# A Software Defined Radio Receiver

Ben Gelb

June 8, 2004

**Abstract**

The current availability of high-performance computing has given rise to the era of digital signal processing. This paper will outline the design of a software radio receiver which combines high speed analog to digital conversion with digital signal processing techniques to demodulate radio-frequency signals.

# 1 Introduction

As an electo-magnetic wave moves past a conductor, it causes the electrons in the wire to oscillate and produce a measurable voltage at points within the wire. The frequency of this oscillation is what is called a signal's *frequency*.

Most radio receivers use analog circuitry to downconvert the signal to a lower frequency which is audible to the human ear, then demodulates the signal into intelligible speech.

A software defined radio receiver, on the other hand, works by directly measuring the voltage present in the conductor (this conductor, by the way, is called an antenna) with an analog to digital converter (ADC) and quantizing it as an integer relative to a fixed voltage reference.

Through a process called sampling, voltage *samples* are collected at a high rate (in the case of the design detailed herein, at a rate of 20MHz).

# 2 Sampling Theory

## 2.1 What is sampling?

Sampling is the process of digitizing a signal by recording discrete samples of the signal's amplitude at specific points in time. If done right, a signal can be captured and perfectly reconstructed without losing *any* information.

## 2.2 The rules

Like any good theory, the sampling theory is bound by a few conditions.

One of the first conditions is that the signal we are digitizing be *band-limited*. This means that we must guarantee that the frequncies contained in the input signal fall within a certain range - or bandwidth. This is accomplished by the use of a filter (usually a low-pass filter)

which allows certain frequencies to pass, and drops others.

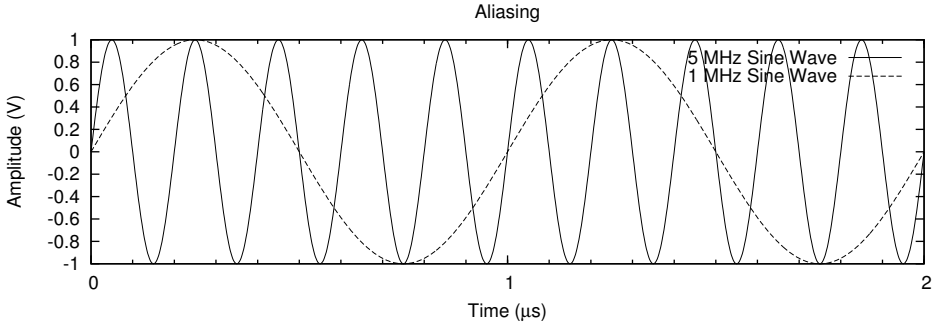
It turns out that any non-sinosoudial waveform is comprised of an infinite series of sine and cosine functions oscillating at different frequencies. A square wave, for example, contains harmonics that are much higher in frequncy than the fundamental wave. Removing these components can alter the shape of a wave, so its important to take this into accont when calculating the bandwidth.

The next condition says that if we sample a signal fast enough, we won't lose any information. But how fast is fast enough?

Eq (1), known as the Nyqist Rate Equation, tells us that our sampling rate must be twice as fast as the bandwidth of the signal we want to digitize (in a simple case, highest frequency we're going to sample).

$$\text{Eq (1) } f_s = 2f_{bw}$$

To see why Nyquist's equation is true, consider what would happen if we ignored it, and sampled the following signals at a rate of 4 MHz.




The 1 MHz wave would be sampled properly, but the 5 MHz wave would not. Notice that if the sample points were at $0\mu s$, $.25\mu s$, $.50\mu s$, $.75\mu s$ and $1\mu s$, the samples will be the same for both signals. As a result, the 5 MHz signal will not appear to be a 5 MHz signal, but rather as an *alias* at 1 MHz, right on top of the real 1 MHz signal. This is bad for business. Two signals on top of each other are worthless, because you can't tell them apart, and can't accurately do any processing with them.

If we wanted to use both signals we'd need to up the sample rate to at least 10 MHz, but if only one was of concern, we'd have a few options.

If we used a low-pass filter with a cutoff of 2 MHz, (half of the sampling rate), the 5 MHz signal would be filtered out and the 1 MHz signal could be digitized properly.

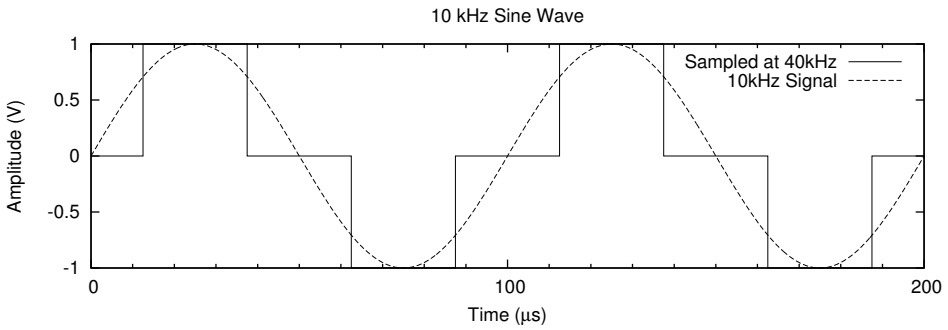
But what if we were concerned with the 5 MHz signal? As long as the passband we care about is less than 2 MHz, we should be able to do it. Suppose we install a bandpass filter that passes signals from 4 to 6 MHz through, and drops all others. The effects of aliasing suddenly turn in our favor. Because we can predict exactly where the alias will appear, and because we've made sure no other signals will collide with that alias, we can effectively downconvert the 4 to 6 MHz passband to 0 to 2 MHz.

In either case, we're limiting the bandwidth as prescribed by Eq (1). The band-limiting filter, incidentally, is called an anti-aliasing filter, because it prevents the appearance of aliases in the sampled data.

## 2.3 But mommy, there's holes in the signal!

It's probably occurred to you by now that despite the fact that an analog signal varies continuously in infinitely small variations of amplitude over infinitely small increments of time, I've claimed that we can capture and reconstruct a signal with a finite number of discrete samples without any loss of information. How could I have the gall to claim such a thing? By capturing only discrete steps of the signal, it seems certain that something must have been left out, or that there are "holes" in the signal.

I must admit the concept that we haven't lost anything is a bit counterintuitive, but my claim is actually quite real. Consider what would happen if we took a 10 kHz sine wave, and sampled it at 40 kHz.




It would appear from this example that we've lost a part of our signal. It definitely looks inferior to the original waveform (dotted line), and the effect of the digitization is obvious. How is it then that we have not lost any information?

The original signal is a sine wave and the digitized version looks like a sort of stair-steppy squareish wave. In any case, it certainly doesn't look like perfect recovery. Remember though, that the input signal was band limited at half the sampling rate. The square wave is composed of many sine waves, one at the fundamental frequency (10kHz), and several higher frequency components - one at the sampling frequency (40kHz), one at every harmonic of the sampling frequency (80kHz, 120kHz, etc.), and the sum and difference of the fundamental and the sampling frequency and each of its harmonics. The overall effect is the occurrence positive and negative *images* of the frequencies in the Nyquist range centered around each of the harmonics of the sampling frequency.

All of the harmonics have one thing in common - they're all outside of the allowable range for the original signal, so it's easy to determine that they aren't supposed to be there. If the same band-limiting filter (now called an anti-imaging filter instead of an anti-aliasing filter) is applied to the output as was applied to the input, the extraneous components are removed and all that's left is the original 10kHz sine wave, perfectly recovered. Cool, huh?

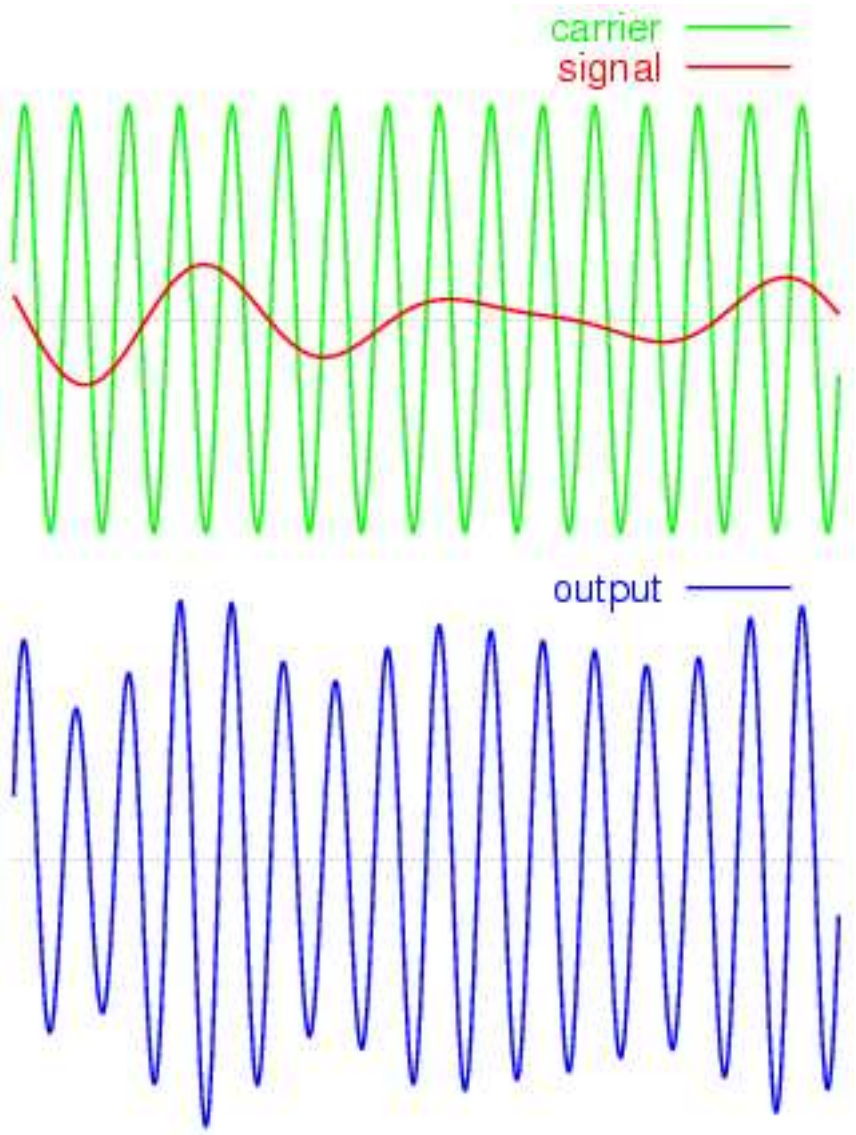
# 3 Modulation Theory

If all radio signals were just simple sine waves, they would not be terribly useful. You'd be severely limited in your ability to communicate, in that you'd be able to communicate two states - on or off. Once upon a time, all radio communication was actually done in this way. It was called Morse Code, and it worked pretty well in its day, but it hardly stacks up to the advanced communication modes of late.

Information is carried on waves by different forms of *modulation*. There are many different forms of modulation, but all of them work by modifying one or more of three properties of a carrier wave; frequency, phase, and amplitude.

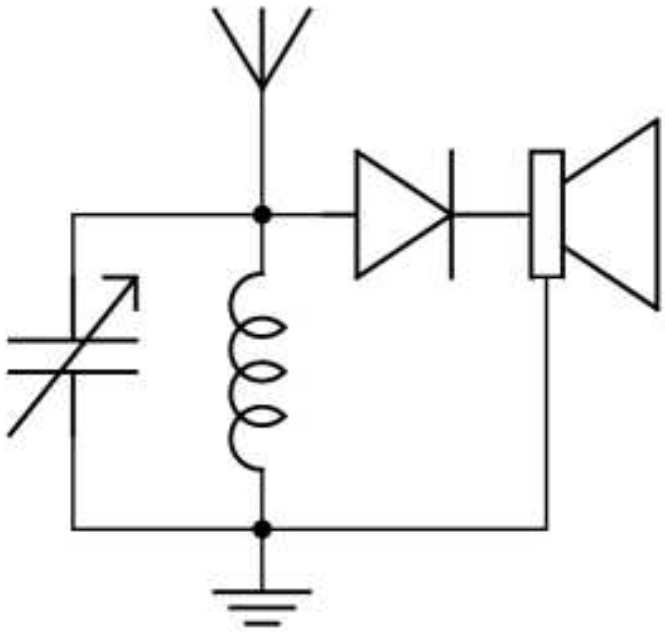
## 3.1 Amplitude Modulation

One of the simplest forms of modulation, called Amplitude Modulation, puts an audio signal on a radio-frequency signal by varying the amplitude of the carrier. The following illustration gives a pretty clear picture of this simple concept.




## 3.2 Traditional AM Demodulation

A electronic circuit, shown below in its absolute simplest form, can reverse this process - seperating out the audio from an AM modulated radio signal.



A nearby radio transmitter will cause a voltage in the antenna wire. The tunable LC circuit will cause all signals except the one on the resonant frequency to be shorted to ground, so they will not be demodulated. This forms a rudimentary filter which allows the tuning of a specific station. The signal that is tuned-in will look like the AM modulated signal in section 3.1.

The average voltage is 0 volts, as the signal is symmetrical with respect to the x-axis. By passing the signal through the diode shown in the circuit above, it's as if the bottom half of the waveform was chopped off (a diode allows current to flow in only one direction). Once the signal is aymmetrical with respect to the x-axis, the voltage is not 0 and it is not constant. It varies exactly as the audio signal used to modulate the wave, and is then sent into the earphone (we can ignore the bumpiness of because it's way beyond hearing range, or, if we feel like being espescially elegant, we could filter out the radio-frequency component with a small capacitor, which looks like a conductor at high frequencies, but an insulator at low ones).

Before we can go any further, we must leave modulation for a minute to explain a

particular method of sampling, critical to digital signal processing.

## 3.3 Quadrature Sampling

If you remember what we said about modulation just a moment ago, there are three properties of a wave that can be used to convey information; phase, frequency (rate of change of phase), and amplitude. A stream of individual discrete samples is great for storing and recalling analog information, for example, a CD player, or computer soundcard. A single discrete sample, however, fails to tell much about any of the three quantities we're so concerned with, which is why a technique called quadrature sampling is employed in DSP applications. The idea behind quadrature sampling is to use two sets of samples which are 90 degrees out of phase with each other. By pairing each in-phase sample with a quadrature sample (90 degrees later) we can determine the instantaneous amplitude and phase of a signal from any one sample by using simple trigonometry.

## 3.4 Demodulating AM in Software

In order to demodulate an AM signal in software, we need to find the amplitude of the carrier wave. The amplitude is what carries the voice information on an AM signal. Below I derive the math for finding the amplitude.

The in-phase component, $I$, can be expressed as

$I = msin\phi$

where $m$ is the amplitude of of the wave, and $\phi$ cycles from 0 to $2\pi$ at the carrier frequency.

A cosine function shifted 90 degrees is equal to a sine function, so the quadrature component $Q$ can be written

$Q = mcos\phi$

To find $m$ start by adding the squares of $I$ and $Q$.

$I^2 + Q^2 = m^2 cos^2\phi + m^2 sin^2\phi = m^2(cos^2\phi + sin^2\phi)$

Remember that Pythagoras said that $cos^2\theta + sin^2\theta = 1$, so,

$I^2 + Q^2 = m^2$

Take the square root of both sides, and voila!

$m = \sqrt{I^2 + Q^2}$

# 4 System Setup

In this section I describe the configuration I used, both the GnuRadio software and the hardware I used for my experimentation with software defined radio technology.

## 4.1 Hardware

There were two main pieces of hardware used in my project (besides the computer itself). First and foremost was a large wire antenna, 102 feet in length, strung between two trees at my residence. The antenna was fed with coaxial cable, which ran from the antenna, inside my house to my computer.

The computer was outfitted with a PCI-DAS4020 data acquisition card supplied free of charge by Measurement Computing. The Measurement Computing card has an analog to digital converter which samples at 20 MHz. The antenna was connected directly to the input of the data acquisition card and the RF voltages on the antenna were measured.

## 4.2 Software Configuration

The GnuRadio package was used as a platform for experimentation with software radio. The GnuRadio codebase is organized into many logical modular pieces of code which can be assembled together into a processing chain. The topology of the processing chain determines

what function the software radio will perform. In my case, it was fairly straightforward. Data was captured from the data acquisition card, decimated into I and Q and downmixed to a 0 Hz center, filtered so that only one signal was being processed (not every AM station at once), demodulated, filtered and decimated again, and written to the sound card.

## 4.3 Data Capture

Two methods were used to capture data from the data acquisition card. The first, and simplest, is a command line utility which collects data from the card at writes it to a file. The data can later be read out of the file and processed. This method has some serious limitations, however. Each sample is 12 bits long, and is stored as a 16-bit integer (there are no 12-bit datatypes). At a sampling rate of 20 MHz, that is 40 MB of data per second! This exceeds the sustained write capability of most hard drives, so data must be collected into RAM. With 512 MB of RAM, this limits capture to about 10 seconds worth.

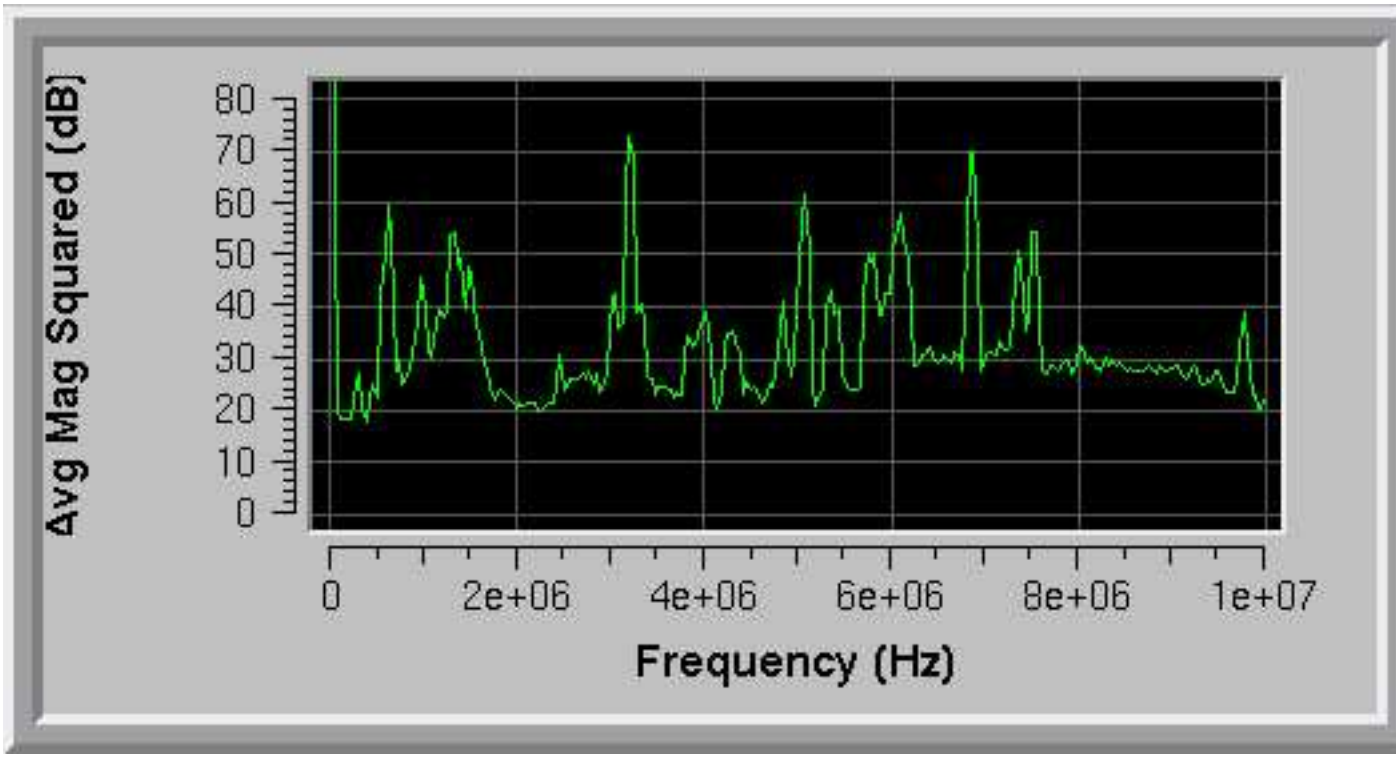
The other method of data capture is to capture the data directly into the program doing signal processing. This way, the data does not have to be written to disk because it is captured and processed in real time. This functionality is provided by a data source object which communicates with the data acquisition board and can be connected to other processing blocks.

## 4.4 Fast Fourier Transform

The Fast Fourier Transform, a type of Discrete Fourier Transform, is a powerful digital signal processing tool. It transforms a signal in the time domain into the frequency domain. This can be used for many purposes. In my case, I used it to help me visualize signals at various stages in the processing chain by plotting the output of an FFT onto a display.

Below is the output of an FFT plot which ranges from 0 to 10 MHz. The peaks show

strong radio signals. The large cluster to the left of the plot shows the American AM broadcast band. The peaks over the rest of the spectrum are from shortwave broadcasting stations, some domestic, and others based elsewhere around the globe.




## 4.5 Digital Filters

Digital filtering is very important to a software defined radio system. Digital filters, just like analog filters, drop signal of certain frequencies while allow others to pass unattenuated. Filters are used in my project to seperate out a single radio station from an entire 10 MHz section of spectrum (the bandwidth allowed by Nyquist equation).

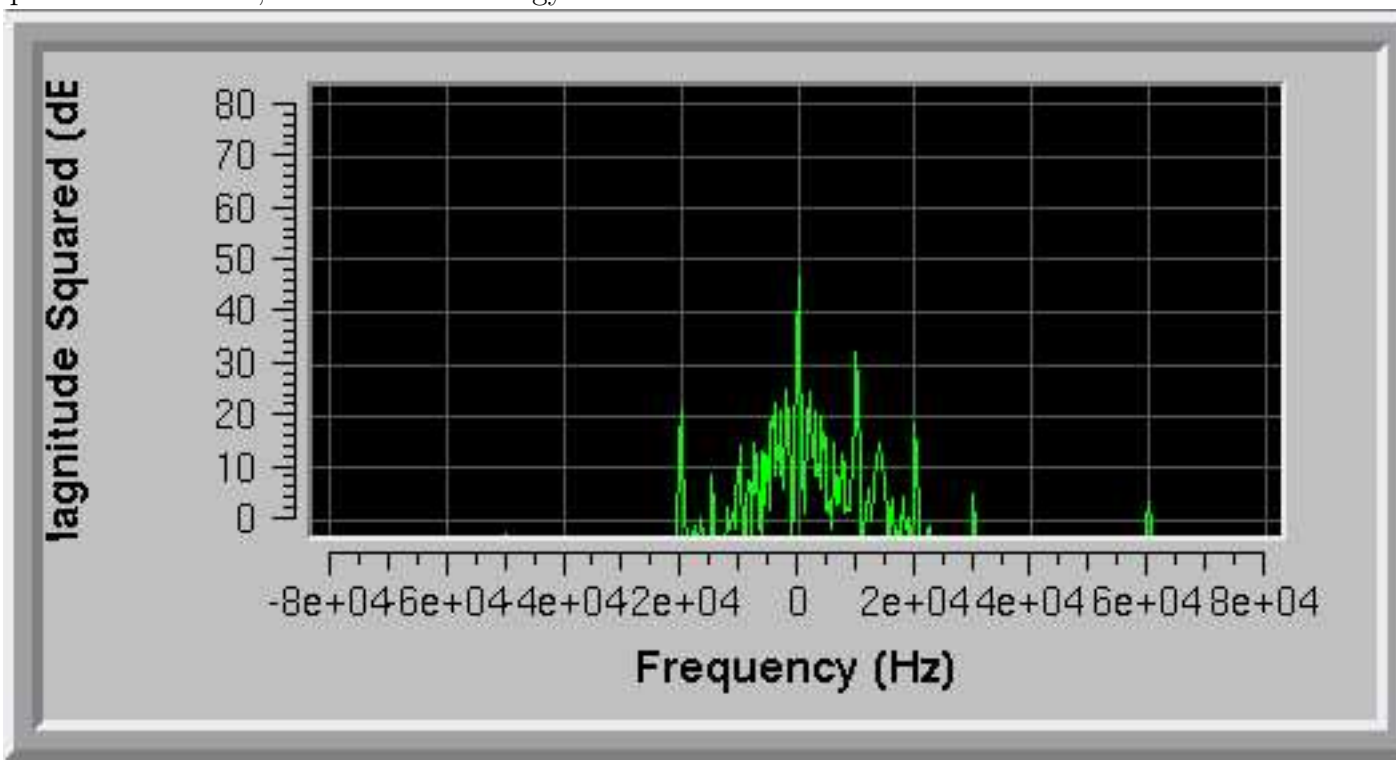
Filters also will *decimate* the sample data. The bandwidth allowed to pass through a filter is often considerably smaller than the range of signals going into a filter. After filtering is performed, the sampling rate is often far faster than is necessary. Decimation combines samples together to lower the sampling rate, which means less data to process, so less CPU time is used. There is no loss of information here (as long as Nyquist's rules are

followed). The other reason for reducing the sampling rate is so the data can be written to the sound card for output. The soundcard supports only a few sampling rates, like 32 kHz, for example. So, the original sampling rate of 20 MHz must be reduced to 32 kHz by the end of the processing chain in order to be ouput.

## 4.6 Decimating Into Quadrature

The data acquisition card does not perform quadrature sampling on its own. A special processing block transforms the one dimensional set of samples into a two dimensional set by pairing up sets of samples and reducing the overall sampling rate. The same processing block also performs a transform with shifts a particular frequency to 0 Hz. This is necessary so that all other signals can be filtered out.

The following figure depicts a filtered and shifted AM broadcast signal. The carrier is the peak in the center, and the audio energy is in the sidebands.



### 4.7 Demodulation

The demodulation math is another processing block. It uses the math outlined in section 3.4. I wrote my own version of this block, which is contained in the appendix of this paper.

# 5 Appendix

## 5.1 am_demod.cc

```
/*
 * File: am_demod.cc
 * Program that uses GNURadio libraries to demodulate
 * AM signals.
 *
 * Date: 11/7/2003
 * (c) 2003 Ben Gelb
 *
 */

#include <make_GrMC4020Source.h>
#include <GrFFTAvgSink.h>
#include <GrFFTSink.h>
#include <VrFixOffset.h>
#include <GrFreqXlatingFIRfilterSCF.h>
#include "GrAMDemod.h" //AM Demodulator Code
#include <GrFIRfilterFSF.h>
#include <VrAudioSink.h>
#include <VrConnect.h>
#include <VrMultiTask.h>
#include <VrGUI.h>
#include <gr_firdes.h>
#include <gr_fir_builderF.h>
#include <VrNullSink.h>
#include <VrFileSource.h>
#include <getopt.h>

const int inputRate = 20000000; // input sample rate from PCI-DAS4020/12

float Freq = 1.5e6;
```

```
const int chanTaps = 75;
const int CFIRdecimate = 125;
const float chanGain = 2.0;


const float demodGain = 6000;

const int RFIRdecimate = 5;
const int audioTaps = 50;
const float audioGain = 1.0;

const int demodRate = inputRate / CFIRdecimate;
const int audioRate = demodRate / RFIRdecimate;

int main (int argc, char **argv)
{
float f;

    VrGUI *guimain = 0;
    VrGUILayout *horiz = 0;
    VrGUILayout *vert = 0;

    guimain = new VrGUI(argc, argv);
    horiz = guimain->top->horizontal();
    vert = horiz->vertical();

cin >> f; //read in desired frequency in kHz.
Freq = f*1000.0;

cerr << "Center Frequency: " << Freq << endl;

    cerr << "Input Sampling Rate: " << inputRate << endl;
    cerr << "Complex FIR decimation factor: " << CFIRdecimate << endl;
    cerr << "AM Demodulator Sampling Rate: " << demodRate << endl;
    cerr << "Real FIR decimation factor: " << RFIRdecimate << endl;
    cerr << "Audio Sampling Rate: " << audioRate << endl;

    VrSource<short> *source;
    // --> short
    source = make_GrMC4020SourceS(inputRate, MCC_CH2_EN | MCC_ALL_1V);
```

```
    // short --> short
    VrFixOffset<short,short> *offset_fixer = new VrFixOffset<short,short>();

    // build channel filter
    vector<float> channel_coeffs =
      gr_firdes::low_pass (1.0,// gain
   inputRate,// sampling rate
   5e3,  //low pass cutoff
   50e3,// width of transition band
   gr_firdes::WIN_HAMMING);

    cerr << "Number of channel_coeffs: " << channel_coeffs.size () << endl;

    // short --> VrComplex
    GrFreqXlatingFIRfilterSCF *chan_filter =
      new GrFreqXlatingFIRfilterSCF (CFIRdecimate, channel_coeffs, Freq);

    // VrComplex --> float
    GrAMDemod<float> *demod = new GrAMDemod<float>(demodGain);

    // float --> short
    double width_of_transition_band = audioRate / 32;
    vector<float> audio_coeffs =
      gr_firdes::low_pass (1.0,// gain
   demodRate,// sampling rate
   8e3,  // low-pass cutoff freq
   width_of_transition_band,
   gr_firdes::WIN_HAMMING);


    cerr << "Number of audio_coeffs: " << audio_coeffs.size () << endl;

    GrFIRfilterFSF *audio_filter = new GrFIRfilterFSF (RFIRdecimate, audio_coeffs);

    VrSink<VrComplex> *fft_sink1 = 0;
    VrSink<float> *fft_sink2 = 0;
    VrSink<short> *fft_sink3 = 0;

// whole passband spectral display
VrSink<short> *band_scope = new GrFFTAvgSink<short>(vert, 0, 80, 512);

// sink1 is channel filter output
```

```
fft_sink1 = new GrFFTSink<VrComplex>(vert, 0, 80, 512);

// sink2 is fm demod output
fft_sink2 = new GrFFTSink<float>(vert, 0, 120, 512);

// sink3 is audio output
fft_sink3 = new GrFFTSink<short>(horiz, 0, 120, 512);

    VrSink<short> *final_sink = new VrAudioSink<short>();

    //connect the modules together

NWO_CONNECT (source, band_scope);
    NWO_CONNECT (source, offset_fixer);
    NWO_CONNECT (offset_fixer, chan_filter);
    NWO_CONNECT (chan_filter, demod);
NWO_CONNECT (chan_filter, fft_sink1);
    NWO_CONNECT (demod, audio_filter);
NWO_CONNECT (demod, fft_sink2);
    NWO_CONNECT (audio_filter, final_sink);
NWO_CONNECT (audio_filter, fft_sink3);

VrMultiTask *m = new VrMultiTask ();

m->add (band_scope);
m->add (fft_sink1);
    m->add (fft_sink3);
    m->add (fft_sink2);
m->add (final_sink);

    m->start ();
    guimain->start ();

    while (1){
guimain->processEvents(10 /*ms*/);
       m->process();
    }
}
```

## 5.2 GrAMDemod.h

```
/*
 * File: GrAMDemod.h
```

```
 * Code to find the magnitude of I an Q, thus demodulating AM.
 * The math was supplied by me (Ben Gelb) but this code is based
 * on VrQuadratureDemod.h in src/pspectra/lib/vrp/.
 *
 * Date: 11/7/2003
 * (c) 2003 Ben Gelb
 */

#ifndef _GRAMDEMOD_H_
#define _GRAMDEMOD_H_

#include <VrHistoryProc.h>
#include <math.h>

template<class oType>
class GrAMDemod : public VrHistoryProc<complex,oType> {
protected:
  float gain;
public:
  virtual const char *name() { return "GrAMDemod"; }
  void setGain(float g){ gain = g; return;}
  virtual int work(VrSampleRange output, void *o[],
   VrSampleRange inputs[], void *i[]);
  virtual void initialize();
  GrAMDemod(oType g);
  GrAMDemod();
};

template<class oType> int
GrAMDemod<oType>::work(VrSampleRange output, void *ao[],
VrSampleRange inputs[], void *ai[])
{
  complex **i = (complex **)ai;
  oType **o = (oType **)ao;
  complex product,val;
  complex lastVal = *i[0]++;
  unsigned int size=output.size;

  for (; size>0; i[0]++,size--) {
    val = *i[0];

//My ONE line addition to this code
```

```
//I is thre real component, J is the imaginary component
//find the magnitude
*o[0]++=(oType)(gain * sqrt(real(val)*real(val)+imag(val)*imag(val)));

  }
  return output.size;
}

template<class oType> void
GrAMDemod<oType>::initialize()
{
  history=2;
}

template<class oType>
GrAMDemod<oType>::GrAMDemod(oType g)
  : VrHistoryProc<complex, oType>(1), gain(g)
{
}

#endif
```

## 5.3 adc_rx.py

```
from GnuRadio import *
import wx
import fftsink
import os
import eng_notation
import slider_control
from math import pi

class rx_FlowGraph (gr_FlowGraph):

def __init__ (self):
gr_FlowGraph.__init__ (self)
self.rx_freq = 1.5e6 #initial frequency
self.sample_rate = 20e6 #sampling rate

#define a filter to carve out a 20kHz (or so) "channel" centered around the rx_freq
#this filter "decimates" by a factor of 125, which means that it averages groups
#of 125 samples together, reducing the overall sampling rate. It also uses this
#process to create I and Q components so the signal can be demodulated using
```

```
#quadrature techniques.

self.channel_coeffs = gr_firdes_low_pass ( 1.0,
self.sample_rate,
20e3,
100e3,
gr_firdes.WIN_HAMMING)

self.filter = GrFreqXlatingFIRfilterSCF ( 125,
self.channel_coeffs,
self.rx_freq);

def set_rx_freq (self, freq):
self.rx_freq=freq
self.filter.setCenterFreq(freq)

def set_audio_amp (self, amp):
self.amp = amp

def set_vol (self, vol):
self.amp.setGain(vol/1000.0)

def get_rx_freq (self):
return self.rx_freq

def get_sampling_rate (self):
return self.sample_rate



def build_graph (win_parent):

fg = rx_FlowGraph ()

sample_rate = fg.get_sampling_rate()

src = make_GrMC4020SourceS (sample_rate, MCC_CH2_EN | MCC_ALL_1V) #this is the ADC card
dst, win = fftsink.makeFFTSinkC (fg, win_parent, "FFT", 512, 160e3) #define an FFT windo

demod = GrMagnitudeCF() #this function demodulates AM by finding the magnitude of the ve
# demod = GrSSBModCF(-2*pi*(300+3e3/2)/160e3,1)
```

```
#define a secondary filter that selects a single station out of the 20kHz passband
channel_coeffs = gr_firdes_low_pass (1.0, 160e3, 6e3, 1e3, gr_firdes.WIN_HAMMING)

audio_filter = GrFIRfilterCCF(5, channel_coeffs)
amp = VrAmpFF(.001) #used to keep from overdriving the speakers
fg.set_audio_amp(amp)
output = GrAudioSinkF (1, "/dev/dsp") #put stuff in the soundcard


#connect the pieces together!
#source (ADC board)->20kHz filter->6kHz filter->demodulator->output amp (attenuator)->si

fg.connect (src, fg.filter)
fg.connect (fg.filter, dst) #connects the FFT plot
fg.connect (fg.filter, audio_filter)
fg.connect (audio_filter, demod)
fg.connect (demod, amp)
fg.connect (amp, output)

return (fg, win)

# radio code really stops here. GUI the rest of the way down.

# ----------------------------------------------------------------

if __name__ == '__main__':

    def str_to_float (str):
        return float (str)

    class TestPanel (wx.Panel):
        def __init__ (self, parent, frame):
            wx.Panel.__init__ (self, parent, -1)

            self.frame = frame

            vbox = wx.BoxSizer (wx.VERTICAL)

            self.fg, fft_win = build_graph (self)
            vbox.Add (fft_win, 1, wx.EXPAND)

            hbox = wx.BoxSizer (wx.HORIZONTAL)
```

```
            hbox.Add (1, 1, 1, wx.EXPAND)
            hbox.Add (wx.StaticText (self, -1, "Set Rx Freqency: "), 0, wx.ALIGN_CENTER)
            self.tc_freq = wx.TextCtrl (self, -1, "", style=wx.TE_PROCESS_ENTER)
            hbox.Add (self.tc_freq, 0, wx.ALIGN_CENTER)
            hbox.Add (1, 1, 1, wx.EXPAND)

            self.vol = wx.Slider(self, 192, 0,0,500, wx.DefaultPosition, wx.Size(250, -1
            hbox.Add (self.vol, 0, wx.ALIGN_CENTER)
    hbox.Add(1,1,1,wx.EXPAND)

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

            self.sizer = vbox

            self.SetSizer (self.sizer)
            self.SetAutoLayout (True)
            self.sizer.Fit (self)
            wx.EVT_CLOSE (self, self.OnCloseWindow)
            wx.EVT_TEXT_ENTER (self, self.tc_freq.GetId(), self.EvtTextEnter)
            wx.EVT_COMMAND_SCROLL (self, self.vol.GetId(), self.EvtVolAdj)
    self.UpdateStatusBar ()

def UpdateStatusBar (self):
rx_freq = self.fg.get_rx_freq ()
msg = "RX Frequency: %s" % (eng_notation.num_to_str(rx_freq))
self.frame.GetStatusBar().SetStatusText (msg, 0)

def EvtVolAdj(self, event):
self.fg.set_vol(event.GetInt())

def EvtTextEnter(self, event):
            str = event.GetString ()
            self.tc_freq.Clear ()
            self.fg.set_rx_freq (eng_notation.str_to_num (str))
            self.UpdateStatusBar ()

def OnCloseWindow (self, event):
self.fg.stop ()
self.Destroy ()
```

```
    class MainFrame (wx.Frame):
        def __init__ (self):
            wx.Frame.__init__(self, None, -1, "MC4020 HF Receiver")

            self.CreateStatusBar ()
            mainmenu = wx.MenuBar ()
            menu = wx.Menu ()
            menu.Append (200, 'E&xit', 'Get outta here!')
            mainmenu.Append (menu, "&File")
            self.SetMenuBar (mainmenu)
            wx.EVT_MENU (self, 200, self.OnExit)
            self.panel = TestPanel (self, self)
            wx.EVT_CLOSE (self, self.OnCloseWindow)

        def OnCloseWindow (self, event):
            self.Destroy ()

        def OnExit (self, event):
            self.Close (True)


    class TestApp (wx.App):

        def OnInit(self):
            print "TestApp: pid = ", os.getpid ()
            frame = MainFrame ()
            frame.Show (True)
            self.SetTopWindow (frame)
            print "FlowGraph: ", frame.panel.fg
            frame.panel.fg.start ()
            return True

    app = TestApp (0)
    app.MainLoop ()


# --------------------------------------------------------------------
```

# 6 References


Bourke, Paul. *Discrete Fourier Transform.* <http://astronomy.swin.edu.au/ pbourke/analysis/dft/>

*A Digital AM FM SSB Demodulator.* <http://thierry.leconte.chez.tiscali.fr/demod.html>

Ford, Steve. *Digital Radio Mondiale.* QST, Oct 2003. American Radio Relay League.

*Fourier and the Frequency Domain.* <http://www.spd.eee.strath.ac.uk/ interact/fourier/fourier.html>

Frohne, Rob. *A High-Performance, Single Signal, Direct Conversion Receiver with DSP Filtering.* <http://www.wwc.edu/f̃rohro/R2_DSP/Motorola_DSP_Receiver.html>

GNU Radio Project. *<http://www.gnuradio.org/>*

Hoffman, Forrest. *An Introduction to Fourier Theory.* <http://aurora.phys.utk.edu/ forrest/papers/fourier/>

*KR7A SSTV Demodulator.* <http://www.agurk.dk/bjarke/DSP/DSP%20FM%20Demodulator%20for%20SSTV.htm>

Lyons, Richard. *Quadrature Signals: Complex, But Not Complicated.* <http://www.dspguru.com/info/tutor/quadsig2.htm>

Panter, Philip F.. *Modulation, Noise, and Spectral Analysis.* 1965. McGraw-Hill Inc: New York.

Pohler, Ken C.. *Principles of Digital Audio.* 3rd Ed. 1995. McGraw-Hill Inc: New York.

Youngblood, Gerald. *A Software Defined Radio for the Masses.* QEX, Jul 2002. The American Radio Relay League. *<http://www.flex-radio.com/articles_files/index.htm>*