Tutorial 9: A Dictionary of the GNU Radio blocks

Dawei Shen*

August 21, 2005

Abstract

A dictionary of the GNU Radio blocks. This article takes a tour around the most frequently used blocks, explaining the syntax and how to use them.

1 Introduction

When we use Matlab to do simulation, it is believed that in order to write the code cleanly and efficiently, we need to memorize a number of Matlab built-in functions and tool boxes well and use them skillfully. The same applies to GNU Radio. About 100 frequently used blocks come with GNU Radio. For a certain number of applications, we can complete the designing using these existing blocks, programming only on the Python level, without the need to write our own blocks. So in this article, we will take a tour around the GNU Radio blocks.

A good way to go through the blocks is to study the two GNU Radio documentations generated by Doxygen. They will be generated when you install `gnuradio-core` and `usrp` packages, assuming you have Doxygen installed. They are located at:

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

You may like to bookmark them in your web browser. The first one is also available on-line.

A block is actually a class implemented in C++. For example, in the FM receiver example, we use the block `gr.quadrature_demod_cf`. This block corresponds to the class `gr_quadrature_demod_cf` implemented in C++. The SWIG creates its interface to Python. The SWIG does some magic work so that from Python's point of view, the block becomes a class called `quadrature_demod_cf` defined in the module `gr`, so that we can access to the block using `gr.quadrature_demod_cf` in Python. When you look at the Doxygen documentations, the prefix such as `gr`, `qa`, `usrp` corresponds to the module name in Python and the part after the prefix is the real name of the block in that module, such as `quadrature_demod_cf`, `fir_filter_fff`. So if we talk about a block named '`A_B_C_...`', we will use it as '`A.B_C_...`' in Python.

*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

2 Signal Sources

2.1 Sinusoidal and constant sources

Block: gr.sig_source_X.

Usage:

```
gr.sig_source_c [f, i, s]   ( double sampling_freq,
                              gr_waveform_t waveform,
                              double frequency,
                              double amplitude,
                              gr_complex [float, integer, short] offset )
```

Notes: The suffix `X` indicates the data type of the signal source. It can be `c` (complex), `f` (float), `i` (4 byte integer) or `s` (2 byte short integer). The `offset` argument is the same type as the signal. The `waveform` argument indicates the type of the wave form. `gr_waveform_t` is an enumeration type defined in `gr_sig_source_waveform.h`. Its value can be:

```
gr.GR_CONST_WAVE
gr.GR_COS_WAVE
gr.GR_SIN_WAVE
```

When you use `gr.GR_CONST_WAVE`, the output will be a constant voltage, which is the amplitude plus the offset.

2.2 Noise sources

Block: gr.noise_source_X.

Usage:

```
gr.noise_source_c [f, i, s] ( gr_noise_type_t type,
                              float amplitude,
                              long seed )
```

Notes: The suffix `X` indicates the data type of the signal source. It can be `c` (complex), `f` (float), `i` (4 byte integer) or `s` (2 byte short integer). The `type` argument indicates the type of the noise. `gr_noise_type_t` is an enumeration type defined in `gr_noise_type.h`. Its value can be:

```
GR_UNIFORM
GR_GAUSSIAN
GR_LAPLACIAN
GR_IMPULSE
```

Choosing `GR_UNIFORM` generates uniformly distributed signal between `[-amplitude, amplitude]`. `GR_GAUSSIAN` gives us normally distributed deviate with zero mean and variance $amplitude^2$. Similarly, `GR_LAPLACIAN` and `GR_IMPLUSE` represent a Laplacian distribution and a impulse distribution respectively. All these noise source blocks are based on the pseudo random number generator block gr.random. You may take a look at `/gnuradio-core/src/lib/general/gr_random.h(cc)` to see how to generate a random number following various distributions.

2.3 Null sources

Block: gr.null_source.

Usage:

```
gr.null_source ( size_t  sizeof_stream_item )
```

Notes: gr.null_source produces constant zeros. The argument sizeof_stream_item specifies the data type of the zero stream, such as gr_complex, float, integer and so on.

2.4 Vector sources

Block: gr.vector_source_X.

Usage:

```
gr.vector_source_c [f, i, s, b] ( const std::vector< gr_complex > & data,
                                  bool    repeat = false )

(gr_complex can be replaced by float, integer, short, unsigned char)
```

Notes: The suffix X indicates the data type of the signal source. It can be c (complex), f (float), i (4 byte integer), s (2 byte short integer), or b (1 byte unsigned char). These sources get their data from a vector. The argument repeat decides whether the data in the vector is sent repeatedly. As an example, we can use the block in this way in Python:

```
src_data = (-3, 4, -5.5, 2, 3)
src = gr.vector_source_f (src_data)
```

2.5 File sources

Block: gr.file_source

Usage:

```
gr.file_source ( size_t         itemsize,
                 const char *   filename,
                 bool           repeat )
```

Notes: gr.file_source reads the data stream from a file. The name of the file is specified by filename. The first argument itemsize determines the data type of the stream, such as gr_complex, float, unsigned char. The argument repeat decides whether the data in the file is sent repeatedly. As an example, we can use the block in this way in Python:

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/payload.dat", TRUE)
```

2.6 Audio source

Block: gr.audio_source

Usage:

```
gr.audio_source (int sampling_rate)
```

Notes: `audio_source` reads data from the audio line-in. The argument `sampling_rate` specifies the data rate of the source, in samples per second.

2.7 USRP source

Block: `usrp.source_c [s]`

Usage:

```
usrp.source_c (s) (int             which_board,
                   unsigned int    decim_rate,
                   int             nchan = 1,
                   int             mux = -1,
                   int             mode = 0 )
```

Notes: when you design a receiver using GNU Radio, i.e. working on the RX path, probably you need the USRP as the data source. The suffix `c` (complex), or `s` (short) specifies the data type of the stream from USRP. Most likely the complex source (I/Q quadrature from the digital down converter (DDC)) would be more frequently used. Some of the arguments have been introduced in tutorial 4. `which_board` specifies which USRP to open, which is probably 0 if there is only one USRP board. `decim_rate` tells the digital down converter (DDC) the decimation factor D. `nchan` specifies the number of channels, 1, 2 or 4. `mux` sets input MUX configuration, which determines which ADC (or constant zero) is connected to each DDC input (see tutorial 4 for details). '-1' means we preserve the default settings. `mode` sets the FPGA mode, which we seldom touch. The default value is 0, representing the normal mode. Quite often we only specify the first two arguments, using the default values for the others. For example:

```
usrp_decim = 250
src = usrp.source_c (0, usrp_decim)
```

3 Signal Sinks

3.1 Null sinks

Block: `gr.null_sink`.

Usage:

```
gr.null_sink ( size_t  sizeof_stream_item )
```

Notes: `gr.null_sink` does nothing but eat up your stream. The argument `sizeof_stream_item` specifies the data type of the zero stream, such as `gr_complex`, `float`, `integer` and so on.

3.2 Vector sinks

Block: `gr.vector_sink_X`.

Usage:

```
gr.vector_sink_c [f, i, s, b] ()
```

Notes: The suffix `X` indicates the data type of the signal sink. It can be `c` (complex), `f` (float), `i` (4 byte integer), `s` (2 byte short integer), or `b` (1 byte unsigned char). These sinks write the data into a vector. As an example, we can use the block in this way in Python:

```
dst = gr.vector_sink_f ()
```

3.3 File sinks

Block: `gr.file_sink`

Usage:

```
gr.file_sink ( size_t         itemsize,
               const char *   filename )
```

Notes: `gr.file_source` writes the data stream to a file. The name of the file is specified by `filename`. The first argument `itemsize` determines the data type of the input stream, such as `gr_complex`, `float`, `unsigned char`. As an example, we can use the block in this way in Python:

```
src = gr.file_source (gr.sizeof_char, "/home/dshen/rx.dat")
```

3.4 Audio sink

Block: `gr.audio_sink`

Usage:

```
gr.audio_source (int sampling_rate)
```

Notes: When you finish the signal processing and wish to play the signal through the speaker, you should use the audio sink. `audio_sink` will output the data to the sound card at the sampling rate of `sampling_rate`.

3.5 USRP sink

Block: `usrp.sink_c [s]`

Usage:

```
usrp.sink_c (s) ( int              which_board,
                  unsigned int     interp_rate,
                  int              nchan = 1,
                  int              mux = -1 )
```

Notes: when you design a transmitter using GNU Radio, i.e. working on the TX path, probably you need the USRP as the data sink. The suffix `c` (complex), or `s` (short) specifies the data type of the stream going into USRP. Most likely the complex sink would be more frequently used. Some of the arguments have been introduced in tutorial 4. `which_board` specifies which USRP to open, which is probably 0 if there is only one USRP board. `interp_rate` tells the interpolator on the FPGA the interpolation factor I. Note that the interpolation rate I must be in the range `[4, 512]`, and must be a multiple of 4. `nchan` specifies the number of channels, 1 or 2. `mux` sets output MUX configuration, which determines which DAC is connected to each interpolator output (or disabled). '-1' means we preserve the default settings. Quite often we only specify the first two arguments, using the default values for the others. For example:

```
usrp_interp = 256
src = usrp.sink_c (0, usrp_interp)
```

4 Simple Operators

4.1 Adding a constant

Block: `gr.add_const_XX`

Usage:

```
gr.add_const_cc [ff, ii, ss, sf] ( gr_complex [float, integer, short] k )
```

Notes: The suffix `XX` contains two characters. The first one indicates the data type of the input stream while the second one tells the type of the output stream. It can be `cc` (complex to complex), `ff` (float to float), `ii` (4 byte integer to integer), `ss` (2 byte short integer to short integer) or `sf` (short integer to float). The `gr.add_const_XX` block adds a constant to the input stream so that the signal has a different offset. Note that for `gr.add_const_sf`, the argument k is float. We add a float constant to a short integer stream, and the output stream becomes float.

4.2 Adder

Block: `gr.add_XX`

Usage:

```
gr.add_cc [ff, ii, ss] ( )
```

Notes: The suffix `XX` indicates the data type of the input and output streams. `gr.add_XX` adds all input streams together. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
adder = gr.add_cc ()
fg.connect (stream1, (adder, 0))
fg.connect (stream2, (adder, 1))
fg.connect (stream3, (adder, 2))
fg.connect (adder, outputstream)
```

Then the output of the `adder` is `stream1+stream2+stream3`.

4.3 Subtractor

Block: `gr.sub_XX`

Usage:

```
gr.sub_cc [ff, ii, ss] ( )
```

Notes: The suffix XX indicates the data type of the input and output streams. gr.sub_XX subtracts across all input streams. output = input_0 - input_1 - input_2.... The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
subtractor = gr.sub_cc ()
fg.connect (stream1, (subtractor, 0))
fg.connect (stream2, (subtractor, 1))
fg.connect (stream3, (subtractor, 2))
fg.connect (subtractor, outputstream)
```

Then the output of the subtractor is stream1-stream2-stream3.

4.4 Multiplying a constant

Block: gr.multiply_const_XX

Usage:

```
gr.add_const_cc [ff, ii, ss] ( gr_complex [float, integer, short] k )
```

Notes: The suffix XX indicates the data type of the input and output streams. The gr.multiply_const_XX block multiplies the input stream by a constant k.

4.5 Multiplier

Block: gr.multiply_XX

Usage:

```
gr.multiply_cc [ff, ii, ss] ( )
```

Notes: The suffix XX indicates the data type of the input and output streams. gr.multiply_XX computes the product of all input streams. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
multiplier = gr.multiply_cc ()
fg.connect (stream1, (multiplier, 0))
fg.connect (stream2, (multiplier, 1))
fg.connect (stream3, (multiplier, 2))
fg.connect (multiplier, outputstream)
```

Then the output of the multiplier is stream1×stream2×stream3.

4.6 Divider

Block: gr.divide_XX

Usage:

```
gr.divide_cc [ff, ii, ss] ( )
```

Notes: The suffix `XX` indicates the data type of the input and output streams. `gr.divide_XX` divides across all input streams. `output = input_0 / input_1 / input_2...`. The type of each incoming stream and must be the same. When we use it in Python, multiple upstream block could be connected to it. For example:

```
divider = gr.divide_cc ()
fg.connect (stream1, (divider, 0))
fg.connect (stream2, (divider, 1))
fg.connect (stream3, (divider, 2))
fg.connect (divider, outputstream)
```

Then the output of the `subtractor` is stream1÷stream2÷stream3.

4.7 Log function

Block: `gr.nlog10_ff`

Usage:

```
gr.nlog10_ff ( float n,
               unsigned vlen )
```

Notes: `gr.nlog10_ff` computes $n \times \log_{10}(input)$. Input and output are both float numbers. Ignore the argument `vlen`, the vector length, using its default value 1.

5 Type Conversions

5.1 Complex Conversions

Block:

```
gr.complex_to_float
gr.complex_to_real
gr.complex_to_imag
gr.complex_to_mag
gr.complex_to_arg
```

Usage:

```
gr.complex_to_float( unsigned int vlen )
gr.complex_to_real( unsigned int vlen )
gr.complex_to_imag( unsigned int vlen )
gr.complex_to_mag( unsigned int vlen )
gr.complex_to_arg( unsigned int vlen )
```

Notes: The argument `vlen` is the vector length, we almost always use the default value 1. So just ignore the argument. These blocks convert a complex signal to separate real & imaginary float streams, just the real part, just the imaginary part, the magnitude of the complex signal and the phase angle of the complex signal. Note that the block `gr.complex_to_float` could have 1 or 2 outputs. If it is connected to only one output, then the output is the real part of the complex signal. Its effect is equivalent to `gr.complex_to_real`.

5.2 Float Conversions

Block:

```
gr.float_to_complex
gr.float_to_short
gr.short_to_float
```

Usage:

```
gr.float_to_complex ( )
gr.float_to_short ( )
gr.short_to_float ( )
```

Notes: These blocks are like 'adapters', used to provide the interface between two blocks with different types. Note that the block `gr.float_to_complex` may have 1 or 2 inputs. If there is only one, then the input signal is the real part of the output signal, while the imaginary part is constant 0. If there are two inputs, they serve as the real and imaginary parts of the output signal respectively.

6 Filters

6.1 FIR Designer

Block: `gr.firdes`

Notes: `gr.firdes` has several static public member functions used to design different types of FIR filters. These functions return a vector containing the FIR coefficients. The returned vector is often used as an argument of other FIR filter blocks.

6.1.1 Low Pass Filter

Usage:

```
vector< float > gr.firdes::low_pass ( double      gain,
                                      double      sampling_freq,
                                      double      cutoff_freq,
                                      double      transition_width,
                                      win_type    window = WIN_HAMMING,
                                      double      beta = 6.76)    [static]
```

Notes: `low_pass` is a static public member function in the class `gr.firdes`. It designs the FIR filter coefficients (`taps`) given the filter specifications. Here the argument `window` is the type of the window used in the FIR filter design. Valid values include

```
WIN_HAMMING
WIN_HANN
WIN_BLACKMAN
WIN_RECTANGULAR
```

6.1.2 High Pass Filter

Usage:

```
vector< float > gr.firdes::high_pass ( double      gain,
                                       double      sampling_freq,
                                       double      cutoff_freq,
                                       double      transition_width,
                                       win_type    window = WIN_HAMMING,
                                       double      beta = 6.76)   [static]
```

6.1.3 Band Pass Filter

Usage:

```
vector< float > gr.firdes::band_pass ( double      gain,
                                       double      sampling_freq,
                                       double      low_cutoff_freq,
                                       double      high_cutoff_freq,
                                       double      transition_width,
                                       win_type    window = WIN_HAMMING,
                                       double      beta = 6.76)   [static]
```

6.1.4 Band Reject Filter

Usage:

```
vector< float > gr.firdes::band_reject ( double      gain,
                                         double      sampling_freq,
                                         double      low_cutoff_freq,
                                         double      high_cutoff_freq,
                                         double      transition_width,
                                         win_type    window = WIN_HAMMING,
                                         double      beta = 6.76)   [static]
```

6.1.5 Hilbert Filter

Usage:

```
vector< float > gr.firdes::hilbert ( unsigned int     ntaps,
                                     win_type    windowtype = WIN_RECTANGULAR,
                                     double      beta = 6.76 )   [static]
```

Notes: `gr.firdes::hilbert` designs a Hilbert transform filter. `ntaps` is the number of taps, which must be odd.

6.1.6 Raised Cosine Filter

Usage:

```
vector< float > gr.firdes::root_raised_cosine ( double    gain,
                                                double    sampling_freq,
                                                double    symbol_rate,
                                                double    alpha,
                                                int       ntaps )   [static]
```

Notes: `gr.firdes::root_raised_cosine` designs a root cosine FIR filter. The argument `alpha` is the excess bandwidth factor. `ntaps` is the number of taps.

6.1.7 Gaussian Filter

Usage:

```
vector< float > gr.firdes::gaussian ( double    gain,
                                      double    sampling_freq,
                                      double    symbol_rate,
                                      double    bt,
                                      int       ntaps )   [static]
```

Notes: `gr.firdes::gaussian` designs a Gaussian filter. The argument `ntaps` is the number of taps.

6.2 FIR Decimation Filters

Block:

```
gr.fir_filter_ccc
gr.fir_filter_ccf
gr.fir_filter_fcc
gr.fir_filter_fff
gr.fir_filter_fsf
gr.fir_filter_scc
```

Usage:

```
gr.fir_filter_ccc (int decimation,
                   const std::vector< gr_complex > & taps )
gr.fir_filter_ccf (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_fcc (int decimation,
                   const std::vector< gr_complex > & taps )
gr.fir_filter_fff (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_fsf (int decimation,
                   const std::vector< float > & taps )
gr.fir_filter_scc (int decimation,
                   const std::vector< gr_complex > & taps )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

Each block has two arguments. The first one is the decimation rate of the FIR filter. If it is 1, then it's just a normal 1:1 FIR filter. The second argument `taps` is the vector of the FIR coefficients, which we get from the FIR filter designer block `gr.firdes`.

6.3 FIR Interpolation Filters

Block:

```
gr.interp_fir_filter_ccc
gr.interp_fir_filter_ccf
gr.interp_fir_filter_fcc
gr.interp_fir_filter_fff
gr.interp_fir_filter_fsf
gr.interp_fir_filter_scc
```

Usage:

```
gr.interp_fir_filter_ccc (unsigned interpolation,
                           const std::vector< gr_complex > & taps )
gr.interp_fir_filter_ccf (unsigned interpolation,
                           const std::vector< float > & taps )
gr.interp_fir_filter_fcc (unsigned interpolation,
                           const std::vector< gr_complex > & taps )
gr.interp_fir_filter_fff (unsigned interpolation,
                           const std::vector< float > & taps )
gr.interp_fir_filter_fsf (unsigned interpolation,
                           const std::vector< float > & taps )
gr.interp_fir_filter_scc (unsigned interpolation,
                           const std::vector< gr_complex > & taps )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

Each block has two arguments. The first one is the interpolation rate of the FIR filter. The second argument `taps` is the vector of the FIR coefficients, which we get from the FIR filter designer block `gr.firdes`.

6.4 Digital Down Converter with FIR Decimation Filters

Block:

```
gr.freq_xlating_fir_filter_ccc
gr.freq_xlating_fir_filter_ccf
gr.freq_xlating_fir_filter_fcc
gr.freq_xlating_fir_filter_fcf
gr.freq_xlating_fir_filter_scc
gr.freq_xlating_fir_filter_scf
```

Usage:

```
gr.freq_xlating_fir_filter_ccc [ccf, fcc, fcf, scc, scf]
```

```
( int      decimation,
  const std::vector< gr_complex [float] > &   taps,
  double      center_freq,
  double      sampling_freq )
```

Notes: These blocks all have a 3-character suffix. The first one indicates the data type of the input stream, the second one tells the type of the output stream, and the last one represents for the data type of the FIR filter taps.

These blocks are used as FIR filters combined with frequency translation. Recall that in tutorial 4, in the FPGA there are digital down converters (DDC) translating the signal from IF to baseband. Then the following decimator downsamples the signal and selects a narrower band of the signal by low-pass filtering it. These blocks do the same things except in software. These classes efficiently combine a frequency translation (typically down conversion) with an FIR filter (typically low-pass) and decimation. It is ideally suited for a 'channel selection filter' and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

6.5 Hilbert Transform Filter

Block: `gr.hilbert_fc`

Usage:

```
gr.hilbert_fc( unsigned int ntaps )
```

Notes: `gr.hilbert_fc` is a Hilbert transformer. The real output is input appropriately delayed. Imaginary output is Hilbert filtered (90 degree phase shift) version of input. We only need to specify the number of taps `ntaps` when using the block. The Hilbert filter designer `gr.firdes::hilbert` is used implicitly in the implementation of the block `gr.hilbert_fc`.

6.6 Filter-Delay Combination Filter

Block: `gr.filter_delay_fc`

Usage:

```
gr.filter_delay_fc( const std::vector< float > &  taps )
```

Notes: This is a filter-delay combination block. The block takes one or two float stream and outputs a complex stream. If only one float stream is input, the real output is a delayed version of this input and the imaginary output is the filtered output. If two floats are connected to the input, then the real output is the delayed version of the first input, and the imaginary output is the filtered output. The delay in the real path accounts for the group delay introduced by the filter in the imaginary path. The filter taps needs to be calculated using `gr.firdes` before initializing this block.

6.7 IIR Filter

Block: `gr.iir_filter_ffd`

Usage:

```
gr.iir_filter_ffd ( const std::vector< double > &  fftaps,
                    const std::vector< double > &  fbtaps)
```

Notes: The suffix `ffd` means float input, float output and double taps. This IIR filter uses the Direct Form I implementation, where `fftaps` contains the feed-forward taps, and `fbtaps` the feedback ones. `fftaps` and `fbtaps` must have equal numbers of taps. The input and output satisfy a difference equation of the form

$$y[n] - \sum_{k=1}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

with the corresponding rational system function

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}}$$

Note that some texts define the system function with a '+' in the denominator. If you're using that convention, you'll need to negate the feedback taps.

6.8 Single Pole IIR Filter

Block: `gr.single_pole_iir_filter_ff`

Usage:

```
gr.single_pole_iir_filter_ff ( double alpha,
                               unsigned int    vlen )
```

Notes: This is a single pole IIR filter with float input, float output. The input and output satisfy a difference equation of the form:

$$y[n] - (1-\alpha)y[n-1] = \alpha x[n]$$

with the corresponding rational system function

$$H(z) = \frac{\alpha}{1-(1-\alpha)z^{-1}}$$

Note that some texts define the system function with a + in the denominator. If you're using that convention, you'll need to negate the feedback tap. The argument `vlen` is the vector length. We usually use its default value 1, so just ignore it.

7 FFT

Block:

```
gr.fft_vcc
gr.fft_vfc
```

Usage:

```
gr.fft_vcc ( int     fft_size,
             bool    forward,
             bool    window )
gr.fft_vfc ( int     fft_size,
             bool    forward,
             bool    window )
```

Notes: These blocks are used to compute the fast Fourier transforms of the input sequence. For `gr.fft_vcc`, it computes forward or reverse FFT, with complex vector in and complex vector out. For `gr.fft_vfc` it computes forward FFT with float vector in and complex vector out.

8 Other useful blocks

8.1 FM Modulation and Demodulation

Block:

```
gr.frequency_modulator_fc
gr.quadrature_demod_cf
```

Usage:

```
gr.frequency_modulator_fc ( double sensitivity )
gr.quadrature_demod_cf ( float gain )
```

Notes: The `gr.frequency_modulator_fc` block is the FM modulator. The instantaneous frequency of the output complex signal is proportional to the input float signal. We have seen `gr.quadrature_demod_cf` in the FM receiver example. It calculates the instantaneous frequency of the input signal.

8.2 Numerically Controlled Oscillator

Block: `gr.fxpt_nco`

Usage: This block is used to generate sinusoidal waves. We can set or adjust the phase and frequency of the oscillator by several public member functions of the class. The functions include:

```
void    set_phase (float angle)
void    adjust_phase (float delta_phase)
void    set_freq (float angle_rate)
void    adjust_freq (float delta_angle_rate)
void    step ()
void    step (int n)
float   get_phase () const
float   get_freq () const
void    sincos (float *sinx, float *cosx) const
float   cos () const
float   sin () const
```

We use `cos()` or `sin()` function to get the sinusoidal samples. They calculate sin and cos values according to the current phase. The `freq` is actually the phase difference between consecutive samples. When we call `step()` method, the current phase will increase by the `freq`.

8.3 Blocks for digital transmission

Block:

```
gr.bytes_to_syms
gr.simple_framer
gr.simple_correlator
```

Usage:

```
gr.bytes_to_syms ( )
gr.simple_framer ( int payload_bytesize )
gr.simple_correlator ( int payload_bytesize )
```

Notes: `gr.bytes_to_syms ( )` converts a byte sequence (for example, a unsigned char stream) to a $\pm$ sequence, i.e. the digital binary symbols. `gr.simple_framer` packs a byte stream into packets, with the length of `payload_bytesize`. Then necessary synchronization and command bytes are added at the head. `gr.simple_correlator` is a digital detector, which synchronizes the symbol and frame, finally detects the digital information correctly. The real implementation of these blocks is a little bit complicated. Please study the FSK example `fsk_r[t]x.py` and the source code of these blocks. These blocks are very helpful when we want to design digital transmission schemes.

9 Wrap up

This tutorial reviews some of the most frequently used blocks in GNU Radio. Using these blocks skillfully would be very important and useful. Certainly we only discuss a fraction of the available blocks. There are some other more advanced, less used blocks that we didn't talk about. Also more and more blocks are coming as GNU Radio gets more popular (maybe including your block one day). Our introduction is very simple. If you wish to know all the details about a block, please go to its documentation page and then read its source code directly. The source code is the best place to understand what's going on in a block thoroughly. Studying some of the examples to see how a block a used is also a very good way.

References

[1] **GNU Radio 2.x Documentation** http://www.gnu.org/software/gnuradio/doc/index.html