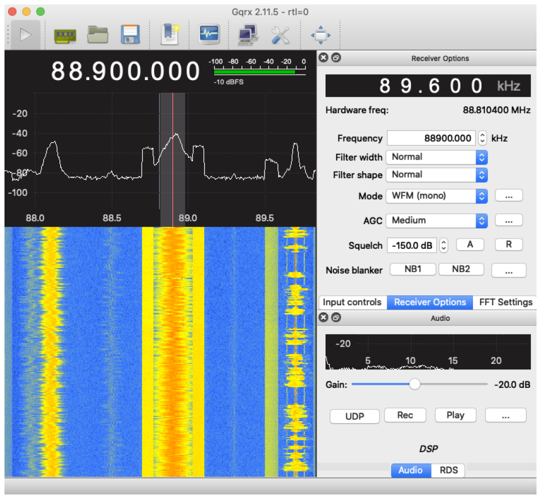
6.341: RTL-SDR Project

Nada Hussein

Fall 2020

1 Data Collection

To start this project, I chose a sampling frequency of 2.048 MHz. I then chose a center frequency of 88.9 MHz, as this was a choice that gave me an interesting band that I could hear clear audio from. I also noted the hardware frequency, 88.810 MHz, as this would be important for modulating down to baseband.


Gqrx 2.11.5 - rtl=0

88.900.000

Receiver Options

89.600 kHz

Hardware freq: 88.810400 MHz

Frequency 88900.000 kHz

Filter width Normal

Filter shape Normal

Mode WFM (mono)

AGC Medium

Squelch -150.0 dB

Gain: -20.0 dB

Figure 1: GQRX Settings for Sampled Data

For the rest of the lab writeup I will be showing visuals for data1.raw, the first blind dataset given, for convenience. For this file, the hardware frequency was 100.3 MHz, and the center frequency was 100.7 MHz, with a sampling rate of 2.048 MHz.

2 Discrete-Time Channel Selection

2.1 Modulating to Baseband

To begin, my signal had the following magnitude response:

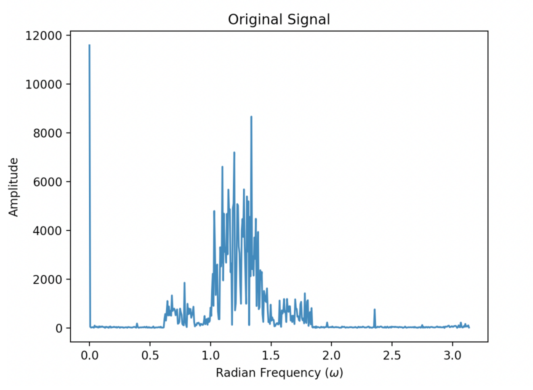


Figure 2: Magnitude Response of Original Signal

Based on the RTL-SDR block diagram, we see that the RTL-SDR is able to modulate our signal from Ω_c down to 0. However, we notice in the GQRX window that our hardware frequency, Ω_h, was different from Ω_c. This means we need to modulate down by an additional amount equivalent to the error between the two frequencies. I computed ω_0, the additional modulation we needed, as follows:

$$\Omega_c = 2\pi f_c$$

$$\Omega_h = 2\pi f_h$$

$$\omega_0 = \frac{\Omega_c - \Omega_h}{f_s} = \frac{2\pi(f_c - f_h)}{f_s} \approx 1.227$$

where $f_c = 100700000$ Hz, $f_h = 100300000$ Hz, and $f_s = 2048000$ Hz. This gave $\omega_0 \approx 1.227$, which, as we can see from the graph of the original signal, is approximately how much the signal needs to be shifted in frequency to be centered around 0. Once I had ω_0, I created a sequence equivalent to $e^{-j\omega_0 n}$ for all n in range of the length of my signal to correspond to a frequency shift by ω_0, and element-wise multiplied them to modulate my signal down to baseband.

This resulted in a magnitude response as follows:

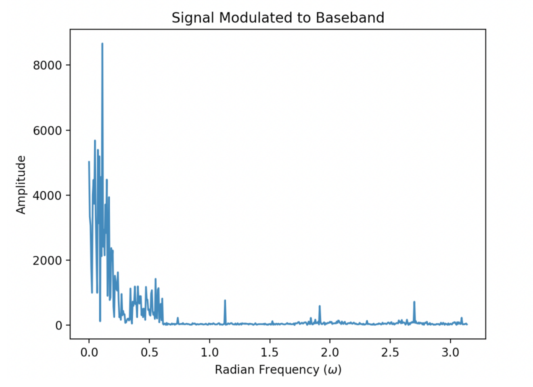


Figure 3: Magnitude Response of Signal Modulated to Baseband

2.2 Downsampling

To downsample, I needed to convert my sampling rate from 2.048 MHz to 256 kHz. I found my downsampling factor M as follows:

$$M = \frac{f_s}{f_s'} = \frac{2048000}{256000} = 8$$

2.2.1 Low Pass Filter

Once we knew M, we first had to implement a low pass filter with $\omega_c = \pi/M$ and gain 1 in order to prevent any aliasing once we decimate. With $M = 8$, this gave us a cutoff frequency $\omega_c = \pi/8$. I then used scipy.remez to create a low pass filter with the following specifications:

$$\omega_p = \pi/8$$
$$\omega_s = \pi/8 + 0.06\pi$$
$$\delta_p = 0.0575$$
$$\delta_s = 0.0033$$

I tuned the transition bandwidth, 0.06π, and the δ values based on finding a filter that sufficiently suppressed the high frequencies, kept the low frequencies with minimal rippling, and dropped off quickly in the transition band so as not to take to include too many frequencies higher than $\pi/8$.

This produced a low pass filter as follows:

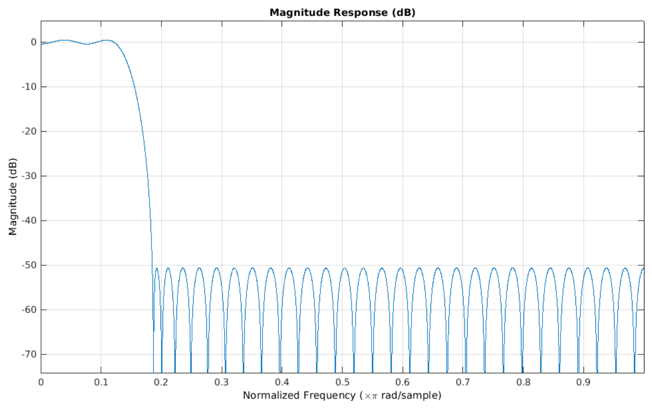


Figure 4: Magnitude Response of Low Pass Filter with $\omega_c = \pi/8$

Here you can see that the filter drops off at $\pi/8$, as we wanted, has minimal rippling in the passband, and sufficiently suppresses the frequencies in the stopband.

I convolved my signal with the impulse response of my low pass filter to apply the anti-aliasing before decimating.

2.2.2 Decimation

After the low pass filter, we are now safe to decimate the signal and avoid aliasing caused by the replicas in the frequency domain created by decimating. I decimated the signal by simply taking every 8th sample, and ignoring everything else. As expected, this compressed the time domain signal by a factor of 8, which expanded the frequency domain transform by 8.

This resulted in a signal magnitude response as follows:

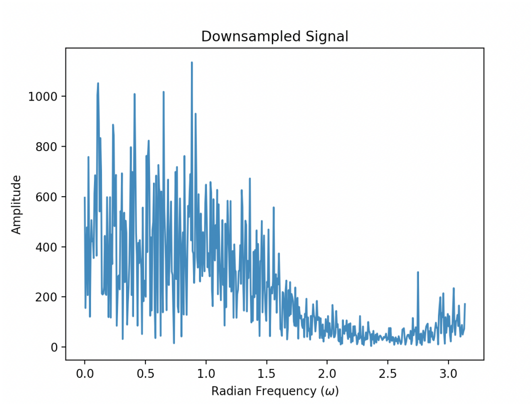


Figure 5: Magnitude Response of Downsampled Signal

3 FM Demodulation

3.1 Frequency Discriminator

3.1.1 Limiter

The first step of the Frequency Discriminator was normalizing the magnitude of each sample of the signal so that we did not conflate the magnitude with the phase once we did the DT Differentiation. I did this by simply setting the new signal $y_1[n]$ as follows:

$$y_1[n] = \frac{y[n]}{|y[n]|}$$

This made the magnitude of any given sample either 1 or -1.

3.1.2 DT Differentiator

Now we were ready to implement the DT Differentiator. To do this, I first looked at the frequency response of an ideal DT differentiator with linear phase:

$$H_{diff}(e^{j\omega}) = (j\omega)e^{-j\omega M/2}, -\pi < \omega < \pi$$

When we convert this to the time-domain, we get:

$$h_{diff}[n] = \frac{cos\pi(n - M/2)}{(n - M/2)} - \frac{sin\pi(n - M/2)}{\pi(n - M/2)^2}, -\infty < n < \infty$$

I decided to implement this filter in the time-domain. To do this, I initially chose M = 10, since an even value of M would mean that my time delay block could be implemented easily in the time domain since M would be an integer. I then sampled $h_{diff}[n]$ from 0 to 10. However, if we just truncated the window we would end up with a lot of artifacts from the effective rectangular window we multiplied by. So in order to smooth out the window I decided to multiply by a Kaiser window with α = M+1, and β = 2.4. When I multiplied my truncated $h_{diff}[n]$ with the Kaiser window, I got the following impulse response:

This also yielded the following magnitude response:

Here we can see that the frequency response closely follows the linearity we would have expected until around 0.8π, but because M = 10 yields a Type III system, we are forced to have a zero at π, which ruins the linearity that we wanted. To fix this, I had to choose M to be even. Here you can see the impulse response when I chose M = 15.

This also yielded the following magnitude response:

Here you can see we are able to get a really good approximation for the linear slope we expected, and our differentiator will behave as expected.

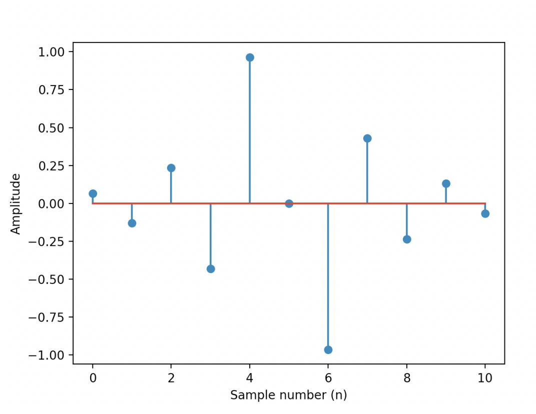


Figure 6: Impulse Response of DT Differentiator for M = 10

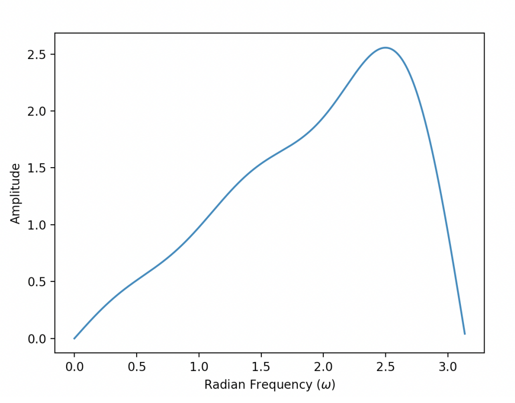


Figure 7: Magnitude Response of DT Differentiator for M = 10

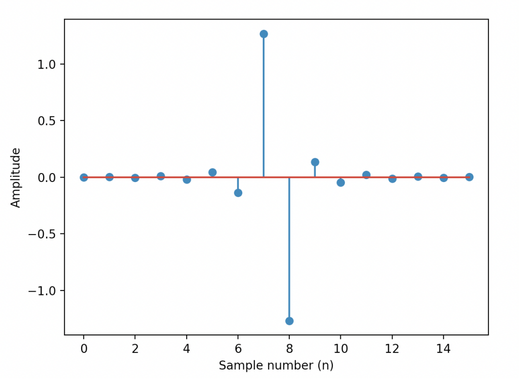


Figure 8: Impulse Response of DT Differentiator for M = 15

3.1.3 Conjugation and Time Delay

After the DT Differentiator was done, I needed to implement the conjugated and time delayed version of $y_1[n]$ so I could multiply it with the output of my differentiator. I first conjugated $y_1[n]$. I then wanted to convert the $e^{-j\omega M/2}$ block into a time delay of $y_1[n-M/2]$. However, because I chose M to be 15, I could not directly implement this time shift since M/2 would not be an integer. To fix this issue, I expanded $y_1[n]$ by 2, performed a linear interpolation of the two neighboring values to create each new value, shifted by M

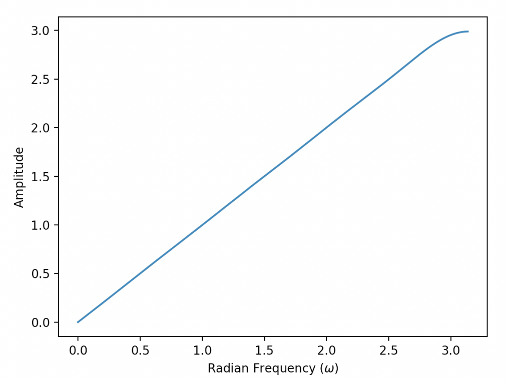


Figure 9: Magnitude Response of DT Differentiator for M = 15

instead of M/2, and decimated by 2. This allowed me to keep M odd to avoid the differentiator having a zero at π, but also allowed me to keep the delay block in the time domain. After this was done, I multiplied my time-shifted, conjugated output with the differentiator's output.

3.2 Imaginary Component

I then completed the frequency discriminator by taking the imaginary part of the output.

3.3 Deemphasis Filter

We defined the continuous time deemphasis filter as follows:

$$H(j\Omega) = \frac{1}{1 + j\Omega\tau_d}$$

To convert this to a DT filter approximation, we needed to take the bilinear transform of $H(j\Omega)$.

$$H(j\Omega) = \frac{1}{1 + j\Omega\tau_d}$$

$$H(s) = \frac{1}{1 + \tau_d s}$$

We can convert this to H(z) by finding a relationship between z and s as follows:

$$z = e^{sT} = \frac{e^{sT/2}}{e^{-sT/2}} \approx \frac{1 + sT/2}{1 - sT/2}$$

We can then rearrange this to become:

$$s \approx \frac{2}{T}\frac{z-1}{z+1}$$

From here we can plug in our relationship between s and z to create H(z):

$$H(z) = H(s)|_{s=\frac{2}{T}\frac{z-1}{z+1}} = \frac{1}{1 + \tau_d \frac{2}{T}\frac{z-1}{z+1}} = \frac{z+1}{z + 1 + \tau_d \frac{2}{T}(z-1)}$$

$$H(z) = \frac{z+1}{z(1 + \tau_d \frac{2}{T}) + 1 - \tau_d \frac{2}{T}} = \frac{z+1}{z(1 + 2\tau_d f_s) + 1 - 2\tau_d f_s}$$

To get this in the same form that MATLAB will return, we then divide everything by $(1 + 2\tau_d f_s)$:

$$H(z) = \frac{\frac{z+1}{1+2\tau_d f_s}}{z + \frac{1-2\tau_d f_s}{1+2\tau_d f_s}}$$

Plugging in $\tau_d = 7.5e-5$ and $f_s = 256000$ yields:

$$H(z) = \frac{0.02538z + 0.02538}{z - 0.9492385}$$

When we compare this to MATLAB's bilinear transform output, we get the exact same transform.

Below we've shown that the DT approximation filter we've created is in fact a good approximation for the CT deemphasis filter.

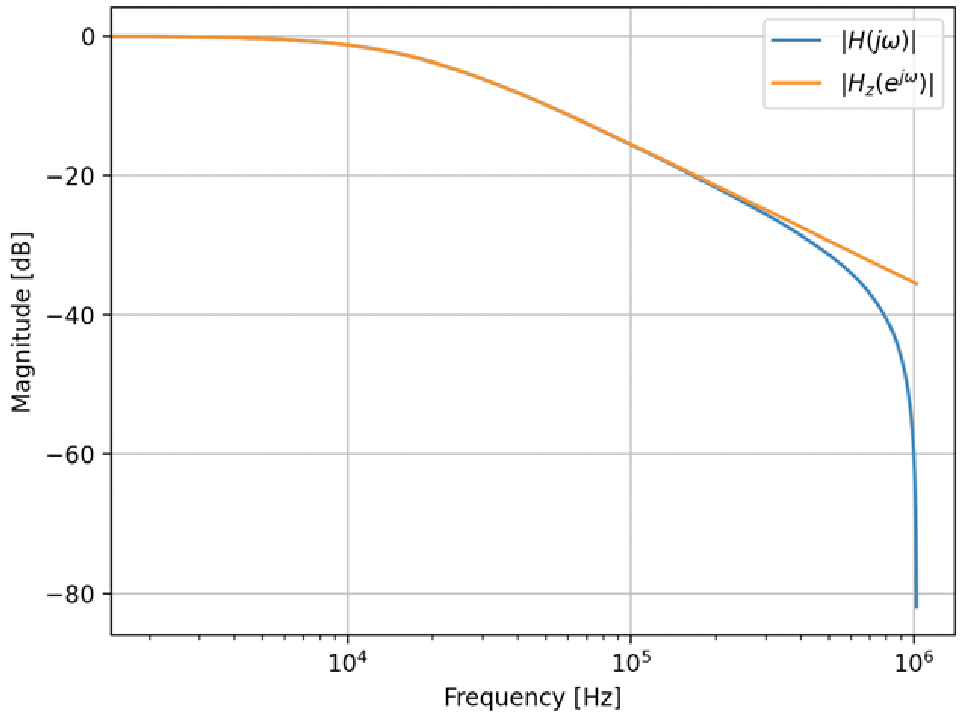


Figure 10: Magnitude Response of CT transfer function vs DT transfer function

To implement this, I converted H(z) into a difference equation:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{az+b}{cz+d}$$

$$Y(z)(cz+d) = X(z)(az+b)$$

$$cy[n+1] + dy[n] = ax[n+1] + bx[n]$$

$$cy[n] + dy[n-1] = ax[n] + bx[n-1]$$

$$y[n] = \frac{ax[n] + bx[n-1] - dy[n-1]}{c}$$

I then implemented this iteratively to create my deemphasis output.

3.4 Low Pass Filter

To extract the mono audio signal from the FM multiplex signal, we needed to take just the frequency band $|f| \leq 15$ kHz. First I needed to convert these frequencies as follows:

$$\omega_p = \frac{2\pi f_c}{f_s} = \frac{2\pi(15000)}{256000}$$

$$\omega_s = \frac{2\pi f_c}{f_s} = \frac{2\pi(18000)}{256000}$$

I then used the same δ values as before:

$$\delta_p = 0.0575$$

$$\delta_s = 0.0033$$

This led to the following lowpass filter:

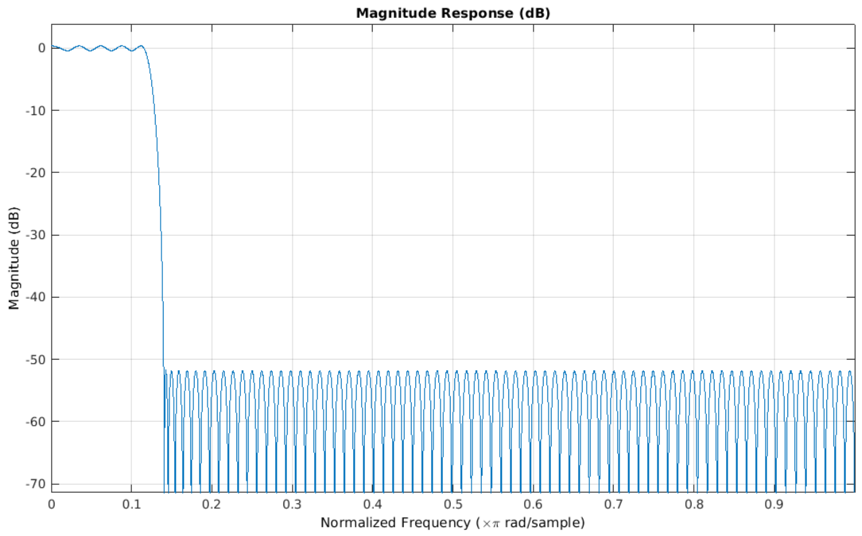


Figure 11: Magnitude Response of Low Pass Filter for $|f| \leq 15$ kHz

3.5 Decimation

To convert the sampling rate from 256 kHz to our final sampling rate of 64 kHz, we need to decimate by a factor of $256/64 = 4$. Because we already implemented a low pass filter with a 15 kHz cutoff, we know that this was $\omega_c = 15000/256000 * 2\pi = \frac{30\pi}{256}$. This is less than $\pi/4$, which is the cutoff needed to prevent aliasing from decimating by 4. This means we don't need to implement another anti-aliasing filter before decimating, so I simply took every 4th sample of the signal to decimate it by 4.

From here I was able to hear clear audio!

4 Blind Test