Trenton Schulz
Inf 3330
Project

GUI Programming with PyQt—By Example

Python has many libraries available for it. This gives people a lot of choice in how they want to accomplish their tasks. One way of writing GUIs was presented in our textbook for this course, Python Scripting for Computational Science [1], using Tkinter. Tkinter is an OK way of writing your GUI, and with some help from Python Mega Widgets is good for solving scientific computing chores. However, it is a bit lacking in the number of widgets you get out of the box and its look and feel on platforms other than X. The primary platform I work on is Mac OS X and the Tkinter scripts look nothing like a native application there. There is also a problem that certain Tcl/Tk parts aren't working on OS X, but that's not necessarily the fault of Tcl/Tk. But it is limiting.

Back when I was getting my bachelor's degree in the U.S., I stumbled upon Qt when I was searching for something, anything, to get me out of the console-based world of programming we all learn. I was very, very impressed with Qt and fell in love with the API. As time wore on, I was able to get a job at Trolltech and have worked on Qt for the last four years. So, naturally, I was very interested in how Qt or more specifically, PyQt, fares in writing GUIs. One way to test out PyQt is to run it over the same problems as the Tkinter scripts that were presented in the textbook. That is what I set out to do and was very successful. What this document will detail is how one can writh the GUI examples presented in the textbook with PyQt. In particular, we will loot at the examples included in scripting-src tarball from the course.

The main way that I decided to do this is similar to how Trolltech presents information in articles in its Qt Quarterly newsletter. That is, that I first explain what we need to do, then show the code for a section, followed by an explanation of what the code does. This does result in fewer comments in the code, but also encourages one to write code that is fairly easy to read. The typical challenge that arises is making sure the code has acceptable formatting for the document.

We shall begin by first explaining what one needs to get up and running with PyQt (assuming that one starts with the setup presented in class). We will also quickly go over some PyQt and Tkinter differences. We will then go over a majority of the examples that introduce new concepts or new classes. We will end with doing a very quick exploration of using Qt Designer and PyQt together. We will conclude with up with some opinions about PyQt and using it for GUI programming.

In order to run the examples on your machine, you will need to install the following packages on your machine. Information on how to install each package is included in their respective package and is beyond the scope of this document.

Qt 3.x: This is the Qt toolkit provided by Trolltech. Version 3 of the toolkit is available via the QPL or GPL on X11, GPL on Mac OS X, and a commercial license on Windows. The latest version is available from Trolltech's ftp servers. If you are running a recent distribution of Linux, you probably have Qt installed already, if you have KDE 3.x, you are in good shape. You must use Qt version 3, as a version of PyQt does not exist for Qt 4 yet.

Source code:
ftp://ftp.trolltech.com/qt/source/

PyQt 3.x: PyQt is the Python binding for Qt and is offered under the same terms as the Qt library. It is offered by Riverbank Computing and is available from them. Again, many Linux distributions include PyQt by default so this step may not be necessary (try 'import qt' in a Python interpreter to see if you have it). If you don't have it installed you may have to install SIP (an alternative to SWIG). Which is also available from the same place as you get PyQt.

Source code:
http://www.riverbankcomputing.co.uk/pyqt/

PyQwt 4.x: PyQwt offers Python bindings for the Qwt library. Qwt is an open source scientific/technical library for use with Qt. We will use it primarily for its plotting widget. Qwt is included in the source for the PyQwt package so everything can be built in one fell swoop.

Source code:
http://pyqwt.sourceforge.net/

The versions that were used when doing this project were Qt/Mac 3.3.5, PyQt 3.15, and PyQwt 4.2. Other versions should work, but have not been tested. Once you have everything installed you are ready to begin, but let's take a quick look at some of the differences between Qt and Tkinter before diving into the code.

Qt differs a little from Tkinter in a few ways. The first and most obvious is the way of signaling when things change. In Tkinter, you essentially bind a variable to a control and then register a Python function to manipulate the data at certain points. Qt solves this using signals and slots. A signal signifies a high-level event such as, a button is pressed, a slider is moved, an item is selected, etc. Signals can be connected to slots, which, in Python, can be any function. It is possible to even create your own signals inside of Python and "emit" them for your own objects. One very nice advantage of signals and slots is that one object's slot needs to know nothing about the object emitting a signal other than the arguments past it. This allows one to rip out one of the widgets, drop in another, connect the proper signals to the right slots, and presto, you are all set. One bonus of using PyQt over Qt is that there is no need for using Qt's meta-object compiler (moc). This is because Python has all the magic of QMetaObject built-in. As a normal Qt

user, this is a refreshing change. All the examples will make use of Qt's signals and slots, so we'll explore this as we progress.

Parent-child relationships are also very important in Qt. Usually you create your GUI by sub-classing from a widget and then create other widgets that you want on top of the widget as children of that widget. This makes classes very important in PyQt.

Another way Qt differs from Tkinter is the way it deals with layouts. The typical Tkinter widget, "packs" its children taking hints where you want them stuffed in the widget. Tkinter also provides a grid layout and that is similar to the approach Qt follows. Here you create a QLayout object and add your widgets to it. The layout will then take care of the widgets' sizes for you.

Because of these differences, it sometimes because impractical to write PyQt exactly the same as they were in Tkinter. This is expected and not a problem because we are interested in how the final script will function if we were to use that instead of the Tkinter scripts.

Documentation for PyQt [2] is rather limited. PyQt's documentation focuses on explaining how the wrapped functions differ from their original Qt versions. Since the wrapper doesn't abstract much away much from Qt, you can look at the original Qt documentation [3] to steer you. This is probably fine if you are used to C++ or Qt's documentation, but could be a little daunting if you've never encountered it before. However if you look at a few of the scripts presented here and then look at the documentation for some of the functions and classes, you should be able to piece things together. Otherwise, I shamelessly and gladly plug Jasmin Blanchette and Mark Summerfield's C++ GUI Programming with Qt 3 [4] as an excellent start to learning Qt.

As a final note, this project was done at home using a Powerbook running Mac OS X. Therefore; certain parts of Tkinter and Pmw were not available (notably the BLT library). While some time was spent trying to get BLT to work, it was dropped and the script was interpreted by hand instead.

We need a place to start and the best place to begin is to try our hand at the first GUI example. Since this is our first encounter with PyQt code, we'll number the lines and walk through it line by line. In future examples, we won't be doing the explicit numbering and we'll discuss what we are doing as we go along.

```
1     #!/usr/bin/env python
2
3     from qt import *
4     import sys, math
5
6     # Create the application for running the event loop
7     app = QApplication(sys.argv)
8
9     # Create the window and its widgets
```

```
10    window = QWidget()
11    label = QLabel("Hello, World! The sine of  ", window)
12    textEdit = QLineEdit("1.2", window)
13    button = QPushButton(" equals ", window)
14    result = QLabel(window)
15
16    # Lay out the widegts
17    layout = QHBoxLayout(window)
18    layout.addWidget(label)
19    layout.addWidget(textEdit)
20    layout.addWidget(button)
21    layout.addWidget(result)
22
23    def computeSin():
24        value = textEdit.text().toDouble()[0]
25        result.setNum(math.sin(value))
26
27    QObject.connect(button, SIGNAL("clicked()"),
28                    computeSin)
29    # Show the widget and start the event loop
30    window.show()
31    app.setMainWidget(window)
32    app.exec_loop()
```

Figure 1: hwGUI1.py listing

And here's what it looks like in action:

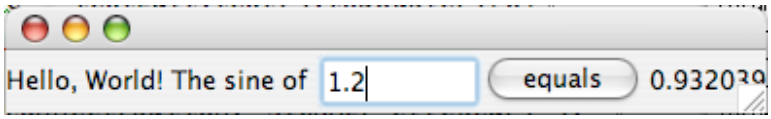

Figure 2: hwGUI.py running

Line 1 is our favorite way of invoking Python. In line 3, we import all of PyQt into our script. We also import the sys and math modules. This allows to access PyQt objects and methods without prefixing them all with "qt.".

On line 7 we create a QApplication. A QApplication is a Qt Object that is the foundation of any Qt GUI program. The main purpose of a QApplication is to run an event loop and dispatch events to widgets. QApplication will also create the connection to the GUI server, so you must create it before you create any widgets or do any painting. Qt has a number of command line arguments that govern what style the application is run in and some other, mostly X11, options. When we pass them along to the QApplication, they are parsed and removed by the QApplication.

On line 10 we create a QWidget. We pass no arguments to it, so it will be a "top-level widget," in other words, a window.

In Lines 11 through 14 we create and set up two QLabels, a QLineEdit, and a QPushButton. When we are creating the widgets, we first pass the text we wish to show in the each widget. We then pass the window we created earlier. This tells PyQt to create these widgets as children of the window. Or rather, making the window the parent of these widgets. It also means that these widgets will be inside of the window instead of being windows themselves.

Now that we've created the widgets, we need to lay them out in the window, otherwise they will all be piled on top of each other. We do this first by creating a QHBoxLayout. This is a horizontal layout object. We pass the window as the parent so the layout knows whom it is managing the widgets for. We then add the widgets to the layout in the order we want them to show up in the layout. The "Hello World" label comes first followed by the line edit, followed by the button, followed by the label to show the result. This happens in lines 17-21.

Lines 23-25 define the function for computing the sine of the value that the user entered into the line edit. How we call this function we'll get to shortly, but we can examine what is happening here first. We access the text of the line edit through its text() method. This method returns a QString. A QString is Qt's own Unicode string which will usually need to be cast to a Python string before it can be used inside of Python. In this case though, we need the numerical value. While we could cast the QString to a Python string and then to a float, we can short circuit the process by calling one of QString's own number converting functions. This function will return a tuple. The first element is double (float) value, the second is a Boolean indicating whether or not the conversion was successful or not. In the spirit of the original script, we forego error checking. Once we have the number, we call math.sin() and set the result as the text of the label. We use QLabel's setNum function for this to avoid another conversion in our script.

On Line 27 we create our signal-slot connection. Here we connect the button's clicked signal to the computSin function we defined above. The clicked signal is emitted whenever someone presses and releases the push button. Every time that the signal is emitted, the computeSin function is synchronously called.

At this point, we are done with all the tough work of the script and all that remains is to show the widget and start the event loop. This is what lines 30 and 32 do respectively. Line 33 tells the QApplication object that window is our "main widget." This means that when this window is closed, the application will stop its event loop and end the script. If we didn't sit the main widget and didn't establish some other way of quitting the application, the script would run indefinitely.

Since PyQt deals with signal and slot connections, it is very trivial to make the hwGUI2.py script. The main purpose of that script was to show make the computation happen when enter is pressed in the line edit. Here's the script with bold lines for what has changed looks like this:

```
#!/usr/bin/env python
```

```
from qt import *
import sys, math

# Create the apllication for spinning the event loop
app = QApplication(sys.argv)

# Create the window and its widgets
window = QWidget()
label = QLabel("Hello, World! The sine of  ", window)
textEdit = QLineEdit("1.2", window)
label2 = QLabel(" equals ", window)
result = QLabel(window)

# Lay out the widegts
layout = QHBoxLayout(window)
layout.addWidget(label)
layout.addWidget(textEdit)
layout.addWidget(label2)
layout.addWidget(result)

def computeSin():
    global textEdit
    global result
    value = textEdit.text().toDouble()[0]
    result.setNum(math.sin(value))

QObject.connect(textEdit, SIGNAL("returnPressed()"),
                computeSin)

window.show()
app.setMainWidget(window)
app.exec_loop()
```

Figure 3: hwGUI2.py listing

As you can see, all we need to do is substitute a QLabel for the QPushButton and connect to the line edit's returnPressed() signal.

In hwGUI3.py, the Tkinter example was similar to the first example, but also adds the ability to quit the script by hitting 'Q'. First we'll look at the function that will bring up the message box asking us to quit.

```
def quitWithMessage():
    if (QMessageBox.question(window, "Quit?",
                             "Are you sure you want to quit?",
                             "Quit", "Stay here") == 0):
        qApp.quit()
```

Figure 4 hwGUI3.py quitWithMessage function

QMessageBox has a few static function that will create a message box, show it and block further execution until a button is pressed in the message box. We create a "Question" message box with a window title of "Quit?" and ask the user if he or she want to quit. We then create two buttons, one with the text of "Quit" and the other with the text "Stay Here." It is usually a good idea to create buttons with text that contains the action that the button will take. The function returns which button was pressed. "Quit" is button zero and "Stay Here" is button one. When the button is "Quit," we tell the script to quit. Here we reference qApp. qApp is a global symbol that is always accessible from Qt and therefore PyQt. We could have easily have used the app object we created earlier, but it is also possible to use the qApp object when you haven't created a QApplication yet (such as in a class). It may also be of interest to note that QApplication's quit() method is also a slot that we could chose to connect to directly and bypass the message box all together.

```
accel = QAccel(window)
accel.insertItem(Qt.Key_Q)

QObject.connect(accel, SIGNAL("activated(int)"), quitWithMessage)
```

Figure 5 hwGUI3.py QAccel Item

Now that we've created the function, we need to create an object to connect to it. Since we want this to happen when we press the 'Q' key, we will use QAccel. QAccel takes a key sequence (or multiple ones) and emits a signal when the sequence is entered. Since we only have one accelerator, we don't need to de-multiplex the values and connect it straight to the quitWithMessage function. Notice that the signal has an extra parameter, an integer, while our function takes no parameters. With PyQt it is possible to make a connection to a function that takes less parameters. These extra arguments are ignored.

When we run the script, we'll see that when we hit 'Q', the message box won't show. Instead the Q will show up in the QLineEdit. This is because of the idea of "focus" and how keyboard events are delivered. By default, PyQt will deliver keyboard events to QApplication's focusWidget(). The widget will get the QKeyEvent and can choose to accept or reject the event. If it chooses to ignore the event, the widget is propagated up the parent chain until it reaches the top of the hierarchy or someone chooses to accept the event. Since it is natural for a QLineEdit to accept all the standard alphanumeric characters, it accepts the Q and shows it. However, if we change the focus widget to the QPushButton (by pressing tab, for example) and hit 'Q' we will get our message box.

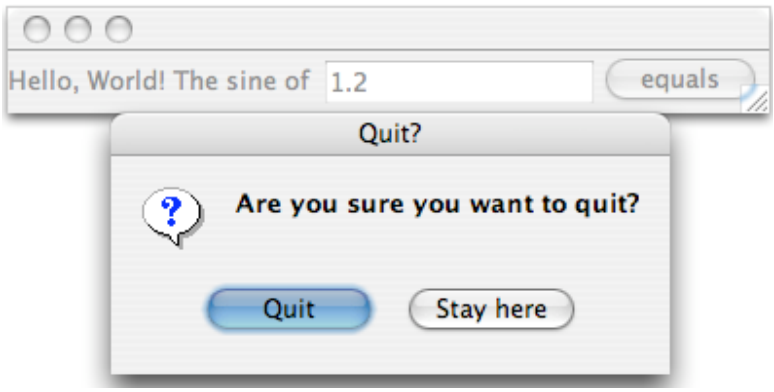

Figure 6: hwGUI3.py running with message box

For disillusioned PyQt fans, it certainly is possible to do this, even in a line edit. It would require installing an "event filter" on the line edit. The filter would catch all events to the line edit. Then we could certainly filter out the Q-key event and do what we wish with it. To create an event filter we need to make a subclass of QWidget though. The code for doing this is shown in hwGUI10a.py, though I won't go through it here.

The examples hwGUI4.py and hwGUI5.py demonstrate some different ways of creating layouts in Tkinter. We can quickly show how similar layouts are created in PyQt. First we can change our layout that we create from a QVBoxLayout to a QHBoxLayout.

```
label2.setAlignment(Qt.AlignCenter)
# Lay out the widgets
layout = QVBoxLayout(window)  # Note the V instead of H
layout.addWidget(label)
layout.addWidget(textEdit)
layout.addWidget(label2)
layout.addWidget(result)
```

Figure 7: hwGUI4.py QVBoxLayout

This will cause the widgets to be laid out from top-to-bottom instead of left-to-right. In order to make the "equals" label look OK in the layout, we set the alignment of label2 to center alignment.

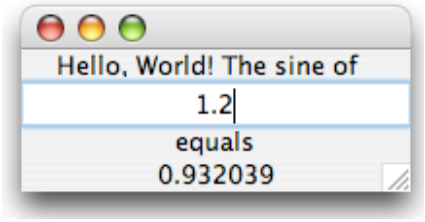

Figure 8: hwGUI4.py running

The hwGUI5.py script shows off stacking of layouts to create a more complex layout.

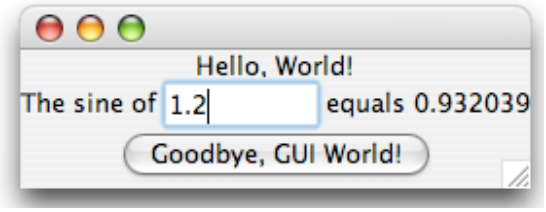

Figure 9: hwGUI5.py showing an alternate layout

The code to do this looks like this:

```
# Lay out the widegts
mainLayout = QVBoxLayout(window)
mainLayout.addWidget(label)
hlayout = QHBoxLayout()
hlayout.addWidget(label2)
hlayout.addWidget(textEdit)
hlayout.addWidget(equalsLabel)
hlayout.addWidget(result)
mainLayout.addLayout(hlayout)
quitLayout = QHBoxLayout()
quitLayout.addStretch()
quitLayout.addWidget(quitButton)
quitLayout.addStretch()
mainLayout.addLayout(quitLayout)
```

Figure 10: hwGUI5.py stacking QLayouts

First we create mainLayout as a QVBoxLayout and we add our first label. We then create a QHBoxLayout (hlayout) and add the rest of our widgets except the quit button to it. We then add hlayout to the mainLayout. In theory, we could just add our quit button to the main layout and be done, but then the quit button would stretch to match the width of the entire widget. Since buttons, like many things, look a bit better when not being stretched, we create another QHBoxLayout (quitLayout) and call addStretch() add the quitButton, call addStretch() again and finally add the quitLayou to mainLayout. The addStretch() method works like a placeholder, it will consume excess space. Since we sandwich the quitButton between two stretches, the quitButton takes up only as much space as it needs in the center and the stretches consume the rest.

The hwGUI6.py script adds no real functionality, but it does give us a chance to look at the font properties of widgets. Here we change the font for the “Hello World” label. The code looks like this:

```
labelFont = QFont("times")
labelFont.setPointSize(18)
labelFont.setBold(True)
label.setFont(labelFont)
```

Figure 11: hwGUI6.py, using QFont

We first create a QFont object and request the "times" font. QFont is a abstraction from the various fonts implementations on windowing systems. QFont will look for a font that matches the name. If it doesn't find a match, it will apply various algorithms to find the closest matching font. If that fails, it will pass us a default font. Regardless, we can manipulate the various font values, such as point size and thickness. Here we change the original font to 18 point and tell it to be bold. We then set the font on the label. When we run the script we see that indeed that "Hello World" is bigger and bolder.

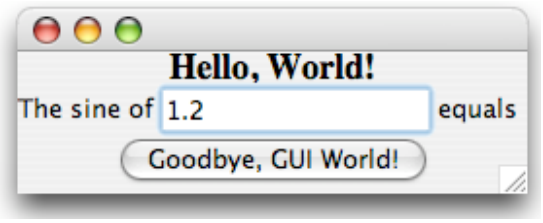

Figure 12: hwGUI6.py running

We play a bit with palettes in the hwGUI7.py script. Here, the idea is to create a yellow button with blue text. The palette of the widget is a QPalette object that contains several colors that describe the colors the widget should be painted. It groups these colors into three groups that you can refer to by QColorGroups. There is a QColorGroup for when the widget is active, inactive (not in the active window), and when the widget is disabled. Each QColorGroup has a list of role that corresponds to a color.

```
palette = quitButton.palette()
colorGroup = palette.active()
colorGroup.setColor(QColorGroup.ButtonText, Qt.blue)
colorGroup.setColor(QColorGroup.Button, Qt.yellow)
palette.setActive(colorGroup)
palette.setInactive(colorGroup)
palette.setDisabled(colorGroup)
quitButton.setPalette(palette)
```

Figure 13: hwGUI7.py, coloring a button

First we take quitButton's palette. While we could have constructed one from scratch, this is usually the method you will use when you are changing the palette of a widget. We then take the active QColorGroup of the palette and set the ButtonText role to Qt.blue and the Button role to Qt.yellow. For this example, we just set this new color group as the color group for the active, inactive, and disabled groups in the palette. We then set the palette back. The result is shown below. I'm showing this script in Windows style because the native commands to draw a button on Mac OS X don't allow for the color to be changed. Still PyQt, running the Mac OS X style shows something better than the Tkinter script.

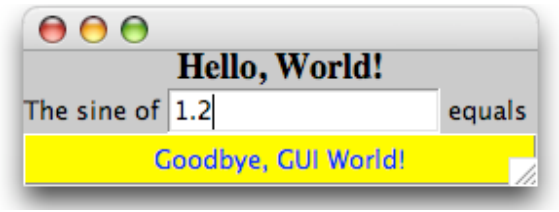

Figure 14: hwGUI7.py running

Since PyQt doesn't have a idea of a "packer" layout and therefore no idea of an anchor, we have to take a slightly different route for creating the layout in hwGUI8.py. Instead of playing with the layout, we instead alter quitButton's size policy. A size policy describes how a widget should interact with a layout. This is accomplished with the following line.

```
quitButton.setSizePolicy(QSizePolicy.Expanding,
                         QSizePolicy.MinumumExpanding)
```

Figure 15: hwGUI8.py, altering a size policy

QSizePolicy has two values, one for horizontal and one for vertical. The key here is the second value which for the vertical element. By default, a QPushButton will take the size it needs vertically and refuse to go smaller. Here we override it to take its minimum, but grow if it possibly can. This makes quitButton greedier in grabbing space. To get the same effect of it being anchored in the right, we remove the stretch before we add the quitButton to its layout.

```
...
quitLayout = QHBoxLayout()
quitLayout.addWidget(quitButton)
quitLayout.addStretch()
mainLayout.addLayout(quitLayout)
...
```

Figure 16: hwGUI8.py, changing a layout

The result is shown below:

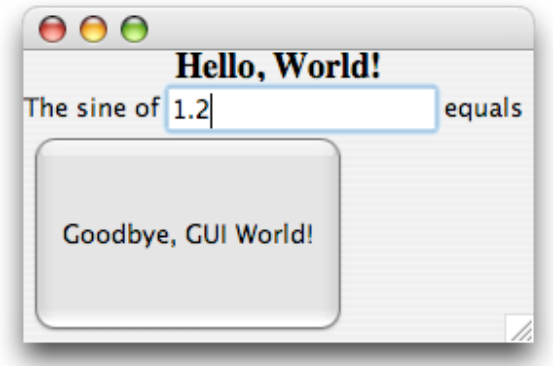

Figure 17: hwGUI8.py running

In hwGUI9.py we change back from a QLabel to a QPushButton, but one that is flat. This is changed very easily in code by calling setFlat() on the button.

```
equalsButton = QPushButton("equals", window)
equalsButton.setFlat(True)
```

Figure 18: hwGUI9.py, creating a flat button

I'd just like to point out that the resulting script when run on Mac OS X, it really has a flat button, unlike the TkInter script that shows a normal button.

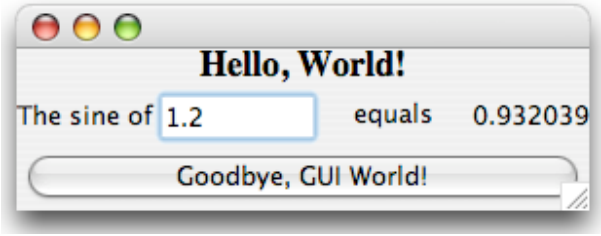

Figure 19: hwGUI9.py with a flat button

One layout class we haven't looked at yet is QGridLayout. QGridLayout works similar to the other box layouts we've seen, except that it is a two-dimensional grid instead of a 1-dimensional array. Let's see how it would look if we used a QGridLayout in hwGUI9.py instead of our stacking of layouts

```
mainLayout = QGridLayout(window)
mainLayout.addMultiCellWidget(label, 0, 0, 0, 3)
mainLayout.addWidget(label2, 1, 0)
mainLayout.addWidget(textEdit, 1, 1)
mainLayout.addWidget(equalsButton, 1, 2)
mainLayout.addWidget(result, 1, 3)
mainLayout.addMultiCellWidget(quitButton, 2, 2, 0, 3)
```

Figure 20: hwGUI9_grid.py: Using a QGridLayout

The main difference here is that we can add widgets to a particular row and column that we want it to be in, or we can have it span rows and columns. We first add the "Hello World" label as a multi-cell widget that will span the width of the widget. We then add the labels, line edit, and flat button to the second row. Finally, we add the quit button as another multi-row widget.

In general, you can compose a hierarchy of QHBoxLayouts and QVBoxLayouts into a corresponding QGridLayout. If you have no special requirements, it is a matter of personal taste to which layout you want to use. There is one case where a QGridLayout is superior which we will examine shortly.

As with Tkinter, it makes much more sense to use PyQt in an object-oriented manner. This is especially true with PyQt as it is based on the heavily object-oriented Qt library.

The typical case in PyQt is to derive from a QWidget subclass that you need and construct the other widgets as children of your subclass. With PyQt's signals and slots, it makes it very easy to make these widgets into components that you can use in other widget hierarchies. Something that does not work well at all with the scripts we've written thus far. We'll see this soon enough, but let's take a look at how we would do our "Hello World" script as a QWidget subclass.

```
class HelloWorld(QWidget):
    def __init__(self, *parent):
        QWidget.__init__(self, *parent)
        # Set up the Widgets
        label = QLabel("Hello, World!", self)
        font = QFont("times")
        font.setPointSize(18)
        font.setBold(True)
        label.setFont(font)
        self.lineEdit = QLineEdit("1.2", self)
        label2 = QLabel("The sine of", self)
        computeButton = QPushButton("equals", self)
        computeButton.setFlat(True)
        self.sineLabel = QLabel(self)
        quitButton = QPushButton("Goodbye, GUI World", self)
```

Figure 21: HelloWorld constructor

Notice here that our class is a subclass of QWidget. This makes it very easy to embed our widget in the future should we choose to. All QWidgets take an optional argument for their parent. This is how we build up our hierarchies. We just need to be sure that we pass it on to our base class to take care of that for us. That's why we call QWidget.__init__() here. Next we create the widgets we need, passing self as the parent to make the widgets our children. Notice that we only actually keep lineEdit and sineLabel as member variables. The others we only need to keep references to for the lifetime of the constructor. Even though we don't keep variables to all the labels or buttons, we could get a hold of them later by asking for a list of our children.

After we've created the widgets, it's good practice to lay them out and then establish connections. First the layout code.

```
        # Lay the widgets out
        mainLayout = QVBoxLayout(self)
        mainLayout.addWidget(label)
        mainLayout.setMargin(6)
        mainLayout.setSpacing(11)
        subLayout = QHBoxLayout()
        subLayout.addWidget(label2)
        subLayout.addWidget(self.lineEdit)
        subLayout.addWidget(computeButton)
        subLayout.addWidget(self.sineLabel)
```

```
        mainLayout.addLayout(subLayout)
        mainLayout.addWidget(quitButton)
```

Figure 22: HelloWorld constructor layout code

The code is similar to what we've seen before, but we also set the margin around the layout and the spacing between each widget. These are specified in pixels and give a better appearance to our widget than what we've had previously.

Finally, we create our connections. And define whatever slots we need.

```
        # Now make the connections
        QObject.connect(self.lineEdit, SIGNAL("returnPressed()"),
                        self.computeSine)
        QObject.connect(computeButton, SIGNAL("clicked()"),
                        self.computeSine)
        QObject.connect(quitButton, SIGNAL("clicked()"),
                        qApp, SLOT("quit()"))

    def computeSine(self):
        self.sineLabel.setNum(
                math.sin(self.lineEdit.text().toDouble()[0]))
```

Figure 23: HelloWorld, connections and computeSin

Here we connect the lineEdit's "returnPressed" signal and the computeButton's "clicked" signal to our computeSine method. We also connect quitButton's "clicked" signal to qApp's quit slot. We could have alternatively specified qApp.quit in the function. But our current way will make the connection at the C++ level and thus be a little more efficient and faster. After all, you certainly can't quit an application fast enough.

Our computeSine function is not much different from earlier versions other than it now accesses our own label and line edit instead of depending on them in a global scale.

One other nice advantage of having the widget contained in a class is that we can easily add a "test" module in our script to see how it looks as a stand-alone widget. The rest of our examples where we create a widget class, we will test with code that looks very similar to the following:

```
if __name__ == "__main__":
    import sys
    app = QApplication(sys.argv)
    window = HelloWorld()
    window.show()
    app.setMainWidget(window)
    app.exec_loop()
```

Figure 24: A typical "__main__"

This should be familiar to you already. But if we run the script by itself, __name__ will be __main__ and in that case we create a QApplication and an instance of our widget and show it.

Now we will present a way of bolting a GUI onto an already existing script. We will try to implement it like it is done in the book; that is that there are sliders for ranges of numbers and line edits for free-form numbers or other text. QLineEdits will work for fine for text, and QSlider would make a logical choice for the sliders. But there's a problem. QSlider only takes integers for getting and setting values and we need it to deal with floating point numbers. We also need to cement the slider together with a couple of QLabels to show the argument and its value. Since we need to do this several times, it makes sense to wrap this into a subclass. Since this subclass will contain multiple widgets and we don't want to go rooting around looking for the slider, it would be nice if the widget would signal its value changes.

```
class SimVizSlider(QWidget):
    def __init__(self, argLabel, floatMin, floatMax, *parent):
        QWidget.__init__(self, *parent)
        self.argLabel = QLabel(argLabel, self)
        self.slider = QSlider(Qt.Horizontal, self)
        self.slider.setMinValue(floatMin * 100)
        self.slider.setMaxValue(floatMax * 100)
        self.realValue = QLabel(self)

        mainLayout = QVBoxLayout(self)
        mainLayout.addWidget(self.argLabel)
        hlayout = QHBoxLayout()
        hlayout.setSpacing(6)
        hlayout.addWidget(self.slider)
        hlayout.addWidget(self.realValue)
        mainLayout.addLayout(hlayout)
        self.realValue.setMinimumWidth(
                            self.fontMetrics().width("WWWWW"))

        QObject.connect(self.slider, SIGNAL("valueChanged(int)"),
                        self.adjustValue)
        self.adjustValue(self.slider.value())
```

Figure 25: SimVizSlider constructor

Our constructor takes a few arguments, the label text, a floating-point minimum and maximum value of the slider, and, of course, the optional parent. We create a label for the "text" of the argument. We then create a horizontal slider. Since we assume we have floating-point values we will "convert" the values to integers by multiplying them by 100. We then create a label to show the float value. Afterwards, we create our layout with the argLabel on top of the slider and realValue label's layout. At the end of creating the layouts, we set the realValue's minimum width to be the width of 5 W's with the help of our QFontMetrics object. This tells the label that it needs at least as much space as 5 W's.

This is for aesthetics, so let's look at what it is trying to prevent. Since there is a very good chance that the widget will be run with a variable-width font. The width of the label will change depending on the text it contains, even if we enforce that the text should contain a certain number of characters, a '1' just takes less space than a '7'. The upshot of this is that since it is in a layout with the QSlider, the slider will also change its width. This is a bit annoying, especially since this happens when you are changing the value. By setting the minimum width, we force the label to be at least a certain size. We use the letter W because it is usually the widest character in a (latin) font. Now our slider won't "slide" on us.

Once we've laid out the widgets, we make connect the valueChanged signal of the slider to our adjustValue slot that will massage the value. At the end we call this slot ourselves with the sliders current value to "kick start" the label with a proper value.

When we created the slider, we set its minimum and maximum values by multiplying our float values by 100. In order to get decent float values returned, we need to do some translation. This happens in our adjustValue slot.

```
def adjustValue(self, newValue):
    adjustedValue = newValue / 100.0
    strAdjusted = "%4.2f" % adjustedValue
    self.realValue.setText(strAdjusted)
    self.emit(PYSIGNAL("argChanged"),
              (str(self.argLabel.text()),
               str(self.realValue.text())))
```

Figure 26: SimVizSlider adjustValue

Here we take the value given to us and divide it by 100. We then make it a nice string that we can set as the label text. Finally, we emit a new signal. Signals in PyQt are a little different than signals in normal Qt. In Qt, signals basically look like functions, but because of how Python does things, this doesn't work. Instead you create a "PySignal" when you are emitting. The emit() function comes from PyQt's QObject and expects a PYSIGNAL() and then a tuple containing any additional data. Here we emit the text of the argLabel and our new float value.

Since, we probably need different default values for the slider, let's also add a set method too.

```
def setArgValue(self, value):
    self.slider.setValue(value * 100) # realValue will catch change
```

Figure 27: SimVizSlider setArgValue

This is straightforward; we simply multiply the value passed in by 100. One thing to note though is that calling setValue() will trigger the slider's valueChanged signal, so our label will get updated automatically without any work on our part.

You may wonder why we needed this method at all, since we could have specified this as an argument in the constructor, and you would be right. The main reason was because adding an extra numerical argument is usually confusing because it's hard to remember the order. Python solves this problem with named arguments, but because of this *parent argument, it is impossible use named arguments with a standard QObject subclass. This is unfortunate, but a small price to pay overall.

Looking at the original Tkinter script, the main idea is to put the arguments in a dictionary that we can easily iterate over later. Since we already have a group of widgets that emit both their argument and value, it is probably also worthwhile to do that with the line edit as well. We can do this very quickly with another subclass.

```
class SimVizLineEdit(QWidget):
    def __init__(self, argLabel, argParam, *parent):
        QWidget.__init__(self, *parent)
        self.argParam = argParam
        self.argLabel = QLabel(argLabel, self)
        self.lineEdit = QLineEdit(self)
        self.connect(self.lineEdit,
                     SIGNAL("textChanged(const QString &)"),
                     self.updateText)
        hbox = QHBoxLayout(self)
        hbox.setSpacing(6)
        hbox.addWidget(self.argLabel)
        hbox.addWidget(self.lineEdit)
```

Figure 28: SimVizLineEdit constructor

The SimVizLineEdit is very simple; argLabel contains what should be in the label and argParam in the actual parameter that will be stored in the dictionary. The widget consists of a label and a line edit. We connect the edit's textChanged signal to our updateText slot. The slot itself looks like this.

```
    def updateText(self, theString):
        self.emit(PYSIGNAL("argChanged"),
                  (self.argParam, str(theString)))
```

Figure 29: SimVizLineEdit updateText

Here, we simply aggregate the signal with the argument for the dictionary. To be complete in our presentation of the class here, the code for setting the value is presented as well.

```
    def setArgValue(self, value):
        self.lineEdit.setText(str(value))
```

Figure 30: SimVizLineEdit setArgValue

Now that we have our parts assembled, we can write the main widget. The constructor is rather long so let's walk through it in parts.

```
class MainWindow(QWidget):
    def __init__(self, *parent):
        QWidget.__init__(self, *parent)
        self.cmdArgs = {}
        sliderM = SimVizSlider("m", 0.0, 5.0, self)
        sliderB = SimVizSlider("b", 0.0, 2.0, self)
        sliderC = SimVizSlider("c", 0.0, 20.0, self)
        sliderA = SimVizSlider("A", 0.0, 10.0, self)
        sliderY0 = SimVizSlider("y0", 0.0, 1.0, self)

        self.connect(sliderY0, PYSIGNAL("argChanged"), self.updateDict)
        self.connect(sliderA, PYSIGNAL("argChanged"), self.updateDict)
        self.connect(sliderC, PYSIGNAL("argChanged"), self.updateDict)
        self.connect(sliderB, PYSIGNAL("argChanged"), self.updateDict)
        self.connect(sliderM, PYSIGNAL("argChanged"), self.updateDict)

        sliderM.setArgValue(1.00)
        sliderB.setArgValue(0.70)
        sliderC.setArgValue(5.0)
        sliderA.setArgValue(5.0)
        sliderY0.setArgValue(0.20)
```

Figure 31: MainWindow constructor start

We begin by creating an empty dictionary (cmdArgs) for the arguments. We then create SimVizSliders for each of the range arguments. We then create connect to the argChanged signal we created to our updateDict slot. We then set the sliders to the default values we want. Again, since we created the connections before we set the values, the updateDict slot will get called which will update cmdArgs.

```
        self.funcEdit = SimVizLineEdit("func:", "func", self)
        self.wEdit = SimVizLineEdit("w:", "w", self)
        self.tStopEdit = SimVizLineEdit("tstop:", "tstop", self)
        self.timeStepEdit = SimVizLineEdit("time step:", "dt", self)
        self.caseNameEdit = SimVizLineEdit("case name:", "case", self)

        self.connect(self.funcEdit, PYSIGNAL("argChanged"),
                     self.updateDict)
        self.connect(self.wEdit, PYSIGNAL("argChanged"),
                     self.updateDict)
        self.connect(self.tStopEdit, PYSIGNAL("argChanged"),
                     self.updateDict)
        self.connect(self.timeStepEdit, PYSIGNAL("argChanged"),
                     self.updateDict)
        self.connect(self.caseNameEdit, PYSIGNAL("argChanged"),
                     self.updateDict)

        self.funcEdit.setArgValue("y")
```

```
self.wEdit.setArgValue(2 * math.pi)
self.tStopEdit.setArgValue(30.0)
self.timeStepEdit.setArgValue(0.05)
self.caseNameEdit.setArgValue("tmp1")
```

Figure 32: MainWindow more construction

We follow the same pattern with our SimVizLineEdits; we create them, make connections and then set their values.

```
computeButton = QPushButton("Compute", self)
quitButton = QPushButton("Quit", self)

QObject.connect(quitButton, SIGNAL("clicked()"),
                qApp, SLOT("quit()"))
QObject.connect(computeButton, SIGNAL("clicked()"),
                self.runScript)

pic = QPixmap(os.path.join(os.environ['scripting'], \
              'src','misc','figs','simviz2.xfig.t.gif'))
picLabel = QLabel(self)
picLabel.setPixmap(pic)
```

Figure 33: MainWindow final construction

We round out the window by creating the “Compute” and “Quit” buttons and connect them up to the proper slots. We then create a QPixmap that contains the figure and set it on a QLabel. The rest of the constructor contains code for laying out the widget that we will ignore for now.

We need to implement our updateDict slot. This is trivial, thanks to our special signals.

```
def updateDict(self, arg, value):
    self.cmdArgs[arg] = value;
```

Figure 34: MainWindow updateDict

Finally we implement the runScript function. This will actually be very similar to the TkInter script.

```
def runScript(self):
   # Walk down the list of variables and put them in a string
   cmd = os.path.join(
           os.environ['scripting'], 'src', 'py',\
           'intro', 'simviz1.py')
   for arg in self.cmdArgs.items():
       cmd += " -%s %s" % (arg[0], arg[1])

   print "final command is " + cmd
```

```
failure = os.system(cmd)
if failure:
    QMessageBox.critical(self, "Error",
                         "The underlying script failed to run",
                         "Bummer!")
```

Figure 35: MainWindow runScript

We create our command line by iterating through the dictionary and call os.system on the final result. If the script fails to run, we display a meesage box alerting the user.

That's it, here's what the thing looks like.

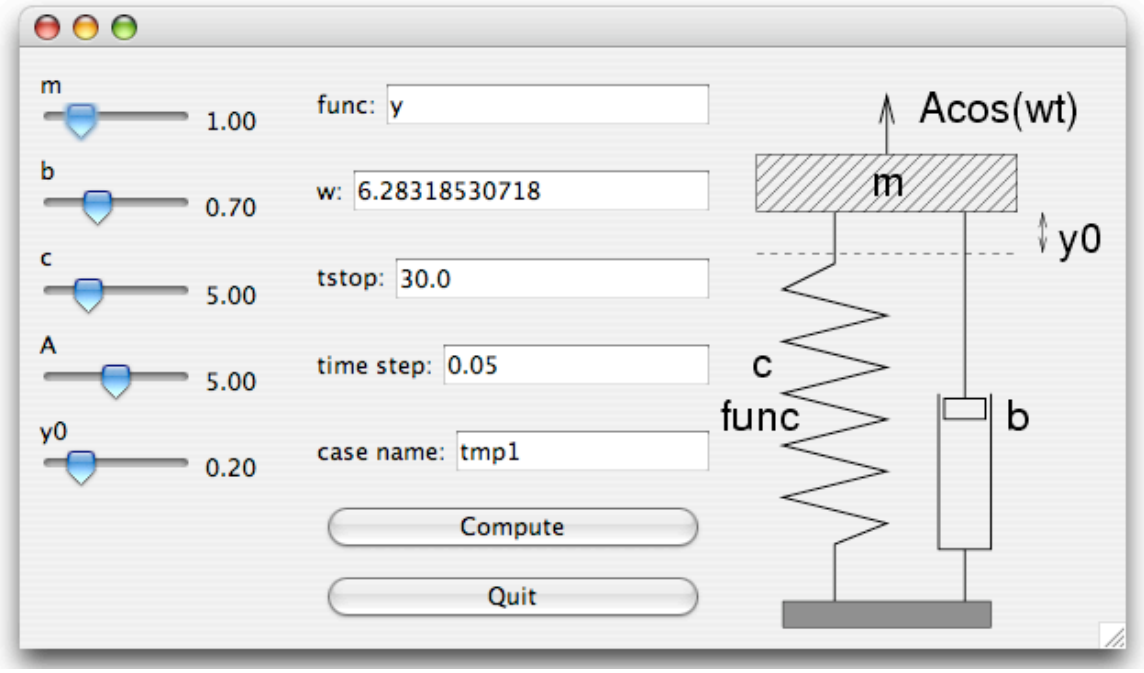

Figure 36: SimViz1GUI.py running

It looks OK, but it is a bit lame to have the line edits not all lined up. The reason this happens is because we stack a bunch of QHBoxLayouts into the QGridLayout. The QHBoxLayouts don't know anything about the other layouts in the layout, so they can only be aligned by chance. The proper way to solve this problem is to remove the QLabel from the SimVizLineEdit and put the labels and SimVizLineEdits into the grid layout individually. Since the grid layout manages both the label and the line edit, it will align them in the grid. At this point though, there isn't much point in having the SimVizLineEdit class anymore. But we can make this work without the subclass. All we need in another dictionary mapping QLineEdits to the proper argument. This is shown in simvizGUI2.py:

```
...
```

```
        self.connect(funcEdit, SIGNAL("textChanged(const QString &)"),
                     self.updateDict2)
        self.connect(wEdit, SIGNAL("textChanged(const QString &)"),
                      self.updateDict2)
        self.connect(tStopEdit, SIGNAL("textChanged(const QString &)"),
                      self.updateDict2)
        self.connect(timeStepEdit,
                     SIGNAL("textChanged(const QString &)"),
                      self.updateDict2)
        self.connect(caseNameEdit,
                      SIGNAL("textChanged(const QString &)"),
                      self.updateDict2)

        self.lineEditDict = { funcEdit: "func", wEdit: "w",
                              tStopEdit: "tstop",
                              timeStepEdit: "dt", caseNameEdit: "case" }
        ...
```

Figure 37: simviz2GUI.py extra dictionary

We need to do this lookup in the updateDict2 slot.

```
    def updateDict2(self, value):
        self.cmdArgs[self.lineEditDict[self.sender()]] = str(value)
```

Figure 38: SimViz2GUI.py updateDict2

Here we use sender() to lookup the line edit that sent the signal. Using sender() is usually not the best way to solve problems as it usually breaks encapsulation and it can be None. In this case though, we know that we only will call this function as a slot and we deal with our own children, so it is probably best solution.

The script will give us this window.

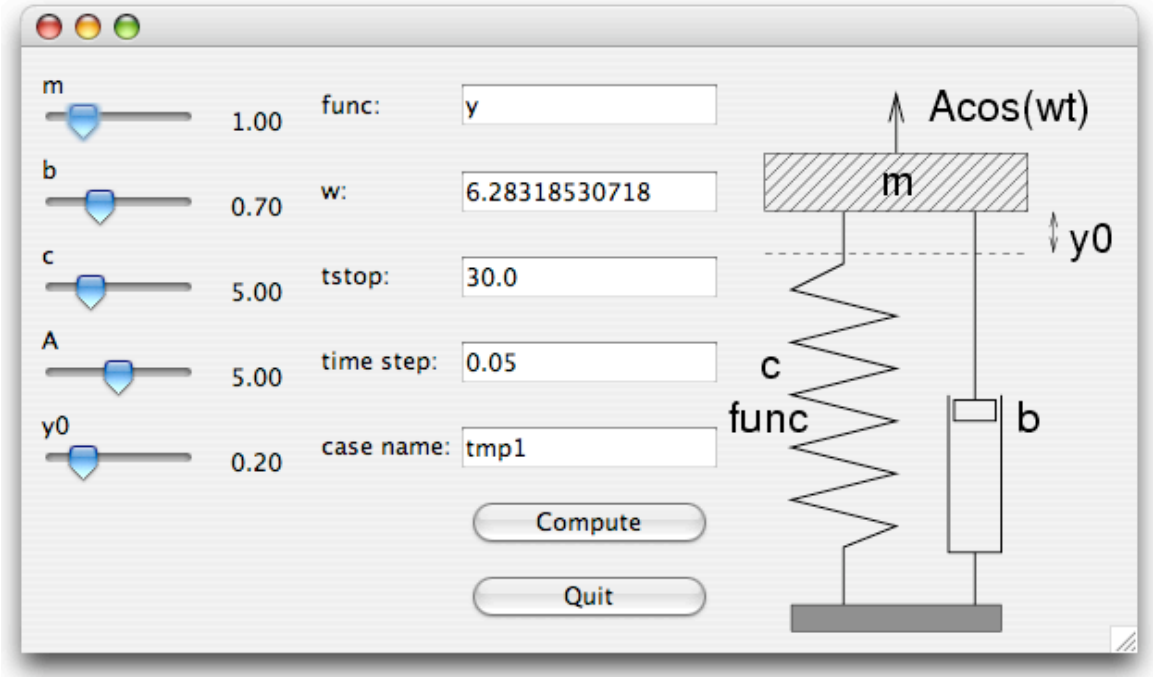

Figure 39: SimViz2GUI.py running

Another example of putting a GUI to a command line script or rather a sequence of commands is guimath.py. This script does not offer much in new material or techniques But it does show a technique for creating throwaway widgets.

```
self.resultWidget = QVBox()
resultLabel = QLabel(self.resultWidget)
button = QPushButton("Close", self.resultWidget)
QObject.connect(button, SIGNAL("clicked()"),
                self.resultWidget, SLOT("deleteLater()"))
resultLabel.setPixmap(QPixmap(self.gifFile))
self.resultWidget.show()
```

Figure 40: guimath.py throwaway dialog

Here we create a QVBox, this is a widget with a built in QVBoxLayout. It's handy for quick widgets, but that about it. We then create a label and button for it. We connect the button's clicked signal to the QVBox's deleteLater slot. The deleteLater slot is a handy feature that will deliver an event to the widget to tell it to delete itself. In C++ and Qt this has several functions, here we use deleteLater() to free up memory when the window is closed. Even though Python handles memory management for us, it's sometimes nice to issue hints like this. On another note, in order to make sure that this window isn't garbage collected immediately, we need to make sure our class keeps a reference to it, otherwise it will be immediately garbage collected when the function finishes. We could have worked around this by using a QDialog instead and called exec_loop to create a modal event loop for the dialog. On the other hand, this allows us to see the output and still access the GUI without closing the resulting window.

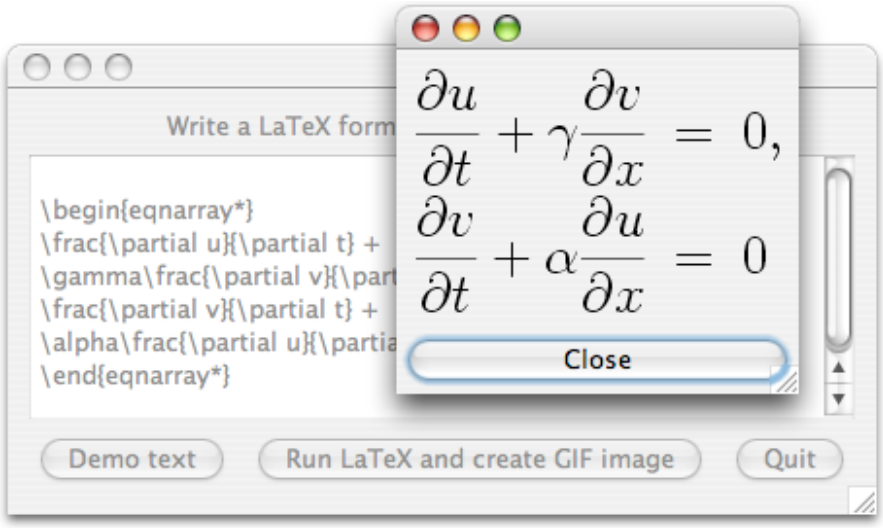

Figure 41: guimath.py running

There are a couple of examples don't provide much in the way of learning anything new with PyQt, but there are included in the source for the book, so they were duly ported and we will say a few words about them here. Feel free to browse the source code accompanying the document.

simplecalc.py: While TkInter does a good job showing that one can write a very simple GUI with a lot of power in around 35 lines, the corresponding PyQt GUI is 30 lines. However, hitting 'Q' won't work.

calulator.py: This demonstrates how you can subclass a QPushButton and aggregate its clicked signal with more information like we did with the textChanged signal in simvizGUI1.py. The only other interesting item is the fact that original script has an out of bounds read in its calculate function. The PyQt corrects this with the possibility of instead running out of memory. Though either way, I don't think the calculator will see much use.

fileshow1.py and **fileshow2.py**: These two scripts are also very trivial with PyQt. The QTextEdit class works well for displaying the content of text files. It was difficult to tell what the real difference was between versions 1, 2, and 3 of the original scripts, but I do know the original fileshow2.py and fileshow3.py both lock up with the message of "Exception in Tkinter callback" on Mac OS X.

Now let's look at an example that will introduce us to events in PyQt. The example is text_tag.py. It basically contains a couple of labels that change their color when the mouse pointer is inside them. Since there is only one way to do this with Qt, we do one version of this script.

Besides signals and slots, Qt also passes information along in the form of events. Events usually are a bit more low-level than signals. They usually indicate that a mouse button

was pressed or released, a key was pressed, a widget was shown, etc. To hook into an event the typical way to do it is to re-implement one of the specialized event handlers that are defined in QWidget or QObject. The two events that we are interested are when the mouse enters the widget and when it leaves. This means we need to re-implement enterEvent and leaveEvent respectively. Our QLabel subclass looks like this:

```
class EnterLeaveLabel(QLabel):
    def __init__(self, color, text, *parent):
        QLabel.__init__(self, text, *parent)
        self.color = color

    def enterEvent(self, event):
        self.oldPalette = QPalette(self.palette())
        self.setPalette(QPalette(self.color))

    def leaveEvent(self, event):
        self.setPalette(self.oldPalette)
```

Figure 42: EnterLeaveLabel class

The class takes a color and a text for its constructor. The key things are the enterEvent and leaveEvent. In enterEvent, we take a copy of the current palette. We then create a new QPalette based on the color passed to us earlier and set this as the palette for the label. On leaveEvent, we restore the old palette. That's it. We also make a subclass of this class to handle the counting:

```
class CountingLabel(EnterLeaveLabel):
    def __init__(self, color, *parent):
        EnterLeaveLabel.__init__(self, color,
                                 "You haven't hit me yet!", *parent)
        self.counter = 0

    def enterEvent(self, event):
        EnterLeaveLabel.enterEvent(self, event)
        self.counter += 1
        # Nice grammer counts
        if self.counter == 1:
            self.setText("You have hit me 1 time")
        else:
            self.setText("You have hit me %d times" % self.counter)
```

Figure 43: CountingLabel class

The only extra bits we add are a counter and extra code to update the text of the label in the enterEvent. With that we have a script that functions like the Tkinter script. This one is half the length though.

The fancylist.py script is an extension of this, but PyQt already provides us a QListBox for items and, even better, a signal when the mouse enters an item. This makes things rather simple, but we still have to do a bit of subclassing. First, we must take a look at the

QListBox class. It is a basically a widget that contains a number of QListBoxItems. We can create our own QListBoxItem subclass to show extra information and paint itself different when the mouse pointer is over it.

```
class FancyListBoxItem(QListBoxText):
    def __init__(self, text, extraInfo):
        QListBoxText.__init__(self, text)
        self.extraInfo = extraInfo
        self.color = None

    def paint(self, painter):
        if self.color != None and not self.isSelected():
            painter.fillRect(0, 0, self.listBox().width(),
                             self.height(self.listBox()),
                             QBrush(self.color))
        QListBoxText.paint(self, painter)
```

Figure 44: FancyListBoxItem class

We start by subclassing QListBoxText. This is a text QListBoxItem that already knows how to paint text. The paint function is what is interesting here. Here we check if our color variable was set and that we aren't selected, if that's the case we fill the area of the item with our color. Regardless we call our base class's paint function to paint the text. Now that we have items that will draw themselves, we can focus on the list box itself.

```
class FancyListBox(QListBox):
    def __init__(self, *parent):
        QListBox.__init__(self, *parent)
        self.setSelectionMode(QListBox.Multi)
        self.mouseOverItem = None
        self.connect(self, SIGNAL("onItem(QListBoxItem *)"),
                     self.updateColor)
```

Figure 45: FancyListBox constructor

In our constructor, we turn on "multi selection" which basically means we can select multiple items by simply clicking on them. We initialize our mouseOverItem to None since the mouse isn't over a particular item and connect QListBox's onItem signal to our updateColor slot.

```
    def updateColor(self, newItem):
        if newItem != self.mouseOverItem:
            if self.mouseOverItem:
                self.mouseOverItem.color = None
                if not self.isSelected(self.mouseOverItem):
                    self.updateItem(self.mouseOverItem)
            self.mouseOverItem = newItem
            if self.mouseOverItem\
               and not self.isSelected(self.mouseOverItem):
                newColor = self.palette().color(QPalette.Active,
```

```
                                        QColorGroup.Link)
        self.mouseOverItem.color = newColor
        self.updateItem(self.mouseOverItem)
        self.emit(PYSIGNAL("infoForItem"),
                  (self.mouseOverItem.extraInfo,))
```

Figure 46: FancyListBox updateColor

In updateColor, we check if the new item that the mouse is on is different from our mouseOverItem. If it is, we change mouseOverItem's color to none and call updateItem on it if it isn't selected. This will repaint our item in its original state. We then assign the newItem to the mouseOverItem and, if the newItem is not selected, change it's color member and call updateItem on that. We then emit the infoForItem signal with that item's extraInfo.

That's it! We can then drop it into a widget to test. That is what the rest of the script does and here's the result.

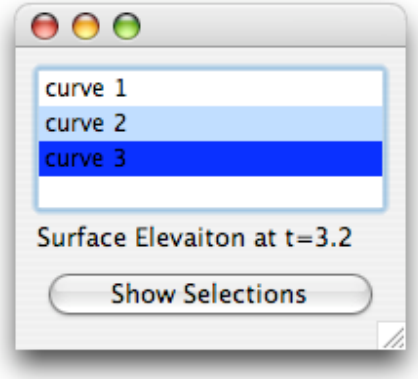

Figure 47: fancylist.py running with curve to selected

Although PyQt provides a lot of widgets, it does not provide a plotting widget such as the one available in PMW. However, since there is quite a community for Qt, one only has to search far to find a solution. This comes in the way of Qwt (Qt Widgets for Technical Applications). One of the classes is a plotter that is quite advanced. Not to be outdone, there is also PyQwt, which provides an interface to PyQt and uses things like Python Numeric (PyNum) and friends. From what I could see, the PyQwt folks have done a very good job and even have provided an interface that could pretty much work as a poor man's *Mathematica*. The bad news is that the documentation for Qwt [5] isn't at the same level as the documentation for, say, Qt. The good news is that we can do the plotting examples with PyQwt and get some nice results.

First up is plotdemo.py.

```
class PlotCanvasDemo(QWidget):
    def __init__(self, *parent):
        QWidget.__init__(self, *parent)
```

```
    frame = QVBox(self)
    frame.setFrameStyle(QFrame.StyledPanel | QFrame.Sunken)
    self.plotter = QwtPlot("Simple QwtPlot Demo", frame)
    label = QLabel("Options:", self)
    moveButton = QPushButton("Move Points", self)
    psButton = QPushButton("Print to File", self)
    symbolsButton = QPushButton("Symbols", self)
    quitButton = QPushButton("Quit", self)
    self.connect(moveButton, SIGNAL("clicked()"), self.movePoints)
    self.connect(psButton, SIGNAL("clicked()"), self.printPS)
    self.connect(symbolsButton, SIGNAL("clicked()"),
                 self.showSymbols)
    self.connect(quitButton, SIGNAL("clicked()"),
                 qApp, SLOT("quit()"))
    ...
    self.ncurves = 3
    self.npoints = 20
    self.vector_x = zeros(self.npoints, Float)
    self.vector_y = [zeros(self.npoints, Float)
                     for y in range(self.ncurves)]
    self.fillVectors()
    self.createPlot()
```

Figure 48: PlotCanvas constructor

This is standard code that creates a QwtPlotter and the buttons for moving points, printing the plot to file, showing symbols on the plot and quitting. We then create our curves in PyNum arrays using the same code from the original plotdemo_sptk.py script and zero them out. We then call functions to fill the vectors with data and plot the graphs.

```
def fillVectors(self, seed = True):
    if seed:
        random.seed(9)
    for index in range(self.npoints):
        self.vector_x[index] = index
        for y in range(self.ncurves):
            self.vector_y[y][index] = random.uniform(0, 8)
```

Figure 49: PlotCanvas fillVectors

Our fillVertors code borrows from code in the plotdemo_sptk.py script. The difference here is that we also add a seed argument that is True by default. If seed is True, we seed the pseudo random number generator with the value 9 and proceed to fill our PyNum arrays with random values to be used in the graph.

```
def createPlot(self):
    colors = [Qt.red, Qt.yellow, Qt.blue,
              Qt.green, Qt.black, Qt.gray]
    self.curves = []
    for i in range(self.ncurves):
```

```
            curve = self.plotter.insertCurve("Line %d" % i)
            self.plotter.setCurveData(curve, self.vector_x,
                                      self.vector_y[i])
            pen = QPen(colors[i], i + 1)
            self.plotter.setCurvePen(curve, pen)
            self.curves.append(curve)
        self.plotter.replot()
```

Figure 50: PlotCanvas createPlot

The createPlot method is the most interesting as it introduces us to the plotter interface. QwtPlot works by creating curves for each array. One can access the curve or properties of the curve through means of a key. We will store our curves in a list. When createPlot is called, we reset our list and go through our list of PyNum arrays and create a curve for each one. We also give each curve a specific color by specifying a pen. Once we've created all the pens we tell the plotter to replot, which will draw the curves in the plotter.

```
    def showSymbols(self):
        diamond = QwtSymbol()
        diamond.setStyle(QwtSymbol.Diamond)
        diamond.setBrush(QBrush(Qt.blue))
        diamond.setSize(QSize(10, 10))
        for curve in self.curves:
            self.plotter.setCurveSymbol(curve, diamond)
        self.plotter.replot()
```

Figure 51: PlotCanvas showSymbols

Here we create a QwtSymbol make it a blue diamond that is 10 pixels by 10 pixels and set it as the symbol for each curve and re-draw the curves. Now at each data point in the curves, we will have a blue diamond.

```
    def movePoints(self):
        self.plotter.clear()
        # Not exactly the same effect, but it gets the same feeling
        self.fillVectors(seed = False)
        self.createPlot()
```

Figure 52: PlotCanvas movePoints

Our movePoints function is basically a cheat, we regenerate the numbers for the arrays and replot them. This creates the illusion of the graph "moving,"

```
    def printPS(self):
        printer = QPrinter()
        # The output is unfortunately platform-specific
        # In general it's postscript on X11 systems,
        # PDF on the mac and driver-specific on windows.
        printerFileName = "tmp2.ps"
        if os.name == "posix":
            if os.uname()[0] == "Darwin":
```

```
        printerFileName = "tmp2.pdf"
elif os.name == "nt":
    printerFileName = "tmp2.prn"
printer.setOutputToFile(True)
printer.setOutputFileName(printerFileName)
self.plotter.printPlot(printer, QwtPlotPrintFilter())
```

Figure 53: PlotCanvas printPS

Finally, we have the print function. We use QPrinter to accomplish this for us. It encapsulates lots of printing issues for us. Unfortunately, this version of Qt doesn't provide a standard format for printing to file, so we will only get postscript on Qt/X11 systems. To compensate, we try to generate a proper name depending on which OS we are running. Once we have the filename, we tell the printer we want to print to a file and what its filename will be. We then pass the printer to the plotter with default print options. The QwtPlotter takes care of the rest. An alternative way of doing this would be to create a QPixmap and pass that along to printPlot instead. At that point, we could have used QPixmap to save the image to disk. Another solution would have been to use grabWidget().

The complete script gives the following window.

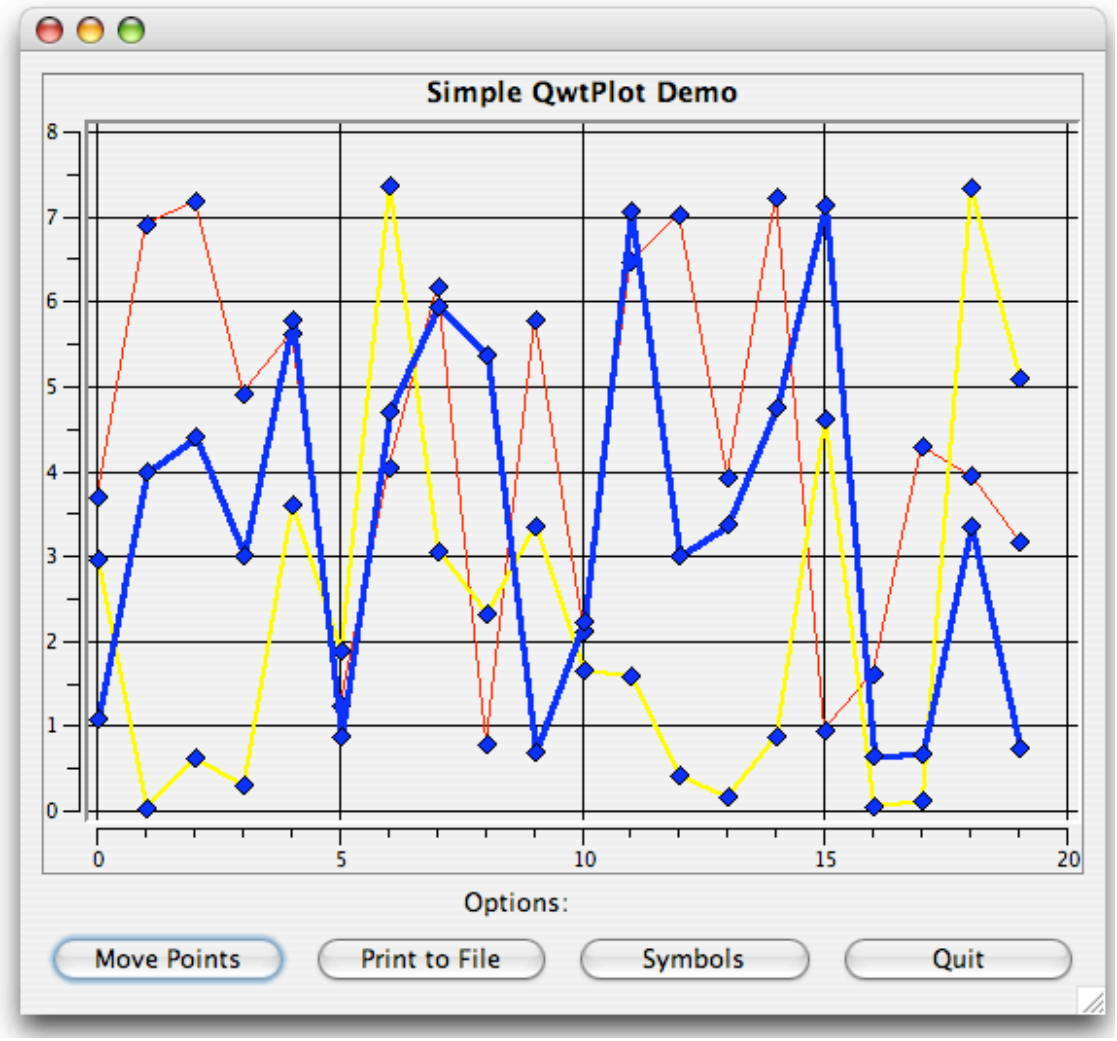

Figure 54: plotdemo.py running

Sometimes it is easier to show what is going on with animating the data points. The Tkinter example shows a nice way of doing an animation by rendering the points and then sleeping for a little while before drawing again. While this works, a better and probably less resource intensive way of doing the animation would be through a timer delivered by the event loop. This has also keeps the GUI responsive in case you want to implement other features (pausing, rewinding, zoom, etc). We'll take a look at a timer-based approach for the PyQt version of the animation script.

```
class AnimatePlot(QWidget):
    def __init__(self, func, xmin, xmax, ymin, ymax,
                 resolution=300, *parent):
        QWidget.__init__(self, *parent)
        self.func = func
        self.xmin = xmin
        self.xmax = xmax
        self.ymin = ymin
```

```
        self.ymax = ymax
        self.resolution = resolution
        self.tstop = 2; self.dt = 0.05
        self.plotter = QwtPlot("Animation Example", self)
        self.plotter.setAxisScale(QwtPlot.yLeft, self.ymin, self.ymax)
        self.plotter.setAxisScale(QwtPlot.xBottom, self.xmin,
                                  self.xmax)
        self.plotter.replot()
        self.button = QPushButton("Start", self)
        mainLayout = QVBoxLayout(self)
        mainLayout.addWidget(self.plotter)
        layout = QHBoxLayout()
        layout.addStretch()
        layout.addWidget(self.button)
        mainLayout.addLayout(layout)
        self.connect(self.button, SIGNAL("clicked()"), self.initPlot)
```

Figure 55: AnimatePlot constructor

The interface to the AnimatePlot for PyQt is very similar to that of the Tkinter class. We copy the values over and then set the tstop and dt variables. We then create a QwtPlotter and set the X- and Y-axis to be within the values passed in. We call replot to get show the new scale of the plot. We also add a button and connect it to our slot for starting the animation.

```
    def initPlot(self):
        self.plotter.clear()
        self.curve = self.plotter.insertCurve(self.func.__name__)
        self.plotter.enableLegend(True, self.curve)
        self.plotter.setCurvePen(self.curve, QPen(Qt.red))
        dx = (self.xmax - self.xmin) / float(self.resolution)
        self.x = sequence(self.xmin, self.xmax, dx)
        self.t = 0
        self.y = self.func(self.x, self.t)
        self.button.setEnabled(False)
        self.startTimer(33)
```

Figure 56: AnimatePlot initPlot

In our initPlot slot, we first clear the plot and insert a curve based on the function that was passed in the constructor. We enable the legend of the curve and give the curve a red pen to draw itself. We compute a PyNum sequence based on how many points we specified in the constructor. We start our time *t* at 0 and compute our first set of values. We disable the button as we don't want to have multiple animations going at the same time, and start a timer. The timer will then deliver a QTimerEvent to us every 33 milliseconds. This corresponds to producing around 30 frames a second, which a nice value to shot for when doing animation, since most human eyes can't see individual frames at this speed. The startTimer function returns us an ID for the timer, but since we only have one timer running, we can ignore this value.

If you've been following along, you'll guess that we deal with the timer in a timerEvent handler, and you would be right.

```
    def timerEvent(self, event):
        if self.t >= self.tstop:
            self.killTimer(event.timerId())
            self.button.setEnabled(True)
            return
        self.doPlot(self.func(self.x, self.t), self.t)
        self.t += self.dt
```

Figure 57: AnimatePlot timerEvent

When we enter the timer event, we first see if we need to continue running the timer or not. If we've reached the end of our animation, we then kill our timer using the timer ID of the event (this is the same value as was given in when we called startTimer). We re-enable the button, since someone may want to see the animation again and we then return, as we don't need to advance the animation anymore. Otherwise, we plot the points for the current time point and advance the time for the next time the timer fires.

```
    def doPlot(self, y, t):
        self.y = y
        self.plotter.setCurveData(self.curve, self.x, self.y)
        self.plotter.setCurveTitle(self.curve, '%s(x,t=%.4f) '
                                   % (self.func.__name__, t))
        self.plotter.updateLegendItem(self.curve)
        self.plotter.replot()
```

Figure 58: AnimatePlot doPlot

The plotting code is straightforward, we set the new set of points given to us in y and set it as the new curve data. We update the text of the curve title to be the current time for the function and then call methods to update the legend item and plot the new curve.

With the class defined all we need to do is create an instance of it and press the start button to show the animation.

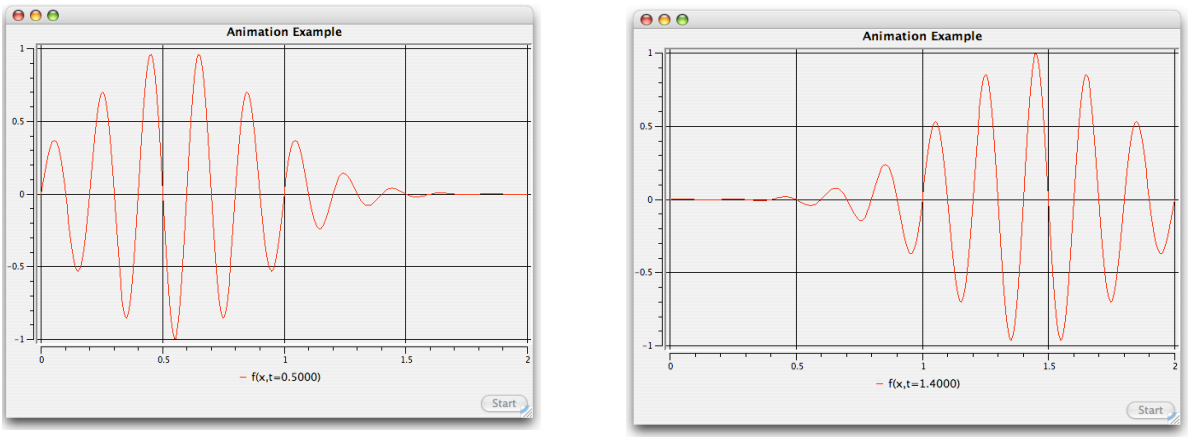

Figure 59: animate.py running

If you look through the Qt documentation, you may notice a QTimer class. This class presents a user-friendly timer that you can connect slots to and start and stop without knowing (or caring about) IDs. However, this is a fairly processor intensive script and creating the extra object and delivering the C++ signal to the Python slot was just a bit slower than using the timerEvent. We could have easily balanced this out by not firing the timer as often, but that makes the animation to jerky. As it stands the performance is acceptable, as long as you don't make the window too large.

One thing nice feature of PyQt that we didn't really get to show off is its ability to use the XML forms generated by Qt Designer (hereafter referred to as Designer) to create widgets for use in PyQt. Designer is a form builder that let's you visually create your widgets and lay them out. This is a big plus when you work on many forms or just would rather not hand-code your layout and widget creation code. This is accomplished via the command line tool pyuic. Giving pyuic a .ui file generated by designer will cause pyuic to dump the corresponding PyQt class. We can put this in a file and access it like we would any other Python class. What's more, we can write some extra Python code in a "ui.h" and pyuic will even insert the Python into the final class.
To illustrate how this works, let's do a "quick port" of the simviz GUI over to Designer. Designer itself has a fair amount of documentation and we won't bother repeating it here. But most people should be able to follow along in our whirlwind tour.

First, we start Designer and create a new widget. We then use the toolbox on the left side of the default Designer set-up to add the sliders, buttons, labels, and line edits. Along the way it is also important to use the property editor and change the names of the sliders and the line edits so they are easier to remember. We can also set default values and minimum sizes, along with many other properties too. The approach that was used here was to name them after the parameters for the script: sliderM, lineEditFunc, etc. We can also add a QLabel and set its content to be the figure we show in the script. Then we can experiment with various ways to layout widgets. The best part is that we can break the layouts and try again if we don't like its current look.

To show off the abilities of the ui.h file, we will implement all the functionality of the widget here. We bring up the edit slots dialog and create 4 slots for our widget, init(), runSimViz(), updateLabel(), and updateSliderLabel(). For the slots that require parameters, we (or rather pyuic) will give them slightly cryptic names of a0, a1, and so on. From there, we can double-click on the widget and be taken to its ui.h file. You can edit the source here, but I prefer save everything, exit Designer and do it in a dedicated text editor. But before we do that, we can use the signal-slot editor to create connections between each slider's valueChanged signal and the updateSliderLabel slot.

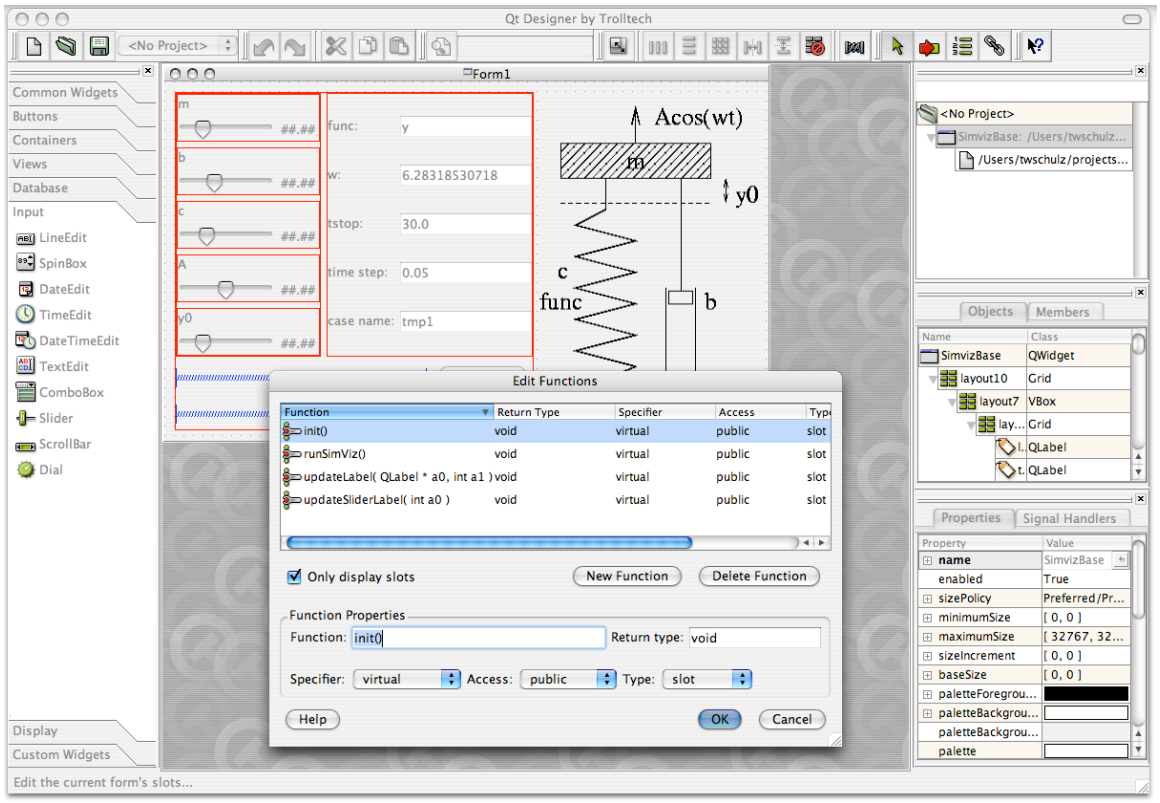

Figure 60: Qt Designer editing slots

Here are the implementations of the slots, first is init(), its a semi-unwritten rule that this function will get called from the class's constructor. Don't worry about the C++ stuff in the file, it can be safely ignored.

```
void SimvizBase::init()
{
    self.updateLabel(self.labelM, self.sliderM.value())
    self.updateLabel(self.labelB, self.sliderB.value())
    self.updateLabel(self.labelC, self.sliderC.value())
    self.updateLabel(self.labelA, self.sliderA.value())
    self.updateLabel(self.labelY0, self.sliderY0.value())
}
```

Figure 61: SimVizBase init

The init slot will make sure that the labels show the correct values when the widget is shown. Since this is no longer handled by a special widget it has to be done in at construction.

```
void SimvizBase::updateLabel(QLabel *a0, int a1)
{
    adjustedValue = a1 / 100.0
    strAdjusted = "%7.2f" % adjustedValue
    a0.setText(strAdjusted)
}

void SimvizBase::updateSliderLabel(int a0)
{
    slider = self.sender()
    label = None
    if slider == self.sliderM:
        label = self.labelM
    elif slider == self.sliderC:
        label = self.labelC
    elif slider == self.sliderB:
        label = self.labelB
    elif slider == self.sliderA:
        label = self.labelA
    else: # slider == self.sliderY0
        label = self.labelY0
    self.updateLabel(label, a0)
}

void SimvizBase::updateLabel(QLabel *a0, int a1)
{
    adjustedValue = a1 / 100.0
    strAdjusted = "%7.2f" % adjustedValue
    a0.setText(strAdjusted)
}
```

Figure 62: SimvizBase additional slots

Our updateSliderLabel uses sender to determine which label we should use and then updates that label by calling updateLabel.

```
void SimvizBase::runSimViz()
{
    import os
    # Collect the data and run it.
    commandStr = os.path.join(os.environ["scripting"], "src", "py",
                              "intro", "simviz1.py")
    commandStr += " -m " + str(self.labelM.text())
    commandStr += " -b " + str(self.labelB.text())
    commandStr += " -c " + str(self.labelC.text())
```

```
    commandStr += " -A " + str(self.labelA.text())
    commandStr += " -y0 " + str(self.labelY0.text())
    commandStr += " -func " + str(self.lineEditFunc.text())
    commandStr += " -w " + str(self.lineEditW.text())
    commandStr += " -tstop " + str(self.lineEditTStop.text())
    commandStr += " -dt " + str(self.lineEditTimeStep.text())
    commandStr += " -case " + str(self.lineEditCaseName.text())

    print "The string is: " + commandStr
    failure = os.system(commandStr)
    if failure:
        QMessageBox.critical(self, "Error",
                             "The underlying script failed to run",
                             "Dang!")
}
```

Figure 63: SimvizBase runSimViz

Our runSimViz script is a little different than the one presented earlier. Instead of walking through the dictionary, we just talk to the widgets directly. Some may argue that this connects the ui and code to closely together, but for the purposes of this example, I think the dictionary is overkill.

Now it is time to turn the .ui file into a Python class. We will usually want to do this whenever we change the files. One way of accomplishing this is to write a Makefile to run pyuic on the file. The rule to do it would look something like this:

```
simvizbase.py : simviz.ui simviz.ui.h
        pyuic simviz.ui -o simvizbase.py
```

Figure 64: A Makefile for pyuic

Once we've done that we can run make and see that we do get a Python class out of it. From browsing the file, you'll see that it doesn't do things much differently than how we originally coded them.

Finally, we can test our class trivially with this short script.

```
from qt import *
from simvizbase import *
import sys

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = SimvizBase()
    window.show()
    app.setMainWidget(window)
    app.exec_loop()
```

Figure 65: test.py

And with that, we have a fully featured GUI that should function like simviz2GUI.py, but with much less typing on our part.

It is worth mentioning that PyQt wraps the QWidgetFactory class via its qtui module. With this you can take the raw UI files generated by Designer and load them at runtime in your running script. Using this, you can make a very dynamic GUI with very little logic inside the GUI at all.

I would like to round out this paper with a conclusion about PyQt. While I have no qualms recommending Qt for a project, I'm a little bit more reserved with PyQt. PyQt is great if you know Qt, but I'm not sure if it is the logical choice for a Python programmer. The problem being that while Qt is a great API designed well for C++, this does necessarily translate into a great API for Python. For instance, Qt has a great property system, you can query what properties a widget has, read and write them, all without knowing the real function names. It would seem to be rather neat to make a QWidget's subclass's constructor more flexible by adding the properties to the constructor and addressing them as named arguments. That way, one could set the necessary properties for a widget all in one go, similar to what one can do with Tkinter.

Still, it certainly would be my choice for doing my Python GUI programming. It certainly is fun to not have to generate a Makefile and compile your programs all the time. Plus, being able to use Designer is a big plus when you want to do a bit more complex things than what is demonstrated here. I also pleasantly surprised by the work done by the PyQwt people [6], they offer a few extras (like the ability to run PyQt inside of an interactive interpreter) that really make for compelling reasons to use PyQt. Perhaps it's best to give it a test drive yourself and see how good a fit it is for you.

Work Cited

[1] Hans Petter Langtangen. Python Scripting for Computational Science. Copyright 2004. Springer-Verlag Berlin Heidelberg. pp. 205-268, 503-515

[2] PyQt Documentation. http://www.river-bank.demon.co.uk/docs/pyqt/PyQt.html

[3] Trolltech Documentation. http://doc.trolltech.com/qt/3.3

[4] Jasmin Blanchette and Mark Summerfield. C++ GUI Programming with Qt 3. Copyright 2003 Addison-Wesly.

[5] Qwt Library. http://qwt.sourceforge.net/index.html

[6] PyQwt Documentation. http://pyqwt.sourceforge.net/doc/pyqwt.html