Creating a GUI Interface with GNU Radio

1. Open a new file to create a python script. Enter the code as described below. Remember, PYTHON is sensitive to indentation.
2. The following line is required at the beginning of any GNU Radio python script.

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

3. Enter any comment lines describing the script. Example:

   ```
   #This is a tutorial designed to introduce GUI in GNU RADIO applications.
   #Sharlene Katz and Jay Flynn 9-24-08
   #Electrical and Computer Engineering Department
   #California State University Northridge
   ```

4. Enter the following import statements. These are required for scripts that contain a GUI application:

   ```
   #import the overall GUI module
   from gnuradio.wxgui import stdgui2

   #import fft module for spectrum analyzers etc.
   from gnuradio.wxgui import fftsink2

   #import scope module for oscilloscope displays
   from gnuradio.wxgui import scopesink2

   #import waterfall module for waterfall displays
   from gnuradio.wxgui import waterfallsink2

   #import number value sink module to display number values of incoming data
   from gnuradio.wxgui import numbersink2

   #import slider module
   from gnuradio.wxgui import slider

   #import module to handle forms
   from gnuradio.wxgui import form

   #import wx python
   import wx
   ```

5. Enter the following import commands. These are required for the various applications. Others may be required.

```
#import the usual gnuradio modules
from gnuradio import gr, eng_notation, blks2
from gnuradio import audio
from gnuradio.eng_option import eng_option
import sys
import math
```

6. Enter the following commands to define your class. Note that the class defintion and initialization statements are different for scripts with a GUI interface.

```
#define class based on stdgui2.std_top_block which is derived from gr.top_block
#this script is for generating a tone, controlling the frequency and displaying
#fft spectrum of the tone.

class tone_flow_graph (stdgui2.std_top_block):
    def __init__(self, frame, panel, vbox, argv):
        stdgui2.std_top_block.__init__ (self,frame,panel,vbox,argv)

# remainder of the script goes here
```

7. End your script with:

```
#this runs everything
if __name__ == '__main__':
    app = stdgui2.stdapp (tone_flow_graph, "Tone Generator")
    app.MainLoop()
```

8. Save this document and run the script. Recall that you will need to change the file permissions with:

   chmod +x *filename*.py

   and run the script with:

   ./filename.py

   An empty frame should appear. Note that it will have a window menu (to be filled in later) and two status bars. Also, the title specified will appear. NOTE: Some systems may display a very narrow window and/or give a “Critical Error “width <0” in the terminal window. If that occurs, continue with the following steps.

9. Add the following initialization statements to your script in the area "#remainder of script goes here". The indentation should be at the same level as "stdqui2.std_top_block…" since they are part of the init definition.

```
tone_rate = int(48000)
freq = 800
ampl = .5

self.frame = frame
self.panel = panel
```

10. We are going to display the output of a tone generator. First we must create the tone generator and connect it to an audio sink. The following code will do this. Enter this under your initialization statements at the same level of indentation.

```
#define tone source
    self.src1 = gr.sig_source_f (tone_rate, gr.GR_SIN_WAVE, freq, ampl)

#create a sink representing the audio card

     audio_sink = audio.sink (tone_rate)

#connect source to sink ****NOTE**** WITHOUT THIS THE PROGRAM FREEZES!!!!!
    self.connect (self.src1, audio_sink)
```

11. Run the program again. You should now hear your 800 Hz tone along with seeing the empty GUI.

12. Now we are going to define a module that allows us to write to the status boxes at the bottom of the GUI. Enter the following definition. It should be at the same level of indentation as the intialization block definition.

```
def _set_status_msg(self, msg, which=0):
     self.frame.GetStatusBar().SetStatusText(msg, which)
```

13. We also need to assign the messages to be displayed. These are added in the initialization area (directly below self.connect…).

```
self._set_status_msg("Left Message", 0)
self._set_status_msg("Right Message", 1)
```

Re-run the script and verify that your status messages are displayed. Later we will vary these to update as the program is run.

14. To begin to add information to the GUI we need to add some additional initializations and definitions. The following statements go into the initialization area to set up the GUI. These are at the same indentation level as self.connect statement and under self._set_status_msg("Right Message",1):

```
# this calls module to build gui
        self._build_gui(vbox, panel, tone_rate)
```

15. Start the definition of the _build_gui module under the self.frame.GetStatusBar… line and at the same indentation as the def _set_status_msg… line with the following:

```
    def _build_gui(self, vbox, panel, tone_rate):
```

16. The first widget that we will add to the panel is an FFT display to view the spectrum of our signal. This is accomplished with the following (remember to indent so these lines are part of the def _build_gui definition):

```
#this defines the fft display, rate, size of sample and reference level at top
#of display
        self.spectrum_display = fftsink2.fft_sink_f (panel, title="TONE OUTPUT", ref_level
= 30, fft_size=1024, sample_rate=tone_rate)


#connect the tone generator output to fft display
        self.connect (self.src1, self.spectrum_display)
#add the display as a box in our panel
#parameters???
        vbox.Add (self.spectrum_display.win, 1, wx.EXPAND)
```

After adding these lines re-run the script. You should now see the FFT display with your 800 Hz tone.

17. We would like to be able to vary the frequency and waveform of this tone from the GUI interface. The following additions are required.

In the initialization statements (under self._build_gui) add:

```
# sets starting frequency to default
        self.set_freq(freq)
#sets starting waveform to sine
       self.set_wfm('SINE')
```

Under the def build_gui add and be sure to indent to be part of the definition:

```
#this passes the value entered in the 'freq' form on the lower left of the window
        def _form_set_freq(kv):
            return self.set_freq(kv['freq'])

#this passes the value entered in the "wfm' form on the lower right of the window
        def _form_set_wfm(kv):
            return self.set_wfm(kv['wfm'])
```

Then the following is added below and at the same level as vbox.Add(self……):

```
# control area form for frequency display at bottom
        self.myform = myform = form.form()

        hbox = wx.BoxSizer(wx.HORIZONTAL)
        hbox.Add((5,0), 0)
        myform['freq'] = form.float_field(
            parent=self.panel, sizer=hbox, label="Freq", weight=1,
            callback=myform.check_input_and_call(_form_set_freq, self._set_status_msg))

#slider for freq control also at bottom
        hbox.Add((5,0), 0)
        myform['freq_slider'] = \
            form.quantized_slider_field(parent=self.panel, sizer=hbox, weight=3,
                                range=(200, 10000, 1),
                                callback=self.set_freq)

#form for entering square or sine
        hbox.Add((5,0), 0)
        myform['wfm'] = form.text_field(
              parent=self.panel, sizer=hbox, label="WFM", weight=1,
              callback=myform.check_input and_call(-form_set_wfm))

#this actually adds the boxes to frame without it, they work but are invisible
        hbox.Add((5,0), 0)
```

```
        vbox.Add(hbox, 0, wx.EXPAND)
```

NOTE: the following is a new definition and is indented one level out.

```
#define module to trap event of slider moving and what to do about it
    def on_rotate (self, event):
        self.rot += event.delta
        if (self.state == "FREQ"):
            if self.rot >= 3:
              self.set_freq(self.freq + 1)
              self.rot -= 3
            elif self.rot <=-3:
              self.set_freq(self.freq - 1)
              self.rot += 3


#define module to actually set frequency
      def set_freq(self, target_freq):

#this is handy subfunction of gr.sig_source
        self.src1.set_frequency(target_freq)


        self.freq = target_freq
        self.myform['freq'].set_value(target_freq)        # update displayed value in form
        self.myform['freq_slider'].set_value(target_freq)  # update displayed value - actually
moves slider to position proportional to freq
        self.update_status_bar()
        self._set_status_msg("OK", 0)
        return True

#             self._set_status_msg("Failed", 0)
#             return False

#define module to set waveform
      def set_wfm(self, waveform):
            if (waveform == “SQUARE”):
                self.src1.set_waveform(gr.GR_SQR_WAVE)
                self.myform[‘wfm’].set_value(waveform)   #update form display
                self.update_status_bar()
                self._set_status_msg(“SQUARE”, 0)

            elif (waveform == “SINE”):
                self.src1.set_waveform(gr.GR_SIN_WAVE)
                self.myform[‘wfm’].set_value(waveform)   #update form display
                self.update_status_bar()
```

```
                    self._set_status_msg("SINE", 0)

#this allows printing at bottom right status bar- that is set by '1' option
    def update_status_bar (self):
        msg = "Tuning..."
        self._set_status_msg(msg, 1)
```

18. Re-run the script. Use the slider switch you have just inserted to vary the frequency of the tone. Then try adding new frequencies into the form box. This code also changes the text box messages. Also enter SINE or SQUARE in the WFM form box.