *When you start the MeqBrowser, a popup appears that allows you to attach the browser to a meqserver. Just press the start button.*

In order to encourage script readability, a highly modular approach is used in the examples. Subtrees are defined in separate TDL scripts, which also contain the definition of their customizing parameters (TDLOptions). In the same way, there are separate modules for tree execution, and for defining useful things like bookmarks.

This Beginners Guide can be devoured by itself, i.e. without a computer handy. The Python code of many of the TDL scripts is reproduced verbatim, and there are screenshots of the MeqBrowser GUI etc. Since we take you quite seriously, and trust that you will not be a Beginner for long, the screenshots contain some detail that is not explained in the text. You can test your general understanding by conjecturing about it. A grimy, well-used look of your BG will tell us something about you. In that sense, its role is similar to the towel of a galactic hitch-hiker [4].

## 1.1 Using the MeqBrowser

MeqTrees has a rather nifty graphical user interface called the MeqBrowser. It is the brainchild of Oleg Smirnov, like so many other things. Please refer to the

large screenshot in fig 1.2. If you have a computer handy, we suggest that you start it up:

```
$ cd /usr/share/meqtrees/examples/beginners_guide
$ meqbrowser&
```

This launches an empty MeqBrowser window, and a popup appears that allows you to attach the MeqBrowser to a *meqserver* (see fig 1.1). The latter deals with the C++ side of things: It generates the C++ versions of the MeqNodes that you have defined in Python with TDL, it executes the resulting trees, and allows the MeqBrowser to interact with the node state records. Multiple meqservers can be active simultaneously, and a meqserver does not have to be associated with a MeqBrowser (i.e. trees can be executed in batch mode). A discussion of the meqserver is outside the scope of this manual. Just press the *start* button. The popup will disappear, and the line *forest is empty* will appear in the MeqBrowser. Select the TDL menu at the top, and load any file starting with *ROOT_*, i.e. a TDL script that has a *_define_forest(ns)* function (see section 2.5).

The MeqBrowser has three main sections, and a choice of menus and other buttons. In fig 1.2, the contents of the file *ROOT_mySecondTree.py* has been loaded, using the *TDL* menu at the top. Its TDL (=Python) code may inspected *and edited* in the in middle (Tabbed Tools) section. This section also has tabs for *Messages* and *Snapshots*, which are very useful for debugging.

After compilation (i.e. the generation of the explicit MeqTrees nodes), the resulting tree is displayed in the left (Trees) section, where it may be browsed in detail. Clicking on a node displays its internal state in various ways in the right (Gridded Viewers) section, where it may then be inspected interactively. Clicking on the Bookmark menu item *mySecondTree()* creates a page of views for a particular group of nodes.

The tree is executed by issuing a Request to a named node (here called 'rootnode'). The execution is done by means of a *_tdl_job()* function in the same TDL script that contains the *_define_forest()* function. It may be called via the *TDL Exec* button. Upon execution, the various display panels on the right will come alive, showing the node Results (NB: if they do not come alive, click on the bookmark again). Shown in fig 1.2 are three views of the same node: Its cache result (top left), its state record (bottom left) and its quickref_help text (top right). The Result (colored panel) consists of a 2D array of complex values, for a 2D domain in frequency (horizontal) and time (vertical). The real an imaginary parts are plotted side-by-side. The mean (m) and stddev (sd) are inticated at the top. Left-clicking on the colored panel produces horizontal and vertical cross-sections through that point. Right-clicking produces a menu of options for inspecting the panel in more detail.

Also shown are the two popup windows: The left one is for specifying compile-time options that customize the tree, and the right one for speciying runtime options, and for executing the tree by means of one of the *_tdl_job_()* functions

defined in the TDL script. The selected values of TDL options are stored in a file, which greatly increases the ease of using MeqTrees in practice.

Note that a TDL script that contains _define_forest() and _tdl_job_() functions is entirely stand-alone, i.e. the MeqBrowser just executes its contents. Therefore, such a TDL script may also be executed in batch mode, without the MeqBrowser.

Obviously, we have described only the basic functionality of the MeqBrowser, which has many other features that make it easy (and fun!) to execute trees and to inspect node Results. For instance, the Purr button activates a rather useful tool for organizing the intermediate files of a data reduction project. Debugging functionality (stop, step, resume etc) is available along the left edge. The Debug menu also has a profiler, which collects processing statistics for all nodes, either individually or by class. Progress messages are displayed along the botton, and any errors may be inspected in various ways (see section 3.4).

Enjoy.

Figure 1.2: *The MeqBrowser GUI, with the TDL script* ROOT_mySecondTree.py *loaded, compiled and executed. A concise description of the MeqBrowser is in section 1.1.*

# Chapter 2

# TDL scripts

The Tree Definition Language (TDL) is just Python code. See chapter 4. All the specifications for building and executing (forests of) trees are stored in Python modules, which are files with extension *.py*. We call them TDL scripts. Most of them contain functions or classes that generate (subtrees of) MeqNodes with the help of a *TDL object* called a *nodescope (ns),* which is passed around as an argument. Its functions and class methods usually return the rootnodes of subtrees, which are then used as children in other subtrees. In that way, very complex trees may be built up, using and re-using TDL scripts that may be imported from all over the place. TDL scripts may also contain definitions of (menus of) TDLOptions, i.e. variables that allow the user to customize the definition or execution of the tree.

All the user-level TDL scripts that are used for demonstration in this Beginners Guide are in the directory */usr/share/meqtrees/examples/beginners_guide*[1]*:*

```
[noordam@baez beginners_guide]$ ls -lrt *.py
-rw-r--r-- 1 noordam noordam 1146 2009-02-08 15:38 execute_general_request.py
-rw-r--r-- 1 noordam noordam 1572 2009-02-09 12:51 execute_sequence.py
-rw-r--r-- 1 noordam noordam 2620 2009-02-09 13:31 execute_request.py
-rw-r--r-- 1 noordam noordam 2218 2009-02-18 17:34 tensor_matrix.py
-rw-r--r-- 1 noordam noordam 1598 2009-02-19 18:04 tensor_matrix22.py
-rw-r--r-- 1 noordam noordam 2465 2009-02-20 10:35 tensor_constant.py
-rw-r--r-- 1 noordam noordam 8737 2009-02-20 10:46 make_bookmark.py
-rw-r--r-- 1 noordam noordam 3950 2009-02-20 16:26 unique_nodestub.py
-rw-r--r-- 1 noordam noordam 7856 2009-02-23 10:49 ROOT_node_definition.py
-rw-r--r-- 1 noordam noordam 1998 2009-02-23 10:52 myThirdTree.py
-rw-r--r-- 1 noordam noordam 2922 2009-02-23 11:18 ROOT_myFirstTree.py
-rw-r--r-- 1 noordam noordam 1926 2009-02-23 11:19 myFirstTree.py
-rw-r--r-- 1 noordam noordam 3741 2009-02-23 11:23 ROOT_myThirdTree.py
-rw-r--r-- 1 noordam noordam 1969 2009-02-23 11:25 ROOT_domain_4D.py
-rw-r--r-- 1 noordam noordam 2822 2009-02-23 11:26 ROOT_mySecondTree.py
-rw-r--r-- 1 noordam noordam 1923 2009-02-23 11:27 ROOT_solving.py
-rw-r--r-- 1 noordam noordam 1976 2009-02-23 11:29 mySecondTree.py
-rw-r--r-- 1 noordam noordam 1990 2009-02-23 11:32 domain_4D.py
-rw-r--r-- 1 noordam noordam 3736 2009-02-23 11:49 solving_11.py
```

[1] Under Ubuntu. Other distributions can have a slightly different path, depending on their conventions

Some TDL scripts are a bit special in the sense that they contain a TDL functions with names that are recognized by the system, and used for special purposes.

- In the first place, the function *_define_forest(ns)* is called when the *compile* button is pressed. It provides the *nodescope* object (ns) which is needed to define nodes.
- In the second place, all functions with names that starts with *_tdl_job_* (e.g. *_tdl_job_execute_request(mqs)*) will be presented to the user in a popup window then pressing the TDL Exec button, and will be executed when clicked upon.

Although it is not required in any way, the names that are given to the modules in this particular directory indicate what they are used for. Thus, the ones whose names start with *ROOT_* have a *_define_forest()* function, and one or more *_tdl_job_* functions. Only these files should be loaded into the browser (the prefix ROOT_ makes them easily recognisable in the file dialog). The files with names that start with *execute_* contain helper functions that are called by *_tdl_job_* fuctions. All the other modules just contain functions and their TDLOptions for defining subtrees. There are two special cases:

- The module *make_bookmark.py* contains helper functions that make it easy to generate bookmarks (very important!).
- The module *unique_nodestub.py* helps with avoiding nodename clashes (a moderately vexing problem in MeqTrees).

In the following, we will discuss the various parts of a TDL script, using as an example *ROOT_myFirstTree.py*.

Apart from the 'core' TDL modules, all the Python files used in the examples are in the same sub-directory. Thus, the examples do not depend on mysterious (and often unfindable) modules in other directories. All files may be copied to your own directory, and modified at will, in keeping with the sacred MeqTrees Principles of contribution, collaboration, cannibalization and competition.

## 2.1 Script description

It is highly recommended to include in your TDL script a decent description of what it is supposed to do, ideally kept up to date. First of all because most people find it difficult to read their own code after a remarkably short time. But it also makes it much easier to exchange scripts with others, which after all is one of the cornerstones of the MeqTrees endeavour. As the MeqTrees *Script Exchange Mechanism* matures, we will propose some sort of preferred format that lends itself to automatic dissection, e.g. for use in a GUI.

```
"""
file:  ../beginners_guide/ROOT_myFirstTree.py
... description of this module
... modification history
... copyright statement
"""
```

For the moment, we just suggest that you use the Python triple-quote string, which keeps the linebreaks. A nice feature of Python is that all modules, functions and classes have a *.__doc__()* method, which returns all the text before the first executable Python statement. If you make sure that this contains useful information, and is kept up-to-date (!), it is very useful as help-text, e.g. in the *quickref_help* field of a node.

## 2.2 Copyright statement

MeqTrees users tend to be nice guys, who generously work together. But there are also some evil people out there, who will steal a script and try to sell it, and even try to make you pay for using your own. For this reason, each TDL script that does something worthwile should contain a copyright statement. But since we do not want to clutter up the example scripts in this Beginners Guide, we will not bother with it here.

## 2.3 Preamble

The preamble imports other Python modules, and does things like initialization etc. The *Timba.TDL* below refers to a directory that contains some core MeqTrees modules. All the other modules are imported from the beginners_guide directory.

```
from Timba.TDL import *              # from a core MeqTrees directory
import myFirstTree as DT             # from this same beginners_guide directory
import execute_request as ER         # idem
import make_bookmark as BM           # idem
```

## 2.4 Compile-time TDLOptions

A TDL script often contains a definition of (menus of) TDLOptions for customizing a tree. They are presented to the user in a popup window (see fig 4.1). Note that (menus of) TDLOptions that are defined in imported modules are automatically included. It is even possible to include them into a menu by including a module into a TDLMenu definition (see chapter 4).

## 2.5 _define_forest(ns)

If a TDL script has a function with this name, it is called by the MeqBrowser when the compile button is pressed. Thus, only TDL scripts with such a function should be loaded into the MeqBrowser. In our examples, the names of such TDL scripts are prefixed with ROOT_, so they can be easily distinguished in the file dialog.

This function defines the forest of trees of nodes. It is called when the TDL script is loaded, or when the 'blue button' is pressed, or when the compile button in the *TDL Compile-time Options* popup window is pressed. Note the argument ns, which is the *nodescope*, a TDL object for defining MeqNodes (see chapter 4). If the compilation is successful, the tree will be displayed in the left (Trees) section of the MeqBrowser.

A minimal example of the *_define_forest()* function is shown here. It just calls the function *.myFirstTree(ns)* in the imported module *myFirstTree.py (DT)*. Note that, in this example, the global Python variable *rootnode* represents the node that will receive the execution request in the runtime function(s) *_tdl_job_...()* discussed below.

```
def _define_forest (ns, **kwargs):
  """
  Define a very simple tree, using the given nodescope (ns).
  """
  global rootnode
  rootnode = DT.myFirstTree(ns)
  return None
```

## 2.6 Other functions and classes

A TDL script may contain all kinds of other functions and classes, which may of course be called from any other Python module. Functions and class methods that define MeqNodes require a nodescope (ns) argument.

## 2.7 Runtime TDLOptions

In the examples of this Beginners Guide, all runtime options are defined in separate modules with names that start with *execute_*. They are automatically included in the execution popup window.

## 2.8 _tdl_job_...(mqs, parent)

When the tree has been defined and compiled, it may be executed by clicking on one of the functions in the *TDL Jobs and Runtime Options* popup window (see fig 4.1). The latter pops up after a succesful compilation, or may be conjured

up by clicking on the *TDL Exec* button. Any functions in the TDL script that have names that begin with *_tdl_job_* (e.g. *_tdl_job_execute_request()*) will appear automatically in the runtime popup. Most of them will execute the tree by issuing one or more Requests to a named node (usually the root node). But they may also do other things, like printing information, or starting another program like an imager.

When a *_tdl_job* is clicked on, the MeqBowser executes it with the arguments *mqs* and *parent*. The first one is a meqserver object, which is needed for executing trees. In this example, the *_tdl_job_execute_request()* just calls the function *.execute_request()* in the module *execute_request.py (ER)*. The latter also defines some runtime TDLOptions, which are automatically included in the popup menu.

```
def _tdl_job_execute_request (mqs, parent):
  """
  Execute by issuing a request to the node in the global variable 'rootnode'.
  The size and cells of the Request domain may be specified in the runtime menu.
  """
  return ER.execute_request(mqs, rootnode.name)
```

## 2.9 Testing outside the MeqBrowser

It is often useful to be able to test (specific parts of) a TDL script outside the MeqBrowser by executing as a Python script:

```
$ python ROOT_myFirstTree.py
```

This will execute the following section of you script (if you have provided one, of course):

```
if __name__ == '__main__':
  print '\n** Start of standalone test of: ROOT_myFirstTree.py:\n'
  ns = NodeScope()

  if False:
     _define_forest(ns)
     ns.resolve()

  if False:
     # ... perform test 1 ...

  if True:
     # ... perform test 2 ...

  print '\n** End of standalone test of: ROOT_myFirstTree.py:\n'
```

# Chapter 3

# Forests, trees, nodes, requests, results

> I don't think I will ever see,
> a thing as lovely as a tree. (*William Wordsworth)*

A node (a.k.a. MeqNode) is a software object of a certain class. Its function is to return a Result when it receives a Request. A Request will contain, among other things, an n-dimensional domain, subdivided into cells (see section 3.1). The Result contains, among other things, a *vellset* of values for the cells of the requested domain (see section 3.2).

Nodes may be linked up with each other in *trees* (or more precisely: non-circular graphs). Each node has zero or more child nodes, and is itself a child of zero or more parent nodes. A node with zero children is called a *leaf* node (e.g. MeqConstant, or MeqParm). A node with zero parents is called a *rootnode*, and will appear in the list of rootnodes in the Trees section of the MeqBrowser.

Whenever a node receives a Request, it passes it on to its children. When they return their Results, our node will combine them into a Result of its own in a way defined by its class (e.g. a node of class MeqAdd will simply add the results of its children). It then passes the new Result to its own parent(s), etc.

A *forest* is a group of parallel, and usually similar, trees, each with its own rootnode. An example is the group of interferometers of a radio aperture synthesis array, each of which has its own tree, from the specialized leaf node MeqSpigot, to the specialized root node MeqSink (see also chapter 9).

Each node has a *state record*, which may be inspected by clicking on it in the left section of the MeqBrowser. It is highly recommended to study the state records of a few different types of nodes, and to guess what the various fields ar for (a detailed explanation is outside the scope of this Beginners Manual). A sample is given in fig B.1.

There is also an overall *forest state record*, which may be inspected by clicking

on it at the top of the left section of the MeqBrowser. Again, try to guess what the various fields mean in the sample of fig B.4.

## 3.1 Domains

A *domain* is a rectangular area in dimension space. It is usually 2-dimensional (freq,time), but up to user-defined 10 dimensions are supported. A domain is sub-divided into *cells*. Although the default is a regular n-dimensional array, the cell positions and sizes may also be specified as vectors of arbitrary numbers *for each dimension*. Thus, the cells do not have to be contiguous, or regular, and they may even overlap. See fig 3.1. The flexibility in specifying domains of arbitrary sizes and arbitrary cell structure is one of the most powerful features of MeqTrees.

The usual way of data processing is to issue a sequence of Requests with different domains. When processing a particular dataset, the successive domains will often be generated by a special interface node like the MeqVisDataMux. But the user has the choice between using many small domains, or a few large ones. The latter is more efficient, because it minimizes the node overhead, incurred by the transfer of Requests and Results between nodes. Therefore, large domains should be used in principle, only limited by the available memory. However, when solving for M.E. parameters, the optimum domain size and cell structure are influenced by other considerations (see section 6).

There are special nodes that modify the number of cells in the Request before it is passed on, or to change the number of cells in the Result by resampling. Obviously, the latter works better if the Result represents a smooth function, which is approximately linear over a cell. Reducing the number of cells that sample a smooth function can have a large impact on efficiency.

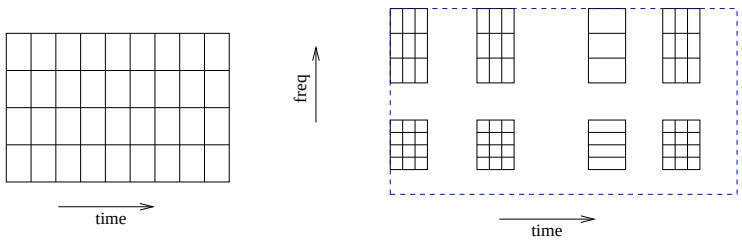


Figure 3.1: *The main function of a MeqTrees node is to return values for the cells of a domain. The default domain on the left is 2D (freq,time) and has identical and contiguous cells on a regular grid. But it is also possible to have irregular grids of non-contiguous cells with different sizes, as illustrated on the right. MeqTrees supports domains with up to 10 dimensions.*

## 3.2 Vellsets

The main function of a node is to return a Result with a *vellset* of values for the cells of the domain specified in a Request. The cell values represent the centre of the cell. The values may be either float or complex.

MeqTrees automatically minimizes the memory size of a vellset by collapsing any axes along which the values do not change. For instance, if all the values of a vellset have the same value, as is the case for a MeqConstant node, only a single value is stored.

## 3.3 Scalars, tensors and arithmetic rules

Most node Results contain a single *vellset*, i.e. an n-dimensional array of values for the requested domain cells. But Results may also contain a list of more than one vellset. If a *shape* is assigned to it (e.g. dims=[2,2]), the list may be interpreted as a vector, a matrix, or a tensor. Tensor nodes can reduce tree clutter, by reducing the number of distinct nodes. They also increase efficiency (a bit).

Nodes are called *scalar* or *tensor* nodes depending on the number of vellsets in their Result. Usually, this is determined by the Results of their children, which may have different shapes. A 2D tensor node can be a *matrix* node. Note that being a tensor or scalar or matrix node is not an *intrinsic* quality of a node type. The following arithmetic rules apply:

- Nodes with one child (e.g. MeqCos) will do something sensible and unambiguous with a tensor child. Their Result will usually be a tensor node with the same shape. There are very few one-child nodes that modify the tensor shape, unless it is their specific function (e.g. MeqTranspose or MeqSelector). For instance, at this moment, there is no way to add the elements of a tensor node.
- Nodes with an arbitrary number of one or more children (e.g. MeqAdd) will usually require all children to have the same tensor shape, or to be a scalar. Their Result will usually be a tensor node with the same shape. An exception is MeqComposer, which takes children with arbitrary shape, and puts all their results together.
- The basic rule for nodes that take two children (e.g. MeqSubtract) is simple: If the two shapes differ, one of them must be a scalar. In that case, it will be applied to all the Results of the tensor child. If the child shapes are the same, the operation is element by element. In special cases, like matrix operations, the rules are consistent with the accepted conventions.

There are various nodes for combining the Results of multiple nodes into a single tensor node, or for making a separate node of a tensor element. A MeqConstant

tensor node may be specified with a list of constants, and an optional shape (e.g.dims=[3,2,4]).

## 3.4 Error reporting and propagation

If you see smoke wafting from your computer when it is processing MeqTrees, there is nothing to worry about. It is just a sign that it is finally doing some honest work, after all your two-fingered editing. It is more troublesome if your machine stops jumping up and down, because it means that something is wrong, and processing has halted. Unlike some other data reduction packages, MeqTrees will try to tell you, in language that you might understand, what the problem is, and what you might do about it. We distinguish two categories of errors:

- *Compile-time errors.* If you have made a Python syntax error, the MeqBrowser will localize it, load the relevant module in its middle section, and indicate the offending line. If you like, you may edit the module right there, and recompile. If you make an error in defining a node, the number of errors will be indicated in the Messages tab of the middle section, and you can investigate by selecting this tab. Reset the error counter by left-clicking below the tab.

- *Runtime errors.* If, for some reason, a node has trouble to produce a Result, its name will turn red in the tree section of the MeqBrowser. Since its parent will obviously also have a problem, it will turn red also, all the way down the tree to its root node. We call this a *trail of blood.* The problem may be inspected either by clicking on a red nodename, which will display its state record. The cache (result) will contain a *fail* message.

While the tree is being executed, it generates progress messages of various kinds along the bottom-left edge of the MeqBrowser. There is a Debug menu at the top, and along the left edge are buttons for debugger functions like set breakpoint, stop, step, resume, etc. These are outside the scope of this manual.

# Chapter 4

# Tree Definition Language

> Thought is not possible without Language. Just think about it!
> *(Douglas Hofstadter)*

The following is an extremely terse summary of the Tree Definition Language (TDL). For a more elaborate (and chatty) treatment, see the *TDL Bedside Companion* [3]. If you are at all serious about using MeqTrees, you should have this document handy.

TDL is just Python. It does two things:

- It provides a Python object called a *nodescope* (ns) that allows you to define *Python node specifications*, i.e. their class (or type), their children (if any), and any other arguments that a specific node type may require (e.g. a MeqConstant requires a value).
- It allows you to define (menus of) TDLOptions, i.e. user-settable variables for customizing the definition and execution of trees.

It is important to realize that TDL does not create actual C++ nodes, just a list of node specifications. These are used by the *meqserver* to generate C++ objects, which may then be executed by passing a Request to them. The tree that is displayed in the left (Trees) section of the MeqBrowser, and the node views in the right (Gridded Viewers) section all relate to C++ nodes, read by the meqserver. Thus Python, an interpreted scripting language, makes it easy for you to define trees, but it does not play a role in execution (except in the special case of pyNodes, see chapter 8).

## 4.1 Defining MeqNodes

Python MeqNodes are defined with the help of a two TDL objects: the *nodescope* object (ns) and the *Meq* object. The Python node specifications are later

converted into C++ nodes by the meqserver. Here, we will concern ourselves only with the Python side.

```
from Timba.TDL import *                    # import all TDL classes and functions
ns = NodeScope()                           # make a nodescope object

# Tree-step process to define a node and assign it to a variable:
nodestub = ns[name]                        # create a node-stub object
nodestub << Meq.Node(children, arguments)  # initialize it with a node definition
variable = nodestub                        # assign it to a variable
type(nodestub) -> <class 'Timba.TDL.TDLimpl._NodeStub'>

# The same thing can be achieved in a single line:
variable = ns[name] << Meq.Node(children, arguments)
```

First of all, a *nodestub* object is generated by qualifying the nodescope object with a name: `v = ns.a` or `v = ns['a']`. The [] form allows names with strange characters, and also names that are variables. Note that we assign the nodestub to a Python variable (v), which is not required, but highly recommended.

For those who know a little Python, *qualification* of a nodescope object makes use of its *__getattr__()* method. The resulting nodestub can be qualified in its turn by using its *__call__()* method, which is the same as a function call: `v1 = v(qual)` or `v1 = ns.a(qual)`. The qualifier may be a string, integer, float, complex etc. So, if qual=54, the result will be a nodestub named *a:54*. Since the result is another nodestub, this process may be repeated ad infinitem: `v2 = v(q1)(q2)(kw=4)`.

A nodestub is converted into a MeqNode (object) by *initialising* it with a node class, or type. This is done with the << operator, and the TDL Meq object, which has a method for each available node class, with arguments to match. The latter will often be its children, but some node classes require other arguments. Thus, the general syntax for MeqNode definition is: `var = ns[name](qual)(..) << Meq[nodeclass]([args])`.

NB: An initialized MeqNode is still a nodestub object. For the purpose of node definition, the main difference between them is the outcome of the test *node.initialized()*, which will return True or False. Very importantly, new nodestubs may be generated by qualifying any nodestub, whether it is initialized or not:

```
stub = ns.stub
node = stub << Meq.Constant(1.0)
stub1 = stub(1)
node2 = stub(2) << Meq.Freq()
stub23 = node2(3)
```

Careful study of the following goups of examples will give you the general idea of this important subject. Again, if you have a computer handy, load the TDL script ROOT_node_definition.py into the MeqBrowser, to see the same examples in action. First the basics:

```
# First define a few nodestubs, and assign them to Python variables:
ns.a1                          # --> a1(None)            #
```

```
a2 = ns.a2                    # --> a2(None)            #
a3 = ns['a3']                 # --> a3(None)            #
a4 = ns['a2+a3']              # --> a2+a3(None)         #
a5 = ns.a5('qual')            # --> a4:qual(None)       # qualifier
a6 = ns.a6(kwqual=5)          # --> a5:kwqual=5(None)   # keyword qualifier
a7 = ns.a7(3)('3')(x=-5.5)    # --> a7:3:3:x=-5.5(None) # multiple qualifiers
a8 = ns.a8(3,-4,5)            # --> a8:3:-4:5(None)     # multiple qualifiers

# Then initialize the nodestubs:
ns.a1 << 1.0                  # --> a1(MeqConstant)            # no variable
a2 << 2                       # --> a2(MeqConstant)            #
a3 << Meq.Constant(-3)        # --> a3(MeqConstant)            #
a4 << Meq.Add(a2,a3)          # --> a2+a3(MeqAdd)              # 2 children
a5 << complex(4,5)            # --> a4:qual(MeqConstant)       #
a6 << Meq.Parm(5.0)           # --> a5:kwqual=5(MeqParm)       # default value
a7 << 5.0                     # --> a7:3:3:x=-5.5(MeqConstant) #
a8 << Meq.Constant([8,9])     # --> a8:3:-4:5(MeqConstant)     # tensor node
```

Leaf nodes have no children. Here are some of the often-used ones. The Meq[class]() argument is usually unambiguous, but it is sometimes better to specify it by name (e.g. Meq.GaussNoise(stddev=..)).

```
b1 = ns.scalar << Meq.Constant(1.0)          # --> b1(MeqConstant)         # value=1.0
b2 = ns.tensor << Meq.Constant([3,4,-1])     # --> b2(MeqConstant)         # tensor node
b3 = ns.parm << Meq.Parm(-1.0)               # --> b3(MeqParm)             # default=-1.0
b4 = ns.freq << Meq.Freq()                   # --> b4(MeqFreq)             #
b5 = ns.time << Meq.Time                     # --> b5(MeqTime)             # same as Meq.Time()
b6 = ns.gauss << Meq.GaussNoise(stddev=1.5)  # --> b6(MeqGaussNoise)       #
```

Most node classes require one or more child nodes. Usually, these are given as the first argument(s) of the relevant method of the TDL Meq object. Multiple children can also be supplied as a list: Meq[class](*cc). However, if there are other arguments as well, the children must be specified with the 'children' keyword: Meq[class](children=cc, arg=..). Note that the order of the child nodes is usually important.

```
c1 = ns.c1 << Meq.Freq()                                 # --> c1(MeqFreq)   # child 1
c2 = ns.c2 << Meq.Time()                                 # --> c2(MeqTime)   # child 2
c3 = ns.c3 << Meq.Add(c1,c2)                             # --> c3(MeqAdd)
c4 = ns.c4 << Meq.Add(*[c1,c2])                          # --> c4(MeqAdd)
c5 = ns.c5 << Meq.Add(children=[c1,c2], option='opt')    # --> c5(MeqAdd)
nodeclass = 'Cos'
c6 = ns.c6 << getattr(Meq,nodeclass)(c5)                 # --> c6(MeqCos)    # node-class as variable
```

If no nodename is specified, the nodescope will do its best to generate automatic nodenames. If an initialized node of this name already exists, it will modify the name until it is unique. If is even clever enough to use the same name if this cannot lead to ambiguity (i.e. if the children are the same too). It is also possible to define subtrees by giving expressions of nodes and numbers. MeqTrees will recognize the Python math operations verb@+,-,*,/@ and the Python math functions abs() and ...?

```
d1 = ns << 67                  # --> (constant)(MeqConstant)    # automatic node
d2 = ns << Meq.Freq()          # --> (freq)(MeqFreq)            #
d3 = ns << Meq.Freq()          # --> (freq)1(MeqFreq)           # unique name
d4 = ns << Meq.Cos(d2)         # --> cos((freq))(MeqCos)        #
d5 = ns << Meq.Cos(d2)         # --> cos((freq))(MeqCos)        # same node!
d6 = ns << d3+d1               # --> add((freq)1,(constant))    # automatic MeqAdd
d7 = ns.mathexpr << (d3-d1)/d5 # --> mathexpr(MeqDivide)        # Python math expression
d8 = ns.mathfunc << abs(d4)    # --> mathfunc(MeqAbs)           # Python math function
```

Finally, there is the tricky issue of avoiding node-name clashes. This slightly more advanced subject is treated in section 8.2 (but illustrated in the same *ROOT_node_definition.py*).

## 4.2 TDL Compile and Runtime Options

The user may customize scripts by specifying values for parameters of tree definition and execution functions. This is discussed in the description of the various components of TDL scripts in chapter 2. The two associated MeqBrowser popup windows are shown in fig 4.1. The syntax of defining (menus of) TDLOptions of both kinds may be studied in the various scripts that are reproduced in this manual. For an example of defining compile-time options, see the the code of the function *.mySecondTree()* in section 5.2. For an example of incorporatinf the TDLOptions from imported modules into a TDLMenu, see the code of TDL script *ROOT_myThirdTree.py* in section 5.3. For an example of defining runtime options, see the code of the function *.execute_request()* in chapter 7. For more details on TDLOptions, see [?].

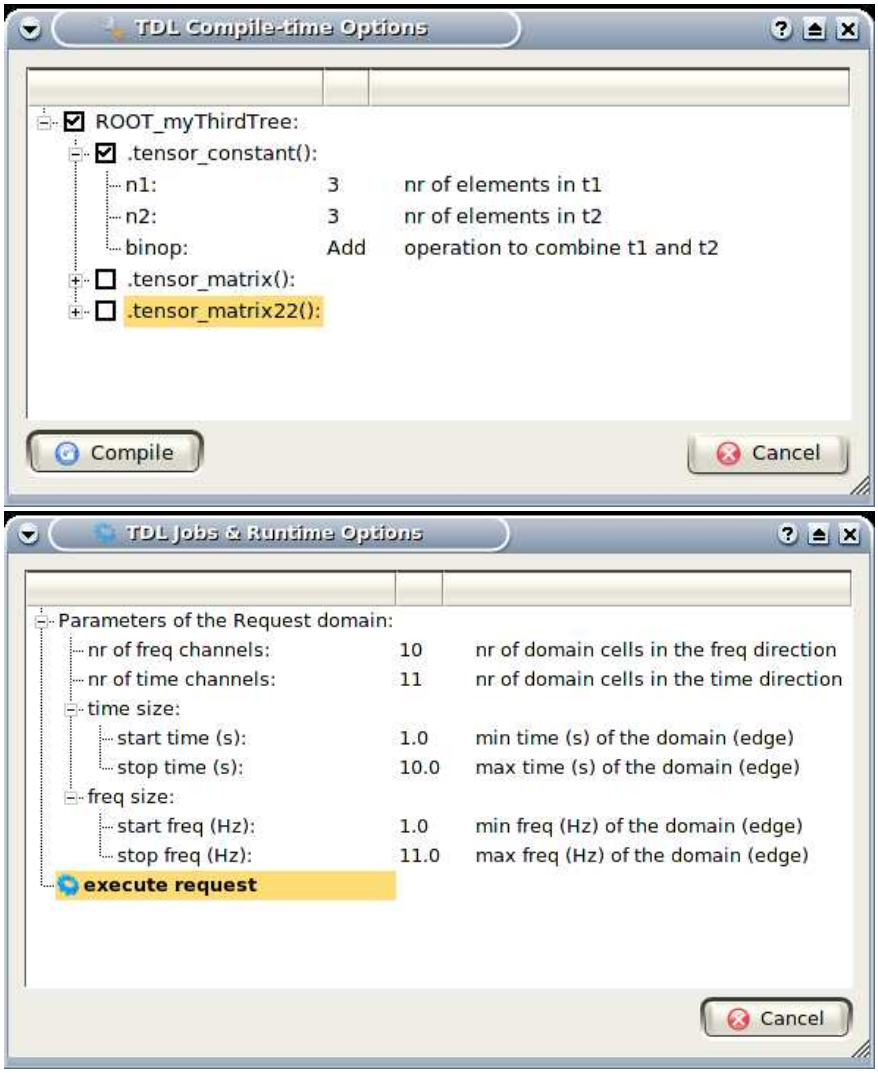


Figure 4.1: *MeqBrowser popup windows for the compilation and execution of trees. At the top is the popup that appears automatically when loading a TDL script (provided it has TDLCompileOptions). It can also be invoked by pressing the TDL Options button. It allows the specification of TDLCompileOptions, i.e. the arguments used in _define_forest() and any other functions it calls. The 'compile' button of the popup does the same as the 'blue button' of the browser. Below is the popup that appears automatically after compiling a tree. It can also be invoked by pressing the TDL Exec button. It allows the specification of TDL-RuntimeOptions, and the execution of _tdl_job(s). NB: Such a _tdl_job_() does not necessarily execute a tree. It can do all kinds of other things, like printing extra information, or starting another program like an imager.*

# Chapter 5

# My first tree(s)

In this chapter, some simple example trees demonstrate various aspects of tree building. Each example has a little more complexity than the last one.

## 5.1 ROOT_myFirstTree.py

The various parts of *ROOT_myFirstTree.py* are reproduced verbatim in chapter 2, illustrating the various components of a typical TDL script. Its function *._define_forest()* calls a separate Python function *.myFirstTree()*, from the imported module *myFirstTree.py*. You are invited to study its code below, line by line, and to relate it to the MeqBrowser screen-shot in fig 5.1:

```
# file ../beginners_guide/myFirstTree.py
from Timba.TDL import *
import make_bookmark as BM

def myFirstTree (ns):
  """ This function defines a tree for: f(x,y) = alpha*sin(b*x+c*y+1.2)
  It uses the given nodescope object (ns), which is a TDL object.
  """
  # This defines a leaf node named "alpha" of class Meq.Constant,
  # initialized with the given value (the fine-structure constant).
  # The node is assigned to a Python variable (alpha), for convenience:
  alpha = ns.alpha << Meq.Constant(value=297.61903062068177)

  # It is easier to just do <<(numeric value).
  # This implicitly defines MeqConstant nodes.
  b = ns.b << 1
  c = ns.c << 2.0
  x = ns.x << -1
  y = ns.y << 1.5

  # Multiply(the results of) two nodes: bx = b*x, the hard way:
  # This defines a node named "bx" of class Meq.Multiply, with two children:
  bx = ns.bx << Meq.Multiply(b,x)

  # Multiply (the results of) two nodes: cy = c*y, the easy way:
  # Arithmetic on nodes implicitly defines the "right" things
  cy = ns.cy << c * y
```

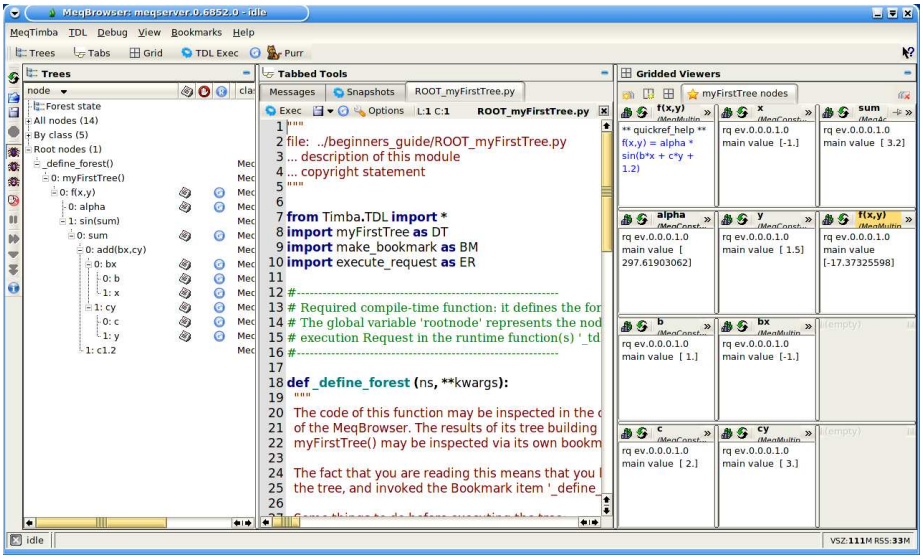


Figure 5.1: *The MeqBrowser with ROOT_myFirstTree.py. Compare the expanded tree in the left (Trees) section with the code of the function .myFirstTree(ns) that is reproduced in section 5.1. The Bookmark item named 'myFirstTree nodes' produces the page of node Results on the right.*

```
# Add (the results of) three nodes: sum = b*x+c*y+1, the easy way
# This will implictly create a Meq.Constant node for the "1.2", etc.
sum = ns.sum << bx + cy + 1.2

# The final result. Note the alternative way of specifying the node name.
# Note that an intermediate node of class Meq.Sin is created automatically
rootnode = ns['f(x,y)'] << alpha * Meq.Sin(sum)

## NB: We could have accomplished the same thing with:
## rootnode = ns['f(x,y)'] << ns.alpha*Meq.Sin(ns.b*ns.x + ns.c*ns.y + 1.2)

# Finishing touches:
BM.bookpage_nodes ([alpha,b,c,x,y,bx,cy,sum,rootnode],
                   name='myFirstTree nodes',
                   help='f(x,y) = alpha * sin(b*x + c*y + 1.2)')
return rootnode
```

## 5.2 ROOT_mySecondTree.py

The module *ROOT_mySecondTree.py* demonstrates a few more features than *ROOT_myFirstTree*.py. Firstly, its function *_define_forest()* calls another tree definition function *.mySecondTree()*, from the imported module *mySecondTree.py*. The latter also defines a few compile-time TDLOptions, which appear in the compile popup window. Secondly, there is an extra *_tdl_job_*

for executing a sequence of requests (see section 7.2). Its runtime TDLOptions appear in the execute popup window. See fig 1.2.

```
import mySecondTree as DT
import make_bookmark as BM
import execute_request as ER
import execute_sequence as ES

def _define_forest (ns, **kwargs):
  global rootnode
  rootnode = DT.mySecondTree(ns)

def _tdl_job_execute_request (mqs, parent):
   return ER.execute_request(mqs, rootnode.name)

def _tdl_job_execute_sequence (mqs, parent):
   return ES.execute_sequence(mqs, rootnode.name)
```

The TDL script *mySecondTree.py* that contains the function *.mySecondTree()*, and the definition of a menu of compile-time TDLOptions looks like this:

```
# file ../beginners_guide/mySecondTree.py
from Timba.TDL import *
import make_bookmark as BM

TDLCompileMenu(".mySecondTree():",
               TDLOption('copt_MeqFreq', 'add MeqFreq', True,
                         doc='if True, add a Meq.Freq node'),
               TDLOption('copt_MeqTime', 'add MeqTime', True,
                         doc='if True, add a MeqTime node'),
               TDLOption('copt_complex', 'add complex', True,
                         doc='if True, add a complex node'),
               TDLOption('copt_MeqGaussNoise', 'add MeqGaussNoise',
                         [0.0, 0.1, 1.0, 10.0], more=float,
                         doc='if True, a MeqGaussNoise(stddev)'),
               toggle='copt_mySecondTree')


def mySecondTree (ns):
  # Make a list of the child nodes specified by TDLOptions:
  cc = []
  if copt_MeqFreq:
    cc.append(ns.MeqFreq << Meq.Freq())
  if copt_MeqTime:
    cc.append(ns.MeqTime << Meq.Time())
  if copt_complex:
    cc.append(ns.complex << Meq.ToComplex(ns << Meq.Freq(), ns << Meq.Time()))
  if copt_MeqGaussNoise>0:
    cc.append(ns.MeqGaussNoise << Meq.GaussNoise(stddev=copt_MeqGaussNoise))

  # Make sure that there is at least one child, and add them all:
  if len(cc)==0:
    cc.append(ns.zero << 0.0)
  rootnode = ns.sum << Meq.Add(*cc)

  # Finishing touches:
  BM.bookpage_nodes (cc+[rootnode], help=None, name='mySecondTree nodes')
  return rootnode
```

## 5.3 ROOT_myThirdTree.py

The file *ROOT_myThirdTree.py* deals with tensor nodes, i.e. nodes that have Results with multiple vellsets (see section 3.3). This is a powerful feature of

MeqTrees, because it allows matrix operations, and visualization of the Results of groups of nodes.

This TDL script also demonstrates how the TDLOptions of multiple tree definition functions may be combined rather neatly into a single TDLMenu. When a particular function is selected in the compile-time popup by ticking its box, its options become visible.

```
"""
file:  ../beginners_guide/ROOT_myThirdTree.py
"""

from Timba.TDL import *
import tensor_constant as TC              # tree definition module
import tensor_matrix as TM                # tree definition module
import tensor_matrix22 as TM22            # tree definition module
import execute_request as ER
import execute_sequence as ES
import make_bookmark as BM

TDLCompileMenu("ROOT_myThirdTree:",
               TC,                        # this includes the TDLOptions from module TC (=tensor_cosntant.py)
               TM,                        # this includes the TDLOptions from module TM (=tensor_matrix.py)
               TM22,                      # this includes the TDLOptions from module TM22 (=tensor_matrix22.py)
               toggle='copt_myThirdTree')


def _define_forest (ns, **kwargs):
  # NB: The tree definition functions (TC, TM, TM22) return a valid node if
  # they are selected, and None otherwise. The first valid node is used.
  global rootnode
  rootnode = TC.tensor_constant(ns)
  if not is_node(rootnode):
    rootnode = TM.tensor_matrix(ns)
  if not is_node(rootnode):
    rootnode = TM22.tensor_matrix22(ns)
  if not is_node(rootnode):
    rootnode = ns.dummy << -0.123456789
  return None
```

The *_tdl_jobs_* are the same as in *ROOT_mySecondTree.py*. We reproduce only one of the three tree definition functions called by *_define_forest()*. The two others are reproduced in chapter 8.

```
""" file ../beginners_guide/tensor_constant.py """
from Timba.TDL import *
import make_bookmark as BM

binops = ['Add','Multiply','Subtract','Divide','Pow',
          'ToComplex','Polar','Composer']

TDLCompileMenu(".tensor_constant():",
               TDLOption('copt_n1', 'n1', [3]+range(1,5),
                         doc='nr of elements in t1'),
               TDLOption('copt_n2', 'n2', [3]+range(1,5),
                         doc='nr of elements in t2'),
               TDLOption('copt_binop', 'binop', binops, more=str,
                         doc='operation to combine t1 and t2'),
               toggle='copt_tensor_constant')

def tensor_constant (ns):
  """
  This function demonstrates tensor nodes, i.e. nodes that have
```

```
 Results with multiple vellsets (scalar nodes have one vellset).
 First make tensors t1 and t2, each with a user-defined number
 of elements. Then combine them into t12 with a user-defined
 operation (e.g. 'Add'). Finally, t1,t12,t2 and t1 (again!)
 are 'composed' into a single tensor node with MeqComposer.
 Since the elements are constants, which do not vary over the
 domain, the node Results are shown as lists of element values.
 This makes it easy to see what happens.
 Note what happens if t1 and t2 have different numbers of elements,
 especially if none of them is a scalar (one element).
 Note that MeqComposer just makes a list of the elements of its
 children, in the order in which they are given. The same child
 can be used more than once.
 Obviously, it would be nice if the elements would be labelled
 in the Result. This has been on Tony's list for some time.
 """

if not copt_tensor_constant:
   return None                                  # not selected

 t1 = ns.t1 << Meq.Constant(range(copt_n1))
 t2 = ns.t2 << Meq.Constant(range(10,10+copt_n2))
 t12 = ns.t12 << getattr(Meq, copt_binop)(t1,t2)
 rootnode = ns['t1,t12,t2,t1'] << Meq.Composer(t1,t12,t2,t1)

 # Finishing touches:
 BM.bookpage_nodes ([t1,t2,t12,rootnode], name='tensor nodes',
                    help=tensor_constant.__doc__)
 return rootnode
```

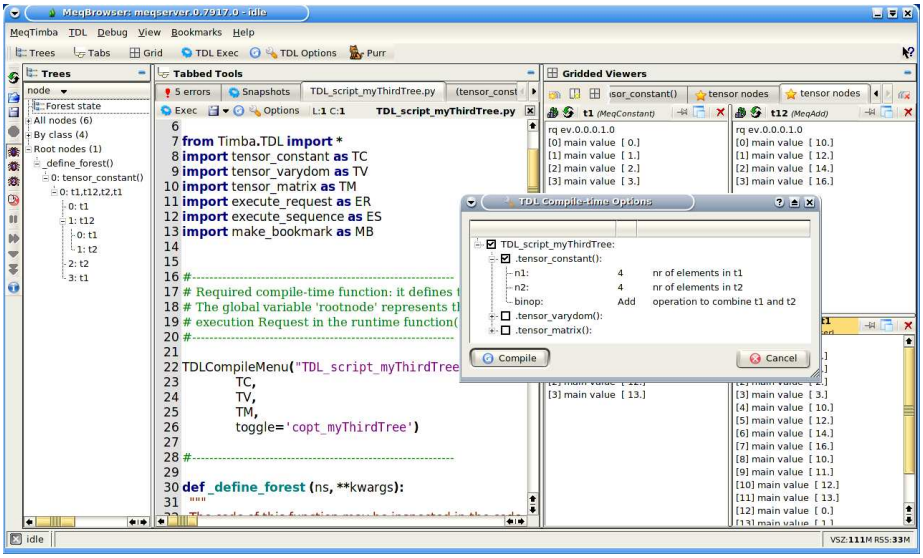


Figure 5.2: *ROOT_myThirdTree.py deals with tensor nodes. Note the multiple vellsets in the Results on the right (we use constants here, because then the Result Plotter shows all the vellsets of a tensor node in a single panel). Relate the node Results, and the expanded tree on the left, with the Python code reproduced in section 5.3.*
*The compile-time popup gives the choice between three different trees, which are defined in different modules (TC, TM and TM22), each of which has its own TDLOptions. The Python code in the middle shows how the TDLOption menus from the imported modules can be included in the menu that is shown in the popup. Clicking on a tick-box reveals the options of that module, which are otherwise hidden.*

# Chapter 6

# Solving for MeqParm coefficients

## 6.1 Some mathematical background

A Measurement Equation (M.E.) may have many parameters $p_k$. We define calibration as the process of finding those values of $p_k$ for which the weighted differences $w_i(D_i - M_i)$ between the available data $D_i$ and predicted (model) values $M_i$ are minimized in a least-squares sense: $\sum_i w_i^2(D_i - M_i)^2 \Rightarrow 0$. This may be done by accumulating linear condition equations of the form:

$$w_i(D_i - M_i) = w_i \sum_k \frac{\partial M_i}{\partial p_k} \Delta p_k \tag{6.1}$$

into a matrix. If the matrix is rectangular, with more equations then unknowns, inversion is equivalent to a least-squares fit. The model values $M_i(p_k)$ are calculated with the M.E., using the best available values of the parameters $p_k$. The M.E. is also used to calculate the partial derivatives $\frac{\partial M_i}{\partial p_k}$. After inversion, the solution is a set of *incremental* improvements $\Delta p_k$ of the parameter values $p_k$. If the M.E. is non-linear, which it usually is, an iterative solution is required. At present, MeqTrees just has a non-linear Levenberg-Marquardt solver, which is suitable for a large class of problems. Other types of solvers will become available later.

The inversion of the solution matrix is done by SVD, which always returns a solution, even if the problem is ill-conditioned, i.e. there is not enough information to solve for all unknowns. In that case, SVD automatically provides a minimum of extra information, perpendicular to the solvable space. This is often a reasonable choice, but it is obviously important for the user to know that this has happened. MeqTrees provides an increasing range of tools for assessing the quality of solutions in a generic way, i.e. independent of the specific problem.

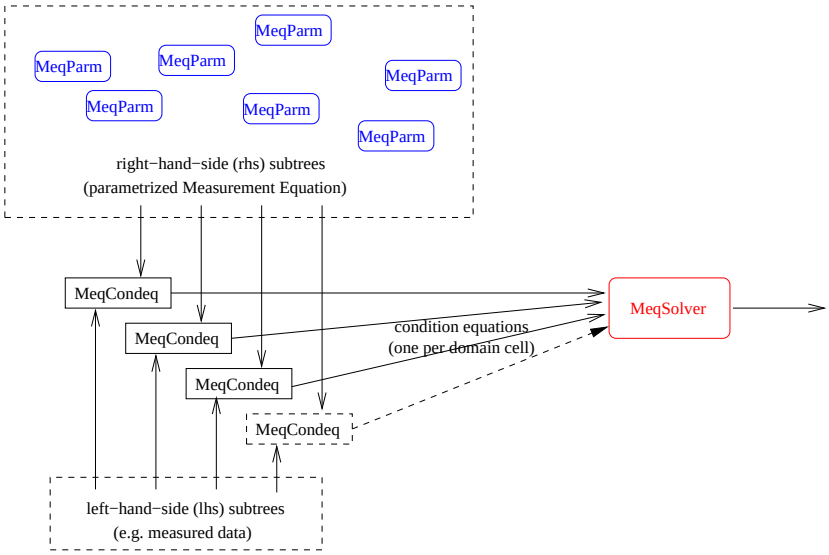


Figure 6.1: *The general structure of a solver subtree. A MeqSolver node has one or more MeqCondeq children, which supply it with condition equations for its solution matrix, a separate one for each domain cell. The goal is to minimize the difference (lhs-rhs) between the results of the MeqCondeq children, by fiddling the values of (an arbitrary subset of) MeqParm nodes. The latter are part of a parametrized Measurement Equation, implemented by MeqTrees. MeqParm values are in the form of polcs, i.e. arrays of polynomial coefficients in frequency and time. Other functions and other dimensions are possible too. In general, a separate solution is made for each Request (domain), generating its own set of MeqParm values. Thus, each MeqParm will be associated with a sequence of values (polcs), which it may interpolate to calculate Result values for an arbitrary Request domain.*

An M.E. parameter is represented by a MeqParm node, which is specified with a default value. One of the most powerful features of MeqTrees is that each parameter is a function of frequency and time, and optionally of other dimensions. The default function is a 2D (time,freq) polynomial. In that case, a parameter value takes the form of a *polc*, i.e. an array of polynomial coefficients. Other functions are possible in principle, each with its own collection of coefficients. Note that these coefficients are the actual $p_k$ for which the solver solves. The number of coefficients may be specified separately for each MeqParm, and for each solution. Each polc (or other set of coefficients) is associated with a domain over which it is valid.

A MeqSolver node solves for a *user-defined subset* of the available MeqParms, which are set to *solvable* by the MeqSolver via the Request. Identifying suitable subsets of MeqParms is greatly helped by the possibility of tagging them, and searching the tree for nodes with specific tags. Multiple solvers may be active at the same time, but at this moment they cannot share the same MeqParms, whether they are set to solvable or not.

A MeqSolver node has one or more MeqCondeq children, each of which has two children, whose Results provide the values for $D_i$ and $M_i$ in equ 6.1. (NB: in the future a third child may be added to provide the weight $w_i$. Until then, $w_i = 1$). A MeqCondeq node makes condition equations of the form 6.1 for the MeqSolver, who accumulates them into its solution matrix. A separate equation is generated for each cell in the domain. The partial derivatives $\frac{\partial M_i}{\partial p_k}$ are calculated by the subtree below $M_i$, starting at the relevant (i.e. solvable) MeqParm nodes. At this moment, only numerical derivatives are calculated, rather than analytical ones. Whatever method is used, the derivatives require an extra vellset for each solvable coefficient, to be processed by every node, and to be carried in the Result. This can be a considerable burden.

## 6.2 Solving_11: One MeqCondeq and one MeqParm

Here is the Python code of the function *.solving_11(ns),* in the TDL script *solving_11.py*, which is imported in the TDL script *ROOT_solving.py:*

```
"""
file ../beginners_guide/solving_11.py
"""
from Timba.TDL import *
import make_bookmark as BM

#-------------------------------------------------------------------
unops = [None,'Cos','Sin','Tan','Acos','Asin','Atan','Cosh','Sinh','Tanh',
         'Exp','Log','Abs','Negate','Invert','Sqrt','Pow2','Pow3','Pow8',
         'Ceil','Floor','Identity']
binops = ['Multiply','Add','Subtract','Divide','Pow',
          'ToComplex','Polar','Composer']

TDLCompileMenu(".solving_11():",
               TDLOption('copt_binop', 'binop', binops, more=str,
```

```
                    doc='ft = MeqFreq binop MeqTime'),
                TDLOption('copt_unop', 'unop', unops, more=str,
                    doc='lhs = unop(ft)'),
                TDLOption('copt_stddev', 'noise', [0.0,0.01,0.1,0.1,1.0,10.0], more=float,
                    doc='stddev of added Gaussian noise'),
                TDLOption('copt_fdeg', 'fdeg', range(6), more=int,
                    doc='degree of MeqParm freq poly'),
                TDLOption('copt_tdeg', 'tdeg', range(6), more=int,
                    doc='degree of MeqParm time poly'),
                TDLOption('copt_num_iter', 'num_iter', [3,1,2,5,10,20,100], more=int,
                    doc='number of iterations'),
                toggle='copt_solving')
#-------------------------------------------------------------------

def solving_11 (ns):
  """
  This function defines a solving tree in which the MeqSolver has a
  single MeqCondeq child, with children lhs and rhs.

  The left-hand-side (lhs) child is a user-defined subtree that produces
  a result that varies (optionally non-linearly) in freq and time:
  -  lhs = ft = binop(freq, time)       (e.g. binop = Multiply)
  -  [lhs = unop(ft)]                   (e.g. unop = Cos)
  -  [lhs = noisy = lhs+noise]       (if stddev>0)

  The right-hand-side (rhs) child is a single MeqParm node that
  represents a polc, i.e. a 2D polynomial in freq and time:
   -  rhs = MeqParm(0.0, shape=[tdeg+1,fdeg+1])

  The shape of the polc (i.e. the polynomial degree in freq and time)
  is specified by the user by means of TDLOptions.
  The solver solves for the polc coefficients, in several iterations.
  The result of the condeq node contains the residuals, i.e. rhs-lhs.
  With increasing polc polynomial degree, the residuals approach zero.
  """

  # Make the left-hand side (the copt_... are TDLCompileOptions):
  # First combine (Multiply or Add or ...) MeqFreq and MeqTime:
  lhs_name = 'f_'+copt_binop+'_t'
  lhs = ft = ns[lhs_name] << getattr(Meq,copt_binop)(ns<<Meq.Freq(), ns<<Meq.Time())

  if isinstance(copt_unop,str):
    # Optionally, apply a unary operation (e.g. Cos) on ft:
    lhs_name = copt_unop+'('+lhs_name+')'
    lhs = ns[lhs_name] << getattr(Meq,copt_unop)(lhs)

  if copt_stddev>0:
    # Optionally, add Gaussian noise:
    noise = ns.noise << Meq.GaussNoise(stddev=copt_stddev)
    lhs_name += '+noise'
    lhs = ns[lhs_name] << Meq.Add(lhs,noise)

  # Make the right-hand side:
  rhs = ns.rhs << Meq.Parm(0.0, shape=[copt_tdeg+1,copt_fdeg+1])

  # Make the MeqConde and the MeqSolver:
  condeq = ns.condeq << Meq.Condeq(rhs,lhs)
  rootnode = ns.solver << Meq.Solver(condeq, solvable=rhs, num_iter=copt_num_iter)

  # Finishing touches:
  help = 'lhs = '+str(copt_unop)+'(ft)'
  help += '\n  in which: ft = (freq '+str(copt_binop)+' time)'
  BM.bookpage_nodes ([ft,lhs,rhs,condeq,rootnode], name='solving_11 nodes', help=help)
  return rootnode
```

Compare this Python code with the MeqBrowser screen-shot in fig 6.2. Note the various MeqParm options in the compile-time popup on the left. The (somewhat

enigmatic) solver plot in the bottom-right panel shows that the MeqSolver has solved for 6 polynomial coefficients, i.e. the top-left triangle of a 3x3 coeff array. The red line shows the convergence after 3 iterations. The condeq plot in the middle-right panel shows that the final difference between right-hand-side (rhs) and left-hand-side (lhs) is very small.

## 6.3 More advanced solving

The simple case of one MeqCondeq and one MeqParm described above already covers a large class of problems, in which an n-dimensional function (e.g. polynomial) is fitted to a given set of data. A more general case is where there is still one MeqCondeq, but there are multiple MeqParms, which may be on either side of the MeqCondeq. The number of possibilities now rapidly increases:

- The MeqSolver will solve for its internal list of MeqParms, which may be a subset of the available ones. The others behave as constants, i.e. they return domain cell values by using all the information that they have available.
- The degree (i.e. the nr of coeff) of the function to be solved for may be specified for each MeqParm separately.
- The solver may be instructed to make a separate solution for multiple tiles of a large Request domain. Again, this may be specified for each MeqParm separately. The default is a single solution over the entire domain. The advantage of *sub-tiling* is that it significantly reduces the node-overhead, because fewer (larger) domains are needed to process a particular observation. The disadvantage is that all the sub-tile solutions are made simultaneously, so that the solution of a sub-tile cannot be used as starting value for the next.
- The cells of a Request domain do not have to be contiguous. Thus, a solution may be made over a large domain, using only one in 100 (say) data-samples in time or freq. This makes it possible

The most general case is of course where the MeqSolver has multiple MeqCondeq children, each with rhs and lhs children that represent subtrees with multiple MeqParms somewhere upstream. The various subtrees will often be connected upstream, i.e. they share nodes, particularly MeqParms. This is the typical case for radio astronomy, where $N(N-1)/2$ interferomeres each have their own MeqCondeq, which share only $N$ antenna subtrees (in the form of socalled Jones matrices) with instrumental parameters.

Figure 6.2: *The MeqBrowser with the TDL script ROOT_solving.py. Compare the expanded tree on the left with the Python code reproduced in section 6.2.*

# Chapter 7

# More on the Runtime menu

In order to encourage modularity and script readability, the actual tree execution code of the _tdl_jobs in the various TDL scripts resides in separate Python modules (which are also TDL scripts, of course). They have been written specifically for the Beginners Guide, and are in the same directory, for you to inspect, *or to copy and cannibalize for your own projects if you like* (perhaps MeqTrees should offer an official version of this kind of function at some point). We feel that the best way to explain what they do is to reproduce their code here.

## 7.1 .execute_request()

The function *.execute_request(mqs)* resides in the Python module *execute_request.py*. Note that this module also defines a menu of Runtime TDL options for customizing the Request domain(f,t). This menu automatically appears in the runtime popup window when this module is imported.

Note the use of a request_counter, which is incremented each time. This is to make sure that the nodes are actually executed again, rather than just returning the contents of their cache (which they do if they get the same Request again). The forest state parameter cache_result is set to 100, to make sure that all nodes cache their results, for you to inspect.

```
"""
file: ../beginners_guide/execute_request.py
"""
from Timba.TDL import * from Timba.Meq import meq
request_counter = 0

TDLRuntimeMenu("Parameters of the Request domain:",
               TDLOption('ropt_num_freq', 'nr of freq channels',
                         [10,11,1,20,50,100], more=int,
                         doc='nr of domain cells in the freq direction'),
               TDLOption('ropt_num_time', 'nr of time channels',
                         [11,10,1,20,50,100], more=int,
```

```
                  doc='nr of domain cells in the time direction'),
              TDLMenu("time size:",
                      TDLOption('ropt_t1', 'start time (s)',
                                [1.0,0.0,-1.0], more=float,
                                doc='min time (s) of the domain (edge)'),
                      TDLOption('ropt_t2', 'stop time (s)',
                                [10.0,1.0], more=float,
                                doc='max time (s) of the domain (edge)'),
                      ),
              TDLMenu("freq size:",
                      TDLOption('ropt_f1', 'start freq (Hz)',
                                [1.0], more=float,
                                doc='min freq (Hz) of the domain (edge)'),
                      TDLOption('ropt_f2', 'stop freq (Hz)',
                                [11.0], more=float,
                                doc='max freq (Hz) of the domain (edge)'),
                      )
              )

def execute_request (mqs, node,
                     freq_offset=0.0, time_offset=0.0,
                     trace=False):
  """
  Execute the given node with the specified time-freq domain (size and cells)
  The (optional) freq and time offsets are fractions of the domain size.
  """

  foffset = (ropt_f1-ropt_f2)*freq_offset
  toffset = (ropt_t1-ropt_t2)*time_offset
  domain = meq.domain(ropt_f1+foffset, ropt_f2+foffset,
                      ropt_t1+toffset, ropt_t2+toffset)
  cells = meq.cells(domain, num_freq=ropt_num_freq,
                    num_time=ropt_num_time)

  global request_counter
  request_counter += 1
  rqid = meq.requestid(request_counter)
  request = meq.request(cells, rqid=rqid)
  result = mqs.execute(node, request)
  return None
```

## 7.2 .execute_sequence()

The function *.execute_sequence(mqs)* resides in the Python module *execute_sequence.py*. It executes a sequence of Requests, with different domains(f,t). It uses the function *.execute_request()* described above, and also imports its Runtime menu. The present module defines an extra Runtime menu with TDL options for customizing the sequence parameters, i.e. the number of steps, and the size of the strides (as a fraction of domain size) in the freq and time directions.

```
"""
file: ../beginners_guide/execute_sequence.py
"""
from Timba.TDL import *

TDLRuntimeMenu("Parameters of the Request sequence:",
               TDLRuntimeOption('ropt_num_steps', 'nr of steps',
                                [3,10,30], more=int,
                                doc='nr of steps in the sequence'),
               TDLRuntimeOption('ropt_tstep', 'time-step (fraction)',
                                [1.0,0.0,0.1,0.5,-0.1], more=float,
                                doc='fraction of the domain time-size'),
```

```
              TDLRuntimeOption('ropt_fstep', 'freq-step (fraction)',
                               [0.0,1.0,0.1,0.5,-0.1], more=float,
                               doc='fraction of the domain freq-size'),
              )

def execute_sequence (mqs, node, trace=False):
  """
  Execute a sequence of 'ropt_num_steps' requests, each shifted by
  the specified fraction of the domain (as specified in execute_request().
  """
  import execute_request as ER
  for i in range(ropt_num_steps):
      ER.execute_request (mqs=mqs, node=node,
                          freq_offset=i*ropt_fstep,
                          time_offset=i*ropt_tstep,
                          parent=parent, trace=trace)
  return None
```

# Chapter 8

# Some more advanced topics

Where will this ever end? Haven't I suffered enough? *(Calvin)*

Just to whet (wet?) your appetite, we will touch on some advanced topics that illustrate the power of MeqTrees. There are more, and more to come.

## 8.1 Extra domain dimensions

The default domain is 2-dimensional (freq and time), which covers the vast majority of cases. However, MeqTrees would not be MeqTrees if it did not offer the possibility of specifying an arbitrary number (up tp 10) of arbitrary dimensions. Two things are needed here. First of all, the Request needs a domain with the desired domensions. In this example, we retain the standard dimensions freq and time, and add two extra ones, L and M.

```
"""
file: ../beginners_guide/execute_general_request.py
"""
from Timba.TDL import *
from Timba.Meq import meq
request_counter = 0

def execute_general_request (mqs, node,
                             dsize=record(freq=(1,10), time=(1,10), L=(-1,1), M=(-1,1)),
                             ncells=record(num_freq=20, num_time=21, num_L=9, num_M=9),
                             trace=False):
  """
  Execute the given node with a domain with an arbitrary number of dimensions,
  i.e. different from the default freq-time (see execute_request() above).
  The size of the domain in the various dimensions is specified by the record dsize.
  The number of cells for the various dimensions is specified by the record ncells.
  """

  domain = meq.gen_domain(**dsize)
  cells = meq.gen_cells(domain, **ncells)

  global request_counter
  request_counter += 1
```

```
  rqid = meq.requestid(request_counter)
  request = meq.request(cells, rqid=rqid)

  result = mqs.execute(node, request)
  return None
```

We can issue such a Request to any tree, without any noticeable difference. The extra dimensions are interpreted as constants. However, if the tree has some nodes whose Results vary over the extra dimensions, things get more interesting. We can achieve this by means of the MeqGrid nodes, which do the same as MeqFreq or MeqTime, but for an arbitrary axis. The tree definition function *4D()* now looks like:

```
"""
file ../beginners_guide/domain_4D.py
"""
from Timba.TDL import * import make_bookmark as BM


TDLCompileMenu(".domain_4D():",
               TDLOption('copt_MeqFreq', 'add MeqFreq', True,
                         doc='if True, add a Meq.Freq node'),
               TDLOption('copt_MeqTime', 'add MeqTime', True,
                          doc='if True, add a MeqTime node'),
               TDLOption('copt_MeqGridL', 'add MeqGrid(axis=L)', True,
                          doc='if True, add a MeqGrid node for dimension L'),
               TDLOption('copt_MeqGridM', 'add MeqGrid(axis=M)', True,
                          doc='if True, add a MeqGrid node for dimension M'),
               toggle='copt_4D')


def domain_4D (ns):
  """
  This function makes a sum of leaf nodes with variation over
  different dimensions like the standard MeqTime and MeqFreq, but also
  some extra dimensions L and M. The latter are specified with the
  MeqGrid(axis='L'), which is a generalized version of MeqFreq/Time.
  When a request is issued with a 4D domain (freq,time,L,M), the
  result will be 4D also.
  """

  # Make a list of the child nodes specified by TDLOptions:
  cc = []
  if copt_MeqFreq:
    cc.append(ns.MeqFreq << Meq.Freq())
  if copt_MeqTime:
    cc.append(ns.MeqTime << Meq.Time())
  if copt_MeqGridL:
    cc.append(ns.MeqGridL << Meq.Grid(axis='L'))
  if copt_MeqGridM:
    cc.append(ns.MeqGridM << Meq.Grid(axis='M'))

  # Make sure that there is at least one child, and add them all:
  if len(cc)==0:
    cc.append(ns.zero << 0.0)
  rootnode = ns.sum << Meq.Add(*cc)

  # Finishing touches:
  BM.bookpage_nodes (cc+[rootnode], name='constituent nodes', help=None)
  return rootnode
```

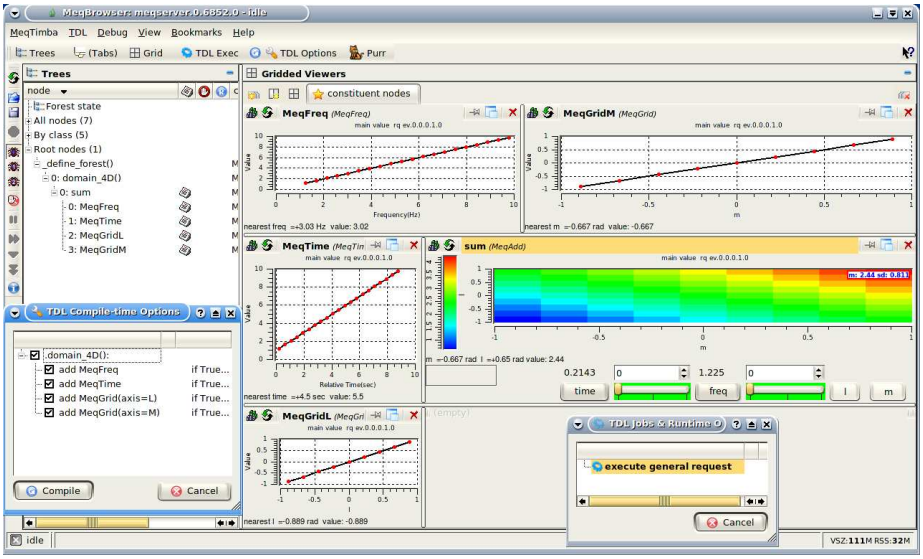


Figure 8.1: *The MeqBrowser showing a Result of executing a tree with a 4D domain. Note the extra controls for the extra dimensions in the display window.*

## 8.2 Avoiding node-name clashes

Since nodes are identified by their names, these must be unique. This is not as easy as it sounds, especially since serious trees will usually be built up with the help of tree-definition modules that are written by someone else. Such modules will define nodes with names that made sense to the developer, and which may easily clash with the node names in your own or other modules.

One way to avoid nodename clashes is to use subscopes, which define a (hopefully) unique namespace by prepending all nodenames with the name of the subscope. A subscope may be derived from a nodescope (ns) or another subscope (ss). It is then used just like a regular nodescope in the definition of MeqNodes. Obviously, it can also be passed to other tree-definition functions instead of the regular nodescope.

```
ss1 = ns.Subscope('sub1')              # derive a subscope object from the nodescope
print 'type(ss) =',type(ss1)           # --> <class 'Timba.TDL.TDLimpl.NodeScope'>
e1 = ss1.e1 << 1.0                     # --> sub1::e1(MeqConstant)
ss2 = ss1.Subscope('sub2')             # subscopes may be derived from another subscope
e2 = ss2.e2 << 2.0                     # --> sub1::sub2::e2(MeqConstant)
ss3 = ns.Subscope('sub3')('aa')(6)     # subscopes may also be qualified
e3 = ss3.e3 << 3.0                     # --> sub3::e3:aa:6(MeqConstant)
```

Another strategy for reducing the chance of nodename clashes is to use tree-definition functions that accept a nodestub (initialized or not) rather than a nodescope, and then generate new nodes by qualifying the given nodestub. The chance of nodename clashes is greatly reduced if the given nodestub has an

unique name. This is the purpose of the helper function *.unique_nodestub(ns, name)* in module *unique_nodestub.py*:

```
stub = UN.unique_nodestub(ns,'g')     # --> g(MeqConstant)      # initialized stub
g1 = stub(1) << 1.0                   # --> g:1(MeqConstant)
g2 = stub(-2.0) << -2.0               # --> g:-2.0(MeqConstant)
stub = UN.unique_nodestub(ns,'g')     # --> g|(MeqConstant)     # initialized stub
stub = UN.unique_nodestub(ns,'g')     # --> g||(MeqConstant)    # initialized stub
g3 = stub(3) << 3.0                   # --> g||:3(MeqConstant)
```

Here is the code of the function *.unique_nodestub(ns, name)*, which is used in the example of *ROOT_node_definition:*

```
"""
file:  ../beginners_guide/unique_nodestub.py
... description of this module
... copyright statement
"""

from Timba.TDL import *
stubtree = None                # global variable, used below

#-------------------------------------------------------------
def unique_nodestub(ns, name, quals=[], kwquals={}, level=0):
  """
  Helper function to generate a unique nodestub with the given
  (node)name, qualifiers (quals) and keyword qualifiers (kwquals).
  If it exists already, its name (or qualifiers) will be changed
  recursively, until the resulting stub is not yet initialized.

  A stub becomes a node when it is initialized: stub << Meq.Node(..).
  But since our unique stub will already be initialized (see below),
  it must be qualified to generate nodes with unique names:
  - import unique_nodestub as UN
  - stub = UN.unique_nodestub(ns, name [, quals=[..]] [,kwquals={..}])
  - node = stub(qual1)(..) << Meq.Node(...)

  Since the stub is unique, chances are good that there will not be
  any nodename clashes if we generate nodes by qualifying it. This is
  not guaranteed, of course. But the chances are maximized by
  consistently generating ALL nodes by qualifying unique nodestubs.

  The new nodestub will be initialized to an actual node so that it
  can be recognized later. To avoid a clutter of orphaned rootnodes
  in the MeqBrowser the initialized stubs are attached to a tree with
  a single (orphaned) rootnode named 'StubTree'.
  """

  # Make the nodestub:
  stub = ns[name](*quals)(**kwquals)

  # Make sure that the nodestub is unique (recursively):
  if stub.initialized():             # the stub represents an existing node
    nn = name+'|'                    #   change the name
    stub = unique_nodestub(ns, nn, quals, kwquals, level=level+1)

  # Initialize the stub, and de-orphan it:
  if level==0:
    global stubtree
    if not is_node(stubtree):
      stubtree = ns['StubTree'] << Meq.Constant(-0.123456789)
    stubtree = stub << Meq.Identity(stubtree)

  # Return the unique nodestub:
  return stub
```

Figure 8.2: *Matrix operations, using ROOT_myThirdTree.py, and selecting the matrix22 tree definition module.*

## 8.3 User-defined nodes

pyNodes, private functions

## 8.4 Matrix operations

As we have seen, a node Result may have more than one primary vellset. The MeqComposer node combines the Results of its children into a list of primary vellsets. Such a list may have a shape by specifying e.g. dims = [2,2]. This opens the possibility of matrix operations, for which MeqTrees has a number of specialized nodes. They are demonstrated in ROOT_myThirdTree.py in section 5.3. As promised there, we here reproduce the two tree definition functions dealing with matrix operations.

# Chapter 9

# Now that you are no longer a Beginner....

> Do not ask what MeqTrees can do for you, ask what you can do for MeqTrees. *President Kennedy, the Obama of his day*

All these little Mickey Mouse exercises are of course very nice and instructive, but you should by now be chomping at the bit to use MeqTrees for some real applications. The good news is that MeqTrees is now mature enough for a certain class of people to do great things with it. But those who just hope to crank a more sophisticated handle than the existing packages can offer are advised, regretfully, to continue cranking the old handles until the MeqTrees vanguard has created some more polished applications for them.

We envisage a converging process. On the one hand we are working very hard to lower the threshold for general users. For instance by providing *frameworks*, i.e. groups of modules that define standard subtrees that can be easily cobbled together, perhaps even by means of a tree-building GUI. Such modules would still be smaller, and thus more flexible, than the programs in a traditional package like AIPS. But they would hide much of the complexity of building up large trees from the basic MeqNodes. On the other hand, we are encouraging the above-mentioned *certain class of people* to join our little group in generating such frameworks, and the many other tools that are made possible by MeqTrees (see some suggestions below). Of course they should do this in the context of their own projects.

Since MeqTrees was originally developed for radio astronomy, we expect that that will be the area of greatest activity in the foreseeable future. Its primary role will be in simulation, and in developing the urgently needed algorithms for *3rd generation* calibration. For the moment, MeqTrees is the only way to implement the Measurement Equations of the new telescopes, and to solve for their parameters. The very short TDL turn-around time (hours rather than

months), and the easy exchange of TDL scripts (and PURR logs) between user-developers, should greatly increase the *rate of evolution* of radio astronomical data reduction software. This is by no means a luxury, after two decades of stagnation, and with up to ten new telescopes being built.

Fortunately, because of the generic nature of the Measurement Equation, there is a lot of communality between the many radio-astronomical projects, so there should be lot of scope for working togther also. For instance, the data-processing trees for many telescopes only differ substantially in their *Jones matrices*, which contain their specific instrumental model. These are 2x2 tensor nodes, in which each matrix element is a subtree crowned with specific MeqParm leaf nodes. MeqTrees already offers one framework of modules for radio-astronomical data processing, and it would be quite straightforward to make some kind of Jones Repository, where developers from all over the world could exchange TDL scripts with their favourite Jones matrices. They could also exchange their PURR logs, which are the results of a strangely convenient tool for keeping track of the many files (inages, plots, etc) generated in a data reduction project, and to assemble a log, with pictures and comments, of the various processing steps. What about that for a beckoning vista?

So, if you belong to a certain class of people, you will not ask what MeqTrees can do for you, but what you can do for MeqTrees. Some suggestions: New kinds of visualization, flagging, Jones matrices for specific telescopes, specific processing scripts, help with the (presumably web-based) Script Exchange Mechanism and the Parameter Management System, etc, etc. Your rewards will be many, ranging from enhanced employability, by any of the new radio telescopes, to the admiration, affection and envy of your peers. In the end, it all comes down to comportment, as defined in the other indispensable book [5] that has sustained and inspired us all these years.

Figure 9.1: *MeqTrees is expected to play an important role in the development of simulation and calibration algorithms for the next generation of giant radio telescopes (LOFAR, ASKAP, MEERKAT, ATA, PAST, PAPER, eVLA, eMERLIN, APERTIF, EMBRACE, SKA). First of all, it can implement an arbitrary Measurement Equation, which is needed for the rather unorthodox new telescopes. Secondly, because of TDL, it offers a very short turn-around time for experimentation (hours rather than months). And thirdly because the ease of exchanging TDL scripts and PURR logs significantly lowers the threshold for inter-telescope collaboration (and creative criticism). All this might just accelerate the rate of evolution enough to sort of meet the largely political deadlines of the new telescopes.*

# Bibliography

[1] *The MeqTrees Wiki,* URL: `http://www.astron.nl/meqwiki`

[2] Noordam J.E., Smirnov O.M. MeqTrees, *A software module for implementing an arbitrary Measurement Equation and solving for its parameters* MeqTrees document .... (2009)

[3] Smirnov O.M. *The TDL Bedside Companion* MeqTrees document ... (2008)

[4] Adams D. *The Hitchhiker's guide to the Galaxy* (1979) Pan Books, London and Sydney

[5] Everitt D., Schechter H. *The Manly Handbook* (1983) Berkeley Books, New York

# Appendix A

# Bookmarks

We have seen that all nodes in the tree may be visualized in various ways by clicking on them the left (Trees) section of the MeqBrowser. This is very nice, but a little cumbersome. Therefore, MeqTrees offers a Bookmarks menu, which offers shortcuts to displaying (groups of) nodes. When you click on a bookmark item, one or more panels will appear in the right (Gridded Viewers) section. Each panel is associated with a specific node, and a specific viewer. The default viewer is the Result Plotter, and those panels will of course be blank if the tree has not been executed yet. The Record Browser viewer displays the node state record. Explanatory text may be displayed by the QuickRef Display viewer, which uses the contents of the quickref_help field of a node state record. Different viewers may be invoked interactively by right-clicking on a panel.

View panels that have been specified via a bookmark are automatically set in *publishing mode*. This means that they will change every time their nodes get a new Result, which is very convenient, and fun to watch. For view panels that are invoked by clicking on a node in the tree, the publishing mode has to be toggled explicitly.

Since bookmarks are such a useful feature, all the examples in this Beginners Guide have them. To increase readability, and because the syntax is a little arcane, we have supplied some simple functions in the file *make_bookmark.py,* to make it easy to specify (pages of) bookmarks. We are reproducing (some of) them here, to give you the basic idea. However, if you have a computer handy, we recommend that you study the module itself.

The next simplest way to make a bookmark for a specific node or page is to use the *Add bookmark for ...* or *Add pagemark for ...* items in the Bookmarks menu. The next simplest way to make a bookmark for a specific node is to use the following function:

```
def make_bookmark (node, name=None, viewer='Result Plotter'):
  """
  The simplest possible bookmark function. Make a bookmark for the given node,
  using the specified viewer. If no name is specified, use the node name for
```

```
  the bookmark menu. NB: A single bookmark like this (i.e. not a bookpage) will
  be displayed in an empty panel of the current pookpage.
  """
  bm = bm_node_result (node, name=name, viewer=viewer)
  append_bookmark(bm)
  return None
```

This uses the following helper functions (also in *make_ bookmark.py*):

```
def bm_node_result (node, name=None, viewer='Result Plotter'):
  """
  Return the bookmark record for the result of the given node,
  using the specified viewer. If a name is specified, use that
  for the bookmark name, otherwise use the node-name.
  """
  bm = record(name=str(name or node.name), viewer=viewer, udi='/node/'+node.name)
  return bm

def append_bookmark(bm):
  """
  Make sure that the forest_state record hase a 'bookmarks' field
  of type list, and that the cache_policy is set to 100 (='always').
  Then append the given bookmark record (bm, may be single or page).
  """
  rr = Settings.forest_state            # the forest state record
  key = 'bookmarks'                     # its relevant field name
  if not rr.has_key(key):
    rr.bookmarks = []
    rr.cache_policy = 100
  elif not isinstance(rr[key],list):
    rr.bookmarks = []
    rr.cache_policy = 100
  rr.bookmarks.append(bm)
  return None
```

It is often desirable to show the results of a group of related nodes on a single page. One way of doing this is the function *.make_ nodepage()*, which shows the result of the specified node and its state record. If a help string has been provided, it is attached to the quickref_help field of the node, and displayed also. Finally, if one or more other nodes are given, their results are displayed also.

```
def bookpage\_nodes (nodes, name=None, help=None):
  """
  Make a (bookmark for a) bookpage for the results of the given
  list of node(s). If no name is provided, a default name is used.
  If help is provided, display it in an extra panel of the bookpage.
  """
  pp = []
  if isinstance(help,str):
    pp.append(bm_node_help(nodes[-1],help=help))
  for node in nodes:
    pp.append(bm_node_result(node))
  if not isinstance(name,str):
    name = nodes[0].name+' ...'
  make_bookpage(pp, name=name, nodes=nodes)
  return None
```

The various helper functions that are used are not shown here, but they may be inspected in *make_ bookmark.py*. There are some more functions there, which provide enough detail for you to contemplate designing some bookpages of your own.

# Appendix B

# Examples of node (and forest) state records

Even though the detailed discussion of the various fields of the node and forest state records are outside the scope of this Beginners Manual, we know that at least some of you will find them very illuminating. The possibility of inspecting the forest state record, and any node state record, by simply clicking on it in the MeqBrowser is part of the unique MeqTree emphasis on visualization. MeqTrees. But since this manual is meant to be read by itself, without a computer handy, we hereby reproduce the screenshots of a few examples of state records, for your insightful perusal.

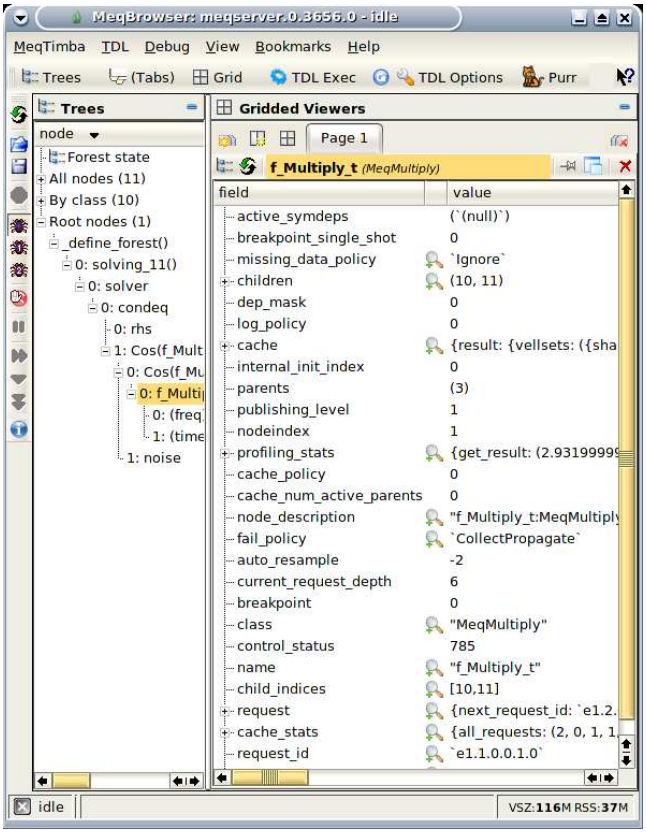


Figure B.1: *Example of a complete node state record. It may be inspected for any node by clicking on it in the left (tree) section of the MeqBrowser. The cache (result) and request fields for different types of nodes are shown in more detail in figs B.2 and B.3. A full description of all its fields is outside the scope of this Beginners Manual, but you are encouraged to guess.*

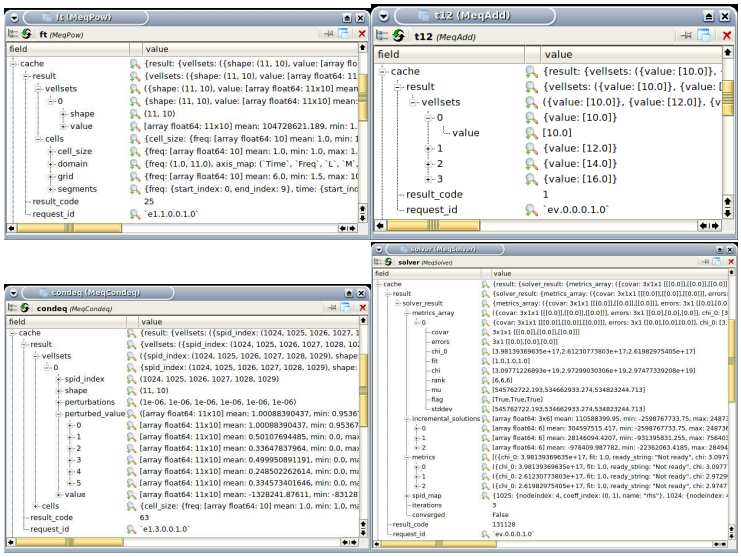

Figure B.2: *The expanded cache (result) fields of the state record of four different types of nodes. Top left is a simple scalar node, with only a single vellset, and no perturbed values. Top-right is a tensor nodes, with multiple vellsets. Bottom-left is a MeqCondeq node, with extra vellsets for the perturbed values that are used in calculating the (numerical) partial derivatives used in condition equations (see section ). Bottom right is a MeqSolver result, which also contains vectors of solution metrics, for all iterations.*

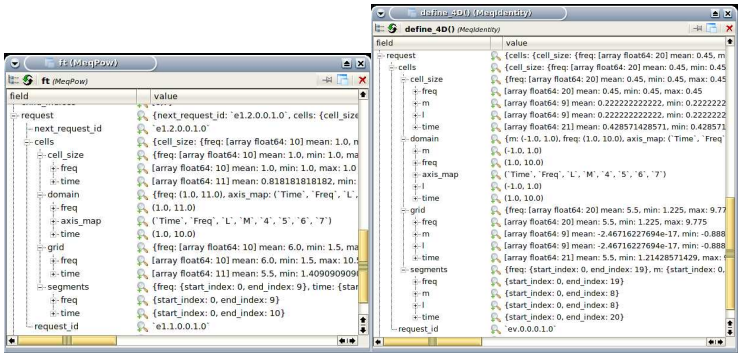

Figure B.3: *The expanded request fields in the state records of two different nodes. On the left is a 'normal' request with a 2-dimensional (freq,time) domain and a regular cell structure. On the right is the 4-dimensional domain used in the example that is discussed in section* 8.1.

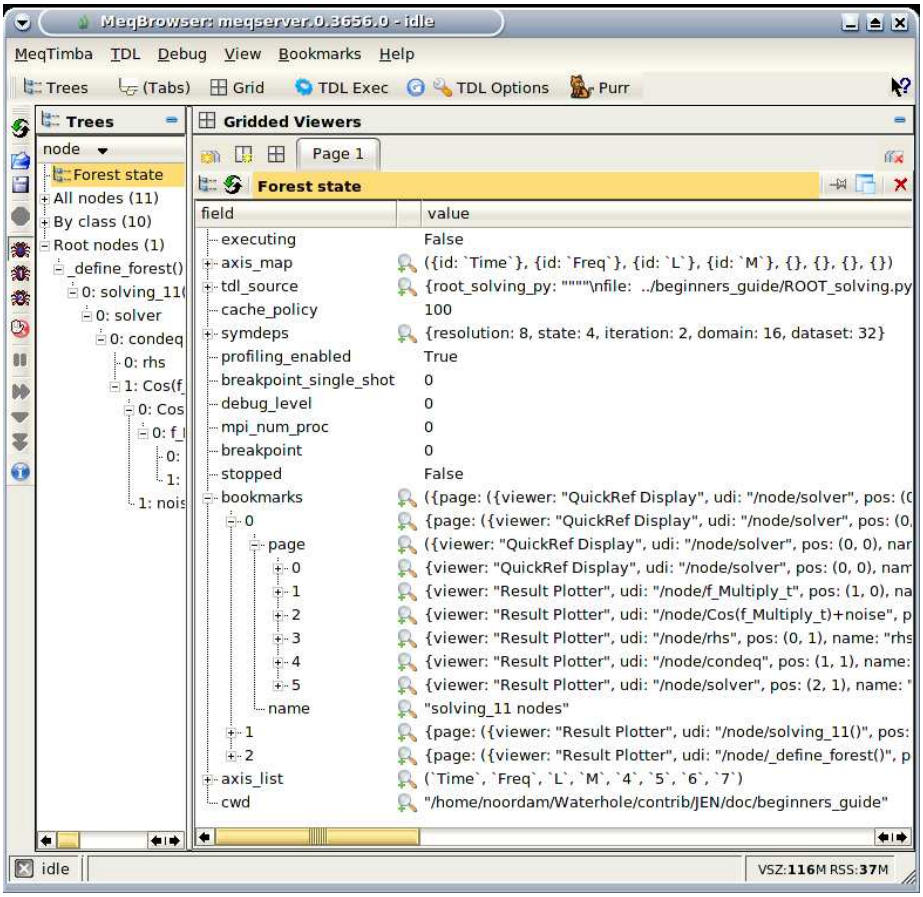


Figure B.4: *The forest state record. It may be inspected by clicking on it at the top of the left (tree) section in the MeqBrowser. A full description of its fields is outside the scope of this Beginners Manual, but you are encouraged to guess.*

# Appendix C

# Available MeqNode classes

Perhaps the most concise description of the capabilities and possibilities of MeqTrees for the discerning user is an overview of the node classes that are available already, and the ones that are desirable in the near future:

- LEAF NODES (nodes without children): Constant, Parm, Freq, Time, Grid, GaussNoise, RandomNoise, Spigot, FITSImage (and several other FITS interface nodes)
- UNARY OPS: (one child): Exp, Log, Abs, Invert, Negate, Sqrt, Pow2(8), Sin, Cos, Tan, Acos, Asin, Atan, Cosh, Sinh, Tanh, Norm, Arg, Real, Imag, Conj, Ceil, Floor, Identity. See also section 3.3.
- BINARY OPS (two children): Subtract, Divide, Pow, Mod, ToComplex(real,imag), Polar(ampl,phase).
- ACCUMULATION OPS (one or more children): Add, Multiply, WSum, WMean. The last two need a vector of weights.
- CELL STATISTICS (one child): Sum, Mean, Product, StdDev, Rms, Min, Max, NElements. They perform cell statistics, along selected axes. The default is all axes, in which case the Result is a scalar.
- TENSOR OPS: Composer, Selector, Paster. Tensor nodes are nodes with multiple vellsets in their Result.
- MATRIX OPS (2D tensors): Transpose, ConjTranspose, MatrixMultiply, MatrixInvert22. The latter operates on 2x2 matrices only, which is sufficient for the RME.
- FLOW CONTROL: ReqSeq, ReqMux, Sink, VisDataMux. They regulate the order in which their children get Requests, and which Result to pass on. They also synchronize the flow of Requests and Results in parallel trees.

- DOMAIN CONTROL: ModRes, Resampler, CoordTransform. ModRes modifies the Request before it is passed on, Resampler modifies the Result itself. CoordTransform modifies the Request passed to its second child.
- FLAGGING: ZeroFlagger, MergeFlags.
- SOLVING: Condeq, Solver, Parm. For the moment, MeqTrees offers only a Levenberg-Marquard non-linear solver.
- VISUALIZATION: All nodes, DataCollect, Inspector(=Composer).
- COORDINATES (mostly radio astronomy): UVW, LMN(radec, radeq0), AzEl(radec, xyz), RaDec(azel, xyz), LMRaDec(lm), ObjectRaDec(name), LST(domain, xyz), ParAngle(radec,xyz), LongLat(xyz). The vector *xyz* is an Earth position in IRTF coordinates.
- TRANSFORMS: FFTBrick, UVInterpol (collectively known as UVBrick).
- USER-DEFINABLE NODES: PyNode, Functional, PrivateFunction. These allow the user to insert his or her own function, written in Python or C++, to read the Results from one or more child nodes, and to generate a customized Result.

A full description of these nodes is outside the scope of this paper. See, for example, the Wiki [1]. We strive towards a core collection of nodes that offer basic functionality, surrounded by collections of more specialised nodes for use in specific areas. The latter should be mostly contributed by users. Some of the contributed nodes will eventually find their way into the core collection, while some of the present nodes will be moved out. Obviously, we will have to solve the technical problem of linking such contributed nodes into MeqTrees with a minimum of fuss.

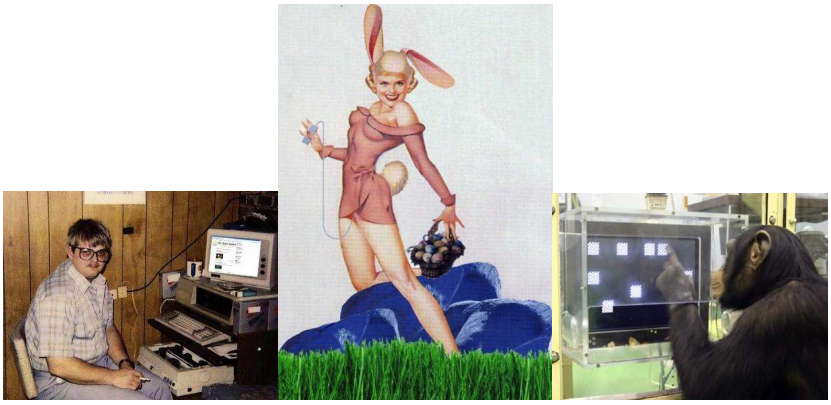

Figure C.1: *MeqTrees can be used for any problem with a Measurement Equation, but it has originally been developed for radio astronomy. In that particular field, the development of data reduction software for Radio Astronomy has been held back by an unfortunate class distinction between developers (left) and users (right). MeqTrees has been designed to encourage a new class of user-developers (middle), propelled by the instant gratification caused by the short time it takes to do exciting new experiments, and the ease of finding partners to share them with. For the moment, however, we are still old-fashioned developers, working away at our dream.*