Tutorial 7: Exploring the FM receiver

Dawei Shen[*]

July 12, 2005

Abstract

In tutorial 5, we skip the discussion on how the FM signal is demodulated, leaving the big '`guts`' as a black box. In this article, the real signal processing techniques for demodulating the FM signal are introduced. We will dig into this black box and see how the signal is processed in the software world.

1 Overview

In previous tutorials, we have introduced the hardware setup of GNU Radio and some programming tips, which form the basis of a real application in the 'soft world'. In this article, we will investigate how the broadcast FM signal is demodulated. The FM receiver is a typical example in GNU Radio. You are able to hear strong stations using just a piece of wire. In tutorial 5, we skipped the introduction to the big box '`guts`', where the real magics of the FM detection are. We will show how the signal is processed from the air to the sound card in this tutorial.

2 From the air to the computer, from real to complex

In tutorial 3 and 4, we have discussed the operations on the USRP, especially the role of the digital down converter (DDC). Basically what the USRP does is to select the part of the spectrum we are interested in and decimate the digital sequence by some factor N. The resulting signal is complex (`gr_complex`) with I/Q two channels. So after we finish writing these lines

```
src = usrp.source_c (0, decim)    # decim = 250, so data rate (quad_rate) is 256kHz
src.set_rx_freq (0, IF_freq)      # IF_freq = our input - 128MHz
src.set_pga(0,20)
```

we've got a 'complex' signal, with a data rate 256k samples per second. We name it 'quadrature rate' - `quad_rate` because the complex signal has I/Q quadrature components.

The IF frequency we choose is an interesting and useful point here. `IF_freq` is equal to the user's input minus 128MHz, as the line in function `parseargs()` indicates:

[*]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

```
return freq1 - 128e6
```

`wfm_rcv_gui` was written assuming that there was no RF front end doing any down conversion. The A/D sample rate is 64M. The Nyquist zones are therefore:

```
[0, 32M]    normal
[32M, 64M]  inverted
[64M, 96M]  normal
[96M, 128M] inverted
```

etc.

The problem is that 96M, one of the folding points, occurs right in the middle of the broadcast band. This means that 95.0 and 97.0 both alias down to the same frequency and can't be distinguished from each other. `freq1 -` 128M will give a valid frequency if freq1 is >= 96M. Bottom line: it's a kludge that sort of works.

3 Getting the instantaneous frequency, from complex to real

It's time to dig into the heart of the '`guts`' now, which we left as a big black box in tutorial 5.

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

`blks` is a package in `gnuradio`. It almost does nothing but refers to another package `blksimpl`. `wfm_rcv` is a class defined in `/gnuradio/blksimpl/wfm_rcv.py`, which is real 'processor' of the FM receiver. The source code is appended at the end.

`wfm_rcv` is derived from `gr.hier_block`. `gr.hier_block` describes a series of blocks in tandem in a flow graph. It assumes that there is at most a single block at the head of the chain and a single block at the end of the chain. Either head or tail may be None indicating a sink or source respectively. `hier_block` could be recognized a sub-graph consisting of several blocks connected one after another. To construct a `hier_block`, we need to specify the flow graph that contains this hierarchical block, the first and the last block in the signal processing chain. A hierarchical block could be treated as a common block, which could be placed and connected in a flow graph, like these lines demonstrate:

```
self.connect (src, guts)
self.connect (guts, (audio_sink, 0))
```

The 'head' block in the chain is `fm_demod`, the instance of `gr.quadrature_demod_cf`. To understand the real work within it, we should know a bit about how FM signals are generated. With FM, the instantaneous frequency of the transmitted waveform is varied as a function of the input signal. The instantaneous frequency at any time is given by the following formula:

```
f(t) = k * m(t) + fc
```

m(t) is the input signal, k is a constant that controls the frequency sensitivity and fc is the frequency of the carrier (for example, 100.1MHz). So to recover m(t), two steps are needed. First we need to remove the carrier fc, then we're left with a baseband signal that has an instantaneous frequency proportional to the original message m(t). The second step is obviously to compute the instantaneous frequency of the baseband signal. Thus, our challenge is to find a way to remove the carrier and compute the instantaneous frequency. Removing the carrier has been done on the FPGA, via the digital down converter (DDC), as introduced in tutorial 3 and 4. We have explained why we tune to (fc - 128MHz) in the preceding section. The resulting signal coming into the 'guts' has already become a baseband signal and the remaining task is to calculate its instantaneous frequency. If we integrate frequency, we get phase, or angle. Conversely, differentiating phase with respect to time gives frequency. These are the key insights we use to build the receiver.

We use the `gr.quadrature_demod_cf` block for computing the instantaneous frequency of the baseband signal. We approximate differentiating the phase by determining the angle between adjacent samples. Recall that the digital down converter produces complex numbers on its output. Using a bit more trigonometry, we can determine the angle between two subsequent samples by multiplying one by the complex conjugate of the other and then taking the arc tangent of the product. Once you know what you want, it doesn't take much code. `gr_quadrature_demod_cf.cc` contains the C++ implementation of this block. We will talk about how to write a signal processing block using C++ in detail in tutorial 10 and 11. But it's useful to give a shot at the code now. The bulk of the signal processing is the three-line loop in `sync_work()` function.

```
Part of gr_quadrature_demod_cf.cc
...

int
gr_quadrature_demod_cf::sync_work (
    int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  gr_complex *in = (gr_complex *) input_items[0];
  float *out = (float *) output_items[0];
  in++;          // ensure that in[-1] is valid
  for (int i = 0; i < noutput_items; i++){
    gr_complex product = in[i] * conj (in[i-1]);
    out[i] = d_gain * arg (product);
  }
  return noutput_items;
}
```

A helpful diagram for the FM receiver can be found here.

```
    fm_demod_gain = quad_rate/(2*math.pi*max_dev)
```

Note that `fm_demod_gain` can be treated as a constant controlling the volume. Its real value doesn't matter. Here `arg (product)` is the phase difference between adjacent samples, if we divide it by the sample interval, i.e. multiply the data rate `quad_rate`, we get the radian frequency ω, which gives us the instantaneous frequency f if we further divide ω by 2π. Finally we normalize the frequency by `max_dev`.

4 Deemphasizer

The second block in the chain is a deemphasizer `deemph`, an instance of the class `fm_deemph`. `fm_deemph` is defined in `fm_emph.py`, also located in the package `blksimpl`.

What is deemphasis? Let's introduce it briefly. It has been theoretically proved that, in FM detector, the power of the output noise increases with the frequency quadratically. However, for most practical signals, such as human voice and music, the power of the signal decreases significantly as frequency increases. As a result, the signal noise ratio (SNR) at the high frequency end usually becomes unbearable. To circumvent this effect, people introduce 'preemphasis' and 'deemphasis' into the FM system. At the transmitter, we use proper preemphasis circuits to manually amplify the high frequency components, and do the converse operations at the receiver to recover the original power distribution of the signal. As a result, we improve the SNR effectively.

In the analog world, a simple first order RLC circuit usually suffices for preemphasis and deemphasis. Here is a nice plot of their transfer functions. In our digital signal processing, a first order IIR filter could be the right choice.

```
Part of fm_emph.py
...

#            1
# H(s) = -------
#          1 + s
#
# tau is the RC time constant.
# critical frequency: w_p = 1/tau
#
# We prewarp and use the bilinear z-transform to get our IIR coefficients.
# See "Digital Signal Processing: A Practical Approach" by Ifeachor and Jervis
#
class fm_deemph(gr.hier_block):
    """
    FM Deemphasis IIR filter.
    """
    def __init__(self, fg, fs, tau=75e-6):
        """
        @type fs: float
        @param tau: Time constant in seconds (75us in US, 50us in EUR)
        @type tau: float
        """
        w_p = 1/tau
        w_pp = math.tan (w_p / (fs * 2)) # prewarped analog freq

        a1 = (w_pp - 1)/(w_pp + 1)
        b0 = w_pp/(1 + w_pp)
        b1 = b0

        btaps = [b0, b1]
        ataps = [1, a1]

        deemph = gr.iir_filter_ffd(btaps, ataps)
        gr.hier_block.__init__(self, fg, deemph, deemph)
```

We start from the transfer function of the analog filter as prototype, and use bilinear transformation to get the digital IIR filter. Note that `gr.iir_filter_ffd` is the IIR filter block with float input, float output and double taps.

5 Audio FIR decimation filter

After passing the deemphasizer, how does the signal look like now? First, it's a real signal with a data rate of 256kHz. Second, it's a baseband signal, with effective frequency range from 0 to about 100kHz, containing all the frequency components of a FM station.

As a side note, the bandwidth of a FM station is usually around 2 * 100kHz. This also explains why we choose 256kHz as the quadrature rate (the decimation rate on the USRP is chosen to be 250). A sample rate of 256kHz is just suitable for the 200kHz bandwidth, without losing any spectrum information. Maybe you have noticed, in the FM receiver, we never use any low-pass filtering operation to 'pick out' the FM station we are interested in. Actually this is done implicitly in the digital down converter (DDC) on the USRP. Recall that digital down converter can be regarded as a low-pass FIR filter followed by a downsampler. As a result, the target station is picked out then spread out to the entire digital spectrum after decimation. Because we choose the right decimation rate, we have eventually done a lot! The bottom line is, we are operating on a single FM station after the signal goes through the USRP.

Now we need to resolve two issues. First, the signal rate is 256kHz now, much higher than what the sound card can adopt. The PC sound cards usually sample up to 96,000 Hz maximum. Second, the 100kHz spectrum contains several channels, L + R, L - R, pilot tones, etc. To keep our life simpler, we just want to design a mono receiver low-passing only the L + R signal. So clearly, an FIR decimation filter is exactly what we want.

To make things clearer, here is a brief introduction to the FM signal band. From 0 to about 16kHz is the left plus right (L + R) audio. The peak at 19kHz is the stereo pilot tone. The left minus right (L - R) stereo information is centered at 2x the pilot (38kHz) and is AM-modulated on top of the FM. Additional subcarriers are sometimes found in the region of 57kHz - 96kHz. We can use the GNU Radio built-in fft block with GUI supports to view the spectrum of the demodulated signal (details will be introduced in tutorial 8). Here is a nice real plot given by Eric. Here is a good illustration of the FM band.

OK, now let's design the FIR decimation filter. The GNU Radio block `gr.fir_filter_fff` gives us an FIR filter with float input, float output, and float taps. Its constructor takes two arguments: the first one is the decimation factor, the second one is the filter coefficients (taps) vector.

```
    self.audio_filter = gr.fir_filter_fff (audio_decimation, audio_coeffs)
```

If we need an FIR filter without changing the data rate, then we just simply set the decimation rate to be 1. If we need an interpolation filter rather than a decimation filter, then the GNU Radio block `gr.interp_fir_filter_xxx` is what we should choose.

The filter coefficients `audio_coeffs` are obtained using the FIR filter design block `gr.firdes`. `low_pass()` is a static public function defined in the class `gr_firdes`. Similarly, it also has `high_pass()`, `band_pass()`, `band_reject()` functions. We use these functions to design the FIR filter taps, provided the filter parameters and specifications. For example, the syntax for design a low-pass filter is:

```
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]
```

The meaning of each argument is quite obvious. Note that `beta` is a parameter for Kaiser window. In our example, we select the audio decimation factor (`audio_decimation`) to be 8, so that the resulting data rate for the sound card is 32kHz. We are only interested in the L + R audio from 0 to 16kHz, so we low pass the output of the quadrature demodulator with a cutoff frequency of 16kHz. This gives us a monaural output that we connect to the sound card outputs. In our example, we choose the cutoff frequency as 15kHz and transition band as 1kHz, which is reasonable.

```
width_of_transition_band = audio_rate / 32
audio_coeffs = gr.firdes.low_pass (volume,         # gain                    (20)
                                   quad_rate,      # sampling rate          (32kHz)
                                   audio_rate/2 - width_of_transition_band, (15kHz)
                                   width_of_transition_band,                (16kHz)
                                   gr.firdes.WIN_HAMMING)
```

OK! Our FM receiver is complete! The signal is at the door of the sound card and is ready to be played. Note that the usage of FIR filters, as well as multirate processing is very important in the digital signal processing.

Finally, we connect these blocks and call the `__init__()` method of `gr.hier_block` to complete the `__init__()` method of the `wfm_rcv` class. Here we need to specify the head and the tail of the pipeline.

```
fg.connect (self.fm_demod, self.deemph, self.audio_filter)
gr.hier_block.__init__(self,
                       fg,
                       self.fm_demod,       # head of the pipeline
                       self.audio_filter)   # tail of the pipeline
```

6 Conclusion

In this article, we have introduced the FM detection techniques and how they are implemented using GNU Radio. Now we can see GNU Radio is really a nice system, providing us so many powerful tools and flexible ways to construct a real application. In next tutorial, we will wrap up the explanation of `wfm_rcv_gui.py`, with an emphasis on the GNU Radio GUI tools.

APPENDIX A: The source code

```
from gnuradio import gr
from gnuradio.blksimpl.fm_emph import fm_deemph
import math

class wfm_rcv(gr.hier_block):
    def __init__ (self, fg, quad_rate, audio_decimation):
        """
        Hierarchical block for demodulating a broadcast FM signal.

        The input is the downconverted complex baseband signal (gr_complex).
        The output is the demodulated audio (float).

        @param fg: flow graph.
        @type fg: flow graph
        @param quad_rate: input sample rate of complex baseband input.
        @type quad_rate: float
        @param audio_decimation: how much to decimate quad_rate to get to audio.
        @type audio_decimation: integer
        """
        volume = 20.

        max_dev = 75e3
        fm_demod_gain = quad_rate/(2*math.pi*max_dev)
        audio_rate = quad_rate / audio_decimation
```

```
# We assign to self so that outsiders can grab the demodulator
# if they need to.  E.g., to plot its output.
#
# input: complex; output: float
self.fm_demod = gr.quadrature_demod_cf (fm_demod_gain)

# input: float; output: float
self.deemph = fm_deemph (fg, quad_rate)

# compute FIR filter taps for audio filter
width_of_transition_band = audio_rate / 32
audio_coeffs = gr.firdes.low_pass (volume,         # gain
                                   quad_rate,      # sampling rate
                                   audio_rate/2 - width_of_transition_band,
                                   width_of_transition_band,
                                   gr.firdes.WIN_HAMMING)
# input: float; output: float
self.audio_filter = gr.fir_filter_fff (audio_decimation, audio_coeffs)

fg.connect (self.fm_demod, self.deemph, self.audio_filter)

gr.hier_block.__init__(self,
                       fg,
                       self.fm_demod,       # head of the pipeline
                       self.audio_filter)   # tail of the pipeline
```

References

[1] Eric Blossom, **Listening to FM Radio in Software, Step by Step**, http://www.linuxjournal.com/article/7505