PyNode Visualisation

(Very Brief) Introduction to PyNodes

MeqTrees provides a large number of pre-defined nodes which you can use to build your trees, but what if you requires some specific functionality which is not present in it? Well of course you will have to write your own node, but a special class Is provided which will make this task easier. This is the PyNode, which allows you to write your node logic in python. These nodes can then be used to build trees as usual.

```
from Timba import pynode

class MyPyNode(pynode.PyNode):
    def __init__(self, *args):
  super(MyPyNode, self).__init__(*args)

 def update_state(self, mystate):
    pass

 def get_result(self, request, *children):
    pass
```

The above interface defines how a custom PyNode would probably look like. The constructor can be omitted, however if you do require one then be sure to call the parent constructor first. The update_state is called by the server whenever the node's state is updated, or needs to be initialised. Here you can define or updates values within the node's state (the same variables will be created and update in the node's python instance as well). The last method is required in every PyNode, and is the method which gets called when a new request is issued. This method should return the result object with the new vells.

Using PyNodes to Create Visualisations

Here we present a new visualisation option in MeqTrees, where users are able to develop their own specialised plots easily. To do this, three special components are required:

– A base PyNode class, which provides helper functions which allow the user to specify what to plot, and how to plot it

– A helper class which encapsulates your PyNode's children and provides various method of extracting their data, specifying subsets, expand vellset ...

– A new plugin in the meqbrowser, which is able to read special 'instructions' in the result object stored within your PyNode and create a plot out of it.

To create your own plot, all you have to do is create a class which inherits from PyBasePlottable.py, define what values you want to use from your children, define curve and plot properties and let the system to the rest. A tutorial of how to do all this now follows.

How to Create Your Own Plotters

Here we will create a typical plotter, which will create a Scatter Plot from its child nodes plotting the vellset means against the 'node's number', with different colors for each vellset within a result:

- ***Define your tree.*** Note that a PyNode can only process results from its immediate children, therefore you have to make sure that any nodes you want to process have your plotter as a parent. It usually might be the case that you need different nodes to be plotted on the axes, and for this you have to make sure that the number of nodes for each axis matches. For examples, you might want to plot the mean or standard derivation of some particular nodes with respect to the baseline distance. The distances can be read from specific nodes, which can be 'mapped' to the x axis. The nodes of interest are then mapped to the axis.

  Generally you would want to plot results from existing trees, without messing around with its tree-building logic. The way forward would be to use the node-searching mechanism to get the nodes of interest. Also, you will need to specify which nodes (by index) will be used to plot which axis, and it would be useful to keep track of the tree that was built and store the sequence of the PyNode's children. Here we will just create 25 MeqComposer nodes, each containing two random MeqConstants. Then we assign this tree as a subtree of our custom PyNode, which we're going to define in the next step.

  ```
  import random
  cc = [ns[str(i)] << Meq.Composer(Meq.Constant(rd.randint(0,100)),
                 Meq.Constant(rd.randint(0,100)))
                for i in range(25) ]
  ns.pynode << Meq.PyNode(children = cc, class_name="PyMyPlotter",
  module_name=__file__)
  ```

- ***Create a new PyNode*** which inherits from PyPlottableBase and override the get_result method. The first statement in this method must be a call to the parent's get_result, as follows:

  ```
  from Timba.Contrib.AxM.pyvis.PyBasePlottable import *
  ```

```
class PyMyPlotter(PyPlottableBase):
  def get_result(self, request, *children):
 super(PyMyPlotter, self).get_result(request, children)
```

- ***Create the ResultVector and MeqResult objects***. A ResultVector object is an encapsulation of the PyNode's children, which allows you to assign labels to your nodes and vellsets, as well as provides helper methods to read data from these nodes. The plotter will need such an object to be passed to it. Make sure you keep track of the result subsets you will want to plot. We are also going to label the child nodes with the index id. The MeqResult object is required to attach the plotting information.

```
rv = ResultVector(children, labels = [str(i) for i in
                              range(len(children))])
vells = meq.vells(meq.shape(request.cells))
result = meq.result(meq.vellset(vells), request.cells)
```

- ***Define your axis***. You have to define at least the values of the y-axis (if nothing is specified for the x-axis then the child number is used, unless you are using it in history mode in which case the x-axis will be ignored anyways and the request number will be used). To define an axis you will need to specify at least what values will be read from the result nodes (means, maximums ...). You can also specify a subset of the nodes to get the values from, restrict the vellset dimensionality (which means that only result which comply with this restriction will be used) as well as specify explicitly which vellsets to use from within the results (by vellset index). In this example, we will just plot the means of each node against its 'node label':

```
y_axis = define_axis(expr = 'means')
x_axis = None
```

- ***Define your styles***. The plotter will allow to define a curve for each vellset, or if you're using the history plotter you can define a curve for each result, and for each vellset within each result. When no curves are defined then the default curve will be used. Here we have two vellsets within each node, so we're going to define two curves, one for each vellset, where each curve will have a different color (both symbols and lines) and the same symbol shape:

```
curves = create_curves(2, colors = [Colors['red'], Colors['blue']],
        symbols = [Symbols['cross'], Symbols['xCross']],
        show_labels = True)
```

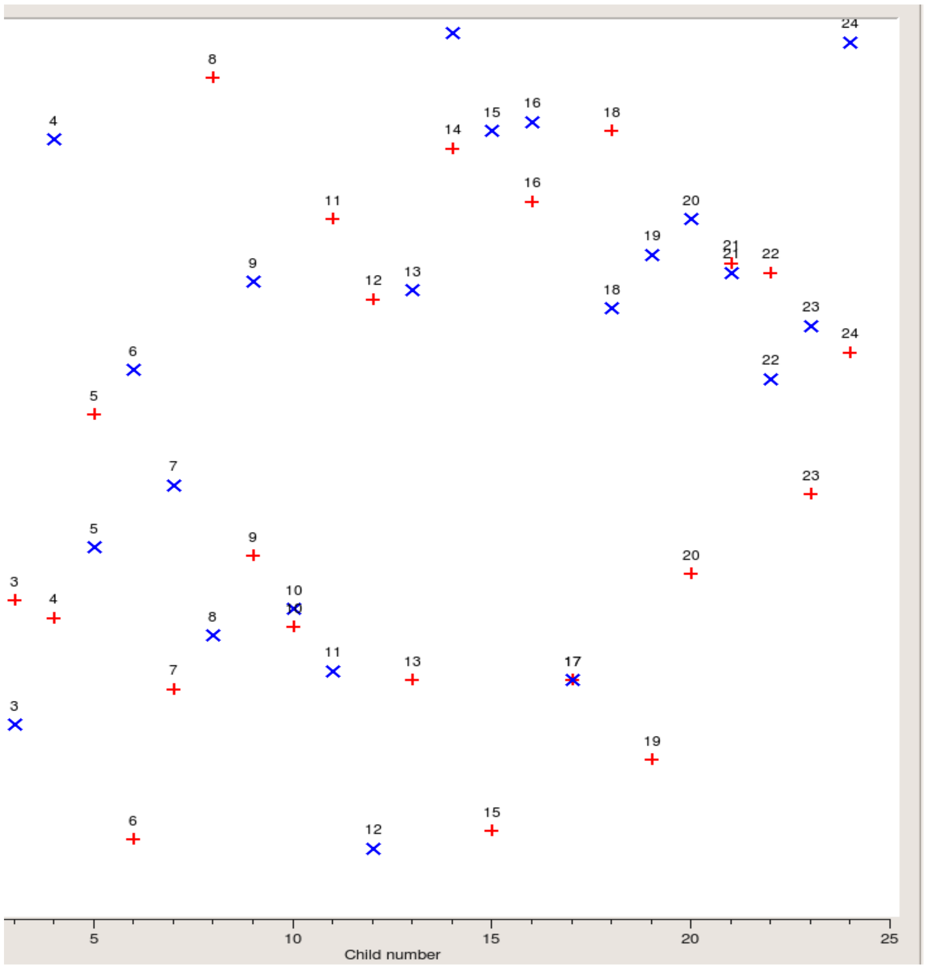
- ***Customise your plot***. The plotter will also allow you set some plot properties, such as the background color and axis properties. Default properties are used if nothing is specified:

```
plot = PlotProperties(axis = [ create_axis(
      axis_id =  AxisId['xBottom'], title = 'Child number'),
      create_axis(axis_id = AxisId['yLeft'], title = 'Means') ],
      title = 'Scatter Means Plot')
```

- ***Create the PyResult***. Once all of the above properties have been set, all that's left is to let the plotter extract the results, assign the properties and attach this to the MeqResult:

```
return self.attach_pyresult(result, rv, y_axis, plot = plot,    curve = curve)
```

Well done, you have created your first plotter, which is depicted in the figure below. The positions of the points will of course be different, but if anything else is then you should probably look through your code and try to see what went wrong



Moving On

The previous section described how to go about creating a simple custom plotter. Now we will build up on this and describe some of the more 'advanced' capabilities which the PyBasePlottable offers. We will start be describing all the arguments which can/must be passed to attach_pyresult.

```
def attach_pyresult(self, meqresult, result_vector, yvalues, xvalues =  None, curves = [],  plot =
      None, separate_complex = False)
```

- meqresult is the MeqResult object to which the PyResult will be attached
- result_vector has to be an instance of ResultVector and contains the child node results from which the values to plot will be extracted
- xvalues and yvalues represent the values to be extracted for the x and y axis. These are dictionaries which can have the following keys:
  - expr is the method to be called from the Result object to get the valus ('means', 'maxs'...)
  - rsubset defines a subset from the ResultVector to use. This must have the form of a list of indices
  - vsubset defines a subset from the Vells within the Result object, again be specifying the vellset indices
  - dims restricts Results to those containing only a certain vellset dimensionality

  A helper method, define_axis, is provided which creates these dictionaries in an easy way.

  NOTE: If xvalues is None then the Result objects will just be enumerated
- curves is a list of curve properties objects (no curve, one curve, or one curve per dataset allowed)
- plot is an object containing extra information for the plot, such as axis, title, background color etc...
- separate_complex defines the behaviour of the plotter when the results contain complex numbers:
  - if False then if there are complex value in the data an argand diagram is create.
  - if True then a separate curve for the real & imaginary parts is created
  - if None then all complex value are ignored

The attach_pyresult_history method is similar to attach_pyresult, except for the fact that when calling this function then the results for subsequent results will be cached and a history plot will be generated.

There are many helper method and classes which will make your life easier, including:

– The CurveProperties class, which allows you to define how the curve will look like. Properties which you can define include:

  - the curve style: let the plotter know if it should draw the curve as a line, in steps, as dots or no line at all:

    CurveStyle['none' | 'lines' | 'sticks' | 'steps' | 'dots']

  - the curve_attribute, which defines if the curve will be fitted or not

    CurveAttribute['inverted' | 'fitted']

  - show_labels which if True will draw a label (the Result object name) above each point on the plot

  - pen which defines the color, size and other properties of the curve's outline (not its fill, which is defined in the brush). This is actually a dictionary containing various properties. A helper function is provided to easily create pens, create_pen. This accepts any key-value pairs as arguments, however only the following ones are allowed:

    - color which defines the outline's color:

      Color['red' | 'darkRed' | 'green' | 'blue' | 'cyan' | 'magneta' | 'yellow' | 'lightGray' | 'gray' | 'darkGray' | 'black' | 'white']

    - width which defines the width of the lines

    - style which defines how the line will be drawn:

      PenStyle['dashDotDot' | 'dashDot' | 'dash' | 'dot' | 'solid' | 'none']

    - join_style which defines how to draw the joint between connecting lines:

      PenJoinStyle['bevel' | 'miter' | 'round' | 'svgMiter']

    - cap_style which defines how non-connecting line edges are drawn

      PenCapStyle['flat' | 'square' | 'round']

  - brush which defines the color and style of how the curve will be filled. Like the pen above, this is defined as a dictionary and the function create_brush can be used to create this:

    - color

    - style which defines the pattern with which the curve/symbol will be filled in

BrushStyle[' backwardDiagonal' | ' concialGradient' | 'cross' | 'dense1/2/3/4/5/6/7' | 'solid' | ' crossingDiagonal' | 'horizontalLines' | 'linearGradient' | 'none' | 'radialGradient' | ' verticalLines']

- symbol which will define the shape of the symbol that will be drawn on each datapoint in the curve, as well as its color, size and so on. This is a dictionary as well, and the function create_symbol is provided to help create this. Acceptable keys include:
  - size which is a tuple with two integers which defines the height and width of the symbol
  - pen
  - brush
  - symbol which defines the shape to draw

    Symbols ['none' | 'ellipse' | 'rectangle' | 'diamond' | 'triangle' | 'downTriangle' | 'upTriangle' | 'leftTriangle' | 'rightTriangle' | 'cross' | 'xCross' | 'horizontalLine' | 'verticalLine' | 'star1' | 'star2' | 'hexagon']

- The PlotProperties class, which allows you to define various plot properties:
  - title to be displayed at the top of the plot
  - background_color of the plotting area
  - margin, which is an integer describing how far away from the sides of the plotting area the curve will be
  - axis, which is a list of at most two axis definitions describing how the axis will be displayed. Like the symbol, this is a dictionary which can be created by calling the helper function create_axis:
    - axis_id which refers to a specific axis

      AxisId['xBottom' | 'yLeft' | 'yRight' | 'xTop']
    - title which represents the axis' text

Installing Everything

First of all we need to checkout the code from the subversion repository (you need subversion for this). The code is located within the Waterhole.

We also need to edit the meqbrowser.py script file in /usr/bin so that it will also import the new PyNodePlotter plugin. Go to the statement:

```
importPlugin('quickref_plotter')
```

and add this before it:

```
import Timba.Contrib.AxM.pyvis.pynode_plotter
```

Now you should be able to use the new plugin and create your own plotters.