

# Tutorial 5: Getting Prepared for Python in GNU Radio by Reading the FM Receiver Code Line by Line – Part I

Dawei Shen\*

*May 27, 2005*

## Abstract

Python plays a key role in GNU Radio programming. GNU Radio provides a framework for building software radios. The signal processing applications – are built using a combination of Python code for high level organization, policy, GUI and other non performance-critical functions, while performance critical signal processing blocks are written in C++. From the Python point of view, GNU Radio provides a data flow abstraction. This article is focused on the Python level, introducing the basic usage of Python and how Python is used in GNU Radio to glue all the signal processing blocks and control the flow of the digital data. A popular and classical example: the implementation of an FM receiver with GUI, is used here. The code is analyzed line by line. The basic grammar of Python and the concept of software radio are introduced in parallel. So this article can be used as both a short Python tutorial and a software radio guidance. How to write the blocks using C++ and other advanced topics will be covered in subsequent chapters.

---

## 1 Overview

GNU Radio's software is organized using a two-tier structure. All the performance-critical signal processing blocks are implemented in C++, while the higher-level organizing, connecting and gluing are done using Python. Many frequently used signal processing blocks have been implemented well and provided to us as parts of the GNU Radio software.

This structure has some similarity with the OSI 7-layer data network model. Lower layer provides service to the higher layer, while the higher layer doesn't care about the implementation details carried on in lower layers, but necessary interfaces and function calls. In GNU Radio, this layer transparency exists in a similar way. From the Python's point of view, what it does is just to select necessary signal sources, sinks and processing blocks, set correct parameters, then connect them together to form a complete application. In fact, all these sources, sinks and blocks are implemented as classes in C++. The parameter setting, connecting operations correspond to some sophisticated functions or class methods in C++. However, Python can't see how sedulously C++ has been working. A piece of lengthy, complicated and powerful C++ code is nothing but an interface to Python.

As a result, no matter how complicated the application is, the Python code is almost always short and neat. The real heavy load is thrown to C++. A thumb of rule should be kept in mind: for any application, what we need to do at the Python level, is always just to draw a diagram showing the signal flow from the source to the sink in your mind, then use the 'nice pen' – Python, to find them and connect them all together, sometimes with the graphical user interfaces (GUI) support.

Obviously Python is crucial in learning GNU Radio. Python is a powerful and flexible programming language, which itself is a long story. But if you have enough C/C++ background, it's just a piece

---

\*The author is affiliated with Networking Communications and Information Processing (NCIP) Laboratory, Department of Electrical Engineering, University of Notre Dame. Welcome to send me your comments or inquiries. Email: dshen@nd.edu

of cake. Considering the fact that Python has some special characteristics when applied to GNU Radio and that some of its cool fancy features may not be necessary in GNU Radio, the article aims at combining the Python programming techniques and software radio concepts together. I am a believer that the best way to grasp the essence of some new knowledge is to go through an example rather than learning the syntax or semantics dryly. So our discussion will surround a popular and classical example: the implementation of an FM receiver with graphical user interfaces. The code will be analyzed line by line and the Python programming, signal processing techniques, software radio concepts and some hardware configuration will be talked about along the way.

This example implements an FM receiver with graphical user interface. The FM signal from the air is received by the USRP board then gets processed in the USRP board and in the computer. Finally the demodulated signal is played using the sound card. No fancy antenna is required. You can hear very high quality FM signal by just inserting a copper wire to the basic RX daughter board. The code can be found at '[gnuradio-examples/python/usrp/wfm\\_rcv\\_gui.py](#)'. Please refer to appendix A to make sure we have the same version of the code. We plan to use two articles to cover the whole materials. This article is the first half, focusing on the basics of Python and GNU Radio. Some advanced topics, such as the GUI tools (wxPython) and the usage of some Python built-in packages, are left to Tutorial 8.

## 2 The first line

If you have read the code of other examples, you can find the first line of these programs is almost always

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

The Python scripts can be made directly executable if we put this line at the beginning of the script and giving the file an executable mode. The '#!' must be the first two characters of the file. The script is given the executable mode by using the 'chmod' command:

```
$ chmod +x wfm_rcv_gui.py
```

Now the script `wfm_rcv_gui.py` becomes executable. You can run this program in the shell using

```
$ ./wfm_rcv_gui.py arguments
```

the Python interpreter will be invoked and the code in this script will be executed line by line orderly. Python is an interpreted language, like Matlab script. No compilation and linking is necessary.

There are several ways to invoke the Python interpreter: you can use

```
$ python ./wfm_rcv_gui.py arguments
```

without the need to give the script the executable mode.

You can also use the interactive mode, by just typing the command in the shell:

```
$ python
```

then the Python interpreter environment will be invoked and you could input your code line by line. However, this is obviously inconvenient. We seldom use the interpreter interactively, unless we write some throw-away programs, test functions or use it as a desk calculator. Most of the time, packing codes in a `.py` file and make the script self-executable is more convenient to us.

## 3 Importing necessary modules

Next, we see a lot of importing stuffs:

```

from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
from gnuradio.wxgui import stdgui, fftsink
import wx

```

Understanding these statements requires the knowledge of '**module**' and '**package**' concepts in Python. The best way to learn them is to go through the Chapter 6 of the Python tutorials.

Here is a brief introduction. If we quit the Python interpreter and enter it again, all the functions and variables we have defined are lost. Therefore, we wish to write a somewhat longer program and save it as a **script**, containing function and variable definitions, and maybe some executive statements. This script can be used as the input of the Python interpreter. We may also want to use a fancy function we've written in several programs without copying its definitions to each program.

To support this, Python provides a **module/package** organization system. A **module** is a file containing Python definitions and statements, with the suffix '**.py**'. Within a module, the module's name (as a string) is available as the value of the global variable '**\_\_name\_\_**'. Definitions in a module can be imported into other modules or into the top-level module. A **package** is a collection of modules that have similar functions, which are often put in the same directory. The '**\_\_init\_\_.py**' files are required to make Python treat the directories as packages. A package could contain both modules and sub-packages (can contain sub-sub-packages). We use 'dotted module names' to structure Python's module namespace. For example, the module name A.B designates a submodule named 'B' in a package named 'A'.

A module can contain executable statements as well as function definitions. The statements are executed only the first time the module is imported somewhere and so that the module is initialized. Each module has its own private symbol table, which is used as the global symbol table by all functions defined within that module. The author of a module can use the global variables in the module without worrying about accidental clashes with the module user's global variables. As a module user, we can access a module's function and global variables using '**modname.itemname**'. Here the '**itemname**' can either be a function or a variable.

Modules can import other modules using '**import**' command. It is customary to place all '**import**' statements at the beginning of a module. Note that the **import** operation is quite flexible. We can import a package, a module, or just a definition within a module. When we try to import a module from a package, we can either use '**import packageA.moduleB**', or '**from package A import module B**'. When using '**from package import item**', the '**item**' can be either a module/sub-package of the package, or some other names defined in the package, like functions, classes or variables.

It's worth taking a while to introduce the modules used in this example amply, because these modules or packages will be frequently encountered in GNU Radio. The top-level package of GNU Radio is '*gnuradio*', which includes all GNU Radio related modules. It is located at

```
/usr/local/lib/python2.4/site-packages
```

By default, this directory is not included in the Python's search path, we need to export the path to the environment variable '**PATHONPATH**'. So we usually add the following line to the users' **.bash\_profile** file:

```
export PATHONPATH=/usr/local/lib/python2.4/site-packages
```

to make sure the Python interpreter could find the *gnuradio* package.

*gr* is an important sub-package of *gnuradio*, which is the core of the GNU Radio software. The type of '**flow graph**' classes is defined in *gr* and it plays a key role in scheduling the signal flow.

*eng\_notation* is a module designed for engineers' notation convenience, in which many words and characters are endowed with new constant values according to the engineering convention. The module *audio* provides the interfaces to access the sound card, while *usrp* provides the interfaces to control the USRP board. *audio* and *usrp* are often used as the signal source and sink. We will see the details about them later in this article. *blks* is a sub-package, which is almost an empty folder if you check its directory. It actually transfers all its tasks to another sub-package *blksimpl* in *gnuradio*, as described in the `__init__.py` file. *blksimpl* provides the implementation of several useful applications, such as FM receiver, GMSK, etc. For this example, the real signal processing part of the FM receiver is performed in this package.

Look at the next line, which is more interesting:

```
from gnuradio.eng_option import eng_option
```

This is exactly what we mentioned just now, we can either import a complete module/sub-package, or, just a function, class or variable definition from this module. In this case, *eng\_option* is a class defined in the module *gnuradio.eng\_option*. We don't need the whole module to be imported, but just a single class definition. *gnuradio.eng\_option* module does nothing but adding support for engineering notation to *optparse.OptionParser*.

This line appears to have a similar format:

```
from gnuradio.wxgui import stdgui, fftsink
```

But the meaning is a little bit different, *gnuradio.wxgui* is a sub-package, not a module, while *stdgui* and *fftsink* are two modules in this sub-package. It's not necessary to import the whole sub-package, so we just import what we want explicitly. *gnuradio.wxgui* provides visualization tools for GNU Radio, which is constructed based on wxPython. The importing operations in Python provide us great flexibility and convenience.

Finally, *optparse*, *math*, *sys*, *wx* are all Python or wxPython's built-in modules or sub-packages, which are not part of the GNU Radio.

At this point, let me emphasize again, these modules imported above may contain executable statements as well as the function or class definitions. The statements will be executed immediately after the modules are imported. After importing the modules and packages, a lot of variables, classes and modules defined in them have been initialized. So don't think nothing has been done. Actually a lot of work behind the stage has been carried on. Many guys are waiting for your order in the workspace.

OK! So far we have taken a quick look at the most frequently used modules in GNU Radio and seen how they are organized together. Maybe it sounds too abstract and you are still confused about their usage. Never mind, we will see them soon.

## 4 The story in the class `wfm_rx_graph`

Familiarizing with object oriented programming (OOP) is important for understanding this section. OOP itself is a long story, which is obviously not what we are focusing on. But we will talk about some OOP concepts along the way. Some digital signal processing techniques will also be discussed as a reminder if we meet the corresponding codes.

### 4.1 Class definition

In this example, a large part of the code is the definition of a class '`wfm_rx_graph`'. The statement

```
class wfm_rx_graph (stdgui.gui_flow_graph):
```

defines a new class '`wfm_rx_graph`', which is inherited (derived) from the base class, or the so called 'father class' -- '`gui_flow_graph`'. The father class `gui_flow_graph` is defined in the module *stdgui* we have just imported from *gnuradio*. By the rules of the namespace, we use `stdgui.gui_flow_graph` to refer to it.

## 4.2 The family of 'FLOW GRAPH' classes

Here an important category of classes, which play a key role in GNU Radio should receive particular attention: the 'flow graph' classes. There are a series of 'GRAPH' related classes defined in GNU Radio. We can find that `stdgui.gui_flow_graph` is derived from `gr.flow_graph`, which is defined in the sub-package `gr`. Further, `gr.flow_graph` is derived from the 'root' class `gr.basic_flow_class`. In GNU Radio, there are also many other classes derived from `gr.basic_flow_graph`. This big 'GRAPH family' makes GNU Radio programming neat and simple, also makes the scheduling of the signal processing clear and straightforward.

What do these 'graphs' do? Suppose you are trying to design a circuit using some commercial software such as Pspice. Probably you will first open a schematic, or a 'canvas', then you put all necessary circuit parts, such as a resistor, an amplifier or some DC powers on this canvas. Finally, you draw lines among these parts to connect them together and complete the circuit design. This scenario applies perfectly into GNU Radio. A **graph** is just like the schematic or the canvas. The circuit parts are replaced by signal sources, sinks and the signal processing blocks in GNU Radio. Finally, those 'wires' correspond to the '**connect**' method of the graph class, which is in charge of gluing these blocks together. Definitely, sometimes, an integrated circuit can serve as a part, the so called sub-circuit, in another schematic. That's also true in GNU Radio, a sub-graph can be used as a whole block in another graph.

In our example, `wfm_rx_graph` is such a graph class belonging to this family, with GUI support. Later we will see it glues the necessary blocks in FM receiver together using the method '**connect**'.

## 4.3 The initialization function: `__init__`

Then we implement the method (or function) '`__init__`' of the class `wfm_rx_graph`. The syntax for defining a new method is

```
def funcname(arg1 arg2 ...)
```

`__init__` is an important method for any class. After defining the class, we may use the class to instantiate an instance. This special method `__init__` is used to create an object in a known initial state. Class instantiation automatically invokes `__init__` for the newly created class instance. Actually in this example, `__init__` is the only method defined in the class `wfm_rx_graph`.

One important feature of Python is worth mentioning before we talk about the details in the function `__init__`. We notice that in this piece of code, there is no explicit signs for where a definition of a class or a function starts and ends. Usually in other programming languages such as C++ or Pascal, we use '**begin**' and '**end**' pair, or a pair of '{' and '}' explicitly to denote the two ends of a group of statements. However, in Python, this is no longer the case. There is **NO** such signs. In Python, statement grouping is done by **indentation** instead of beginning and ending brackets. So be careful about your editing and layout of the code when you write programs using Python.

Now let's see what's going on in the function `__init__`.

```
def __init__(self,frame,panel,vbox,argv):
```

declares the initialization method `__init__` with four arguments. Conventionally, the first argument of all methods are often called **self**. This is nothing more than a convention: the name **self** has absolutely no special meaning to Python. However, methods may call other methods by using method attributes of the **self** argument, such as '`self.connect()`' we will meet later.

The first thing `__init__` does, is to call the initialization method of `stdgui.gui_flow_graph`, its 'father class', with exactly the same four arguments.

```
stdgui.gui_flow_graph.__init__(self,frame,panel,vbox,argv)
```

You may like to take a look at `stdgui.gui_flow_graph`'s initialization method. Since `wfm_rx_graph` is derived from it, we can safely think of `wfm_rx_graph` as a 'special' `gui_flow_graph`. So it's natural that this 'son class' should do something to make himself look like his father first, then do something fancy to make himself a different guy.

## 4.4 Constructing the graph with source, sink and signal processing blocks

### 4.4.1 Defining constants

From the next line, we kind of start to see the real signals coming in, which is less boring

```
IF_freq = parseargs(argv[1:])
```

sets the IF frequency from the return value of the function `parseargs`.

What does IF frequency stand for? IF is short for 'intermediate frequency'. Roughly speaking, it's the center frequency of the frequency band we are interested in. We move the 'real' frequency band, the so called RF band, which is usually very high, to some intermediate frequency band (IF) where the ADC can work obeying the Nyquist theorem. This isn't the key point in this article. I hope you have already got enough background in communications and DSP at this point.

The function `parseargs` is defined in this example later, right after the definition of the class `wfm_rx_graph`. It accepts the user's input arguments when the program is executed in shell. Let's talk about it later.

Besides, you may have noticed that Python doesn't require variable or argument declarations. This is totally different with the 'declare before use' concept in C.

```
adc_rate = 64e6
```

This line defines the sampling frequency of the AD converter, which should be set to 64MHz for the USRP users. According to Nyquist theorem, the maximum frequency component of the interested signal should be less than 32MHz in order not to lose spectrum information after sampling.

```
decim = 250
quad_rate = adc_rate / decim           # 256 kHz
```

Decimation is a concept in DSP world. After sampling the analog signal, we get a digital signal with very high data rate, which is a heavy burden for the CPU and storage. Usually, we can down-sample the digital sequence (decimation) without losing the spectrum information. In this example, the decimation rate is chosen to be 250 so that the resulting data rate is 256K samples per second, which is quite reasonable and acceptable for our CPU speed. `quad_rate` represents for **quadrature data rate**. The reason why it is called quadrature rate will be explained later.

'# 256 kHz' is nothing but a piece of comment. In Python, a comment starts after the symbol '#'. All statements after '#' in each line will be ignored by Python interpreter.

```
audio_decimation = 8
audio_rate = quad_rate / audio_decimation   # 32 kHz
```

After processing the digital FM signal, we wish to play the signal using the sound card of the computer. However, the data rate that a sound card can adopt is rather limited. 256kHz is usually too high and more than necessary. So we need to further decimate the data rate. 32kHz is a common choice for most sound cards.

### 4.4.2 The signal source

The following several lines provide a very high level abstraction for the signal processing procedure, which basically includes three components: the signal source, the signal sink, and a series of signal processing blocks. This example gives those signal processing blocks a very nice name: **guts**.

The signal source for the FM receiver is the USRP board in this example

```
# usrp is data source
src = usrp.source_c (0, decim)
src.set_rx_freq (0, IF_freq)
src.set_pga(0,20)
```

The USRP board receives the analog FM signal from the air via the RX daughter board and samples the signal using the AD converter with a sampling rate 64MHz. The resulting digital sequence then goes into the FPGA chip equipped on the USRP board. It is down-sampled there according to the decimation rate that the user has set (250 in our case). Another important digital signal processing is also done within the FPGA: the real IF-band signal becomes complex base-band signal two I/Q quadrature components. This also explains why the data rate after decimation is called ‘quadrature rate’. Of course, the real story behind is much more complicated, we will leave the details to subsequent chapters. Finally, the complex base-band signal is sent to the software module in the computer via the USB 2.0 cable. A complex number is represented using a **real/imag** pair, which actually requires two real values.

OK, let’s look at the code. *usrp* is the module we’ve imported at the beginning. The *usrp* module is located at:

```
/usr/local/lib/python2.4/site-packages/gnuradio/usrp.py
```

It tells us the module *usrp* is a wrapper for the USRP sinks (transmitting) and sources (receiving). When a sink or a source is instantiated, the *usrp* module first probes the USB port to locate the requested board number, then use the appropriate version specific sink or source. `source_c` is a function defined in *usrp* module. It returns a class object that represents the data source. The suffix `_c` means the data type of the signal is ‘**complex**’, because the signal coming into the computer is complex (actually in real/imag pair). In contrast, we also have `source_s` method in the *usrp* module, which is designed for 16-bit short integer data type.

In this example, `source_c` takes two arguments. ‘0’, specifies which USRP board is going to be opened. Just set it to zero if we work with only one USRP board. The second parameter tells the decimation rate to the USRP board. `set_rx_freq` and `set_pga` are two methods of the source ‘**src**’. `set_rx_freq` tells the USRP board the IF frequency. Just now we have mentioned the USRP board processes the real IF-band signal, into complex base-band signal with two I/Q quadrature components. To do this, the USRP board requires the knowledge of the IF frequency. **pga** is short for ‘Programmable Gain Amplifier’. We can set its value (in db) using the `set_pga` method, which is 20db in our case.

Where are these methods defined? At this level, Python is insufficient to explain all these stuffs. Actually all these methods are implemented using C++. The **SWIG** provides the interfaces between C++ and Python, so that we can call these functions directly in Python, without worrying about the implementation details in C++. **Boost**, a smart pointer system, is also used here to facilitate the interaction between C++ and Python. So the story seems to be rather complicated. We skip the implementation details at this point. Let’s just use these methods happily in Python!

An important document about USRP generated using Doxygen is located at:

```
/usr/local/share/doc/usrp-x.xcvs/html
```

We can look up all the methods provided by USRP in this document. The top level interfaces to the USRP are `usrp_standard_rx` and `usrp_standard_tx`. Also we should take a look at their base classes, `usrp_basic_rx`, `usrp_basic_tx` and `usrp_basic`. There are many other methods, to control and interact with the USRP board. Feel relaxed if you are still worried about the interfaces between Python and C++, we will get back to them when we talk about how to use C++ to write blocks later.

#### 4.4.3 The big signal processing ‘gut’

There is a long story behind

```
guts = blks.wfm_rcv (self, quad_rate, audio_decimation)
```

‘guts’ is the central processing block of this FM receiver. All signal processing blocks, such as deemphasizing, noncoherent demodulation are glued together in this ‘gut’. This more interesting story can be found at

[/usr/local/lib/python2.4/site-packages/gnuradio/blksimpl/wfm\\_rcv.py](/usr/local/lib/python2.4/site-packages/gnuradio/blksimpl/wfm_rcv.py)

The detailed FM receiving techniques will be discussed in Tutorial 7. Now let’s just give it a shot from a high level point of view. `wfm_rcv` is a class defined in the module `blksimpl`. Its base class is `hier_block`, defined in the module `gr.hier_block`. `gr.hier_block` can be thought as a sub-graph, containing several signal processing blocks, which is used as a single sophisticated block in another bigger graph. In this statement, we create an object ‘guts’ as the instantiation of the class `wfm_rcv`. All real signal processing is done within this big block.

#### 4.4.4 The signal sink

Finally, we will play the demodulated FM signal using the sound card. So the audio device is the signal sink in this example:

```
# sound card as final sink
audio_sink = audio.sink (int (audio_rate))
```

`sink` is a global function defined in the module `audio`. It returns an object as the signal sink block. `audio_rate` is a parameter describing the the data rate of the signal entering the sound card, which is 32kHz in our example.

#### 4.4.5 Gluing them together

The next two lines finally complete our signal flow graph

```
# now wire it all together
self.connect (src, guts)
self.connect (guts, (audio_sink, 0))
```

Just now we have talked about the family of the flow graph classes, to which the new class `wfm_rx_graph` belong. All those flow graph classes are derived from the ‘root’ class `gr.basic_flow_graph`. The `connect` method is defined in `gr.basic_flow_graph`. This method is designed for the flow graph to bind all the blocks together. We will investigate more on flow graph classes and their methods in Tutorial 6.

The signal flow graph is done at this point!

## 5 Conclusion

OK! Let’s stop here and have a rest for a while. The rest of the code requires some advanced knowledge. We will add GUI support to the FM receiver, which is cool and attractive. It is built upon `gr-wxgui`, which is based on wxPython and FFTW. We haven’t talked about how to receive arguments from the user screen and how to start the flow graph. Some arguments passing among classes and functions may also seem to be confusing at the point. All these more advanced materials will be covered again in Tutorial 8.

This article is focusing on the basic syntax of Python and how Python plays its role in GNU Radio. Some software radio concepts and signal processing techniques are also mentioned along the way. I hope after reading this article, you can have a rough sense about how to program in GNU Radio.

## APPENDIX A: The source code

```

#!/usr/bin/env python

from gnuradio import gr, eng_notation
from gnuradio import audio
from gnuradio import usrp
from gnuradio import blks
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math

from gnuradio.wxgui import stdgui, fftsink
import wx

class wfm_rx_graph (stdgui.gui_flow_graph):
    def __init__(self, frame, panel, vbox, argv):
        stdgui.gui_flow_graph.__init__(self, frame, panel, vbox, argv)

        IF_freq = parseargs(argv[1:])
        adc_rate = 64e6

        decim = 250
        quad_rate = adc_rate / decim          # 256 kHz
        audio_decimation = 8
        audio_rate = quad_rate / audio_decimation # 32 kHz

        # usrp is data source
        src = usrp.source_c (0, decim)
        src.set_rx_freq (0, IF_freq)
        src.set_pga(0,20)

        guts = blks.wfm_rcv (self, quad_rate, audio_decimation)

        # sound card as final sink
        audio_sink = audio.sink (int (audio_rate))

        # now wire it all together
        self.connect (src, guts)
        self.connect (guts, (audio_sink, 0))

        if 1:
            pre_demod, fft_win1 = \
                fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
                    512, quad_rate)
            self.connect (src, pre_demod)
            vbox.Add (fft_win1, 1, wx.EXPAND)

        if 1:
            post_deemph, fft_win3 = \
                fftsink.make_fft_sink_f (self, panel, "With Deemph",
                    512, quad_rate, -60, 20)
            self.connect (guts.deemph, post_deemph)
            vbox.Add (fft_win3, 1, wx.EXPAND)

        if 1:
            post_filt, fft_win4 = \

```

```
        fftsink.make_fft_sink_f (self, panel, "Post Filter",
                                512, audio_rate, -60, 20)
self.connect (guts.audio_filter, post_filt)
vbox.Add (fft_win4, 1, wx.EXPAND)

def parseargs (args):
    nargs = len (args)
    if nargs == 1:
        freq1 = float (args[0]) * 1e6
    else:
        sys.stderr.write ('usage: wfm_rcv freq1\n')
        sys.exit (1)

    return freq1 - 128e6

if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()
```

## References

- [1] **Python on-line tutorials**, <http://www.python.org/doc/current/tut/>
- [2] Eric Blossom, **Exploring GNU Radio**,  
<http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html>