ECE 4670 Spring 2014 Lab 6 Software Defined Radio and the RTL-SDR USB Dongle

1 Introduction

Software defined radio (SDR) is an exciting merger of digital signal processing and wideband radio hardware [1]. The term SDR came into more common usage in 1992 by Dr.Joe Mitola, but actually had its beginnings back in 1984 at E-Systems. See the Wikipedia footnote mentioned above for more details on the history of SDR. The ideal SDR receiver consists of an antenna connected to an analog-to-digital converter (ADC) followed by a digital signal processing system (DSPS) to extract the signal of interest. The ideal SDR transmitter consists again of a DSPS where information you wish to send is input followed by a digital-to-analog converter (DAC) which directly interfaces with an antenna. Note I said *ideal*. Placing the ADC and DAC right at the antenna proves to be a challenge.

The basic elements of a practical SDR transceiver are shown in Figure 1. In this block you see both an ADC and a DAC converter, so this system indeed represents a transceiver. Note that the

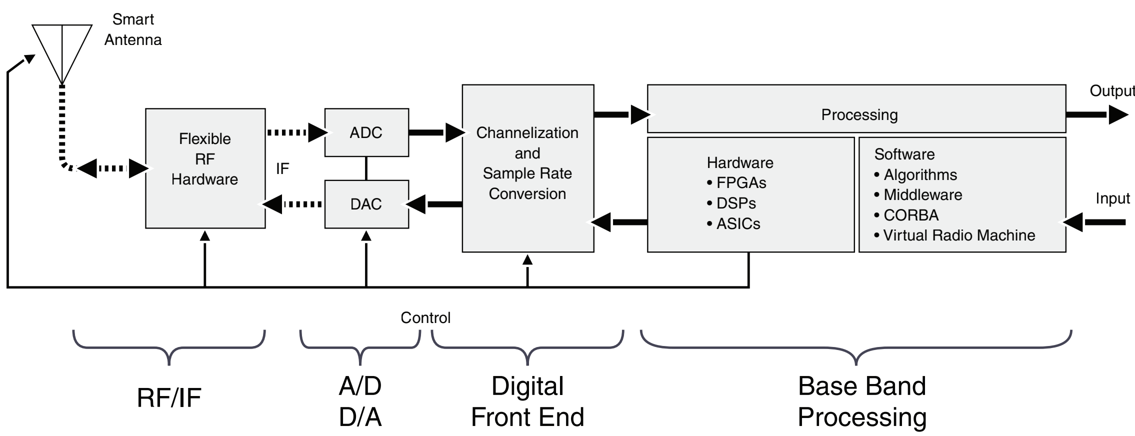


Figure 1: The SDR transceiver concept in block diagram form [1].

ADC and DAC are not placed directly at the antenna. Practical RF/microwave circuit design make it necessary to include the flexible hardware block you see in Figure 1.

The SDR hardware platform chosen for this lab is the RTL-SDR[1]. The is a very popular platform with a very active user community. The form factor of the RTL-SDR is similar to a large USB memory stick. It is referred to as the *RTL-SDR USB dongle*. In the experiments that follow you will write software for demodulating a variety of waveform types. I have written the lab to allow you to use either Python or MATLAB as the algorithm development tool. In the future iteration of this lab, I hope to be able to implement a streaming solution, and thus have you write real-time

[1] `http://rtlsdr.org/`

code. For now, I have you capture 5 to 10 seconds of I/Q signal samples at 2.4 Msps, and then work with those samples after the capture is complete. Although the processing is not done in real-time, you will be able to listen to the results of your work by playing signals back via the PC sound system. For the case of digital modulation, you will perform error checking on the recovered bits using the fact that the transmitted bits are known in advance. To make this possible you will make use of the m-sequence generator developed in Lab 2. Your first exposure to the hardware will have you test drive the RTL-SDR dongle using the software app SDR#[2].

Before jumping into some hands-on work, I want to introduce some of the details of the RTL-SDR and develop a behavioral level model, that explains in a mathematical sense, the inner workings of the device. The lab work that follows will consist of first working with SDR#, then developing MATLAB or Python code to demodulate FM and FM-stereo (multiplexed FM), and finally demodulating frequency shift keying (FSK). The FSK demodulator will include a bit synchronizer so to account for the fact that the transmit and receiver clocks are asynchronous.

2 Overview of the RTL-SDR USB Dongle

The RTL-SDR dongle contains two primary chips: (1) the Raphael Micro R820T radio tuner and the Realtek RTL2832U which contains an 8-bit ADC and USB *data pump*. The original intent of this design was for use as a digital video broadcasting (DVB) receiver. If you look on Amazon you will find that most variations of the RTL-SDR are sold with a TV remote and some DVB software.

The basic chip configuration is depicted in the block diagram of Figure 2. The tuner chip serves as the radio frequency (RF) front-end for the SDR. Following a miniature coax connector for the antenna is a low noise amplifier (LNA) providing a noise figure (NF) of about 3.5 dB. The advertised tuning range of the R820T is 24 MHz to 1850 MHz. Not shown in Figure 2 are the inputs to set the sampling rate f_s and the tuner RF gain. These two SDR attributes will be discussed more later. A simplified view of the R820T tuner internals is shown in Figure 3. The

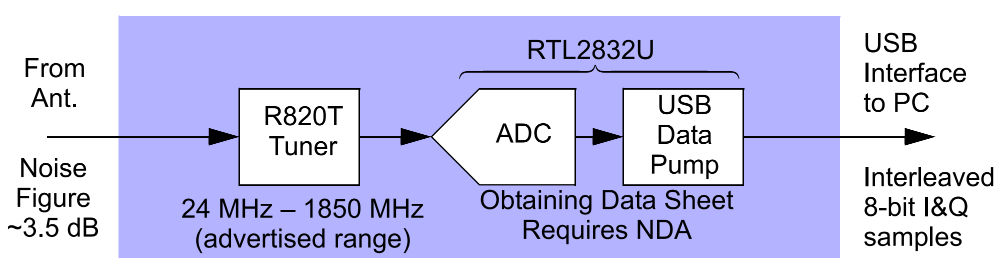


Figure 2: The RTL-SDR high level block diagram.

complete data sheet is also available on the internet[3]. Notice from the data sheet that the actual RF tuning range is listed as being only 42 to 1002 MHz. Information found on the internet confirms that the tested range is somewhere between the two extremes. A frequency synthesizer inside the R820T generates a local oscillator (LO) signal which is responsible for down converting the

[2]`http//sdrsharp.com`

[3]`http://superkuh.com/rtlsdr.html`

received RF to an intermediate frequency (IF). The tuning resolution is 1 Hz, or so it seems from the information available on the internet. Gain control is also provided, both at the LNA and at the output via a variable gain amplifier (VGA). The term automatic gain control (AGC), seen in Figure 3, refere to the use of a signal strength sensing circuit/algorithm to feedback a control signal to the gain control circuitry of an RF receiver. In this case that is the VGA and perhaps the LNA.

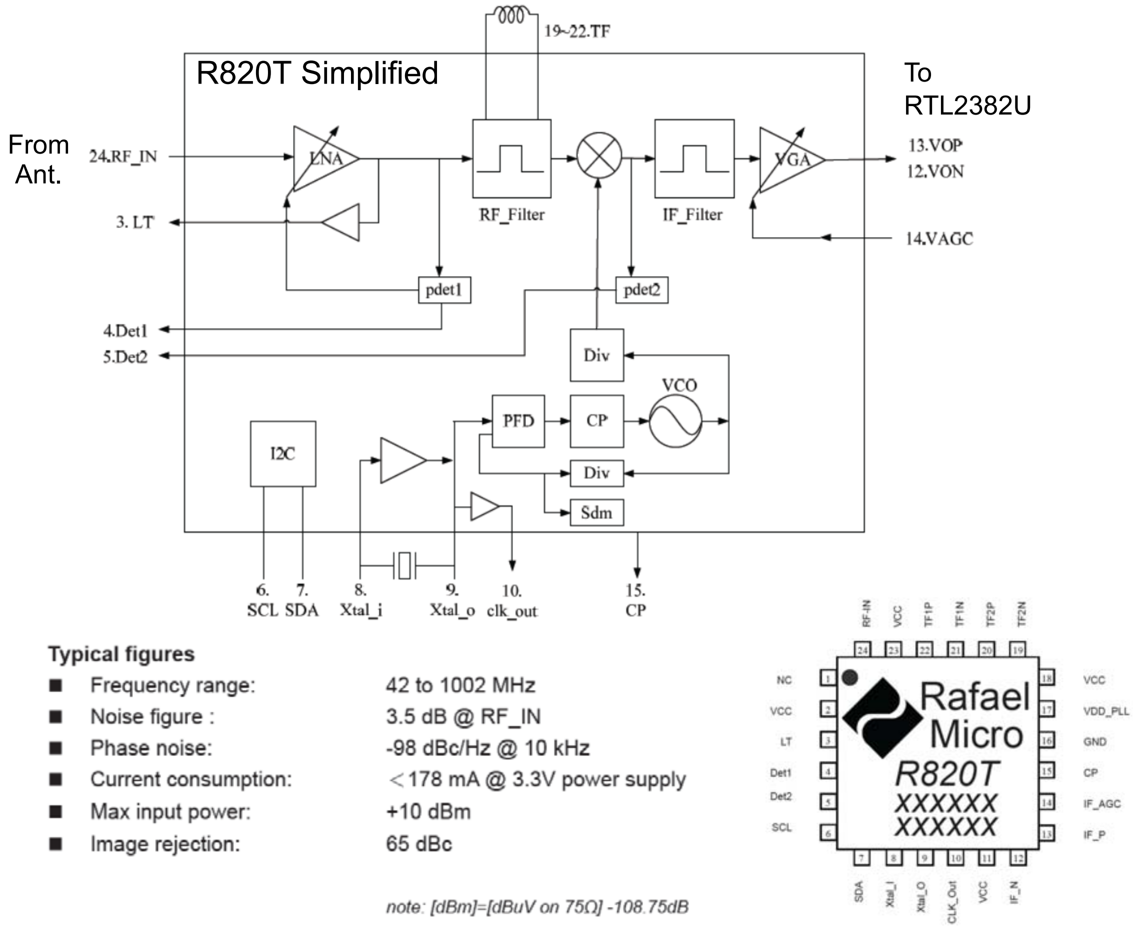


Figure 3: Simplified block diagram of the R820T tuner which is used in the RTL-SDR.

Information on the Realtek chip is not available, unless you have an nondisclosure agreement (NDA) in place from Realtek. This leaves the rest of the SDR open to speculation. It is the RTL2832U where the digital signal processing (DSP) takes place, which includes additional filtering and down sampling of the IF signal delivered by the R820T. The ADC produces 8-bit real/inphase (I) and imaginary/quadrature (Q) interleaved sample values, in an unsigned format. When you get your hands on the sample they are finally converted to signed 8-bit values and parallel I and Q streams. I will describe this further when I discuss the behavioral level model of the RTL-SDR shortly. The RTL2832U also contains a USB interface that sends samples to the PC.

For the curious, yes there is a tear-down page on the internet which describes the internal configuration of the RTL dongle. This photograph is shown in Figure 4 and can be found on the internet at the same location as the R820T data sheet[4]. Notice that the crystal is marked as having frequency 28.8 MHz. This crystal is responsible for setting the frequency accuracy of the tuner. You can find some discussion on the internet of hacking this crystal to obtain higher frequency accuracy and better temperature stability.

Figure 4: A photograph of the RTL-SDR opened up [5].

To better understand the functionality of the RTL-SDR consider the behavioral level model shown in Figure 5. A model of this type allow you to focus on the signal processing details of greatest interest. In this case I am concerned with a mathematical representation of the signal flow from the input to the output. The model shown is linear if you ignore the 8-bit ADC, which is denoted by the quantizer function $Q[\,]$.

A simple model for the input signal, $r(t)$, is that it consists of the desired radio signal $s(t)$ plus background noise $n(t)$ due to the receiver front-end in combination with the antenna. For details on receiver noise modeling see Appendix A of Ziemer and Tranter [2]. In reality, since the receiver has a very wide bandwidth, there are multiple signals present at the front end. This in fact is one the big challenges of SDRs in general. From a modeling standpoint, however, I initially assume just one signal is present.

You can apply superposition to study the impact of multiple signals. The gain control ot the front end serves to keep the signal processing linear, at the expense of dynamic range to receive weak signals. Adding a bandpass filter in front of the LNA can also be considered as a means to reject strong unwanted signals lying out of band.

Following the LNA is a multipler having LO input $e^{-j2\pi f_c t}$. This is a behavioral representation of negative frequency translation by f_c Hz. Recall the Fourier transform theorem

$$x(t)e^{j2\pi f_0 t} \stackrel{\mathcal{F}}{\Longleftrightarrow} X(f - f_0) \tag{1}$$

[4] http://superkuh.com/rtlsdr.html

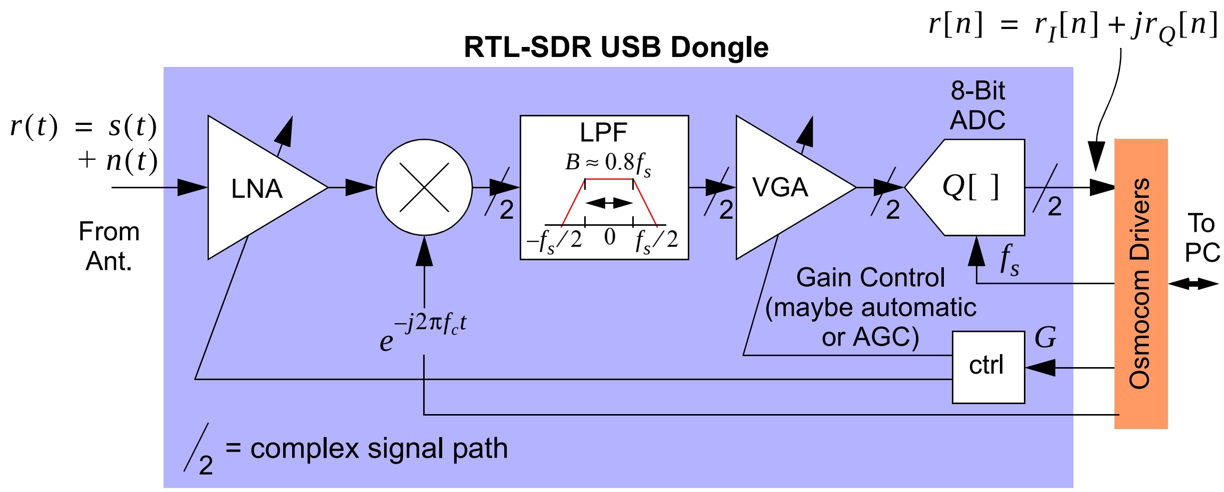


Figure 5: A behavioral level model of the RTL-SDR.

and consider the spectrum sketches of Figure 6. By choosing $f_0 = -f_c$, the frequency translation theorem shifts the input spectrum to the left by f_c Hz. A signal of interest centered at f_c will now be located at 0 Hz following multiplication by $e^{-j2\pi f_c t}$. From a behavioral level standpoint I

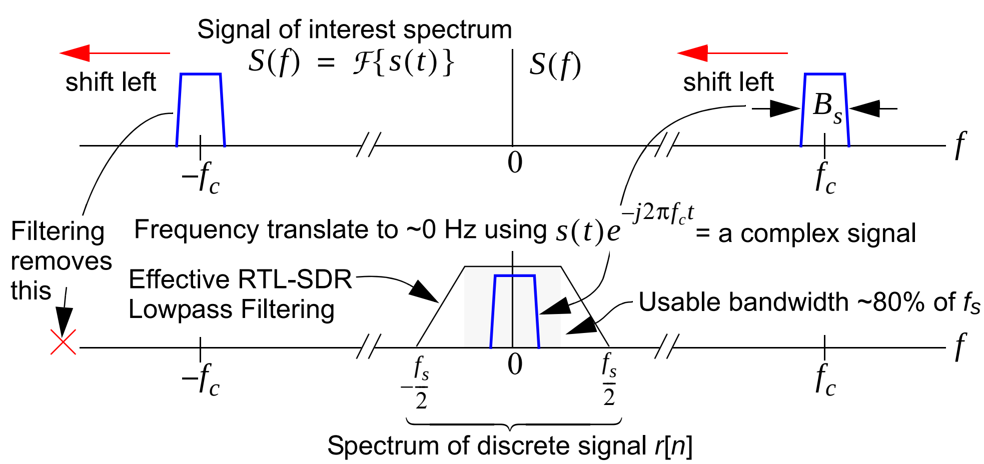


Figure 6: A frequency domain view of the RTL-SDR.

assume that the output of the multiplier, now a complex signal (why?), is passed through a lowpass shaping filter (LPF) that is a function of the sampling rate entered into the RTL-SDR. From the sampling theorem the input signal must be band limited prior to $f_s/2$ Hz. The usable bandwidth is 80% of f_s since a realizable filter requires a transition band to go from the passband gain to the stopband gain. All signal processing following the multiplier is complex due to the fact that $e^{j\theta} = \cos\theta + j\sin\theta$. In the behavioral level model a lowpass filter is required for both the real and imaginary parts. In the actual hardware this filtering is split between the R820T, where it is a single real bandpass filter, and the RTL2382U, where I/Q digital filtering is likely employed.

In any case, it is work noting that since the down converted signal is now complex, the entire spectrum from $-f_s/2$ to $+f_s/2$ is unique. This is in contrast to frequency translation using cos( ) and the corresponding modulation theorem from Fourier transforms, which says

$$x(t)\cos(2\pi f_0 t) \overset{\mathcal{F}}{\Longleftrightarrow} \frac{1}{2}\big[X(f-f_0)+X(f+f_0)\big]. \tag{2}$$

In this case, assuming $x(t)$ is a real signal, the spectrum $X(f)$ has magnitude that is even about $f=0$, so $|X(f-f_0)+X(f+f_0)|$ is even about $f=0$.

Up to the quantizer, $Q[\,]$, the complex signal $r[n]$ can be written as

$$r[n] = \underbrace{G}_{\text{AGC}} \cdot \text{LP}\left\{r(t)e^{-j2\pi f_c t}\right\}\Big|_{t=nT=n/f_s} \tag{3}$$

where LP represent the LPF filter action. In the frequency domain I can write

$$R\big(e^{j2\pi f/f_s}\big) \simeq G\cdot f_s\cdot R(f+f_c)\cdot H_{\text{LP}}(f),\ \ f_s/2 \le f \le f_s/2. \tag{4}$$

I have further assumed that $H_{\text{LP}}(f)=0$ for $|f|>f_s/2$.

The final stage is the quantizer. Knowing that only 8-bits are available to represents the real and imaginary parts, means that significant quantization noise is generated. Using concepts found in [3], I can approximate the quantization noise impact on the noise free $r[n]$ as an additive noise process. The signal out of the quantizer is the sum signal

$$r[n]+e[n] = \big(r_I[n]+jr_Q[n]\big)+\big(e_I[n]+je_Q[n]\big), \tag{5}$$

where the signals $e_I[n]$ and $e_Q[n]$ are noise signals approximately uniformly distributed over one quantization interval of $Q[\,]$. The signal-to-quantization ratio of the $r[n]$ signal is

$$\text{SNR}_q = 6.02\cdot B+10.8-20\log_{10}\left(\frac{R_{\max}}{\sigma_r}\right) \simeq 6B+1.26\text{ dB}. \tag{6}$$

where $\pm R_{\max}$ is quantizer dynamic range, σ_r^2 is the variance or power of $r[n]$, and B is the number of bits used to quantize the magnitude of $r[n]$. Here $B=8-1=7$ as one bit is needed to represent the sign. The final form of SNR_Q assumes that $\pm R_{\max}=3\sigma_r$. Plugging in numbers I arrive at

$$\text{SNR}_Q \approx 43.4\text{ dB} \tag{7}$$

for the 8-bit quantizer of the RTL-SDR. This may seem low, but once the demodulation algorithms are implemented, the signal will undergo filtering and downsampling, which will increase the signal dynamic range and hence increase the effective SNR_Q value.

To actually talk to the RTL-SDR and capture the complex $r[n]$ samples, you need software drivers. The source for these drivers is Osmocom[5]. On a Windows 64-bit PC the driver are available prebuild from the Osmocom site. The file collection is shown in Figure 7. When you experiment with demodulation algorithms, you will need to have these files in the same directory when you develop either MATLAB or Python code. The ZIP package for Lab6 includes these files for 64-bit windows. The driver for the USB dongle itself is managed by the Windows application Zadig[6].

[5] http://sdr.osmocom.org/trac/wiki/rtl-sdr

[6] http://rtlsdr.org/softwarewindows

Name	Type	Modified	Size
rtlsdr_static.lib	LIB File	11/5/2013 2:11 PM	148,136
rtlsdr.lib	LIB File	11/5/2013 2:11 PM	9,370
rtlsdr.dll	Application e...	11/5/2013 2:11 PM	48,128
rtl_test.exe	Application	11/5/2013 2:11 PM	15,360
rtl_tcp.exe	Application	11/5/2013 2:11 PM	19,456
rtl_sdr.exe	Application	11/5/2013 2:11 PM	15,360
rtl_power.exe	Application	11/5/2013 2:11 PM	25,600
rtl_fm.exe	Application	11/5/2013 2:11 PM	25,600
rtl_eeprom.exe	Application	11/5/2013 2:11 PM	17,408
rtl_adsb.exe	Application	11/5/2013 2:11 PM	18,432
pthreadVC2-w64.dll	Application e...	10/26/2012 5:13 PM	87,040
libusb-1.0.dll	Application e...	4/12/2013 1:04 PM	87,552

Figure 7: The osmocom interface software library (windows 64-bit version).

3 Using the RTL-SDR Dongle with SDR#

Now its time to get started working with the RTL-DSR dongle. An easy on-ramp is provided by using a fully build SDR receiver app. On Windows the most popular of these apps is SDR#[7]. Your lab instructor will help you configure Zadig (if needed) and help you get SDR# up and connected to the RTL-SDR dongle.

Once SDR# is properly connected you will see a screen similar to that shown in Figure 8. You immediately see that there are many controls and two graphical displays. The upper graphical display emulates a spectrum analyzer and the lower display is a *waterfall* display which is a spectrogram of the received signal(s). Sliders on the side of the plots allow you to make further display adjustments.On the left is where you enter the frequency tuning information and select what type of demodulation algorithm to have SDR# invoke. The play button sets everything into real-time motion.

It is important that you have the proper SDR device selected before you hit play. Figure 10 shows you how to select the proper dongle and also covers how to configure the RF gain of the device.

The two frequency displays can be confusing at first. The input given to the text box labeled Center sets f_c as described in the behavioral level model of Figure 5. The valued entered into the text box labeled Frequency is the center frequency including an offset to tune above or below f_c by as much as $\pm f_s/2$.The range of values displayed along the frequency axis is $[f_c - f_s/2, f_c + f_s/2]$. The true frequency axis corresponding to the output signal is just $[-f_s/2, f_s/2]$.

You may notice that there is always a spectrum spike at the center frequency f_c. This is due to a small dc bias present in the ADC outputs. Take a look at Figure 10 to see what I mean. If the spike due to ADC bias is interfering with reception, you can move f_c over slightly and then tune in your desired signal using the Frequency control. With this technique you are letting SDR# perform

[7] `http://www.rtl-sdr.com/big-list-rtl-sdr-supported-software/`

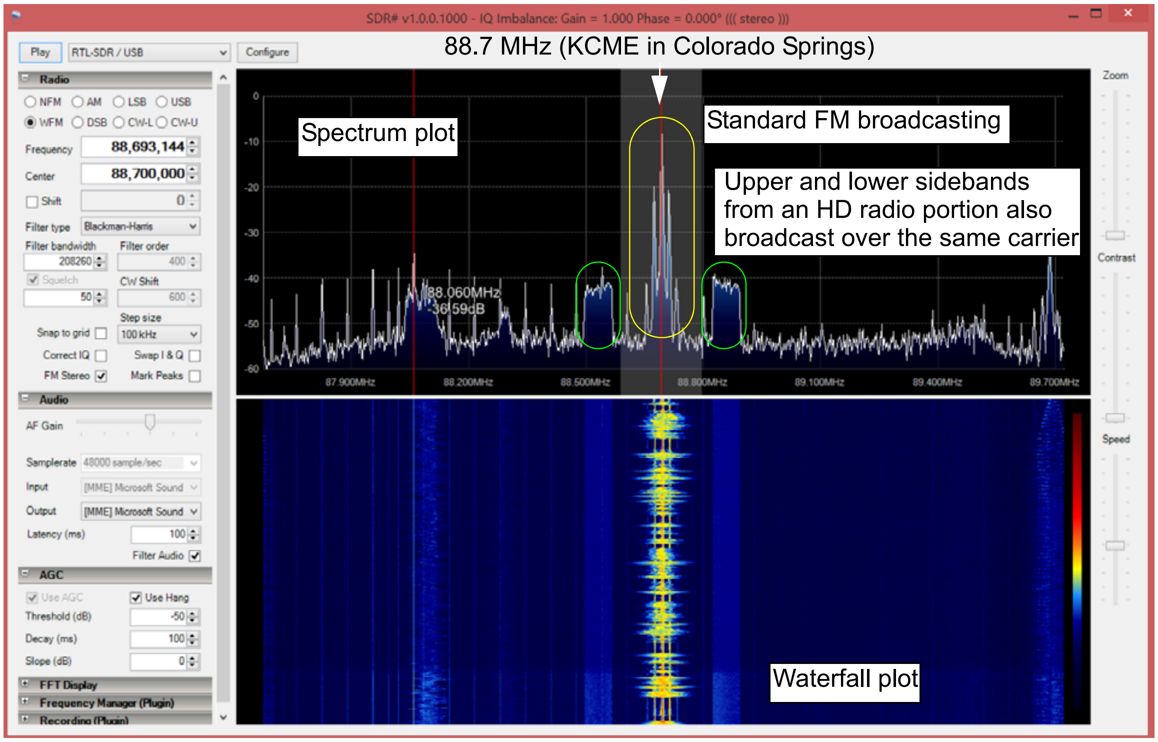


Figure 8: The SDR# GUI.

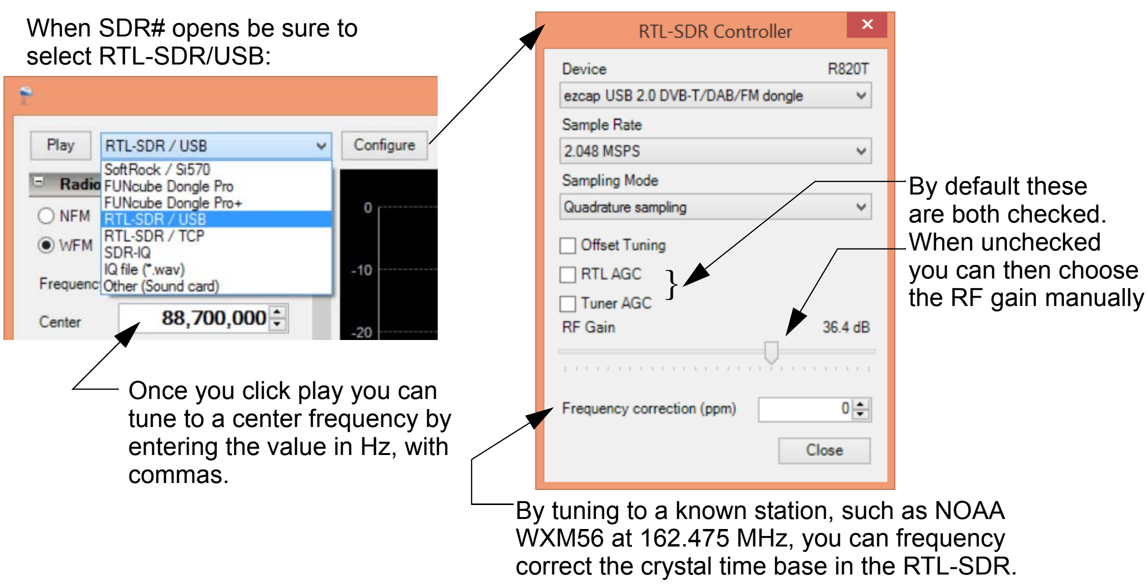


Figure 9: Setting up SDR#.

additional frequency translation on $r[n]$. In the software you find that this is actually what you are doing when you make mouse clicks on either the spectrum or waterfall displays. When you click and drag on the frequency axis itself you then change the center frequency interactively. Further-

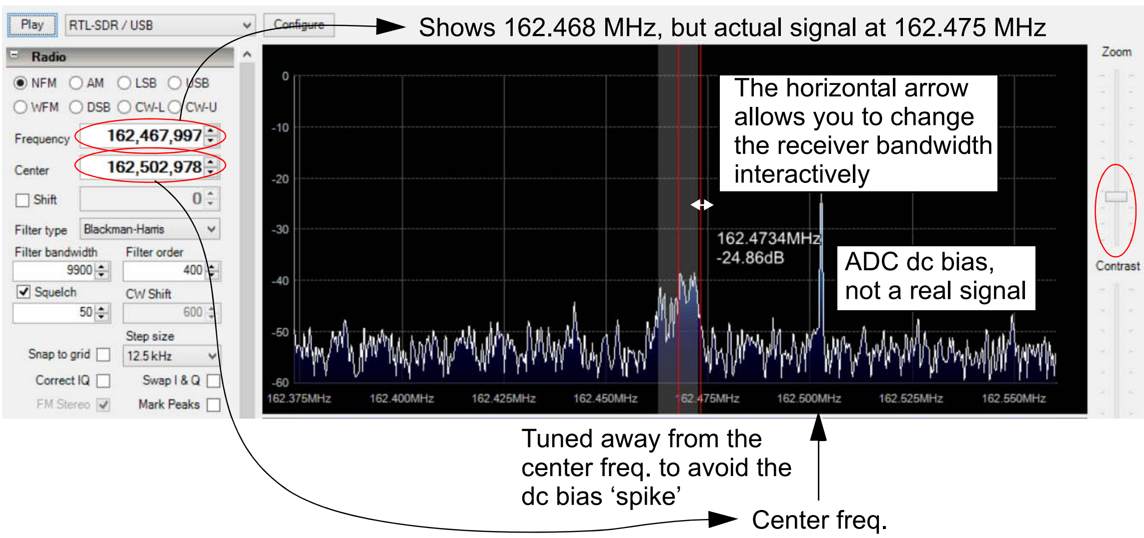


Figure 10: Dealing with dc bias at the center frequency, f_c, and taking note of frequency error.

more, you can change the filter bandwidth used by the demodulation algorithms by dragging with the mouse when you see a horizontal double arrow as you approach the gray shaded region near the tuned frequency. This is very nice!

Additionally, you may notice once you start tuning to known frequencies that they are not quite located where you expect to find them. The crystal oscillator used in the RTL-SDR is not perfect. It is close. Figure 9 points out where you can make a calibration adjustment to correct for the crystal frequency errors.

3.1 Laboratory Exercises

1. Now its time to really do something. Tune in the NOAA weather station WXM56, found at 162.475 MHz. From Lab 4 you know that this is a narrow band FM signal. In SDR# under radio you need to click NFM for narrowband FM demodulation. You may need to try the offset frequency technique described in Figure 10 to avoid interference from the dc spike.

   (a) Play the demodulated audio out through your PC speakers and demonstrate this to your instructor.

   (b) Make note of the receiver Filter Bandwidth that gives you the best reception. Note this is the bandwidth used by the FM demodulator, not $f_s/2$. See Figure 10 for information on how to change the bandwidth interactively using the mouse.

   (c) Calibrate you RTL-SDR dongle to this know signal at 162.475 MHz using the Frequency correction text box described in the lower right of Figure 9. Record this value. Note if you change dongles at some point, this correction factor will likely change. The frequency error is also temperature dependent, so be sure your dongle is warm before calibrating.

2. Tune in and listen to several (at least three) Broadcast FM stations. Recall broadcast FM runs from 88 - 108 MHz. You will need to switch the demodulator to wideband FM (WFM in SDR#). Start with KCME which is at 88.7 MHz. This is the station I have used in my examples.
   (a) For each station you tune into note the frequency (recall channel spacing is every 200 kHz but at odd multiples of 100 kHz, e.g., 88.7 MHz).
   (b) Make note of the program type, is the signal stereo, our the large rectangular sidebands present indicating an HD radio broadcast.
   (c) Speaking of HD radio take a look at what Wikipedia has to say about HD radio. Discuss in your report and confirm that what you see of the FM radio spectra that HD radio is what you are seeing. You will get into stereo multiplexing and the radio data service (RDS) in a later part of this lab.
3. Set up a custom transmitter at 70 MHz using the Agilent 33250A. For details see Figure 11. Experiment with different modulation types. You will have to choose your carrier frequency near 70 MHz, yet avoid the transmission frequencies used by neighboring lab benches. Interference from your neighbors is real, and maybe will make you appreciate why the Federal Communications Commission (FCC) exists.

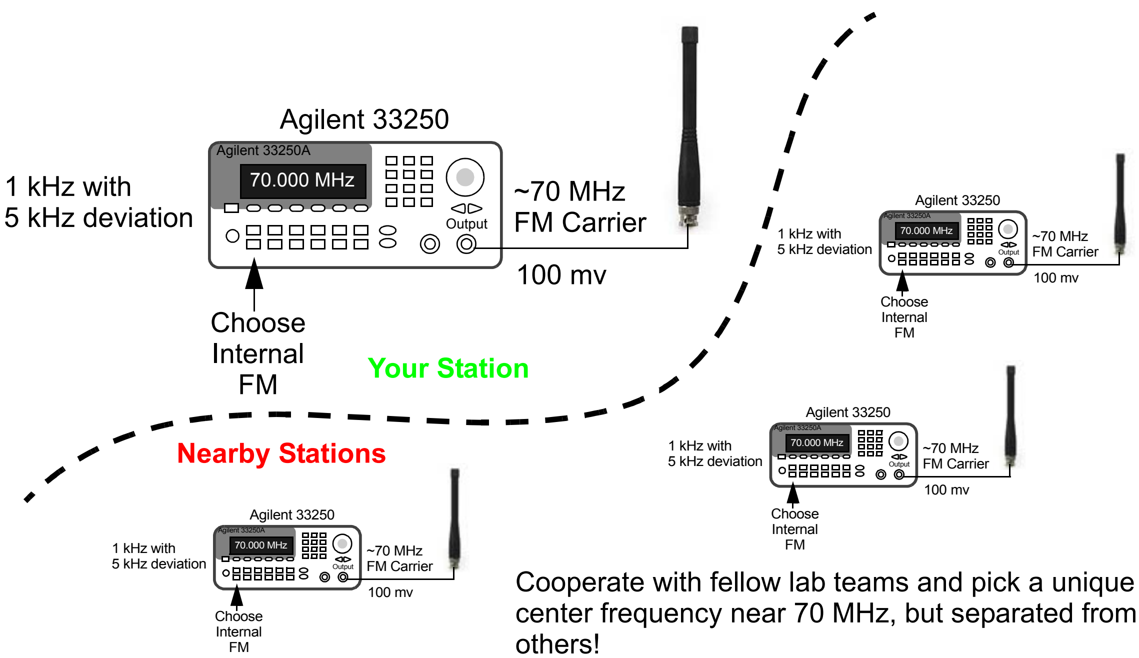


Figure 11: Transmitting an FM test signal using the Agilent 33250.

   (a) Verify that you can receive and hear the 1 kHz FM modulation tone.
   (b) Verify that as the carrier amplitude (initially 100mv) is reduced the signal gets noisy and eventually fades away. Try turning the AGC on and off (see Figure 9) and make manual gain adjustments to compensate for changes in your transmitted signal level.

(c) Switch from FM to AM modulation with your transmitter. Verify that you can again demodulate AM with SDR# by switching the receiver mode to AM. Which do you prefer, AM or FM?

4 Writing Your Own Demodulator Algorithms

The lab work now turns to writing code in Python or MATLAB to implement your own demodulator structures. The general demodulator block diagram used in the remainder of this lab is shown in Figure 12. I now refer to the signal output from the RTL-SDR as $x[n]$ rather than $r[n]$ as was done

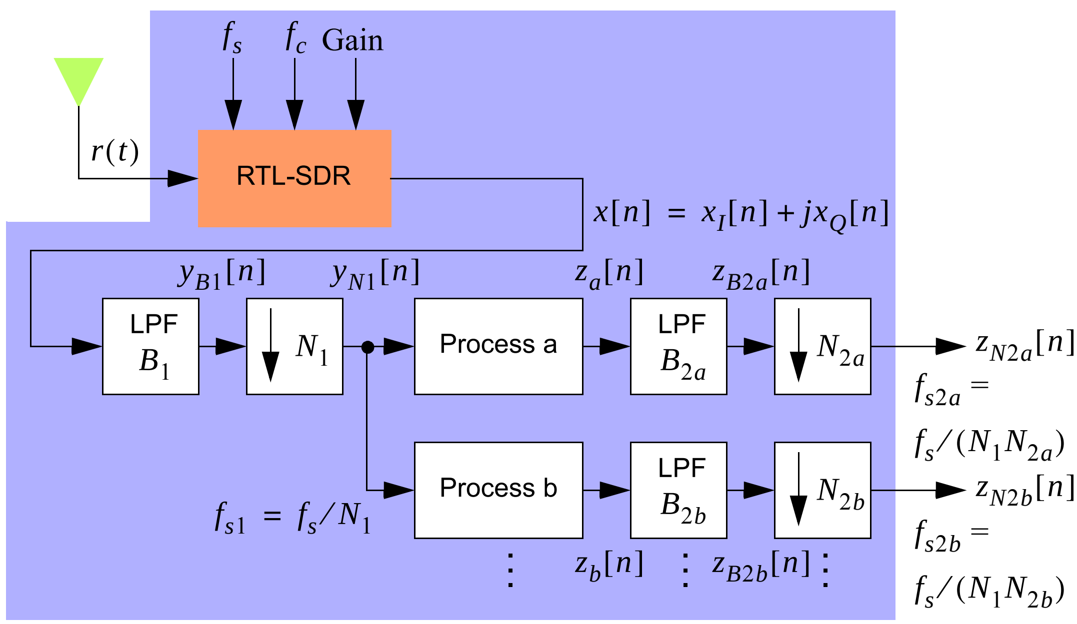


Figure 12: The general demodulator structure used in this lab.

in Figure 5. This done so I can have an *alphabetical* flow of signal names, i.e., x, y, z. No matter the label, $r[n]$ or now $x[n]$, contains at least one signal, receiver front-end noise, and quantization noise. Signals not of interest can also be thought of as interference.

The signal flow makes use of *multirate* signal processing techniques, in particular *decimation*. From your understanding of sampling theory aliasing can be avoided so long as the sampling rate is greater than twice the highest frequency in the signal being sampled. When you lowpass filter a signal that is already in the discrete time domain, the bandwidth reduction may mean that the effective sampling is greater than needed. The downsamping block (arrow pointing down followed by an integer factor) means keep every Mth sample and discard the rest. The combination of the lowpass filter follwed by the downsampler forms a decimator. As described here, decimation by M. If the input sampling rate is f_s the output sampling rate becomes f_s/M.

The need for multiple decimation operations makes sense if the *Process* blocks reduce the signal bandwidth. By reducing the sampling rate as quickly as possible in the demodulator you reduce the burden on the computation engine.

4.1 A Design Example

Consider the following scenario and deduce workable values for N_1 and N_{2a} (the upper path).

- Suppose the RTL-SDR sample rate, $f_s = 2,400,000$ sps (2.4 Msps)
- Secondly, suppose the signal $x[n]$ is centered on 0 Hz through proper choice of f_c, and requires a two-sided bandwidth of at least 200 kHz (further assuming, the one-sided bandwidth is 100 kHz).
- Thirdly, the output sampling rate is required to be 48 ksps.

With $f_s = 2.4$ Msps the Nyquist or folding frequency is $2.4/2 = 1.2$ MHz. With decimation you need the decimated sampling rate to be at least 100 kHz. The maximum allowable decimation factor N_1 is 1200/100 = 12. The sample rate into the second decimator is f_s/N_1. The third bullet implies that $f_{s2a} = f_s/(N_1 N_{2a}) = 48$ ksps. With $f_s = 2400$ ksps it follows that $N_1 N_2 = 2400/48 = 50$. In summary to make the design work you need $N_1 \leq 12$ and $N_1 N_{2a} = 50$. Clearly $N_1 = 12$ is not acceptable, because then you cannot find an integer N_{2a} that will make $f_{s2a} = 48$ ksps. A workable solution is to let $N_1 = 10 \leq 12$, then $N_{2a} = 5$. The intermediate sampling rate $f_{s1} = 240$ ksps.

4.2 MATLAB Coding

To prepare you for upcoming the coding needs, this subsection discusses the special MATLAB resources you will be using. The native MATLAB base allows vector/matrix programming. SDR algorithms rely on digital signal processing (DSP) mathematics, hence you will be using the DSP Toolbox™ as well as other custom functions to implement demodulators. A summary of the key functions needed throughout the rest of this lab are described in Figure 13. This figure also tells where the functions reside.

All of the specialized `.m` files identified in Figure 13 can be found in the Lab 6 ZIP package. A few examples of their use are scattered throughout the following pages. Specific information on how to capture I/Q samples from the RTL-SDR into MATLAB can be found in Appendix A. To play demodulated signals through the PC sound system you can use the MATLAB function `sound(z,fs)`, where `z` is a matrix containing one or columns of signal samples. One column for mono sound and two columns for stereo sound. To avoid distortion the values in `z` must be bounded on $(-1, 1)$. You can use `max(abs(z))` to normalize.

4.3 Python Coding

An exciting alternative to using MATLAB is open source Python, and in particular *IPython*[8]. To install IPython and a scientific configuration of Python follow the link `http://ipython.org/install.html`. I have used both Anaconda and Enthough Canopy. Both have powerful free versions. On the lab computers you will find Canopy installed. In my recent book Signals and Systems for Dummies [4] I chose to use Python (IPython) in place of MATLAB .

[8] http://ipython.org/

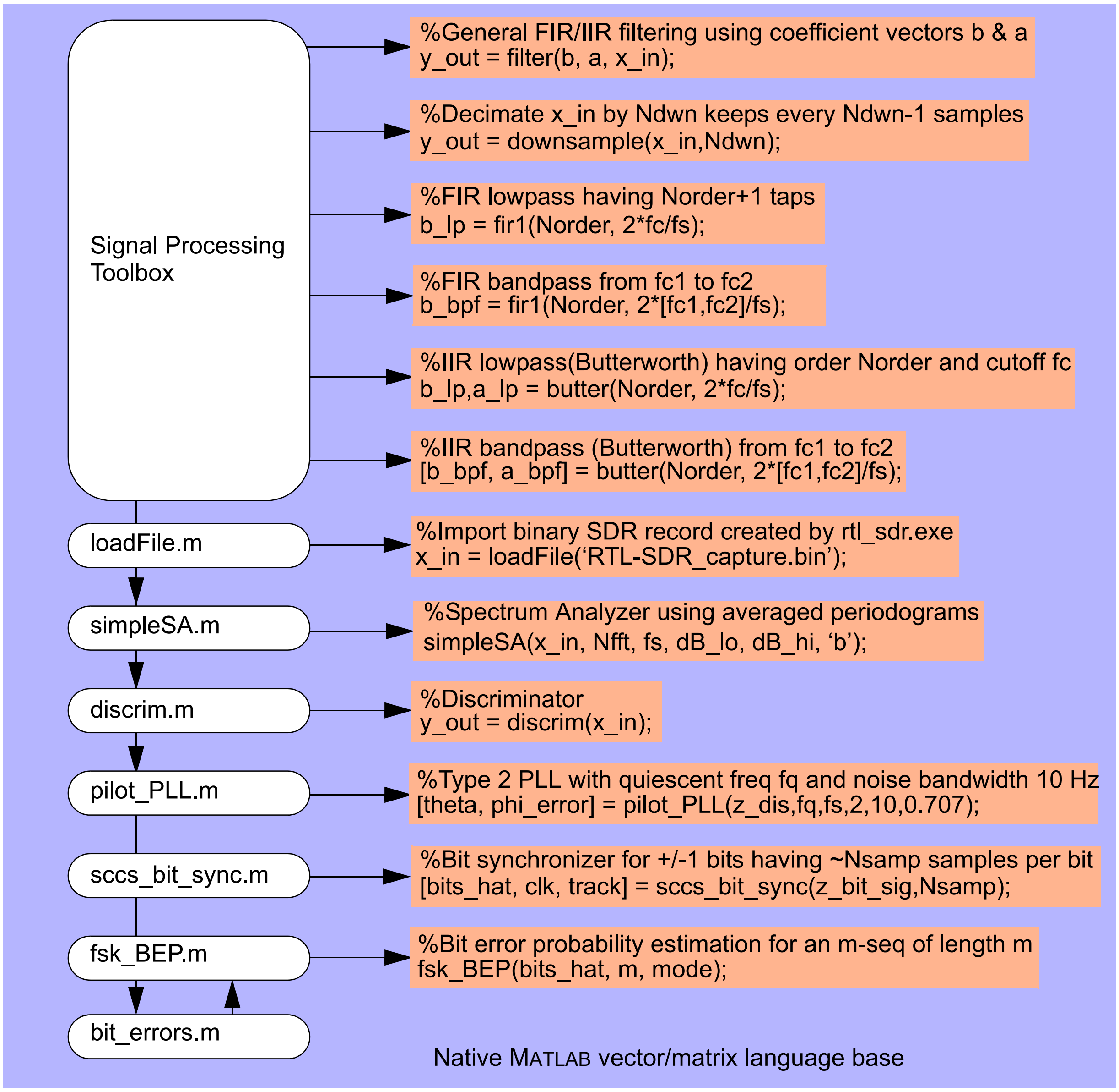


Figure 13: Overview of MATLAB functions used to develop demodulator algorithms for this lab.

Since Python is a general purpose object-oriented programming language, it relies on *packages* to give it a vector/matrix capability. To create a Python environment similar to MATLAB all you need to do is `import` the packages `numpy` and `matplotlib`. When using the IPython command shell the automatic launch configuration starts in `pylab`, which gives you a vector/matrix and graphics base. Then all you need to add via the `import` command is `scipy.signal`, which brings in the signal processing subpackage that is just one part of the complete `scipy` (scientific Python) package.

A summary of the key functions needed throughout the rest of this lab for Python users, is presented in Figure 14. This figure also tells you how to access these functions. For example if you import just `lab6` you will have to access the corresponding functions from the *namespace* lab6 using the dot notation lab6.pilot_PLL(). The code module lab6.py can be found in the Lab 6 ZIP package.

In Python integration with the Osmocom drivers of Figure 7 is obtained using the Python package pyrtlsdr[9]. To see if the package is already installed type pip freeze in the terminal (windows cmd.exe). If you find it in the list, you can install it by typing pip install pyrtlsdr. For help on using the package installer pip, type pip help at the command prompt.

The interface for capturing samples from the RTL-SDR is object oriented. The function x = capture(T) which captures T s of I/Q samples, is the following:

```
import rtlsdr
import numpy as np

def capture(Tc,fc=88.7e6,fs=2.4e6,gain=40):
    # Setup SDR
    sdr = rtlsdr.RtlSdr() #create an RtlSdr object
    #sdr.get_tuner_type()
    sdr.sample_rate = fs
    sdr.center_freq = fc
    #sdr.gain = 'auto' #uncomment to use AGC
    sdr.gain = 40
    # Capture samples
    Nc = np.ceil(Tc*fs)
    x = sdr.read_samples(Nc)
    sdr.close()
    return x
```

Note default values for the center frequency, f_c, the sampling rate, f_s, and the front end gain, G, but you can override them by providing your own values. If you wish to archive a capture the function complex2wav(filename,rate,x) will store the samples in a .wav file. The can be restored using fs,x = wav2complex(filename).

To play demodulated signals through the PC sound system, the easiest approach is to create a *wav* file and use the PC media player to play the file. The function to_wav_stereo(filename, rate,x_l,x_r=None) is available from the Python module lab6.py. To create a mono sound file only supply the x_l ndarray. To avoid distortion the values in z must be bounded on $(-1, 1)$. You can use max(abs(z)) to normalize. A nice means of playing back audio files is available in the Notebook described in Appendix B. You will need to make sure you have IPython 2.0 or greater installed.

Lastly, to plot a long signal vector, the function strips(x, Ns) is available in the module digitalcomm.py. This function is imported into lab6.py. This function is similar to the MATLAB strips() function.

[9] https://github.com/roger-/pyrtlsdr

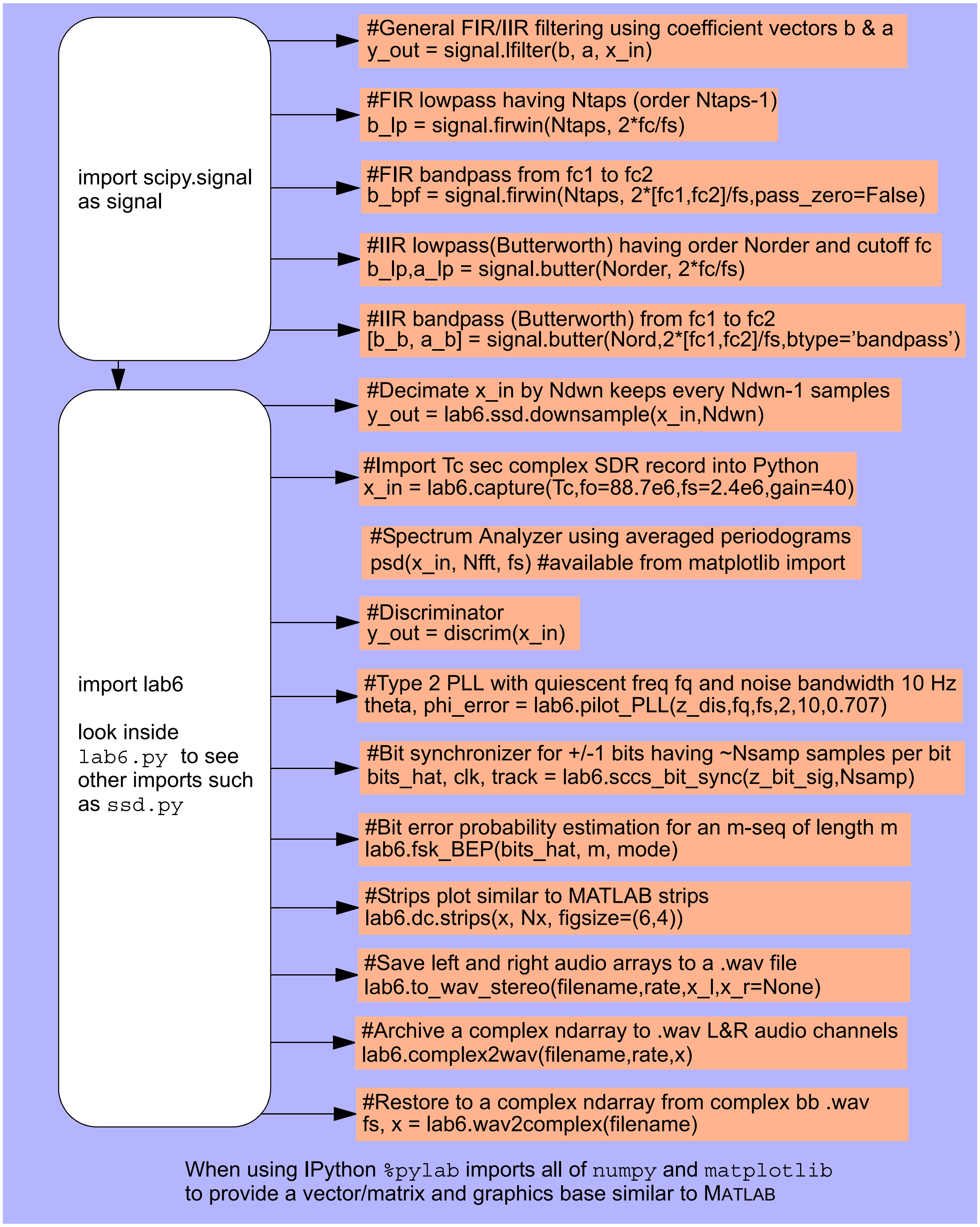


Figure 14: Overview of Python functions used to develop demodulator algorithms for this lab.

5 Developing Algorithms for FM Demodulation

Now it is finally time to write some code to demodulate the message contained in a FM carrier signal. With SDR# you just clicked WFM or NFM and the correct demodulation was utilized. Now you have to take full responsibility for making the right things happen. The basic demodulator architecture is shown in Figure 15. The block labeled *discriminator* is key to demodulating FM.

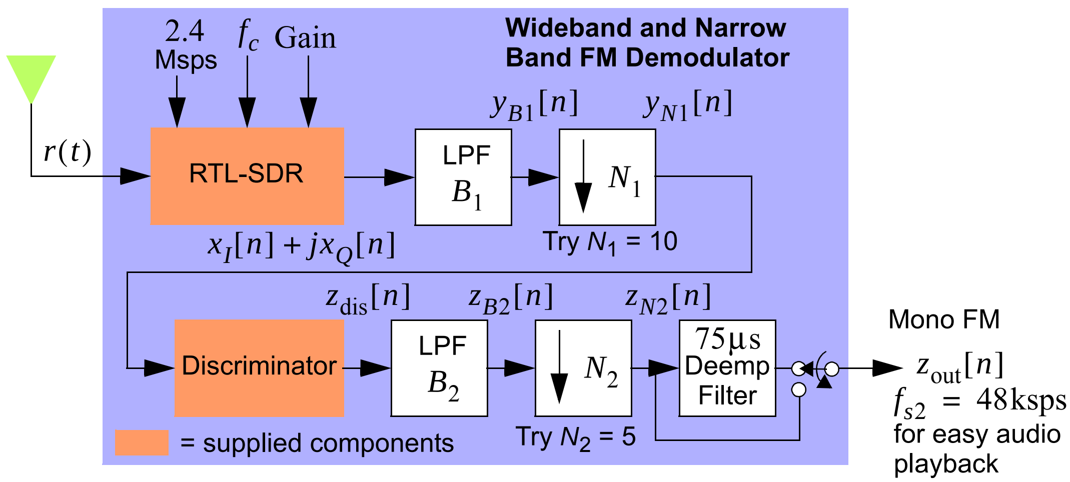


Figure 15: An analog FM receiver for both wideband and narrowband requirements.

5.1 FM Modulation Theory Review

As a quick review, an FM modulated carrier applies the message signal $m(t)$ to the carrier signal $x_c(t)$ such that the derivative of the phase deviation, $d\phi(t)/dt$, (also the frequency deviation) is proportional to the message:

$$x_c(t) = A_c \cos\left[2\pi f_c t + \phi(t)\right] = A_c \cos\left[2\pi f_c t + 2\pi k_d \int^t m(\alpha)d\alpha\right], \tag{8}$$

where k_d is the modulator frequency deviation constant.

To demodulate FM you first consider the ideal discriminator which takes in $x_c(t)$ and operates on the phase deviation to produce

$$y_D(t) = \frac{1}{2\pi} K_D \frac{d\phi(t)}{dt} \tag{9}$$

where K_D is the discriminator gain constant. Notice that for FM, that is $\phi(t) = 2\pi f_D \int^t m(\alpha)d\alpha$ as defined above,

$$y_D(t) = \underbrace{K_D}_{\text{v/Hz}} \cdot \underbrace{f_D}_{\text{v/Hz}} \cdot \underbrace{m(t)}_{\text{v}} \tag{10}$$

To demodulate FM, the *complex baseband discriminator*, also known as the *quadricorrelator*, has a convenient DSP implementation. At complex baseband $x_c(t)$ is of the form

$$\tilde{x}_c(t) = \cos[\underbrace{2\pi\Delta f t + \phi(t)}_{\theta(t)}] + j\,\sin[\underbrace{2\pi\Delta f t + \phi(t)}_{\theta(t)}] = x_I(t) + jx_Q(t), \tag{11}$$

where I have assumed a small frequency error Δf in the frequency translation of $x_c(t)$ to baseband. The frequency discriminator obtains $d\theta(t)/dt$ where in terms of the I and Q signals

$$\theta(t) = \tan^{-1}\left(\frac{x_Q(t)}{x_I(t)}\right) \tag{12}$$

The derivative of $\theta(t)$ is

$$\frac{d\theta(t)}{dt} = \frac{x_I(t)x'_Q(t) - x'_I(t)x_Q(t)}{x_I^2(t) + x_Q^2(t)} \tag{13}$$

In DSP $x_I(t) \Rightarrow x_I(nT) = x_I[n]$ and $x_Q(t) \rightarrow x_Q(nT) = x_Q[n]$, where T is the sample spacing and $1/T = f_s$ is the sampling rate. The derivatives, $x'_I(t)$ and $x'_Q(t)$, are approximated by the *backwards difference* $x_I[n] - x_I[n-1]$ and $x_Q[n] - x_Q[n-1]$ respectively.

Code for implementing the baseband discriminator in Python (MATLAB code almost identical) is given below:

```
import numpy as np
import scipy.signal as signal

def discrim(x):
    """
    disdata = discrim(x)
    where x is an angle modulated signal in complex baseband form.

    Part of the Lab6 ZIP package inside the module lab6.py
    Mark Wickert
    """
    X=np.real(x)        # X is the real part of the received signal
    Y=np.imag(x)        # Y is the imaginary part of the received signal
    b=np.array([1, -1]) # filter coefficients for discrete derivative
    a=np.array([1, 0])  # filter coefficients for discrete derivative
    derY=signal.lfilter(b,a,Y)  # derivative of Y,
    derX=signal.lfilter(b,a,X)  #    "          X,
    disdata=(X*derY-Y*derX)/(X**2+Y**2)
    return disdata
```

5.2 Detailed Complex Baseband Discriminator Analysis

To better understand how the discriminator works and its limitations, you can plug $\theta(nT)$ into the discrete-time implementation, ignoring the $K_D/(2\pi)$ scale factor and the $1/T$ in the derivative approximation, since the code does not include these terms:

$$\begin{aligned} y_D[n] &= \frac{x_I[n]\big(x_Q[n] - x_Q[n-1]\big) - x_Q[n]\big(x_I[n] - x_I[n-1]\big)}{x_I^2[n] + x_Q^2[n]} \\ &= \frac{\cos(\theta[n])\big[\sin(\theta[n]) - \sin(\theta[n-1])\big] - \sin(\theta[n])\big[\cos(\theta[n]) - \cos(\theta[n-1])\big]}{\cos^2(\theta[n]) + \sin^2(\theta[n])} \\ &= \sin\big(\theta[n] - \theta[n-1]\big) \end{aligned} \tag{14}$$

I now insert the sampled continuous-time values for $\theta[n]$ and $\theta[n-1]$

$$\begin{aligned} y_D[n] &= \sin\big(2\pi\Delta f T + \phi(nT) - \phi((n-1)T)\big) \\ &= \sin\left(2\pi\Delta f T + 2\pi f_d \int_{(n-1)T}^{nT} m(\alpha)\, d\alpha\right) \\ &\simeq \sin\left(2\pi\Delta f / f_s + \frac{2\pi f_d}{f_s} \cdot \frac{1}{2}\big[m(nT) + m((n+1)T)\big]\right) \end{aligned} \tag{15}$$

where the last line follows from the trapezoidal integration formula. By assuming that f_s is large enough to assume that $m(t)$ is constant over a T s interval you get

$$\boxed{y_D[n] \simeq \sin\left\{2\pi\left[\frac{\Delta f}{f_s} + \frac{f_d}{f_s}m(nT)\right]\right\}.} \tag{16}$$

The result of (16) shows you that the message is still wrapped inside a sin( ) term, which suggests a nonlinear response. Under the assumption that f_s is large relative to $|2\pi\Delta f + 2\pi f_d m(nT)|$, you can approximate the sine of the argument as the argument itself, that is

$$\boxed{y_D[n] \simeq 2\pi\left[\frac{\Delta f}{f_s} + \frac{f_d}{f_s}m(nT)\right],\ \text{peak freq. dev.} \ll f_s.} \tag{17}$$

Under the assumptions you finally see that the original modulation is recovered to within a scale factor! It is also nice to know that small frequency error introduces a bias on the discriminator output $y_D[n]$. This bias can be used to aid receiver tuning by feeding the bias back to the frequency translation block so as to drive the frequency error to zero. This is known as *automatic frequency control* (AFC).

With this final assumption it should be clear that the complex baseband discriminator has limitations, in particular if the peak frequency deviation, including the tuning frequency error Δf, becomes too large relative to the sampling rate f_s, nonlinear distortion results. Be aware of this limitation. Figure plots the discriminator output via (16) and (17) as the input frequency deviation about zero is swept from $-f_s/2$ to $f_s/2$. The nonlinear characteristic is clearly evident. The frequency deviation limit is $\Delta f = \pm f_s/4$.

It is also important to note that this analysis, although somewhat tedious, has from the very start assumed that the input is a pure FM signal. In reality noise and interference is also present. It is important to bandlimit the input to the discriminator to just the band of frequencies occupied by the signal of interest. Noise and interference impair the ability of the complex baseband discriminator to perfectly recover the modulation $m(t)$.

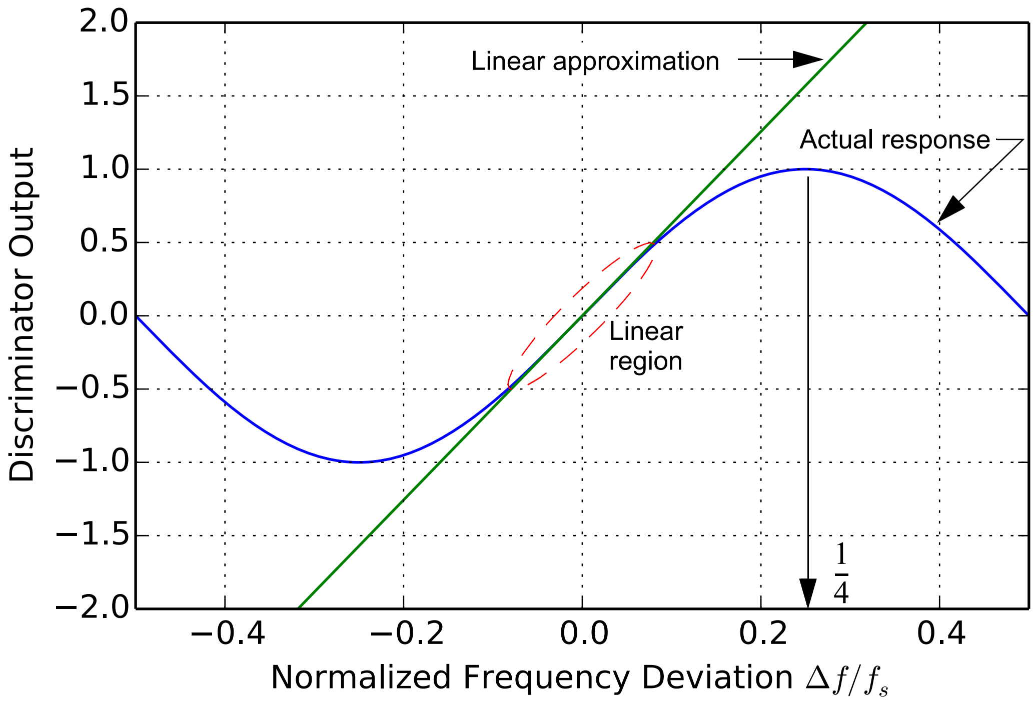


Figure 16: Complex baseband discriminator response characteristic for a static frequency offset of Δf normalized by the sampling frequency f_s.

5.3 Laboratory Exercises

1. In Python or MATLAB , your choice, implement the receiver structure of Figure 15 for broadcast FM. Choose $f_s = 2,400,000$ sps, $N_1 = 10$, and $N_2 = 5$. Note this makes the intermediate sample rate, $f_{s1} = 240$ ksps and the final sampling rate, $f_{s2} = 48$ ksps. When you have completed the steps below you should have a function of the form:

   `[z_out,z_B2,z_N2,z_dis,y_N1,y_B1] = FM_demod(x,B1,N1,B2,N2,fs)` (MATLAB)

   or

   `z_out,z_B2,z_N2,z_dis,y_N1,y_B1 = FM_demod(x,B1,N1,B2,N2,fs)` (Python),

   that returns outputs from each downsampler, the discriminator, and the deemphasis filter (a one-pole lowpass with time constant of 75μs).

   (a) Begin by capturing 5 – 10 seconds of input using either `lab6.capture()` (Python) or the approach outlined in Appendix A for MATLAB. Choose a station of interest. Working with $f_c = 88,700,000.$ Hz, which is KCME, is fine. Set the RF gain to 25 dB or so. Set the gain higher only if you think your signal level is low. If really unsure take a look in SDR#. **Note**: SDR# cannot be running when you are doing RTL-SDR captures.

   (b) View spectrum of you captured complex baseband input signal, $x[n]$ using `psd()` (Python) or `simpleSA( )` (MATLAB) as outlined in Appendix A. Be sure to properly scale the frequency axis. Your capture should look similar to Figure 27. Zoom in to see spectral detail of your signal of interest and also note the bandwidth of the signal for the LPF filter design coming up next. The required bandwidth should be no more that ± 100 kHz to capture the analog message portion of the FM signal. Why?

(c) Choose the cutoff frequency, B_1, for the first LPF based on your assessment of what the design requires. Remember that you want to keep out of band noise and interference away from the discriminator if at all possible. For this lowpass choose a windowed FIR or Butterworth IIR design. This is your choice. For the FIR design limit the number of taps to 64 (order 63) and for the Butterworth limit the order to six.

(d) Decimate the output of the first LPF $y_{B1}[n]$ to produce $y_{N1}[n]$. Calculate the power spectrum at the downsampler output. Comment on what you see.

(e) Send $y_{N1}[n]$ through the discriminator function to produce $z_{\text{dis}}[n]$. Plot the spectrum of this signal and compare it with the idealized spectrum shown in Figure 17. On your plot identify the spectrum features you see from Figure 17. If the station you picked does not have radio broadcast data service (RDS) on a subcarrier of 57 kHz, choose another station that does. Note: KCME does not have RDS!

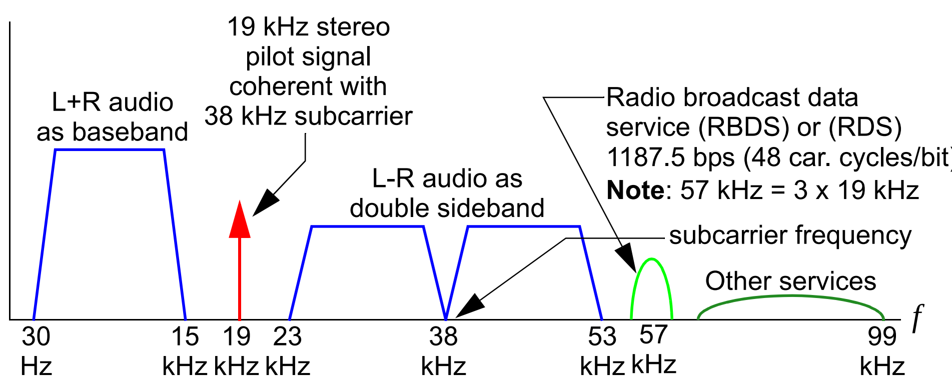


Figure 17: The broadcast FM transmit spectrum prior to the frequency modulator.

(f) Next move on to designing the second LPF. Choose B2 under the assumption that you only want to recover the $L + R$ baseband audio signal. Note: This is a LPF, not a BPF with lower cutoff at 30 Hz. Implement the downsample by five so the sampling rate is finally down to 48 ksps. Listen to the downsampled signal vector by saving the vector as a `.wav` file if using Python and playing the sound vector using `sound(z,fs2)`. The audio should be *crisp* sounding because the lowpass deemphasis filter is not in place yet (it should a bit like having the treble control on your radio turned way up). Comment on any noise you might hear as well.

(g) To finish the demodulator you need to put a one-pole lowpass filter in place to *deemphasize* the high frequencies. In an analog receiver deemphasis can be acomplished with an analog RC (one-pole) lowpass filter having impulse response of the form

$$h(t) = \frac{1}{\tau} e^{-t/\tau} u(t) \tag{18}$$

where $\tau = RC$ is the time constant. In broadcast FM $\tau = 75\mu$s is the standard for US. Through a design technique known as *impulse invariance*, the corresponding discrete-time domain filter has impulse response, to within a scale factor, of the form

$$h[n] = h(nT) = e^{-nT/\tau} u[n] = \left(e^{-T/\tau}\right)^n u[n]. \tag{19}$$

This impulse response corresponds to the first-order difference equation having input and output $x[n]$ and $y[n]$ respectively:

$$y[n] = a_1 y[n-1] + (1 - a_1)x[n], \tag{20}$$

where $a_1 = e^{-T/\tau} = e^{-2\pi \cdot f_3/f_s}$ is the filter feedback coefficient. The gain coefficient on $x[n]$ is $1-a$ so the filter unity gain at dc. Note f_3 is the RC lowpass 3dB frequency, which is related to the time constant τ via $f_3 = 1/(2\pi\tau)$. The `a` and `b` vectors for `y = filter(b,a,x)` or `y = lfilter(b,a,x)` (Python) are `a = [1, -a1]` and `b = [1-a1]`.

Implement this IIR filter in your code and again listen to the recovered audio clip. The sound will be *duller* as a result of deemphasis, but now represents a properly equalized audio signal. By equalized, I mean the frequency response of the audio channel is flat out to at least 12 kHz.

2. Set up a custom transmitter at 70 MHz using the Agilent 33250A, just as you did earlier when working with SDR#. For details again see Figure 11. This time you will only be generating and receiving single tone FM. You will again have to choose your carrier frequency near 70 MHz, to avoid the transmission frequencies used by neighboring lab benches. You will briefly explore differences in the design of narrowband and wideband FM receivers.

   (a) Set the FM deviation on the Agilent 33250 to 5 kHz with a sinusoid tone at 1 kHz. The RF amplitude to 100 mv. Obtain a 5 to 10 s capture record from the RTL-SDR at your chosen carrier frequency.

   (b) Choose appropriate values for B_1, N_1, B_2, and N_2 so that the output sampling rate is 48 ksps. Justify your choice of N_1 to be sure the discriminator is operating in the linear region and make use of Carson's rule to determine B_1. Choose B_2 to allow message bandwidths up to 5 kHz. Finally, choose N_2 so the output sampling rate is 48 ksps.

   Note: The nice thing about software radio design is that you can test your filter demodulator code using simulated signals quite easily. With about one line of code you can write a complex baseband FM transmitter. In IPython, with `pylab` imported, you can write:

```
# An inline function
In [523]: def bb_FM_tx(m,fd,fs):
...:     return exp(1j*2*pi*fd/float(fs)*cumsum(m))
# Here m is a 1kHz tone with fs = 240kHz, fd = 25kHz
In [524]: fm = 1000.; fs = 240e3; fd = 25e3;
In [525]: n = arange(0,10000)
In [526]: x_tx = bb_FM_tx(cos(2*pi*fm/fs*n),fd,fs)
```

```
In [527]: psd(x_tx,2**14,fs);
In [528]: axis([-75e3,75e3,-80,-20])
Out[528]: [-75000.0, 75000.0, -80, -20]
# Include a tuning frequency error of +500 Hz
In [529]: x_tx *= exp(1j*2*pi*500./fs*n)
```

Similarly in MATLAB :

```
>> n = 0:10000;
>> fm = 1000; fs = 240e3; fd = 25e3;
>> x_tx = exp(j*2*pi*fd/fs*cumsum(cos(2*pi*fm/fs*n)));
% Plot the spectrum
>> simpleSA(x_tx,2^14,fs);
% Apply a +500 Hz frquency error
>> x_tx = x_tx.*exp(j*2*pi*500/fs*n);
```

(c) Utilizing the RTL-SDR capture record plot the spectrum of $x[n]$, $y_{B1}[n]$, and $z_{\text{dis}}[n]$. On the spectrum of $z_{\text{dis}}[n]$ note how many dB down the second harmonic distortion is relative to the 1 kHz fundamental. What do think is causing this distortion? Plot about 10 cycles of the final output sinusoid at $z_{\text{out}}[n]$.

(d) Verify that you can receive and hear the 1 kHz FM modulation tone on the PC speakers. Note do not include the deemphasis filter since no preemphasis is included at the transmitter.

(e) Repeat steps (a)–(c) with the deviation increased to 25 kHz.

(f) Which gives better recovered audio quality, the wide wideband or the narrowband scheme? Explain.

(g) Optional: Using either the narrowband or wideband configuration investigated above, verify with an audio source such as a phone or iPod, that music can be sent over your comm link. You will have to set the Agilent 33250 to external FM modulation for this test.

6 Developing Algorithms for a Broadcast FM Stereo Receiver

Here the receiver is more complex as the demodulated FM carrier is a multiplexed signal consisting of at least $L+R$ at baseband, a 19 kHz *pilot* tone, and the $L-R$ signal double-sideband modulated on a 38 kHz subcarrier. Yes, $38 = 2 \times 19$ to allow coherent demodulation using the 19 kHz pilot tone. The left and right audio channels are also *preemphasized* at the transmitter, so they must be *deemphasized* at the receiver. A phase-locked loop (PLL) will be used to track the pilot tone and then frequency double it to 38 kHz. Numerous filters are also employed. The complete receiver block diagram is shown in Figure 18. The PLL is a supplied function block available in the Lab 6 ZIP package. The block diagram may look complex, but your starting point is the receiver function created in Problem 1 of Subsection 5.3.

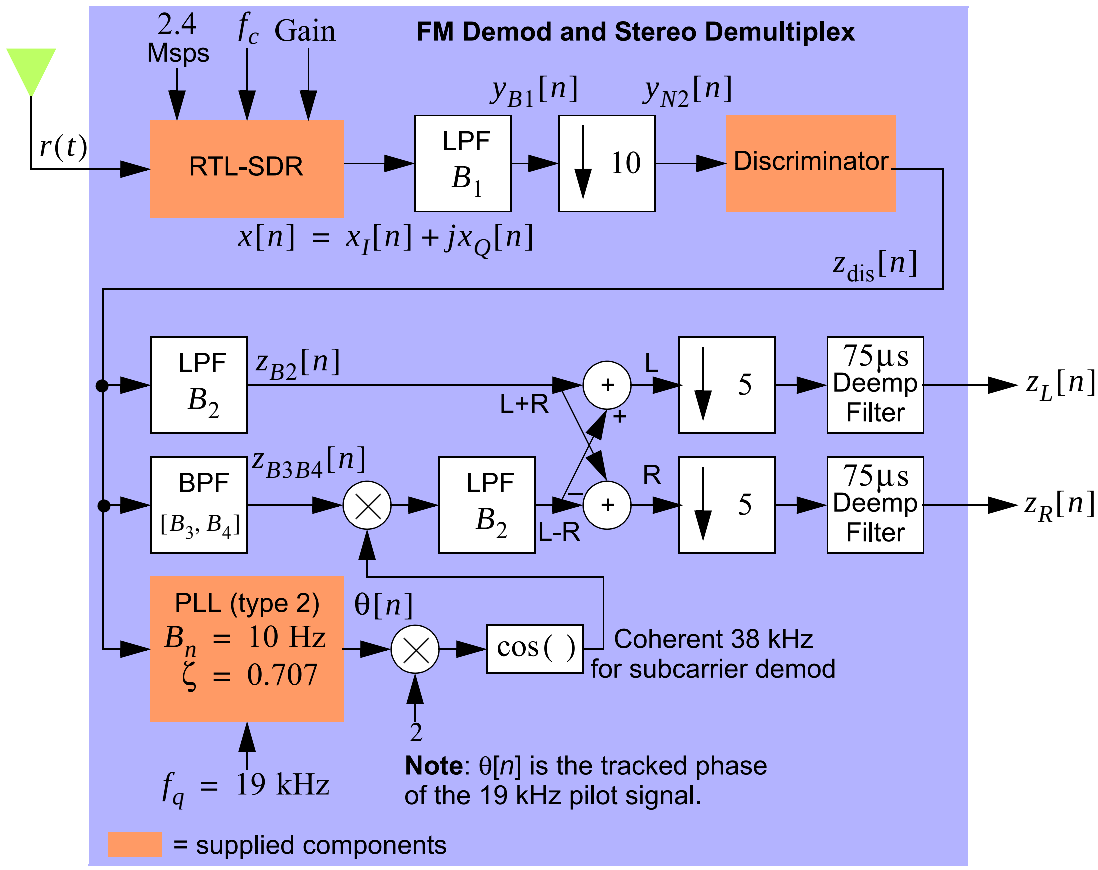


Figure 18: Enhancing the FM demodulator with stereo demultiplexing.

6.1 Laboratory Exercises

There are just a couple of new building blocks in this lab exercise. Designing a bandpass filter is new and using a PLL is new. Beyond the new blocks, all that is required is more programming to *wire* all of the blocks together.

1. Choose an FM station, KCME is fine, and capture 5-10 s from the RTL-SDR. You may use one of your previous captures.

2. Design a bandpass filter to extract the $L - R$ subcarrier signal from the discriminator output $z_{\text{dis}}[n]$. Plot the frequency response in dB to be confident you have a reasonable design.

3. Test your filter by comparing the spectrum of$z_{\text{dis}}[n]$ with the spectrum of $z_{B3B4}[n]$.

4. Test the PLL by passing $z_{\text{dis}}[n]$ into the function `pilot_PLL`. You need to set the VCO quiescent frequency to 19 kHz and enter the proper sampling rate (should be $f_{s1} = 240,000$). To verify that the loop is locking to 19 kHz first take a look at the phase error, e.g., in MATLAB do the following:

```
>> [theta, phi_error] = pilot_PLL(z_dis,19000,fs1,2,10,0.707);
% I have skipped some points since the sampling rate is high
% relative to the bandwidth of the PLL error signal
>> plot(n(1:100:1000000)/fs1,phi_error(1:100:1000000))
```

In Python the steps are similar. In the plotted out you should see the loop go through a transient, similar to that of Figure 19, as the PLL *acquires* the 19 kHz pilot.

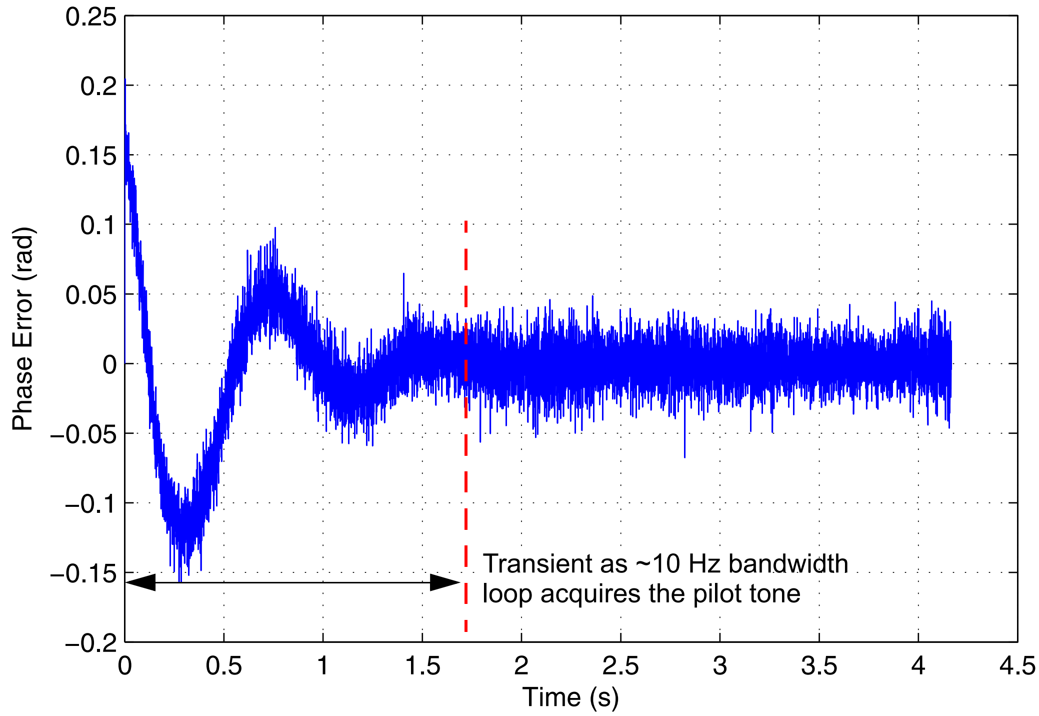


Figure 19: The pilot PLL phase error transient as it locks to the 19 kHz pilot tone embedded in the FM demodulated signal.

5. Continuing the test of the `pilot_PLL`, plot the spectrum of the PLL output by forming the new signal $\cos(\theta[n])$. The spectral line you see should be exactly at 19 kHz. Now plot the spectrum of $\cos(2 \times \theta[n])$. The spectral line should be exactly at 38 kHz. The 38 kHz sinusoid is the signal you will use to coherently demodulate the $L - R$ 38 kHz subcarrier signal (refer to Figure 18) for details.

6. Complete the coding of all the remaining pieces that for the complete stereo receiver. The stereo receiver code should be packaged in a function similar that of Subection 5.3. In particular you should have a function:

   `[z_L,z_R,z_dis,y_N1,y_B1] = FM_stereo_demod(x,B1,N1,B2,N2,fs)` (MATLAB)

   or

   `z_L,z_R,z_dis,y_N1,y_B1 = FM_stereo_demod(x,B1,N1,B2,N2,fs)` (Python), that returns at minimum the stereo outputs `z_L` and `z_R`.

7. Now comes the moment of truth. Listening to the stereo signal following all the processing steps. In MATLAB you use the `sound()` function but you need to concatenate the left and right signal vectors as side-by-side columns, e.g., `z_stereo = [z_L,z_R];`. In Python you write a stereo wave file using the suppled function `to_wav_stereo` (see the discussion under Subsection 4.3). You may want to play the PC sound output through ear buds to be sure the stereo is really present. Additionally jump back to playing the mono version to help discern the difference. Demonstrate the stereo audio playback to your instructor.

8. To satisfy lingering curiosity, replace the 38 KHz coherent reference derived from the `pilot_PLL` with a 38 kHz sinsuoid created locally as in `cos(2*pi*38000./240000*n)`, where `n` is an index vector of the same length as the $z_{\rm dis}[n]$. Again listen to the stereo audio signal and compare it with the stereo signal obtained using the PLL to track the pilot. What differences do you hear? Explain. Are you surprised?

7 Developing Algorithms for a Frequency Shift Keying Receiver

In this section you develop multirate signal processing algorithms to implement a frequency shift keyed (FSK) receiver. FSK is a simple form of digital modulation. If you have ever dialed into a FAX machine by accident, you likely have heard the sound of FSK. In this portion of the lab you will recover the bits of an FSK transmission originating from an *mbed* PN data source driving an Agilent 33250A at about 70 MHz. Of special consideration is the need for a bit synchronizer to properly re-sample the incoming data bearing waveform. The simple fact is the transmit and receive clocks cannot be made synchronous to the point where the clock phase does not cause bit errors. The bit synch algorithm is form of PLL.

7.1 Introduction to FSK

The simplest form of FSK just switches the frequency of the transmitted carrier above and below the nominal carrier frequency by Δf Hz.

$$x_c(t) = A_c \cos\left[2\pi\left(f_c + \frac{\Delta f}{2} d(t)\right)t + \phi\right] \tag{21}$$

where $d(t)$ is a non-return-to-zero (NRZ) data waveform taking on values of ± 1 for T_bs and ϕ is an arbitrary phase. The bit rate is $R_b = 1/T_b$. Note unlike earlier discussions with FM, the Δf term used with FSK defines the peak-to-peak frequency deviation of the carrier as the data waveform bits change form 0 to 1 or 1 to 0.

In complex baseband form the carrier f_c is nominally zero(no tuning error) so the signal becomes

$$\tilde{x}_c(t) = A_c \exp\left[j2\pi\left(f_c + \frac{\Delta f}{2} d(t)\right)t + \phi\right]. \tag{22}$$

To give a better taste of FSK I construct a simple waveform simulation in MATLAB (in Python similar) that follows from (22):

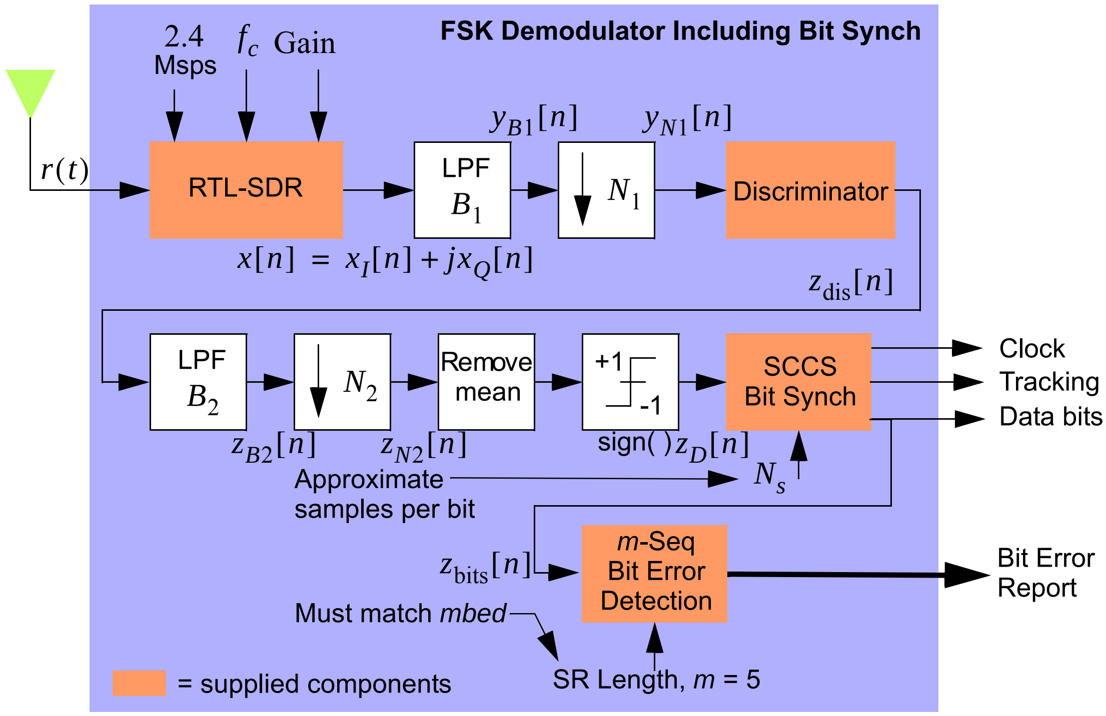


Figure 20: An FSK demodulator including a bit synchronizer.

```
>> Ns = 120; Nbits = 100000; Rb = 1000; Df = 5000.0; fs = Rb*Ns;
>> d_NRZ = filter(ones(1,Ns),1,upsample(2*randi(0:1,1,Nbits)-1,Ns));
>> t_samp = [0:length(d)]/fs; %time axis vector in seconds
>> xc = exp(j*2*pi*Df/2/fs*d.*[0:length(d)-1]);
```

In the above the bit rate is $R_b = 1$ kbps and $\Delta f = 2.5$ kHz. I use 120 (2,400/20 as with RTL-SDR) samples per bit and generate 100,000 bits. The input NRZ data $d(t)$, the baseband spectrum, and the discriminator output are shown in Figure 21. The NRZ data (top) and the discriminator output (bottom) are as expected. The bit duration for both is indeed 1 ms and the discriminator has converted the frequency deviation of $\tilde{x}_c(t)$ back into an NRZ data waveform. The spectrum is perhaps the most interesting. With $\Delta f = 5$ kHz the peak deviation is 2.5 kHz. A property of FSK is the appearance of *horns* at the peak deviation frequencies, i.e., $\pm\Delta f/2 = \pm 2.5$ kHz. The spectral lobe sitting under each horn has shape similar to the main lobe of the $\text{sinc}((f \pm \Delta f/2)T_b)$ function, with null-to-null spacing of $2R_b = 2$ kHz.

7.2 The SCCS Bit Synchronizer

Since the transmit clock generating the NRZ data, $d(t)$, and the receive clock, in this case the RTL-SDR sampling clock, are not perfectly synchronized, you need a means to locally re-sample the waveform output from the discriminator. The supplied function `[rx_symb_d,clk,track]`

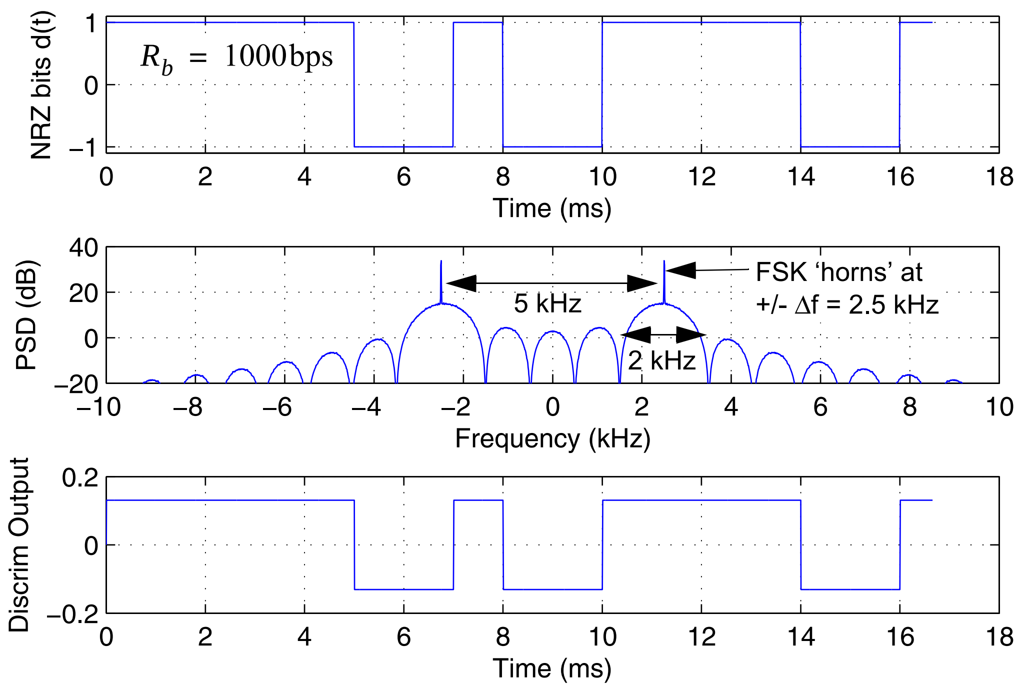


Figure 21: Waveforms and spectrum of an FSK transceiver having $R_b = 1$ kbps and deviation $\Delta f = 2.5$ kHz.

`= sccs_bit_sync(y,Ns)` implements a pure DSP bit synchronizer known as sample-correlate-choose-smallest (SCCS) [6]. For this lab you can for the most part consider this to be a *black box*. The details of the algorithm can be found in the paper by Chen [6] and you can also look at the code and code comments in the provided function.

To further motivate the need for this function, consider the waveform plots of Figure 22. In the upper plot you see a case of synchronous sampling at $N_s = 4$ samples per bit. In the lower plot the sampling rate is asynchronous at $N_s = 4.1$ samples per bit. To obtain the ± 1 bit values from the lower waveform re-sampling is required. This is the job of a bit synchronizer. Figure 23 shows the track signal output from the SCCS for the synchronous and asynchronous sampling depicted in Figure 22. As expected, for a signal having synchronous samples already, the SCCS simply finds the proper index mod(4) to make accurate determination of the bit value, e.g., ± 1. For the slightly over sampled signal, the index needs to be be bumped ahead every so often to keep up with the higher sampling rate. In the end the SCCS function outputs correct bit decisions for both waveforms, with the exception of a start-up transient:

```
rx_bits_sync  =  1    -1     1     1     1     1    -1     1    -1     1 ...
rx_bits_async = -1     1     1     1     1     1    -1     1    -1     1 ...
```

The output variable `rx__symb_d` contains the recovered bits at ± 1 amplitude values. To return to 0/1 logic values just translate the values, e.g. `(rx_symb_d + 1)/2`.

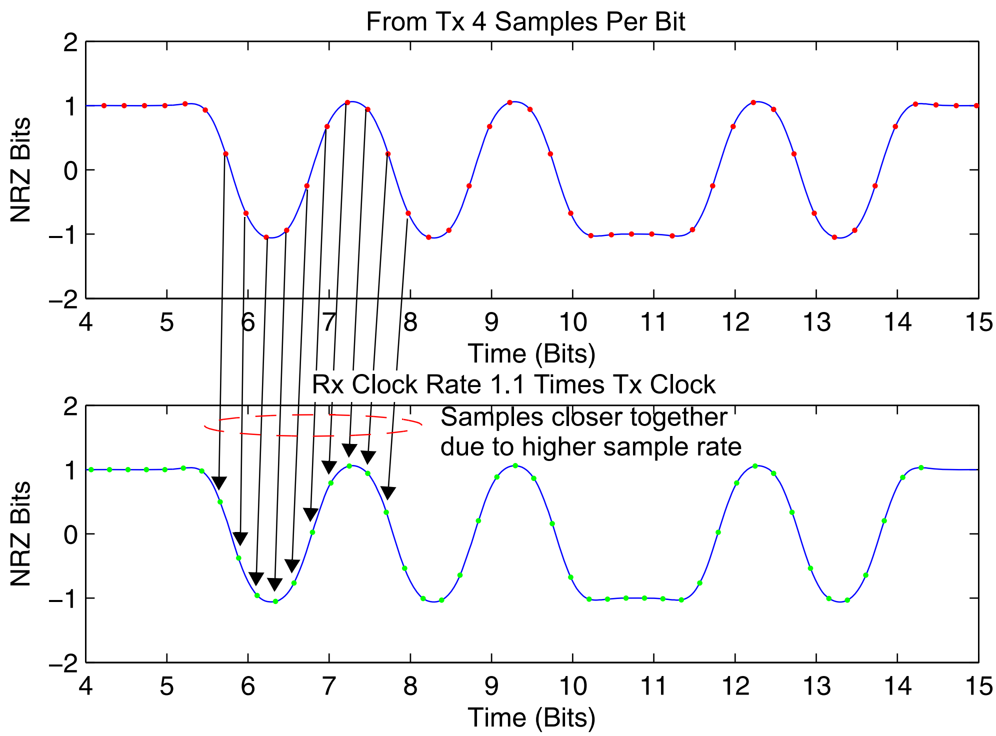


Figure 22: Waveform plots of a filtered NRZ signal with an overlay of synchronous 4 samples per bit (top) and 4.1 samples per bit (bottom).

7.3 Laboratory Exercises

1. To experiment with the FSK receiver you will need to first set up a transmitter. The Agilent 33250 generator will again be used. In order determine if the system is reliable you need to perform *bit error probability* (BEP) testing, also commonly referred to as *bit error rate* (BER) testing. To obtain a known source of data bits you will drive the external modulation port of the Agilent 33250 with a level shifted out from the mbed microcontroller. Recall your experiences with this hardware back in Lab 2. The block diagram of the transmitter is shown in Figure 24.

   (a) Design an test an op-amp level shifter that takes to 0–3.3v output from the mbed to an amplitude swing of ± 2.5v. It is important that the data waveform not be inverted, hence a second op-amp is used.

   (b) On the spectrum analyzer observe the FSK spectrum when using a 31 bit long m-seq from the mbed. First test with a 1 kbps data sequence and then with a 10 kbps data sequence. Set the frequency deviation to 5 kHz and 50 kHz respectively. Note the 33250 achieves a peak frequency deviation of the output when the input reaches ± 5 v. Since your input only goes to ± 2.5v, the total FSK deviation you generate should be $\Delta f = 5$ kHz and 50 kHz, respectively. The theoretical spectrum can be found using the code snippet found in Subsection 7.1.

   (c) With the Agilent transmitting FSK on a carrier near 70 MHz, make two 5–10 second

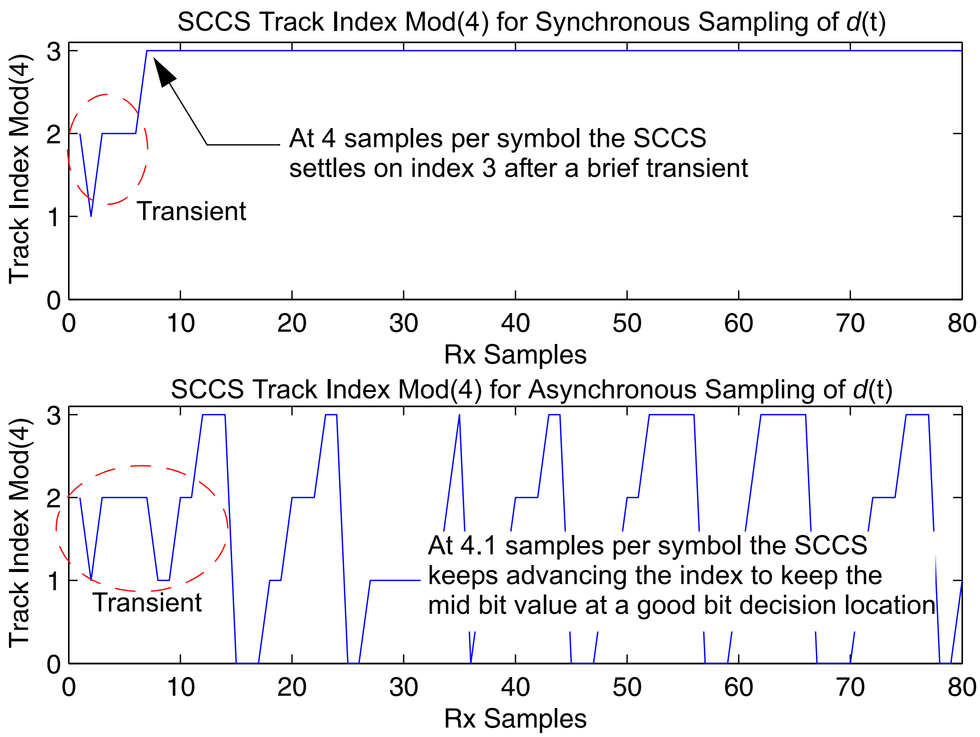


Figure 23: The mod(N_s) tracking signal, here $N_s = 4$, developed by the SCCS to bit synchronize to a waveform having 4 samples per bit (top) and 4.1 samples per bit (bottom).

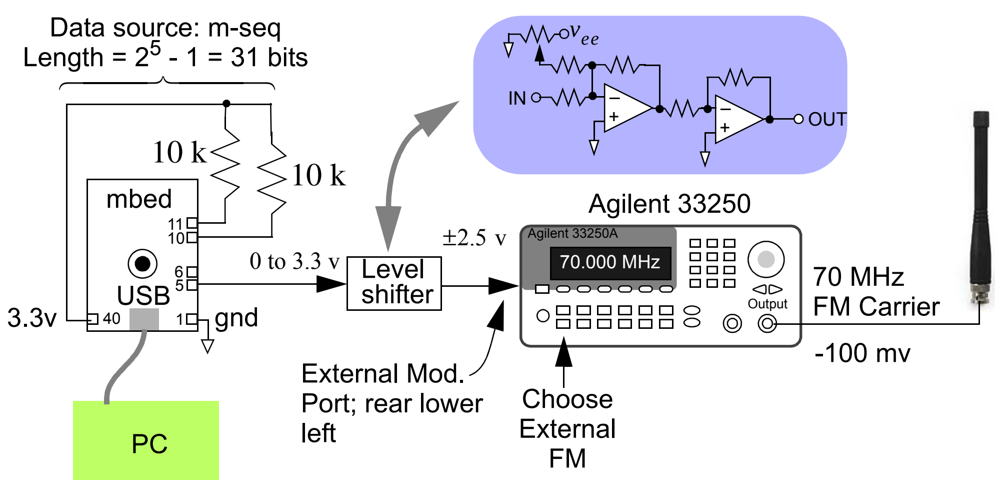


Figure 24: Setting up an FSK transmitter using the mbed, a level shifter, and the Agilent 33250.

captus using the RTL-SDR dongle. One capture should be with the mbed producing 1 kbps, 5 kHz pp effective frequency deviation, and the second with 10 kbps, 50 kHz pp efective frequency deviation.

(d) Develop an FSK receiver function based on your earlier FM demodulator of Subsection 5.3. Your modifications will focus on the new blocks that follow the N_2 downsampler of Figure 20. The *remove mean* block centers the waveform $z_{N2}[n]$ about zero. You simply subtract the mean of the signal from itself, e.g., `z_N2 = z_N2 - mean(z_N2)`. Note the `mean( )` function is natively available in MATLAB and imported into Python from `numpy` in Python. In IPython this is automatically taken care of when you start up via `pylab`. In the next block you pass the signal through the MATLAB/Python function `sign( )`, which thresholds (hard limits) the signal to take on values of ± 1. This type of signal is needed by the SCCS function block.

2. Test your FSK demodulator function using the captured FSK signals.

   (a) Plot the spectrum of the signal input to the discriminator, $y_{N1}[n]$. This will serve as a check to see that you have chosen B_1 correctly. You want to at least pass the main FSK lobes and block as much noise and interference as possible. Note: The bandwidth of an FSK signal is about

   $$B_{\text{FSK}} \simeq \Delta f + 2R_b \text{ Hz}. \tag{23}$$

   (b) Plot the recovered data waveform $z_{N2}[n]$ after the mean removal and compare it with the hard-limited signal $z_D[n]$.

   (c) Obtain the BEP/BER report that the MATLAB or Python function `fsk_BEP(z_bits,5,1)` returns, e.g. in MATLAB :

```
>> fsk_BEP(rx_symb_d(1:end),5,1)
//////////////////////////////////////////////////////
Bit Errors: 0
Bits Total: 89552
       BEP: 0.00e+00
//////////////////////////////////////////////////////
```

   Hint: The `strips( )` plotting function in MATLAB or the strips(x,Nx) function in Python, makes it convenient to quickly view a large number of output bits graphically to see if the correct 31-bit m-seq has been recovered. An example of

```
>> strips(rx_symb_d(2000:3000),62)
```

   in action is shown in Figure 25. From this plot it very easy to see where bit errors occur. This is useful because sometimes you might have some interference that is causing problems in just few areas.

References

[1] `http://en.wikipedia.org/wiki/Software-defined_radio`.

[2] Rodger E. Ziemer and William H. Tranter, *Principles of Communications*, 6th edition, John Wiley, New York, 2009.

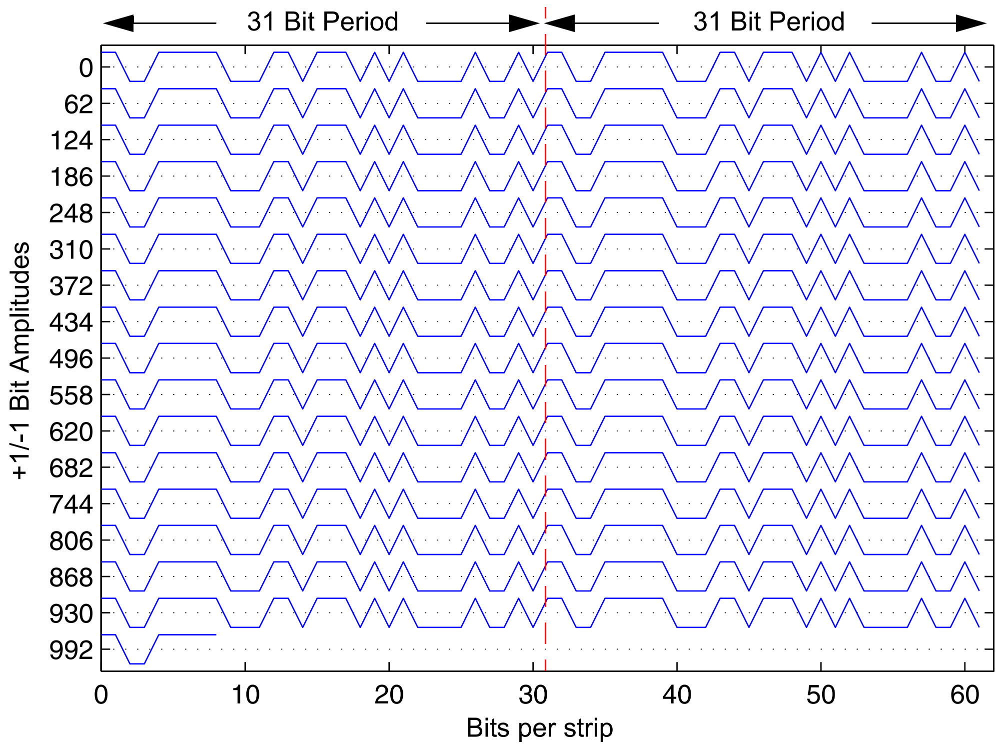


Figure 25: Recovered data bits from the SCCS plotted using `strips( )`.

[3] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing* (3rd Edition), Prentice Hall, New Jersey, 2010.

[4] Mark Wickert, *Signals and Systems for Dummies*, Wiley, New York, 2013. ISBN: 978-1-118-47581-2.

[5] `http://superkuh.com/rtlsdr.html`

[6] K. Chen and J. Lee, "A Family of Pure Digital Signal Processing Bit Synchronizers," *IEEE Trans. on Commun.*, Vol. 45, No. 3, March 1997, pp. 289–292.

Appendix A: Command Line MATLAB

The main body of the lab has described working with the RTL-SDR using Python. Interface libraries have been written for other languages, such as MATLAB . In particular the MathWorks, makers of MATLAB , have a complete RTL-SDR package available at:

`http://www.mathworks.com/hardware-support/rtl-sdr.html`

which uses *Simulink*™. Besides needing Simulink, you must have the *Communication System Toolbox*™ installed.

In this appendix I describe a simpler no frills means of capturing RTL-SDR output. The result is a capture of 8-bit per sample complex (I/Q) data saved to a file. You can then import this file into MATLAB . Once the data is in the MATLAB workspace, you work with the signals samples in much the same way as I have shown using the Python `IPython` console.

The starting point on Windows is to follow the Zadig driver setup discussed in the first part of the lab. Next you need to have the the same dll's and exe's in your path, also as described in the Python set-up. Specific to MATLAB you need to have the m-file `loadFile.m` on your path. The file is listed below and is also included in the Lab 6 ZIP package.

```
function y = loadFile(filename)
%  y = loadFile(filename)
%
% reads  complex samples from the rtlsdr file
%

fid = fopen(filename,'rb');
y = fread(fid,'uint8=>double');

y = y-127;
y = y(1:2:end) + i*y(2:2:end);
```

The function `loadFile('filename.bin')` takes a raw RTL-SDR raw capture file and converts it to a complex signal vector in MATLAB . To capture sample you use the program `rtl_sdr.exe` which is one of the command line programs in the Windows package found at `http://sdr.osmocom.org/trac/wiki/rtl-sdr`. The command line consists of the sampling rate `-s xxx` in samples per second, the center frequency `-f xxx` in Hz, the RF gain `-g xx` in dB, and the name of the binary file to write the results to, e.g. `capture.bin`.

```
some-path>rtl_sdr -s f_in_sps -f f_in_Hz -g gain_dB capture.bin
```

Figure 26 is screen shot of a capture. I recommend about five seconds, that is count off five seconds after you start the capture, then hit `Ctrl-c` to end the capture. The file size grows rapidly. Expect a file size of 20-30 Mbits.

In MATLAB you can verify a successful capture via spectrum analysis using the supplied function `simpleSA.m`, included in the Lab 6 ZIP package. Here I capture an FM broadcast station at 88.7 MHz (KCME, Colorado Springs):

```
>> x = loadFile('capture.bin');
>> simpleSA(x,2^14,2400);
```

The station is strong so I set the RF gain to 35 dB and sample at 2.4 Msps. The spectrum result is shown in Figure 27 using `simpleSA()`. Note since this is a complex baseband signal, 0 Hz corresponds to the capture center frequency of 88.7 MHz. The rectangular shaped spectra visible above and below the analog FM spectra is the *HD*[10] signal overlayed on top of the legacy transmission.

```
>> x = loadFile('capture.bin');
>> simpleSA(x,2^14,2400);
>> print -depsc -tiff capture.eps
```

[10] http://en.wikipedia.org/wiki/HD_Radio

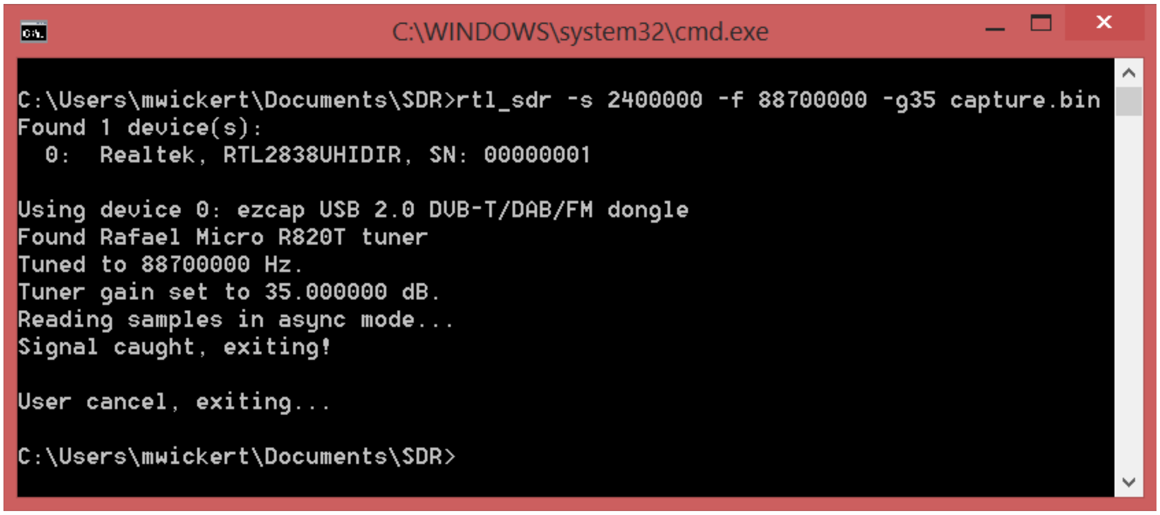


Figure 26: Windows command window showing how to capture receiver output to a binary file.

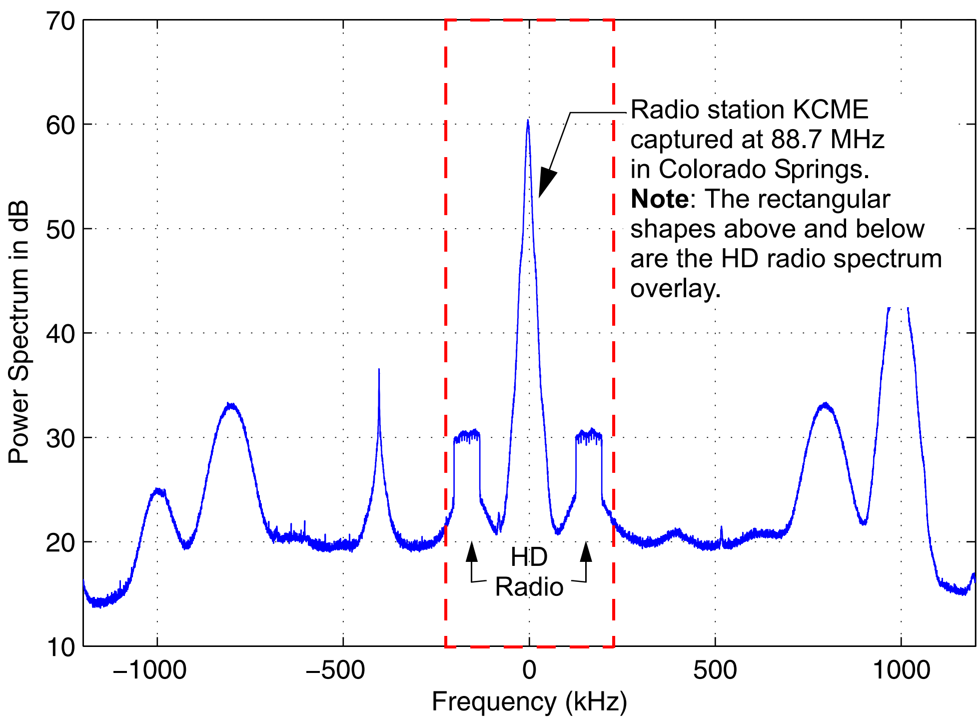


Figure 27: Power spectrum of capture obtained from MATLAB using `simpleSA.m`.

Appendix B: Using Python and the Python Notebook

The IPython notebook[11] provides a very nice way of documenting your work in Python. Code, text, math equations, and plots can all be contained in the notebook. With the advent of version

[11] `http://ipython.org/notebook.html`

2.0, GUI widgets are also available. Of particular note is the `Audio` widget, which lets you play sounds through you computer's audio system. In Figure 28 I show an example of the notebook as it relates to the RTL-SDR. A sample notebook is included in the Lab6 Python ZIP package.

Note the sample notebook does not contain all of the content and function alluded to. In particular `mono_FM` is a variation on one of the functions you write for the lab. The .wav files are not available either, but you do see how to load your own files and play them back.

Appendix C: Setting Up the Python Package pyrtlsdr

Linked off of the Osmocom driver link given earlier, you will find a very nice Python package `https://github.com/roger-/pyrtlsdr`. You can install this package using `easy_install pyrtlsdr` or `pip pyrtlsdr`. Once this package is installed you can use the functions in the package to set up the RTL-SDR to capture I/Q samples into a complex values numpy `ndarray`. You are now set to develop receiver algorithms in like fashion to the MATLAB description given in Appendix A. The important difference is that you never need to go to the OS shell window. Everything is done right in IPython or in the Python functions and scripts you write.

Appendix D: Setup on Windows

Details on how to set-up the RTL-SDR dongle on Windows and SDR# can be found at:

`http://rtlsdr.org/softwarewindows.`

IP[y]: Notebook SDR_Demo (autosaved)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

In [24]:
```
import lab6
import scipy.signal as signal
from IPython.display import Audio, display
```

Capture 5 Seconds of I/Q Samples

In [71]:
```
#x = lab6.capture(Tc, fo=88700000.0, fs=2400000.0, gain=40)
x = lab6.capture(5)
```

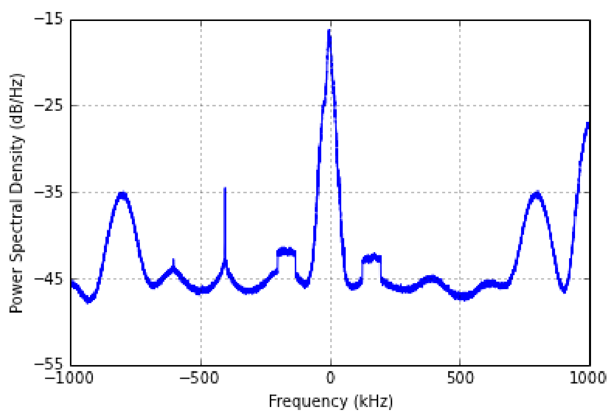
Archive the capture in a wave file

In [77]:
```
lab6.complex2wav('clean_KCME.wav',2400000,x)
```

```
Saved as binary wav file with (I,Q)<=>(L,R)
```

Look at the Complex Baseband Spectrum from the RTL-SDR

In [76]:
```
psd(x,2**14,2400);
xlim(-1000,1000);
xlabel('Frequency (kHz)');
```



Process Samples Through a Mono FM Demodulator

In [74]:
```
z_bb, z_out = mono_FM(x,fs=2.4e6,file_name='new_KCME.wav')
```

```
Done!
```

In [39]:
```
Audio('new_KCME.wav')   # From file
```

Out[39]: 0:01 0:05

Figure 28: Using the IPython notebook to document your work in Python.