Gnu and Python

Objectives:

- Get familiar with Python Blocks
- OOT modules make the actual apps / functionality (GR is the API!)
- How to add OOTs
- How to add Python blocks with gr_modtool and how to code them
- QPSK mapping
- How to add GRC bindings for block

Resources:

Follow the link below for a step-by-step tutorial, we call this TUT3.

A very good tutorial on python can be found in:

https://www.youtube.com/watch?v=rfscVS0vtbw&t=5590s&ab_channel=freeCodeCamp.org

Documentation and manuals:

https://www.gnuradio.org/doc/doxygen/classgr_1_1analog_1_1sig__source__f.html

Notes:

Open example in: Home\src\gr-tutorial\example\tutorial3\grc_files\tutorial_three_1.grc

Before installing any built module on your library, make sure that it works well, then try to install it. Uninstalling and reinstalling could be burdensome.

Check point: Once your QPSK Module is working, please contact the lab instructor.

Intro to Using GNU Radio with Python

This tutorial goes through three parts. The first is how to modify, create, or simply understand the Python generated files GRC produces for us. The second is how to create our own custom out-of-tree (OOT) modules from the ground up. Lastly we go through an actual project to get more practice and build intuition on how we can use GNU Radio in our own project. As with the last tutorial, all the content - pictures, source code, and grc files - is included in the gr-tutorial repository which we should have a local copy if we followed the directions from the GRC Tutorial.

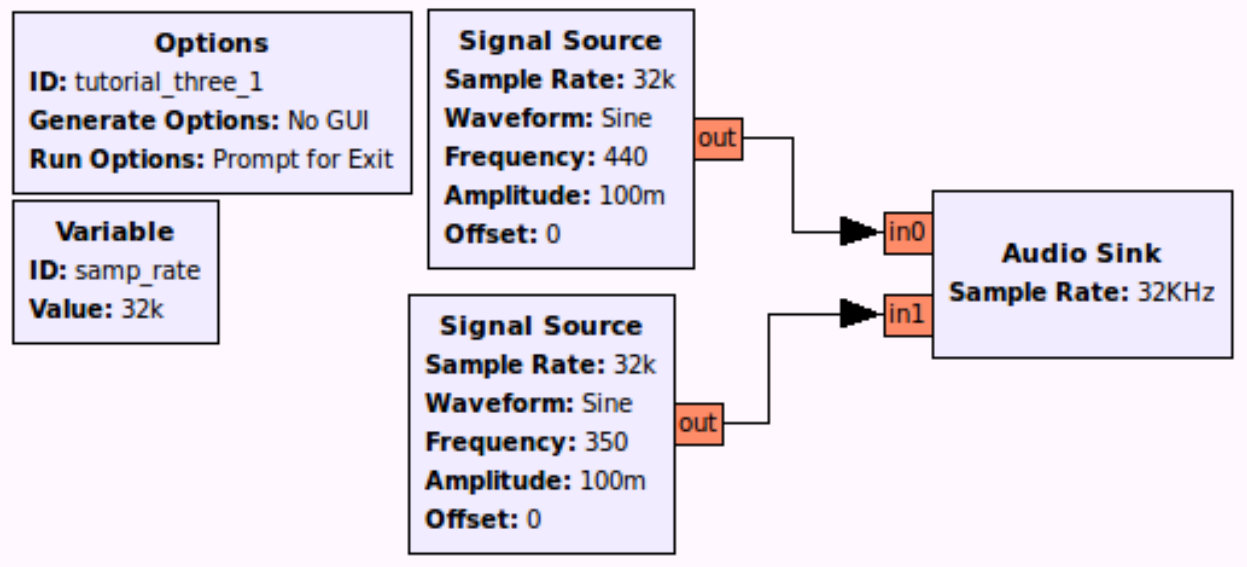
Again we should have a directory with the solutions and a directory with our work as below:

```
1 /home/user/gnuradio/tutorials/solutions
2 /home/user/gnuradio/tutorials/work
```

As a rule, if we hover over the GRC flowgraph images, we will be able to see the corresponding filename. Same applies for other images. Full code files are collapsed with the filename in the collapsed handle.

GRC Generated Python Files

Let us look at a dial-tone example on the GRC:



When we click the **Generate** button, the terminal tells us it produced a .py file so let's open that to examine its code which is reproduced below:

```
#!/usr/bin/env Python
##################################################
# Gnuradio Python Flow Graph
# Title: Tutorial Three
# Generated: Wed Mar 12 15:35:18 2014
##################################################

from gnuradio import analog
from gnuradio import audio
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from optparse import OptionParser

class tutorial_three(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self, "Tutorial Three")

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 32000

        ##################################################
        # Blocks
        ##################################################
        self.audio_sink_0 = audio.sink(samp_rate, "", True)
        self.analog_sig_source_x_1 = analog.sig_source_f(samp_rate, analog.GR_SIN_WAVE,
350, .1, 0)
        self.analog_sig_source_x_0 = analog.sig_source_f(samp_rate, analog.GR_SIN_WAVE,
440, .1, 0)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.analog_sig_source_x_1, 0), (self.audio_sink_0, 1))
        self.connect((self.analog_sig_source_x_0, 0), (self.audio_sink_0, 0))

```

```
39
40 # QT sink close method reimplementation
41
42     def get_samp_rate(self):
43         return self.samp_rate
44
45     def set_samp_rate(self, samp_rate):
46         self.samp_rate = samp_rate
47         self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)
48         self.analog_sig_source_x_1.set_sampling_freq(self.samp_rate)
49
50 if __name__ == '__main__':
51     parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
52     (options, args) = parser.parse_args()
53     tb = tutorial_three()
54     tb.start()
55     raw_input('Press Enter to quit: ')
56     tb.stop()
57     tb.wait()
```

The first thing for us to realize is that the GRC can generate Python files that we can then modify to do things we wouldn't be able to do in GNU Radio Companion such as perform simulations. The libraries available in Python open up a whole new realm of possibilities! For now, we will explore the structure of the GRC Python files so we are comfortable creating more interesting applications.

Hello World Dissected

While examining the code, we need to get familiar with documentation. GNU Radio uses Doxygen (the software) for the GNU Radio Manual. The easiest way to go through the documentation is to go through the functions that we use so let us simplify our code by only including the bare bones needed to run the dial-tone example.

```
1 #!/usr/bin/env Python
2 from gnuradio import gr
3 from gnuradio import audio
4 from gnuradio import analog
5
6 class my_top_block(gr.top_block):
7     def __init__(self):
8         gr.top_block.__init__(self)
9
10         sample_rate = 32000
11         ampl = 0.1
12
13         src0 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 350, ampl)
14         src1 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 440, ampl)
15         dst = audio.sink(sample_rate, "")
16         self.connect(src0, (dst, 0))
17         self.connect(src1, (dst, 1))
18
19 if __name__ == '__main__':
20     try:
21         my_top_block().run()
22     except [[KeyboardInterrupt]]:
23         pass
```

Let us examine this line by line:

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

Tells the shell that this file is a Python file and to use the Python interpreter to run this file. Should always be included at the top to run from the terminal.

```
from gnuradio import gr
from gnuradio import audio
from gnuradio import analog
```

Tells Python the modules to include. We must always have **gr** to run GNU Radio applications. The audio sink is included in the audio module and the signal_source_f is included in the analog module which is why we include them. PEP8 tells us we should import every module on its own line.

```
class my_top_block(gr.top_block):
```

Define a class called "my_top_block" which is derived from another class, **gr.top_block**. This class is basically a container for the flow graph. By deriving from gr.top_block, we get all the hooks and functions we need to add blocks and interconnect them.

```
def __init__(self):
```

Only one member function is defined for this class: the function "*init*()", which is the constructor of this class.

```
gr.top_block.__init__(self)
```

The parent constructor is called (in Python, this needs to be done explicitly. Most things in Python need to be done explicitly; in fact, this is one main Python principle).

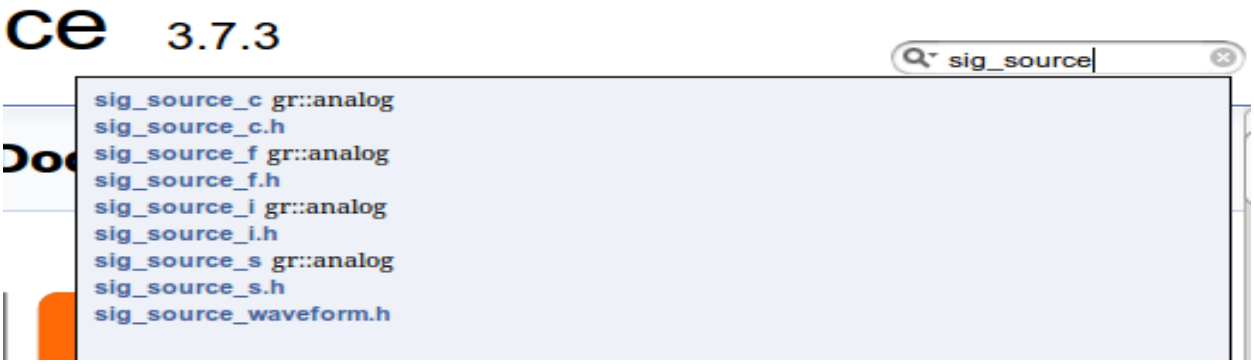
```
sample_rate = 32000
ampl = 0.1
```

Variable declarations for sampling rate and amplitude that we will later use.

A Look at Documentation

```
src0 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 350, ampl)
src1 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 440, ampl)
```

Here we are using functions from GNU Radio so let's have a look at the documentation for **analog.sig_source_f** which is available in the GNU Radio manual. We can find it easily by using the search function as below:



We can then scroll down to **Member Function Documentation** to see how the function is used and the parameters it accepts as below:

```
static sptr gr::analog::sig_source_f::make ( double                    sampling_freq,
                                             gr::analog::gr_waveform_t waveform,
                                             double                    wave_freq,
                                             double                    ampl,
                                             float                     offset = 0
                                           )
```

Build a signal source block.

Parameters

sampling_freq	Sampling rate of signal.
waveform	wavetform type.
wave_freq	Frequency of waveform (relative to sampling_freq).
ampl	Signal amplitude.
offset	offset of signal.

We can see that our function **analog.sig_source_f** takes in 5 parameters but in our code we are only using 4. There is no error because the last input **offset** is set to "0" by default as shown in the documentation. The first input is the **sampling_freq** which we defined as **sample_rate** in our code. The second input is asking for a **gr::analog::gr_waveform_t** waveform so let's click that link to find out more.

Enumeration Type Documentation

enum gr::analog::gr_waveform_t

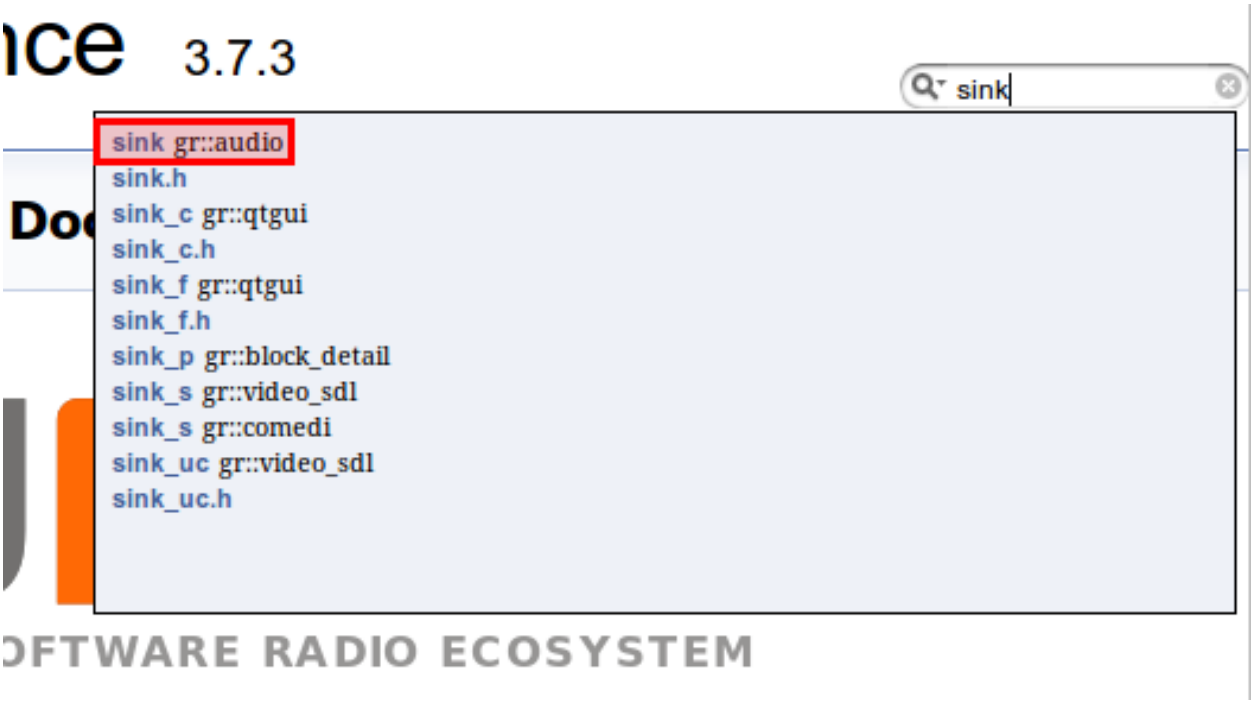
Types of signal generator waveforms.

Enumerator	
GR_CONST_WAVE	
GR_SIN_WAVE	
GR_COS_WAVE	
GR_SQR_WAVE	
GR_TRI_WAVE	
GR_SAW_WAVE	

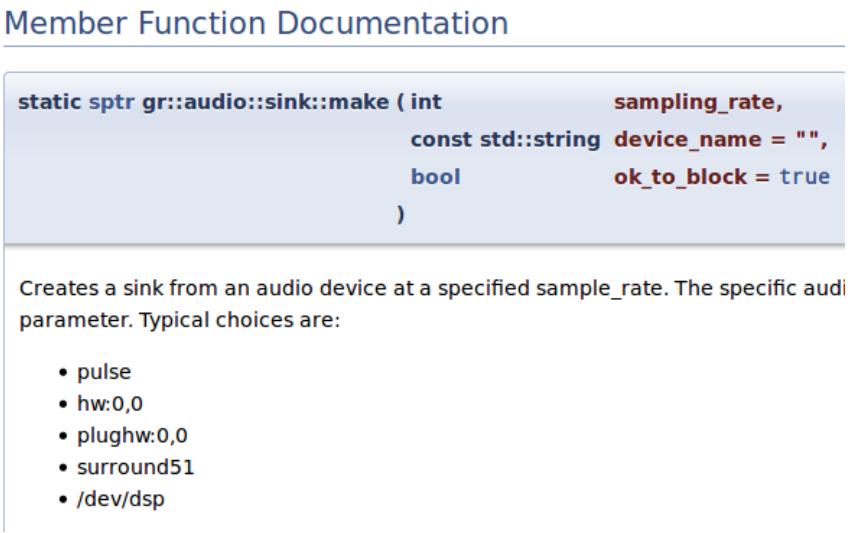
We can see that there are a couple of options to choose from. In this case we chose **analog.GR_SIN_WAVE**. The third input is **wave_freq** which we input "350" or "440". The fourth input is **ampl** which we defined as **ampl**.

```
dst = audio.sink(sample_rate, "")
```

Because documentation is so important, let's look at another example. Again, we can look at the documentation for **audio.sink** which is available on the GNU Radio Manual through the search function:



We can then as before scroll down to the **Member Function Documentation** to see the parameters it accepts:



This time we have 3 inputs with the last being optional. In our code, for the first input **sampling_rate** we used are **sample_rate** variable. In the second input, we have a choice for what device to use for audio output. If we leave it alone as "" then it'll choose the default on our machine.

Connecting the Block Together

```
self.connect(src0, (dst, 0))
self.connect(src1, (dst, 1))
```

The general syntax for connecting blocks is self.connect(block1, block2, block3, ...) which would connect the output of block1 with the input of block2, the output of block2 with the input of block3 and so on. We can connect as many blocks as we wish with one connect() call. However this only work when there is a one-to-one correspondence. If we go back to our initial flowgraph, there are 2 inputs to the **Audio Sink** block. The way to connect them is by using the syntax above. The first line connects the only output of src0 (350 Hz waveform) to the first input of dst (Audio Sink). The second line connects the only output of src1 (440 Hz waveform) to the second input of dst (Audio Sink). The code so far is equivalent to the flowgraph we have created in the beginning; the rest of the lines simply start the flowgraph and provide a keyboard interrupt.

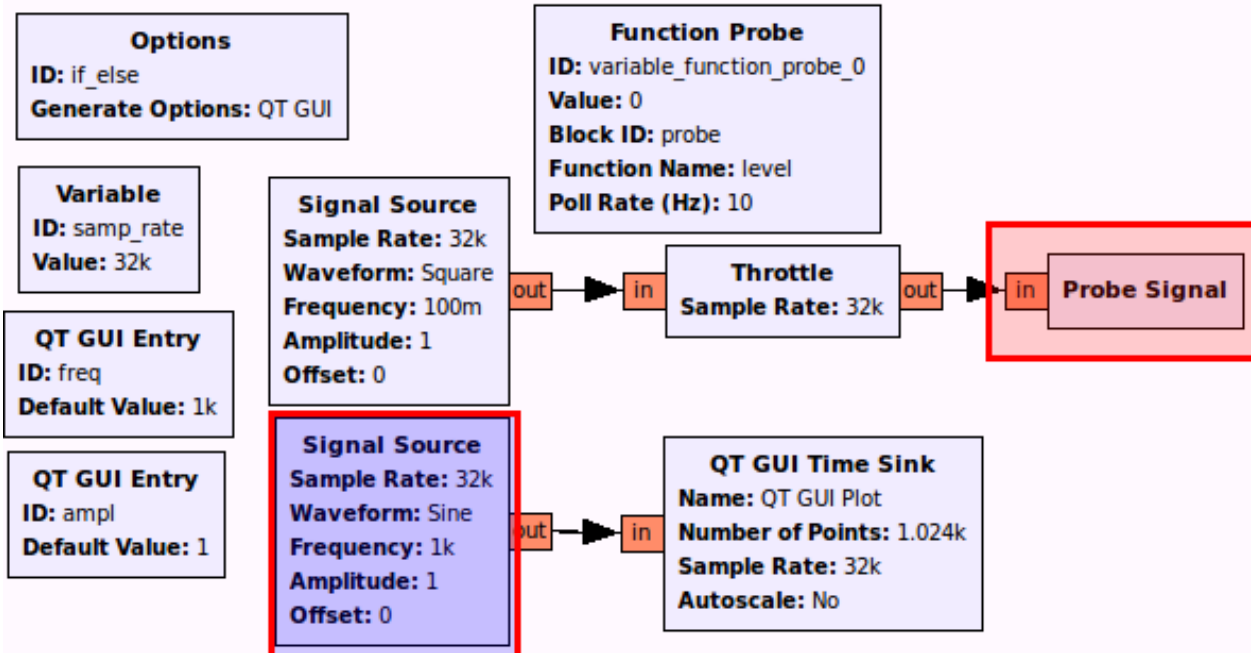
```
if __name__ == '__main__':
    try:
        my_top_block().run()
    except KeyboardInterrupt:
        pass
```

Luckily we are past the early years of GNU Radio when there was no GRC to make the Python files for us. Nowadays we can simply click things together in GRC instead of having to write code in Python to build flowgraphs. Still, a good understanding of what is going on every time we run GRC is good to know as it gives us more control of what we want the program to do.

Modifying the GRC Generated Python File

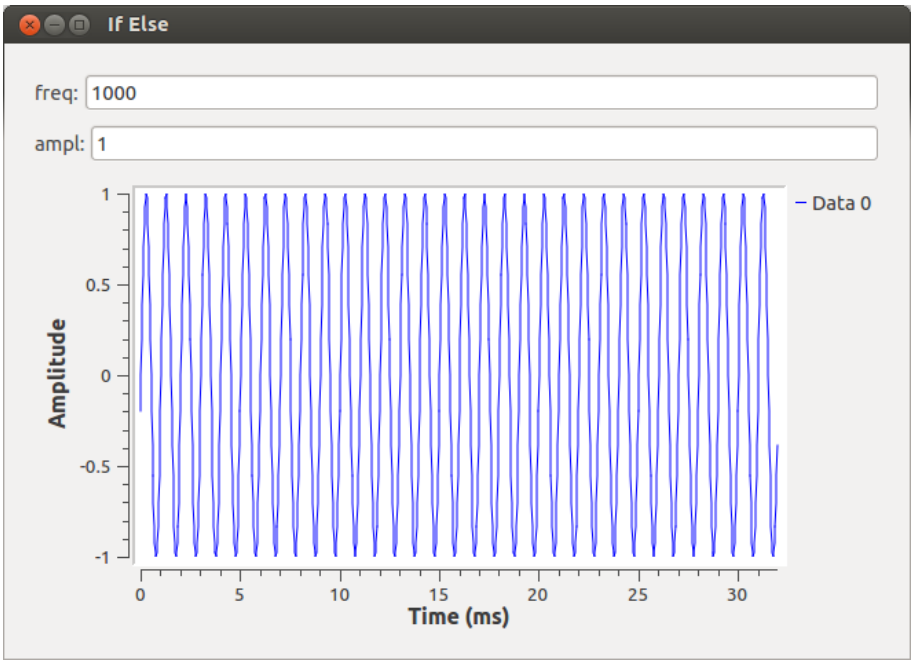
For instance, what if we wanted to change variables such as frequency and amplitude when a certain event occurs. How do we implement if statements, state machines, etc in GNU Radio? One way is to create our own blocks which we will delve into at length later. Another is to modify our GRC generated python file.

Our friend heard we were into RF so being cheap he has asked us to power his house using RF. He wants us to give him high power during the day so he can watch TV and play video games while at night give him low power so he power his alarm clock to wake up for work in the morning. We first need to setup the clock which keeps track of the day and gives us 1 for day or 0 for night. Once we get the time we can send him power through a sine wave using our massive terawatt amplifier and massive dish in our backyard. We did the calculations and we want to pulse at 1kHz, 1 amplitude during the day and 100Hz, 0.3 amplitude at night. Here's what we came up with in GRC:



- **Frequency** to "freq", **Amplitude** to "ampl"
- **ID** to "probe"
- Everything else is visible

The top section keeps track of time and will switch from 0 to 1 while the bottom section sends the pulse. The problem we encounter however is that there is no if-statement block. Sure we can tie the probe to the frequency as we did in tutorial2 for the singing sine wave but that only allows changing by a factor. What if we wanted to change multiple things and not by a linear factor? Let's start by running the flowgraph to make sure we get the output as below:



Now we can open up the GRC generated python file if_else.py which is copied below:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: If Else
# Generated: Thu Sep 13 11:39:57 2018
##################################################

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt4 import Qt
from gnuradio import analog
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import qtgui
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from optparse import OptionParser
import sip
import sys
import threading
import time
from gnuradio import qtgui


class if_else(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "If Else")
        Qt.QWidget.__init__(self)
```

```
40        self.setWindowTitle("If Else")
41        qtgui.util.check_set_qss()
42        try:
43            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
44        except:
45            pass
46        self.top_scroll_layout = Qt.QVBoxLayout()
47        self.setLayout(self.top_scroll_layout)
48        self.top_scroll = Qt.QScrollArea()
49        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
50        self.top_scroll_layout.addWidget(self.top_scroll)
51        self.top_scroll.setWidgetResizable(True)
52        self.top_widget = Qt.QWidget()
53        self.top_scroll.setWidget(self.top_widget)
54        self.top_layout = Qt.QVBoxLayout(self.top_widget)
55        self.top_grid_layout = Qt.QGridLayout()
56        self.top_layout.addLayout(self.top_grid_layout)
57
58        self.settings = Qt.QSettings("GNU Radio", "if_else")
59        self.restoreGeometry(self.settings.value("geometry").toByteArray())
60
61
62        ##################################################
63        # Variables
64        ##################################################
65        self.variable_function_probe_0 = variable_function_probe_0 = 0
66        self.samp_rate = samp_rate = 32000
67        self.freq = freq = 1000
68        self.ampl = ampl = 1
69
70        ##################################################
71        # Blocks
72        ##################################################
73        self.probe = blocks.probe_signal_f()
74        self._freq_tool_bar = Qt.QToolBar(self)
75        self._freq_tool_bar.addWidget(Qt.QLabel("freq"+": "))
76        self._freq_line_edit = Qt.QLineEdit(str(self.freq))
77        self._freq_tool_bar.addWidget(self._freq_line_edit)
78        self._freq_line_edit.returnPressed.connect(
79         lambda: self.set_freq(int(str(self._freq_line_edit.text().toAscii()))))
80        self.top_grid_layout.addWidget(self._freq_tool_bar)
```

```
81          self._ampl_tool_bar = Qt.QToolBar(self)
82          self._ampl_tool_bar.addWidget(Qt.QLabel("ampl"+": "))
83          self._ampl_line_edit = Qt.QLineEdit(str(self.ampl))
84          self._ampl_tool_bar.addWidget(self._ampl_line_edit)
85          self._ampl_line_edit.returnPressed.connect(
86           lambda: self.set_ampl(int(str(self._ampl_line_edit.text().toAscii()))))
87          self.top_grid_layout.addWidget(self._ampl_tool_bar)
88
89          def _variable_function_probe_0_probe():
90              while True:
91                  val = self.probe.level()
92                  try:
93                      self.set_variable_function_probe_0(val)
94                  except AttributeError:
95                      pass
96                  time.sleep(1.0 / (10))
97          _variable_function_probe_0_thread =
threading.Thread(target=_variable_function_probe_0_probe)
98          _variable_function_probe_0_thread.daemon = True
99          _variable_function_probe_0_thread.start()
100
101          self.qtgui_time_sink_x_0 = qtgui.time_sink_f(
102          1024, #size
103          samp_rate, #samp_rate
104          'QT GUI Plot', #name
105          1 #number of inputs
106          )
107          self.qtgui_time_sink_x_0.set_update_time(0.10)
108          self.qtgui_time_sink_x_0.set_y_axis(-1, 1)
109
110          self.qtgui_time_sink_x_0.set_y_label('Amplitude', "")
111
112          self.qtgui_time_sink_x_0.enable_tags(-1, True)
113          self.qtgui_time_sink_x_0.set_trigger_mode(qtgui.TRIG_MODE_FREE,
qtgui.TRIG_SLOPE_POS, 0.0, 0, 0, "")
114          self.qtgui_time_sink_x_0.enable_autoscale(False)
115          self.qtgui_time_sink_x_0.enable_grid(False)
116          self.qtgui_time_sink_x_0.enable_axis_labels(True)
117          self.qtgui_time_sink_x_0.enable_control_panel(False)
118          self.qtgui_time_sink_x_0.enable_stem_plot(False)
119
```

```
120         if not True:
121           self.qtgui_time_sink_x_0.disable_legend()
122 
123         labels = ['', '', '', '', '',
124                   '', '', '', '', '']
125         widths = [1, 1, 1, 1, 1,
126                   1, 1, 1, 1, 1]
127         colors = ["blue", "red", "green", "black", "cyan",
128                   "magenta", "yellow", "dark red", "dark green", "blue"]
129         styles = [1, 1, 1, 1, 1,
130                   1, 1, 1, 1, 1]
131         markers = [-1, -1, -1, -1, -1,
132                    -1, -1, -1, -1, -1]
133         alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
134                   1.0, 1.0, 1.0, 1.0, 1.0]
135 
136         for i in xrange(1):
137             if len(labels[i]) == 0:
138                 self.qtgui_time_sink_x_0.set_line_label(i, "Data {0}".format(i))
139             else:
140                 self.qtgui_time_sink_x_0.set_line_label(i, labels[i])
141             self.qtgui_time_sink_x_0.set_line_width(i, widths[i])
142             self.qtgui_time_sink_x_0.set_line_color(i, colors[i])
143             self.qtgui_time_sink_x_0.set_line_style(i, styles[i])
144             self.qtgui_time_sink_x_0.set_line_marker(i, markers[i])
145             self.qtgui_time_sink_x_0.set_line_alpha(i, alphas[i])
146 
147         self._qtgui_time_sink_x_0_win = sip.wrapinstance(self.qtgui_time_sink_x_0.pyqwidget(),
Qt.QWidget)
148         self.top_grid_layout.addWidget(self._qtgui_time_sink_x_0_win)
149         self.blocks_throttle_0 = blocks.throttle(gr.sizeof_float*1, samp_rate,True)
150         self.analog_sig_source_x_1 = analog.sig_source_f(samp_rate, analog.GR_SIN_WAVE,
freq, ampl, 0)
151         self.analog_sig_source_x_0 = analog.sig_source_f(samp_rate, analog.GR_SQR_WAVE,
0.1, 1, 0)
152 
153 
154 
155         ##################################################
156         # Connections
157         ##################################################
```

```
158         self.connect((self.analog_sig_source_x_0, 0), (self.blocks_throttle_0, 0))
159         self.connect((self.analog_sig_source_x_1, 0), (self.qtgui_time_sink_x_0, 0))
160         self.connect((self.blocks_throttle_0, 0), (self.probe, 0))
161 
162     def closeEvent(self, event):
163         self.settings = Qt.QSettings("GNU Radio", "if_else")
164         self.settings.setValue("geometry", self.saveGeometry())
165         event.accept()
166 
167     def get_variable_function_probe_0(self):
168         return self.variable_function_probe_0
169 
170     def set_variable_function_probe_0(self, variable_function_probe_0):
171         self.variable_function_probe_0 = variable_function_probe_0
172 
173     def get_samp_rate(self):
174         return self.samp_rate
175 
176     def set_samp_rate(self, samp_rate):
177         self.samp_rate = samp_rate
178         self.qtgui_time_sink_x_0.set_samp_rate(self.samp_rate)
179         self.blocks_throttle_0.set_sample_rate(self.samp_rate)
180         self.analog_sig_source_x_1.set_sampling_freq(self.samp_rate)
181         self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)
182 
183     def get_freq(self):
184         return self.freq
185 
186     def set_freq(self, freq):
187         self.freq = freq
188         Qt.QMetaObject.invokeMethod(self._freq_line_edit, "setText", Qt.Q_ARG("QString",
str(self.freq)))
189         self.analog_sig_source_x_1.set_frequency(self.freq)
190 
191     def get_ampl(self):
192         return self.ampl
193 
194     def set_ampl(self, ampl):
195         self.ampl = ampl
196         Qt.QMetaObject.invokeMethod(self._ampl_line_edit, "setText", Qt.Q_ARG("QString",
str(self.ampl)))
```

```
197         self.analog_sig_source_x_1.set_amplitude(self.ampl)
198
199
200 def main(top_block_cls=if_else, options=None):
201
202     from distutils.version import StrictVersion
203     if StrictVersion(Qt.qVersion()) >= StrictVersion("4.5.0"):
204         style = gr.prefs().get_string('qtgui', 'style', 'raster')
205         Qt.QApplication.setGraphicsSystem(style)
206     qapp = Qt.QApplication(sys.argv)
207
208     tb = top_block_cls()
209     tb.start()
210     tb.show()
211
212     def quitting():
213         tb.stop()
214         tb.wait()
215     qapp.connect(qapp, Qt.SIGNAL("aboutToQuit()"), quitting)
216     qapp.exec_()
217
218
219 if __name__ == '__main__':
220     main()
```

We are only concerned about a couple of parts namely the part where the probe is being read:

```
    def _variable_function_probe_0_probe():
        while True:
            val = self.probe.level()
            try:
                self.set_variable_function_probe_0(val)
            except AttributeError:
                pass
            time.sleep(1.0 / (10))
```

We can see that the variable **val** is obtaining the value of the probe block. We can write our if-else statement here based on the value of **val** to change the amplitude and frequency of our sine wave. But how do we change the frequency and amplitude? We can use the part where the **QT GUI Entry** updates the flowgraph. For the variable freq:

```
    def set_freq(self, freq):
        self.freq = freq
        Qt.QMetaObject.invokeMethod(self._freq_line_edit, "setText", Qt.Q_ARG("QString",
str(self.freq)))
        self.analog_sig_source_x_1.set_frequency(self.freq)
```

and for the variable ampl:

```
    def set_ampl(self, ampl):
        self.ampl = ampl
        Qt.QMetaObject.invokeMethod(self._ampl_line_edit, "setText", Qt.Q_ARG("QString",
str(self.ampl)))
        self.analog_sig_source_x_1.set_amplitude(self.ampl)
```

We can see that the functions set_ampl and set_freq can be used for just that - setting the amplitude and the frequency. Thus we can go back and modify our probe function with the if-else statement to give power to our friend.

```
    def _variable_function_probe_0_probe():
        while True:
            val = self.probe.level()
            print val
            if val == 1:
                self.set_ampl(1)
                self.set_freq(1000)
            else:
                self.set_ampl(.3)
                self.set_freq(100)
            try:
                self.set_variable_function_probe_0(val)
            except AttributeError:
                pass
            time.sleep(1.0 / (10))
```

Now there is one more thing we need to take care of. GRC has compiled the python file in the order of creation of the elements, which was okay as long as there were no crossreferences. With the introduced adaptation (calling set_ampl and set_freq inside the _variable_function_probe_0_probe()) we need to fix the order of declarations. As set_ampl and set_freq both modify parameters of analog_sig_source_x_1 but analog_sig_source_x_1 is not

instantiated before line 150, we have to move the declaration of the _variable_function_probe_0_probe() and everything related below that.

```
        self._qtgui_time_sink_x_0_win = sip.wrapinstance(self.qtgui_time_sink_x_0.pyqwidget(), Qt.QWidget)
        self.top_grid_layout.addWidget(self._qtgui_time_sink_x_0_win)
        self.blocks_throttle_0 = blocks.throttle(gr.sizeof_float*1, samp_rate,True)
        self.analog_sig_source_x_1 = analog.sig_source_f(samp_rate, analog.GR_SIN_WAVE, freq, ampl, 0)
        self.analog_sig_source_x_0 = analog.sig_source_f(samp_rate, analog.GR_SQR_WAVE, 0.1, 1, 0)
```

Full code copied below:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: If Else
# Generated: Thu Sep 13 11:39:57 2018
##################################################

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt4 import Qt
from gnuradio import analog
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import qtgui
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
```

```
from optparse import OptionParser
import sip
import sys
import threading
import time
from gnuradio import qtgui


class if_else(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "If Else")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("If Else")
        qtgui.util.check_set_qss()
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except:
            pass
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QVBoxLayout(self.top_widget)
        self.top_grid_layout = Qt.QGridLayout()
        self.top_layout.addLayout(self.top_grid_layout)

        self.settings = Qt.QSettings("GNU Radio", "if_else")
        self.restoreGeometry(self.settings.value("geometry").toByteArray())


        ##################################################
        # Variables
        ##################################################
        self.variable_function_probe_0 = variable_function_probe_0 = 0
        self.samp_rate = samp_rate = 32000
        self.freq = freq = 1000
```

```
        self.ampl = ampl = 1

        ##################################################
        # Blocks
        ##################################################
        self.probe = blocks.probe_signal_f()
        self._freq_tool_bar = Qt.QToolBar(self)
        self._freq_tool_bar.addWidget(Qt.QLabel("freq"+": "))
        self._freq_line_edit = Qt.QLineEdit(str(self.freq))
        self._freq_tool_bar.addWidget(self._freq_line_edit)
        self._freq_line_edit.returnPressed.connect(
            lambda: self.set_freq(int(str(self._freq_line_edit.text().toAscii()))))
        self.top_grid_layout.addWidget(self._freq_tool_bar)
        self._ampl_tool_bar = Qt.QToolBar(self)
        self._ampl_tool_bar.addWidget(Qt.QLabel("ampl"+": "))
        self._ampl_line_edit = Qt.QLineEdit(str(self.ampl))
        self._ampl_tool_bar.addWidget(self._ampl_line_edit)
        self._ampl_line_edit.returnPressed.connect(
            lambda: self.set_ampl(int(str(self._ampl_line_edit.text().toAscii()))))
        self.top_grid_layout.addWidget(self._ampl_tool_bar)

        self.qtgui_time_sink_x_0 = qtgui.time_sink_f(
            1024, #size
            samp_rate, #samp_rate
            'QT GUI Plot', #name
            1 #number of inputs
        )
        self.qtgui_time_sink_x_0.set_update_time(0.10)
        self.qtgui_time_sink_x_0.set_y_axis(-1, 1)

        self.qtgui_time_sink_x_0.set_y_label('Amplitude', "")

        self.qtgui_time_sink_x_0.enable_tags(-1, True)
        self.qtgui_time_sink_x_0.set_trigger_mode(qtgui.TRIG_MODE_FREE,
qtgui.TRIG_SLOPE_POS, 0.0, 0, 0, "")
        self.qtgui_time_sink_x_0.enable_autoscale(False)
        self.qtgui_time_sink_x_0.enable_grid(False)
        self.qtgui_time_sink_x_0.enable_axis_labels(True)
        self.qtgui_time_sink_x_0.enable_control_panel(False)
        self.qtgui_time_sink_x_0.enable_stem_plot(False)
```

```
        if not True:
          self.qtgui_time_sink_x_0.disable_legend()

        labels = ['', '', '', '', '',
                  '', '', '', '', '']
        widths = [1, 1, 1, 1, 1,
                  1, 1, 1, 1, 1]
        colors = ["blue", "red", "green", "black", "cyan",
                  "magenta", "yellow", "dark red", "dark green", "blue"]
        styles = [1, 1, 1, 1, 1,
                  1, 1, 1, 1, 1]
        markers = [-1, -1, -1, -1, -1,
                   -1, -1, -1, -1, -1]
        alphas = [1.0, 1.0, 1.0, 1.0, 1.0,
                  1.0, 1.0, 1.0, 1.0, 1.0]

        for i in xrange(1):
            if len(labels[i]) == 0:
                self.qtgui_time_sink_x_0.set_line_label(i, "Data {0}".format(i))
            else:
                self.qtgui_time_sink_x_0.set_line_label(i, labels[i])
            self.qtgui_time_sink_x_0.set_line_width(i, widths[i])
            self.qtgui_time_sink_x_0.set_line_color(i, colors[i])
            self.qtgui_time_sink_x_0.set_line_style(i, styles[i])
            self.qtgui_time_sink_x_0.set_line_marker(i, markers[i])
            self.qtgui_time_sink_x_0.set_line_alpha(i, alphas[i])

        self._qtgui_time_sink_x_0_win = sip.wrapinstance(self.qtgui_time_sink_x_0.pyqwidget(),
Qt.QWidget)
        self.top_grid_layout.addWidget(self._qtgui_time_sink_x_0_win)
        self.blocks_throttle_0 = blocks.throttle(gr.sizeof_float*1, samp_rate,True)
        self.analog_sig_source_x_1 = analog.sig_source_f(samp_rate, analog.GR_SIN_WAVE, freq,
ampl, 0)
        self.analog_sig_source_x_0 = analog.sig_source_f(samp_rate, analog.GR_SQR_WAVE, 0.1, 1,
0)

        def _variable_function_probe_0_probe():
            while True:
                val = self.probe.level()
                print val
                if val == 1:
```

```
                self.set_ampl(1)
                self.set_freq(1000)
            else:
                self.set_ampl(.3)
                self.set_freq(100)
            try:
                self.set_variable_function_probe_0(val)
            except AttributeError:
                pass
            time.sleep(1.0 / (10))
        _variable_function_probe_0_thread =
threading.Thread(target=_variable_function_probe_0_probe)
        _variable_function_probe_0_thread.daemon = True
        _variable_function_probe_0_thread.start()

        ##################################################
        # Connections
        ##################################################
        self.connect((self.analog_sig_source_x_0, 0), (self.blocks_throttle_0, 0))
        self.connect((self.analog_sig_source_x_1, 0), (self.qtgui_time_sink_x_0, 0))
        self.connect((self.blocks_throttle_0, 0), (self.probe, 0))

    def closeEvent(self, event):
        self.settings = Qt.QSettings("GNU Radio", "if_else")
        self.settings.setValue("geometry", self.saveGeometry())
        event.accept()

    def get_variable_function_probe_0(self):
        return self.variable_function_probe_0

    def set_variable_function_probe_0(self, variable_function_probe_0):
        self.variable_function_probe_0 = variable_function_probe_0

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.qtgui_time_sink_x_0.set_samp_rate(self.samp_rate)
        self.blocks_throttle_0.set_sample_rate(self.samp_rate)
        self.analog_sig_source_x_1.set_sampling_freq(self.samp_rate)
```

```
        self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)

    def get_freq(self):
        return self.freq

    def set_freq(self, freq):
        self.freq = freq
        Qt.QMetaObject.invokeMethod(self._freq_line_edit, "setText", Qt.Q_ARG("QString",
str(self.freq)))
        self.analog_sig_source_x_1.set_frequency(self.freq)

    def get_ampl(self):
        return self.ampl

    def set_ampl(self, ampl):
        self.ampl = ampl
        Qt.QMetaObject.invokeMethod(self._ampl_line_edit, "setText", Qt.Q_ARG("QString",
str(self.ampl)))
        self.analog_sig_source_x_1.set_amplitude(self.ampl)


def main(top_block_cls=if_else, options=None):

    from distutils.version import StrictVersion
    if StrictVersion(Qt.qVersion()) >= StrictVersion("4.5.0"):
        style = gr.prefs().get_string('qtgui', 'style', 'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)

    tb = top_block_cls()
    tb.start()
    tb.show()

    def quitting():
        tb.stop()
        tb.wait()
    qapp.connect(qapp, Qt.SIGNAL("aboutToQuit()"), quitting)
    qapp.exec_()


if __name__ == '__main__':
```

```
main()
```

We can then simply run our flowgraph from outside of GRC using

```
$ python if_else_mod.py
```

We should be able to see the numbers 0 and 1 on the terminal and the sine wave changing amplitude and frequency as the numbers change.

This tutorial is merely an introduction on using python in GNU Radio, for a more advanced tutorial see TutorialsWritePythonApplications .

Where Do Blocks Come From?

Now that we have covered some of the ways we can modify our GRC generated Python files, we can see how to make our own blocks to add functionality to suit our specific project needs. Let us start off simple to get a feel for the waters. We will be making a block that is able to multiply a signal by the number we specify. The first thing we need to realize is that GNU Radio comes with gr_modtool, a utility that makes creating OOT Modules easy. Let us open up a terminal and begin in a project folder for the tutorial three.

Using gr_modtool

Before we begin, we need to figure out what the commands for gr_modtool are so let's ask for help.

```
$ gr_modtool help
Usage:
gr_modtool  [options] -- Run  with the given options.
gr_modtool help -- Show a list of commands.
gr_modtool help  -- Shows the help for a given command.

List of possible commands:

Name      Aliases         Description
==========================================================
==========
disable   dis             Disable block (comments out CMake entries for files)
info      getinfo,inf     Return information about a given module
remove    rm,del          Remove block (delete files and remove Makefile entries)
makexml   mx              Make XML file for GRC block bindings
add       insert          Add block to the out-of-tree module.
newmod    nm,create       Create a new out-of-tree module
```

We immediately see there are many commands available. In this tutorial we will only cover **newmod** and **add**; however, the thorough explanation should enable smooth usage of the other gr_modtool commands without guidance.

First, we notice that in addition to getting help seeing the commands we can use, we can also request more information on individual commands. Let us start with newmod as that is the command to create a new out-of-tree module.

```
$ gr_modtool help newmod
Usage: gr_modtool nm [options].
 Call gr_modtool without any options to run it interactively.

Options:
  General options:
    -h, --help          Displays this help message.
    -d DIRECTORY, --directory=DIRECTORY
                        Base directory of the module. Defaults to the cwd.
    -n MODULE_NAME, --module-name=MODULE_NAME
                        Use this to override the current module's name (is
                        normally autodetected).
    -N BLOCK_NAME, --block-name=BLOCK_NAME
                        Name of the block, where applicable.
    --skip-lib          Don't do anything in the lib/ subdirectory.
    --skip-swig         Don't do anything in the swig/ subdirectory.
    --skip-Python       Don't do anything in the Python/ subdirectory.
    --skip-grc          Don't do anything in the grc/ subdirectory.
    --scm-mode=SCM_MODE
                        Use source control management (yes, no or auto).
    -y, --yes           Answer all questions with 'yes'. This can overwrite
                        and delete your files, so be careful.

  New out-of-tree module options:
    --srcdir=SRCDIR     Source directory for the module template
```

Now that we have read over the list of commands for newmod, we can deduce that the one we want to pick is -n which is the default so we can simply type the MODULE_NAME after newmod. It is actually advised to avoid using "-n" as for other commands it can override the auto-detected name. For now, let's ignore the other options.

Setting up a new block

```
$ gr_modtool newmod tutorial
Creating out-of-tree module in ./gr-tutorial... Done.
```

```
Use 'gr_modtool add' to add a new block to this currently empty module.
```

We should now see a new folder, gr-tutorial, in our current directory. Let's examine the folder to figure out what gr_modtool has done for us.

```
gr-tutorial$ ls
apps  cmake  CMakeLists.txt  docs  examples  grc  include  lib  Python  swig
```

Since we are dealing with Python in this tutorial we only need to concern ourselves with the Python folder and the grc folder. Before we can dive into code, we need to create a block from a template. There are actually four different types of Python blocks, however it's a little too soon to discuss that. We will dive into the synchronous 1:1 input to output block to make explaining things easier (this is a block that outputs as many items as it gets as input, but don't worry about this right now).

Now we know the language we want to write our block in (Python) and the type of block (synchronous block) we can now add a block to our module. Again, we should run the gr_modtool help command until we are familiar with the different commands. We see that the **add** command is what we are looking for. Now we need to run help on the add command in order to see what we need to enter given the information we have so far.

```
gr-tutorial$ gr_modtool help add
... (General Options from Last Help)
Add module options:
  -t BLOCK_TYPE, --block-type=BLOCK_TYPE
                        One of sink, source, sync, decimator, interpolator,
                        general, tagged_stream, hier, noblock.
  --license-file=LICENSE_FILE
                        File containing the license header for every source
                        code file.
  --argument-list=ARGUMENT_LIST
                        The argument list for the constructor and make
                        functions.
  --add-Python-qa     If given, Python QA code is automatically added if
                        possible.
  --add-cpp-qa        If given, C++ QA code is automatically added if
                        possible.
  --skip-cmakefiles   If given, only source files are written, but
                        CMakeLists.txt files are left unchanged.
  -l LANG, --lang=LANG
                        Language (cpp or Python)
```

We can see the **-l LANG** and the **-t BLOCK_TYPE** are relevant for our example. Thus when creating our new block, we know the command. When prompted for a name simply enter

"multiply_py_ff", when prompted for an argument list enter "multiple", and when prompted for Python QA (Quality Assurance) code type "y", or just hit enter (the capital letter is the default value).

```
gr-tutorial$ gr_modtool add -t sync -l python
GNU Radio module name identified: tutorial
Language: Python
Enter name of block/code (without module name prefix): multiply_py_ff
Block/code identifier: multiply_py_ff
Enter valid argument list, including default arguments: multiple
Add Python QA code? [Y/n] y
Adding file 'Python/multiply_py_ff.py'...
Adding file 'Python/qa_multiply_py_ff.py'...
Editing Python/CMakeLists.txt...
Adding file 'grc/tutorial_multiply_py_ff.xml'...
Editing grc/CMakeLists.txt...
```

We notice 5 changes: Two changes to CMakeLists.txt files, one new file `qa_multiply_py_ff.py` which is used to test our code, one new file `multiply_py_ff.py` which is the functional part, and one new file `tutorial_multiply_py_ff.xml`, which is used to link the block to the GRC. Again all this happens in the Python and grc subfolders.

What's with the `_ff`?

For blocks with strict types, we use suffixes to declare the input and output types. This block operates on floats, so we give it the suffix `_ff`: Float in, float out. Other suffixes are `_cc` (complex in, complex out), or simply `_f` (a sink or source with no in- or outputs that uses floats). For a more detailed description, see the FAQ or the BlocksCodingGuide.

Modifying the Python Block File

Let's begin with the multiply_py_ff.py file found in the Python folder. Opening it without any changes gives the following:

```
import numpy
from gnuradio import gr

class multiply_py_ff(gr.sync_block):
    """
    docstring for block multiply_py_ff
    """
    def __init__(self, multiple):
        gr.sync_block.__init__(self,
            name="multiply_py_ff",
```

```
            in_sig=[<+numpy.float+>],
            out_sig=[<+numpy.float+>])
        self.multiple = multiple


    def work(self, input_items, output_items):
        in0 = input_items[0]
        out = output_items[0]
        # <+signal processing here+>
        out[:] = in0
        return len(output_items[0])
```

Let's take this one line by line as our first Python examples. We are already familiar with the imports so we will skip those lines. We are familiar with the constructor (init) of Python so can immediately see that if we want to use our variable "multiple", we need to add another line. Let us not forget to preserve those spaces as some code editors like to add tabs to new lines. How do we use the variable multiple?

How to use variable multiple...

```
def __init__(self, multiple):
        self.multiple = multiple
        gr.sync_block.__init__(self,
```

We notice that there are "<...>" scattered in many places. These placeholders are from gr_modtool and tell us where we need to alter things

```
in_sig=[<+numpy.float+>]
out_sig=[<+numpy.float+>]
```

The **gr.sync_block.*init*** takes in 4 inputs: self, name, and the size/type of the input and output vectors. First, we want to make the item size a single precision float or numpy.float32 by removing the "*<" and the ">*". If we wanted vectors, we could define those as in_sig=[(numpy.float32,4),numpy.float32]. This means there are two input ports, one for vectors of 4 floats and the other for scalars. It is worth noting that if in_sig contains nothing then it becomes a source block, and if out_sig contains nothing it becomes a sink block (provided we change return `len(output_items[0])` to return `len(input_items[0])` since output_items is empty). Our changes to the first placeholders should appear as follows:

```
in_sig=[numpy.float32]
out_sig=[numpy.float32]
```

The other piece of code that has the placeholders is in the work function but let us first get a better understanding of the work function:

```
def work(self, input_items, output_items)
```

The work function is where the actual processing happens, where we want our code to be. Because this is a sync block, the number of input items always equals the number of output items because synchronous block ensures a fixed output to input rate. There are also decimation and interpolation blocks where the number of output items are a user specified multiple of the number of input items. We will further discuss when to use what block type in the third section of this tutorial. For now we look at the placeholder:

```
in0 = input_items[0]
out = output_items[0]
# <+signal processing here+>
out[:] = in0
return len(output_items[0])
```

The "in0" and "out" simply store the input and output in a variable to make the block easier to write. The signal processing can be anything including if statements, loops, function calls but for this example we only need to modify the out[:] = in0 line so that our input signal is multiplied by our variable multiple. What do we need to add to make the in0 multiply by our multiple?

How to Multiple...

```
out[:] = in0*self.multiple
```

That's it! Our block should now be able to multiply but to be sure we have these things called Quality Assurance tests!

QA Tests

Now we need to test it to make sure it will run correctly when we install it unto GNU Radio. This is a very important step and we should never forget to include tests with our code! Let us open up qa_multiply_py_ff.py which is copied below:

```
from gnuradio import gr, gr_unittest
from gnuradio import blocks
from multiply_py_ff import multiply_py_ff

class qa_multiply_py_ff (gr_unittest.TestCase):

    def setUp (self):
        self.tb = gr.top_block ()

    def tearDown (self):
        self.tb = None

    def test_001_t (self):
        # set up fg
        self.tb.run ()
        # check data


if __name__ == '__main__':
    gr_unittest.run(qa_multiply_py_ff, "qa_multiply_py_ff.xml")
```

gr_unittest adds support for checking approximate equality of tuples of float and complex numbers. The only part we need to worry about is the def test_001_t function. We know we need input data so let us create data. We want it to be in the form of a vector so that we can test multiple values at once. Let us create a vector of floats

```
src_data = (0, 1, -2, 5.5, -0.5)
```

We also need output data so we can compare the input of the block to ensure that it is doing what we expect it to do. Let us multiply by 2 for simplicity.

```
expected_result = (0, 2, -4, 11, -1)
```

Now we can create a flowgraph as we have when we first introduced using Python in GNU Radio. We can use the blocks library specifically the vector_source_ffunction and the vector_sink_f function which are linked to the doxygen manual which we should be able to read and understand. Let us assign three variables "src", "mult", and "snk" to the blocks. The first is shown below:

```
src = blocks.vector_source_f(src_data)
```

The rest are hidden below as an exercise:

What do we assign snk and mult?

```
mult = multiply_py_ff(2)
snk = blocks.vector_sink_f()
```

Now we need to connect everything as src ~~> mult~~> snk. Instead of using self.connect as we did in our other blocks we need to use self.tb.connect because of the setUp function. Below is how we would connect the src block to the mult block.

```
self.tb.connect (src, mult)
```

How would we connect the other blocks together?

```
self.tb.connect (mult, snk)
```

Then we can run the graph and store the data from the sink as below:

```
self.tb.run ()
result_data = snk.data ()
```

Lastly we can run our comparison function that will tell us whether the numbers match up to 6 decimal places. We are using the assertFloatTuplesAlmostEqual instead of the "regular assert functions"https://docs.python.org/2/library/unittest.html#assert-methods included in python's unittest because there may be situations where we cannot get a=b due to rounding in floating point numbers.

```
self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
```

All together the new test_001_t function should appear as below:

```
src_data = (0, 1, -2, 5.5, -0.5)
expected_result = (0, 2, -4, 11, -1)
src = blocks.vector_source_f (src_data)
mult = multiply_py_ff (2)
snk = blocks.vector_sink_f ()
self.tb.connect (src, mult)
self.tb.connect (mult, snk)
```

```
self.tb.run ()
result_data = snk.data ()
self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
```

We can then go to the python directory and run:

```
gr-tutorial/python$ python qa_multiply_py_ff.py
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

While we are here, we should take a break to change one of the numbers in the src_data to ensure that the block is actually checking the values and to simply see what an error looks like. Python allows for really quick testing of blocks without having to compile anything; simply change something and re-run the QA test.

XML Files

At this point we should have written a Python block and a QA test for that block. The next thing to do is edit the XML file in the grc folder so that we can get another step closer to using it in GRC. GRC uses the XML files for all the options we see. We actually don't need to write any Python or C++ code to have a block display in GRC but of course if we would connect it, it wouldn't do anything but give errors. We should get out of the python folder and go to the grc folder where all the XML files reside. There is a tool in gr_modtool called makexml but it is only available for C++ blocks. Let us open the tutorial_multiply_py_ff.xml file copied below:

```
1 <?xml version="1.0"?>
2 <block>
3   <name>multiply_py_ff</name>
4   <key>tutorial_multiply_py_ff</key>
5   <category>tutorial</category>
6   <import>import tutorial</import>
7   <make>tutorial.multiply_py_ff($multiple)</make>
8   <!-- Make one 'param' node for every Parameter you want settable from the GUI.
9        Sub-nodes:
10       * name
11       * key (makes the value accessible as $keyname, e.g. in the make node)
12       * type -->
13  <param>
14    <name>...</name>
15    <key>...</key>
```

```
  <type>...</type>
 </param>

 <!-- Make one 'sink' node per input. Sub-nodes:
      * name (an identifier for the GUI)
      * type
      * vlen
      * optional (set to 1 for optional inputs) -->
 <sink>
  <name>in</name>
  <type><!-- e.g. int, float, complex, byte, short, xxx_vector, ...--></type>
 </sink>

 <!-- Make one 'source' node per output. Sub-nodes:
      * name (an identifier for the GUI)
      * type
      * vlen
      * optional (set to 1 for optional inputs) -->
 <source>
  <name>out</name>
  <type><!-- e.g. int, float, complex, byte, short, xxx_vector, ...--></type>
 </source>
</block>
```

We can change the name that appears and the category it will appear in GRC. The category is where the block will be found in GRC. Examples of categories tag are **Audio** and **Waveform Generators** used in previous examples. Examples of names tag are the **QT GUI Time Sink** or the **Audio Sink**. Again, we can go through the file and find the modtool place holders. The first is copied below:

```
<!-- Make one 'param' node for every Parameter you want settable from the GUI.
     Sub-nodes:
     * name
     * key (makes the value accessible as $keyname, e.g. in the make node)
     * type -->
```

This is referring to the parameter that we used in the very beginning when creating our block: the variable called "multiple". We can fill it in as below:

```
1
2  <param>
3    <name>Multiple</name>
4    <key>multiple</key>
5    <type>float</type>
6  </param>
```

The next placeholder can be found in the sink and source tags:

```
1
2  <sink>
3    <name>in</name>
4    <type><!-- e.g. int, float, complex, byte, short, xxx_vector, ...--></type>
5  </sink>
```

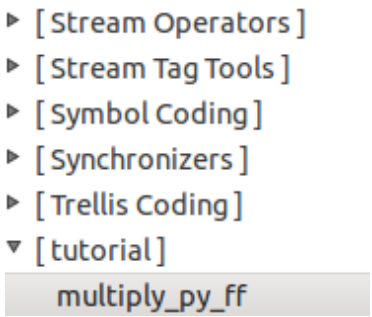
We can see that it is asking for a type so we can simply erase everything in the tag and replace it with "float" for both the source and the sink blocks. That should do it for this block. The best way to get more experience writing xml files is to look at the source code of previously made blocks such as the existing multiple block. Let's go back to this and use the documentation tag!

Installing Python Blocks

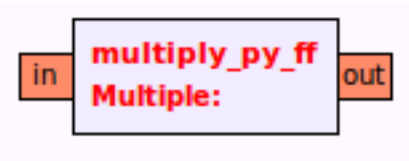
Now that we have edited the XML file; we are ready to install the block into the GRC. First, we need to get out of the /grc directory and create a build directory called "build". Inside the build directory, we can then run a series of commands:

```
cmake ../
make
sudo make install
sudo ldconfig
```

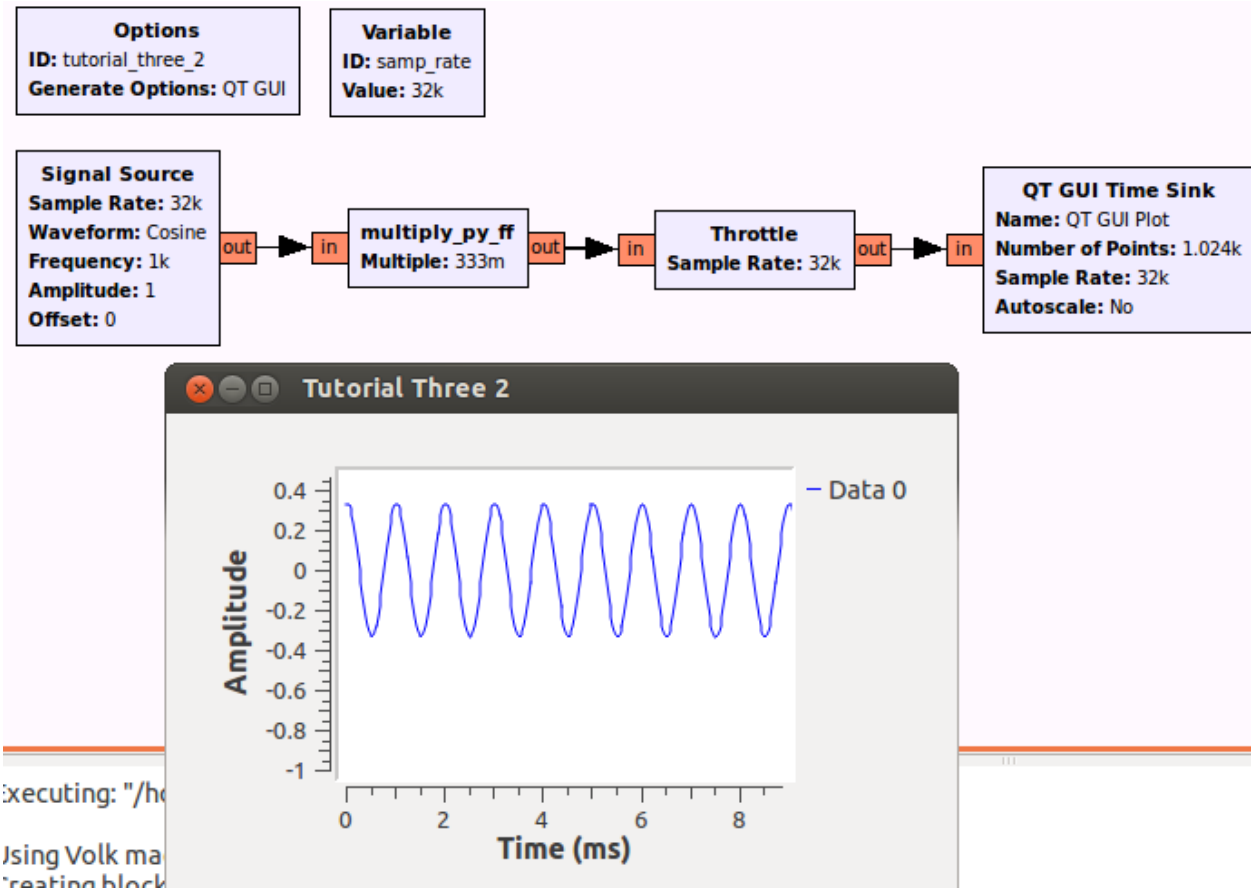
We should then open up the GRC and take a look at the new block.



We can see the category and the block name. When we drag it into the GRC workspace, we can see the multiple variable we set in the param tag of the XML file.



Now that we can see our block, let us test it to make sure it works. Below is an example of one of the many ways to check whether it is actually multiplying.



My QPSK Demodulator for Satellites

Now that we have a better understanding on how to add new blocks to GNU Radio for use in the GRC, we can do another example. This time we will be creating a QPSK demodulator block using the same process as before in the same module. Let's first setup the scenario. There is a "noise source" that outputs data in complex float format but let's pretend it comes from a satellite being aimed at our computer. Our secret agent insider tells us this particular satellite encodes digital data using QPSK modulation. We can decode this using a QPSK demodulator that outputs data into bytes. Our insider tells us the space manual doesn't specify whether it's gray code or not. We want to read the bytes using a time sink. What would our flowgraph look like?

Incomplete
Flowgraph...

Now that we know the input type, output type, and parameters, we can ask the question we skipped with our multiply_py_ff block. What type of block do we want?

Choosing a Block Type

The GNU Radio scheduler optimizes for throughput (as opposed to latency). To do this it needs to know the number of samples each block is inputting and outputting thus came the creation of decimation and interpolation blocks where we specify the multiple factor. This is directly related to the sampling rate discussion we had in tutorial 2 where these special types of blocks are able to change the sampling rate to something else in order to comply with a specific piece of hardware such as our soundcard or a specific modulation/demodulation.

- Synchronous (1:1) - Number of items at input port equals the number of items at output port. An example is the multiply_py_ff block.
- Decimation (N:1) - Number of input items is a fixed multiple of the number of output items. Examples include filters such as decimating high pass and low pass filters (e.g., for sampling rate conversion).
- Interpolation (1:M) - Number of output items is a fixed multiple of the number of input items. An example is the Interpolating FIR Filter.
- General/Basic (N:M) - Provides no relation between the number of input items and the number of output items. An example is the rational resampler block which can be either a sync, decimator, or interpolator.

When we insert data into our block, do we need more samples or less samples? Or put another way, should our sample rate change?

What Type of Block Should we Use?

- Sync Block or Basic Block

Adding Another Block to our OOT Module

Now we know everything we need to know to create the block in gr_modtool. As a refresher, what would our gr_modtool command be?

gr_modtool command...

```
gr-tutorial$ gr_modtool add -t sync -l python
```

name: qpsk_demod_py_cb (c for complex input and b for byte/char/int8 output)
args: gray_code
QA Code: y

Now we have our files setup so we can begin with editing the qpsk_demod_py.py file in the /python folder

The QPSK Demodulator

The code here is a little more than we have done before so let's take it in steps. First what do we expect our block to do? We can start with a constellation plot which shows the four different quadrants and their mappings to bit 0 and bit 1

With Gray coding (adjacent only flips by 1). Note that the mapping we use here is different from the mapping on wikipedia:

```
(-1+1j) 10 | 00 (1+1j)
-----------+-----------
(-1-1j) 11 | 01 (1-1j)
```

Without Gray coding:

```
(-1+1j) 11 | 00 (1+1j)
-----------+-----------
(-1-1j) 10 | 01 (1-1j)
```

We can see that we will need to write an if-else statement to take into account the gray_code variable. We will also need four other if-else statements inside the main if-else so that we can pick the mapping. Our pseudocode will look as follows:

```
if gray_code
    if 1+1j
        binary "00" = 0
    ....
    elif -1-1j
        binary "11" = 3
else
    if 1+1j
        binary "00" = 0
    ....
    elif -1-1j
```

```
binary "10" = 2
```

So we have everything we need to implement. Let's go ahead and fill in our gr_modtoool placeholders. We can begin with def init. There are three changes. How do we use the variable gray_code outside the function (similar to what we did with multiple in the last example)? What are the input and output types in numpy

Changes to def *init*...

```
def __init__(self, gray_code):
    self.gray_code=gray_code
    gr.sync_block.__init__(self,
        name="qpsk_demod_py",
        in_sig=[numpy.complex64],
        out_sig=[numpy.uint8])
```

Once we have our constructor setup, we can go onto the work function. For simplicity and beauty, let us call the pseudocode we made above a function "get_minimum_distance" that takes samples as input arguments. In our multiply_py_ff example, we took all the samples and multiplied them with with out[:] = in0*self.multiple. The in0 is actually a vector so contains many samples within it. The multiply example required the same operation for each sample so it was okay to simply operate on the entire vector but now we need to have different operations per sample so what do we do?

How can we operate on samples in a vector?

- loops!

```
    for i in range(0, len(output_items[0])):
        sample = in0[i]
        out[i] = self.get_minimum_distances(sample)
```

Now we can move onto the get_minimum_distances(self, sample) function. We already have pseudo code so the next step is translating to Python. Below is a snip of what the code can look like. Again there are multiple ways to do this

```
1  def get_minimum_distances(self, sample):
2      if self.gray_code == 1:
3          if (sample.imag >= 0 and sample.real >= 0):
4              return 0 # 1+1j
5          elif (sample.imag >= 0 and sample.real < 0):
6              return 2 # -1+1j
```

Let us try to fill in the other cases for gray code and non-gray code. Below is what the entire file Python file can look like once complete:

qpsk_demod_py_cb.py

```
1 import numpy
2 from gnuradio import gr
3
4 class qpsk_demod_py(gr.sync_block):
5     """
6     docstring for block qpsk_demod_py
7     """
8     def __init__(self, gray_code):
9         self.gray_code=gray_code
10        gr.sync_block.__init__(self,
11            name="qpsk_demod_py",
12            in_sig=[numpy.complex64],
13            out_sig=[numpy.uint8])
14
15    def get_minimum_distances(self, sample):
16        if self.gray_code == 1:
17            if (sample.imag >= 0 and sample.real >= 0):
18                return 0 # 1+1j
19            elif (sample.imag >= 0 and sample.real < 0):
20                return 2 # -1+1j
21            elif (sample.imag < 0 and sample.real < 0):
22                return 3 # -1-1j
23            elif (sample.imag < 0 and sample.real >= 0):
24                return 1 # 1-1j
25        else:
26            if (sample.imag >= 0 and sample.real >= 0):
27                return 0 # 1+1j
28            elif (sample.imag >= 0 and sample.real < 0):
29                return 3 # -1+1j
30            elif (sample.imag < 0 and sample.real < 0):
31                return 2 # -1-1j
32            elif (sample.imag < 0 and sample.real >= 0):
33                return 1 # 1-1j
34
35    def work(self, input_items, output_items):
36        in0 = input_items[0]
37        out = output_items[0]
```

```
38
39      for i in range(0, len(in0)):
40          sample = in0[i]
41          out[i] = self.get_minimum_distances(sample)
42
43      return len(output_items[0])
```

Now that we have code, we know what's next!

Multiple QA Tests

We can test our qpsk_demod_py for when it is in gray_code mode and when it's not in gray_code mode. To do that we need to setup multiple tests in our single QA file. QA tests generally follow the same setup from before. We select some inputs as tests and we check them against what we expect the outputs to be. The only difference from the multiply qa test is that this qa test requires more cases. There are four quadrants we need to map and two modes so in total there are eight test cases. We can open up our qa_qpsk_demod_py_ff.py file to change things.

We can copy the def test_001_t from the qa_multiply_py_ff code which is copied below:

```
1   def test_001_t (self):
2       src_data = (-3, 4, -5.5, 2, 3)
3       expected_result = (-6, 8, -11, 4, 6)
4       src = blocks.vector_source_f (src_data)
5       mult = multiply_py_ff (2)
6       dst = blocks.vector_sink_f ()
7       self.tb.connect (src, mult)
8       self.tb.connect (mult, dst)
9       self.tb.run ()
10       result_data = dst.data ()
11       self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
```

This time we are working with a complex input so our src = blocks.vector_source_f must change. If we use the search bar in the manual we can find the other options:

PIC of SEARCH

b - bytes/unsigned char/int8
c - complex
f - float
i - int
s - short

What do we change our source and sink vectors to?
src = blocks.vector_source_c (src_data)
dst = blocks.vector_sink_b ()

Before we move onto actual test cases, let us decide which mode we are testing for the test_001_t. We can create a new variable and assign it False (translates to 0) to test non-Gray code

```
gray_code = False
```

Once we know we want to test non gray_code mappings, we can refer to our chart above and start placing in the proper inputs and outputs into the src_data and the expected_results. For instance if we were testing only two cases for non gray_code, we would do:

```
1 src_data = ((-1-1j), (-1+1j))
2 expected_result = (2, 3)
```

Last thing to do is call upon our new block in the "qpsk =" line and pass it the gray_code parameter

qpsk = ?

- qpsk = qpsk_demod_py_cb (gray_code)

Now that we are done with the non gray_code test, we can simply create another test "def test_002_t (self):" and copy the contents underneath making sure that for this test we set gray_code = True and change the expected_result so it matches gray_code mapping. The full test is copied below:

Full QA QPSK Demod Code...

```
 1 from gnuradio import gr, gr_unittest
 2 from gnuradio import blocks
 3 import numpy
 4 from qpsk_demod_py_cb import qpsk_demod_py
 5
 6 class qa_qpsk_demod (gr_unittest.TestCase):
 7
 8    def setUp (self):
 9        self.tb = gr.top_block ()
10
11    def tearDown (self):
12        self.tb = None
13
14    def test_001_t (self):
15        gray_code = False
16        src_data = ((-1-1j), (-1+1j), (1+1j), (1-1j))
17        expected_result = (2, 3, 0, 1)
18        src = blocks.vector_source_c (src_data)
```

```
19        qpsk = qpsk_demod_py (gray_code)
20        dst = blocks.vector_sink_b ()
21        self.tb.connect (src, qpsk)
22        self.tb.connect (qpsk, dst)
23        self.tb.run ()
24        result_data = dst.data ()
25        self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
26
27    def test_002_t (self):
28        gray_code = True
29        src_data = ((-1-1j), (-1+1j), (1+1j), (1-1j))
30        expected_result = (3, 2, 0, 1)
31        src = blocks.vector_source_c (src_data)
32        qpsk = qpsk_demod_py (gray_code)
33        dst = blocks.vector_sink_b ()
34        self.tb.connect (src, qpsk)
35        self.tb.connect (qpsk, dst)
36        self.tb.run ()
37        result_data = dst.data ()
38        self.assertFloatTuplesAlmostEqual (expected_result, result_data, 6)
39
40 if __name__ == '__main__':
41    gr_unittest.run(qa_qpsk_demod, "qpsk_demod_py_cb.xml")
```

We can then run the test in Python and all should say something similar to:

```
Ran 2 tests in 0.005s

OK
```

So once we verify it works as expected, we can then edit our XML file so that it is usable inside GRC.

XML Mods, Installation, and Running

This XML is very similar to the XML file for the multiply_py_ff block so all we need to do is set the gray_code parameter and pick the correct input (complex) and output (byte) types. A copy of the full XML file is below:

XML File for QPSK Demod

```
1 <?xml version="1.0"?>
```

```
<block>
  <name>qpsk_demod_py</name>
  <key>tutorial_qpsk_demod_py</key>
  <category>tutorial</category>
  <import>import tutorial</import>
  <make>tutorial.qpsk_demod_py($gray_code)</make>
  <!-- Make one 'param' node for every Parameter you want settable from the GUI.
       Sub-nodes:
       * name
       * key (makes the value accessible as $keyname, e.g. in the make node)
       * type -->
  <param>
    <name>Gray Code</name>
    <key>gray_code</key>
    <type>int</type>
  </param>

  <!-- Make one 'sink' node per input. Sub-nodes:
       * name (an identifier for the GUI)
       * type
       * vlen
       * optional (set to 1 for optional inputs) -->
  <sink>
    <name>in</name>
    <type>complex</type>
  </sink>

  <!-- Make one 'source' node per output. Sub-nodes:
       * name (an identifier for the GUI)
       * type
       * vlen
       * optional (set to 1 for optional inputs) -->
       <!-- e.g. int, float, complex, byte, short, xxx_vector, ...-->
  <source>
    <name>out</name>
    <type>byte</type>
  </source>
</block>
```

We can then install as we did for the multiply block however we need to rerun cmake in order to take into account the new block:

```
cd build
cmake ../
make
sudo make install
sudo ldconfig
```

Then we can open up our GRC file from the beginning and place our missing block we just made.

What is the Expected Output?