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.

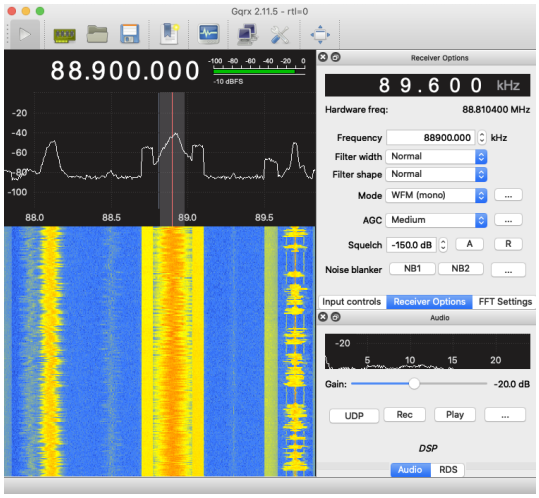

Figure 1: GQRX Settings for Sampled Data

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 data1 raw signal had the following magnitude response:

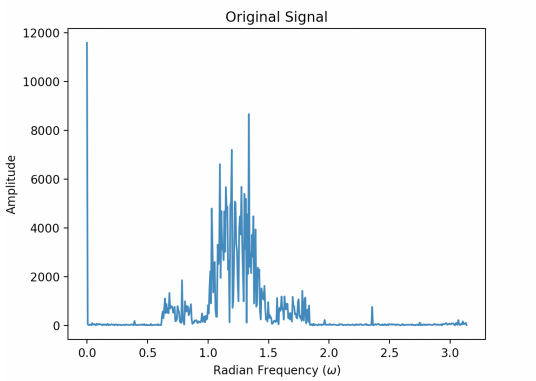

Figure 2: Magnitude Response of Original Signal

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 our hardware frequency, Ω_h, is 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 time-domain 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, where it's clear that the signal is now centered around zero:

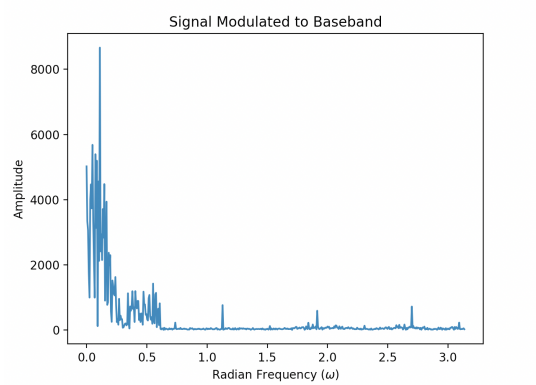

Figure 3: Magnitude Response of Signal Modulated to Baseband

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:

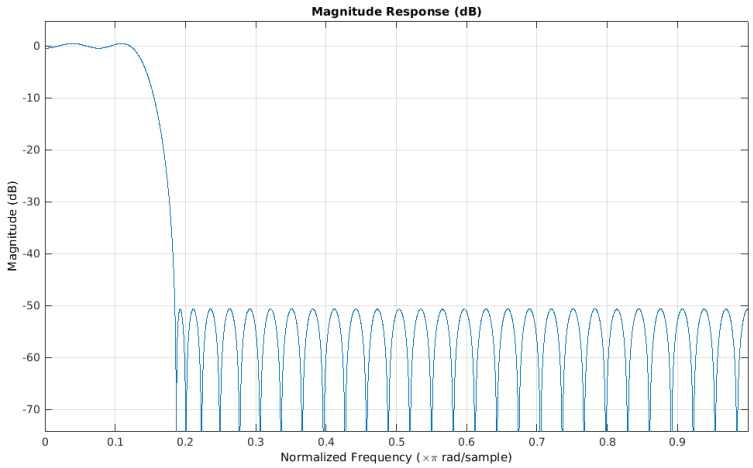

Figure 4: Magnitude Response of Low Pass Filter with $\omega_c = \pi/8$

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.

Below you can see the result of the anti-aliasing filter – the signal has been cut off as expected starting at approximately $\omega = \pi/8 \approx 0.4$, so that our decimation by 8 does not introduce aliasing among the replicas.

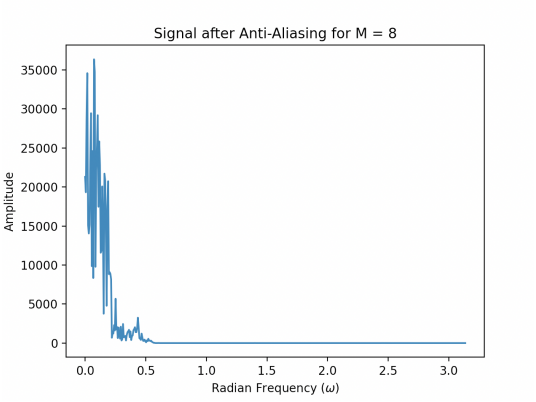

Figure 5: Magnitude Response of Signal after Anti-Aliasing Filter for Decimation by 8

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 response by 8.

This resulted in a signal magnitude response as follows:

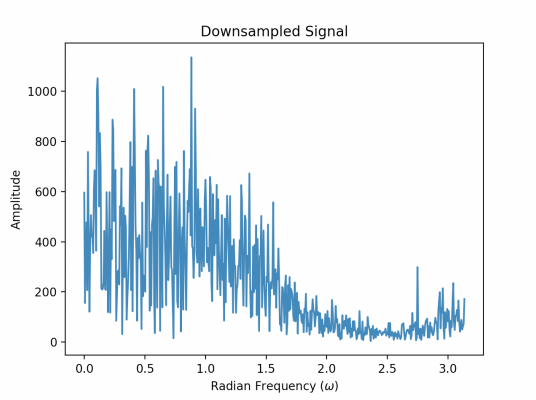

Figure 6: Magnitude Response of Downsampled Signal

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 derivative 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:

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

When we inverse transform this to the time-domain, we get:

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

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/2 would be an integer. I then sampled $h_{\text{diff}}[n]$ from 0 to 9. However, if we just truncated the filter like this, 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 $h_{\text{diff}}[n]$ with the Kaiser window, I got the following impulse response and associated magnitude response:

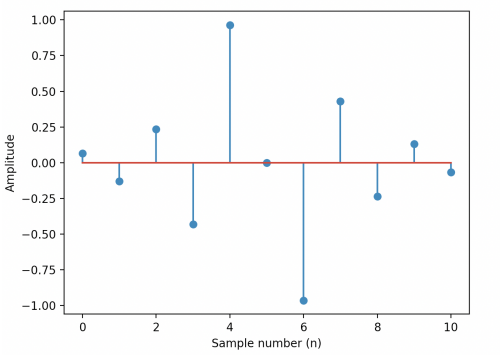

Figure 7: Impulse Response for M = 10

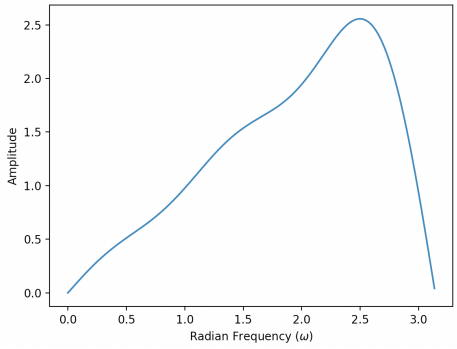

Figure 8: Magnitude Response for M = 10

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 interferes with the linearity that we wanted. To fix this, I had to choose M to be odd.

Here you can see the impulse response and associated magnitude response when I chose M = 15.

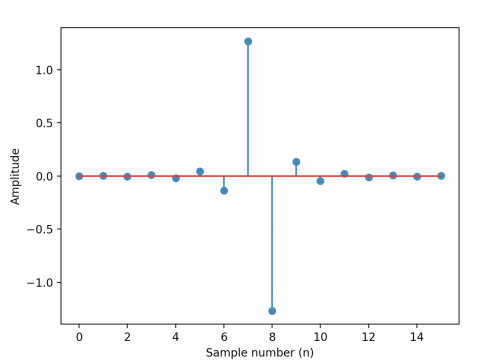

Figure 9: Impulse Response for M = 15

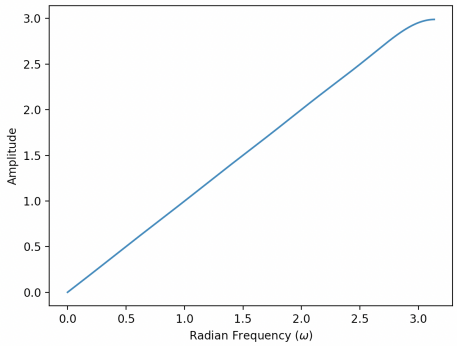

Figure 10: Magnitude Response for M = 15

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.

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 the value at each new zero point, shifted by M 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.

3.1.4 Imaginary Component

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

Below is the magnitude response of the signal after frequency discrimination, where we can see we've been able to create the FM multiplex signal with the mono audio signal, pilot tone, stereo audio signal, and message band in distinct frequency bands:

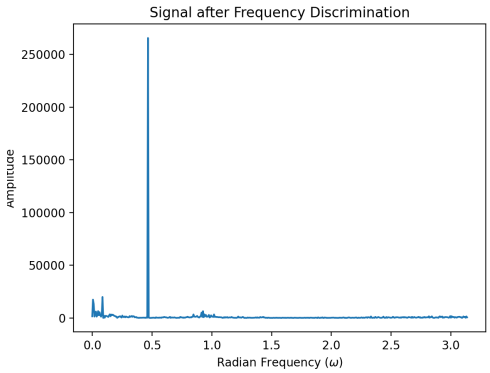

Figure 11: Magnitude Response of Signal After Frequency Discriminator

3.2 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 $H(j\Omega)$, after converting it to $H(s)$.

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

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 $(1 + 2\tau_d f_s)$, and plug in $\tau_d = 7.5 \times 10^{-5}$ and $f_s = 256000$:

$$H(z) = \frac{\frac{z+1}{1+2\tau_d f_s}}{z + \frac{1-2\tau_d f_s}{1+2\tau_d f_s}} = \frac{0.02538z + 0.02538}{z - 0.9492385}$$

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

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

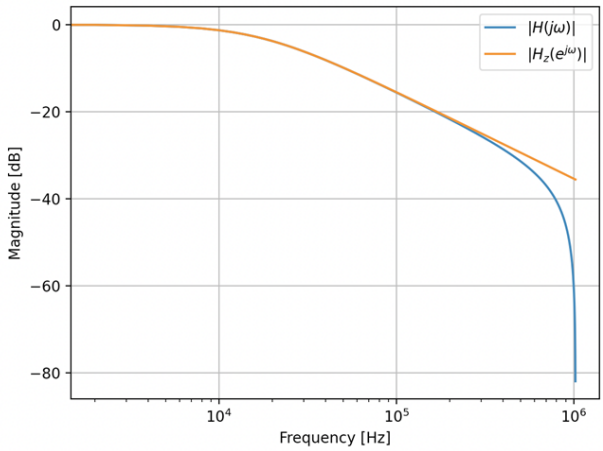

Figure 12: Magnitude Response of CT transfer function vs DT transfer function

To implement this, I converted $H(z)$ into a difference equation by following this form:

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

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

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

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

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

I then implemented this iteratively to create my deemphasis output.

Below you can see the magnitude response of the signal after this point, where we can see that, as expected, the higher frequency components now have much lower intensities than before:

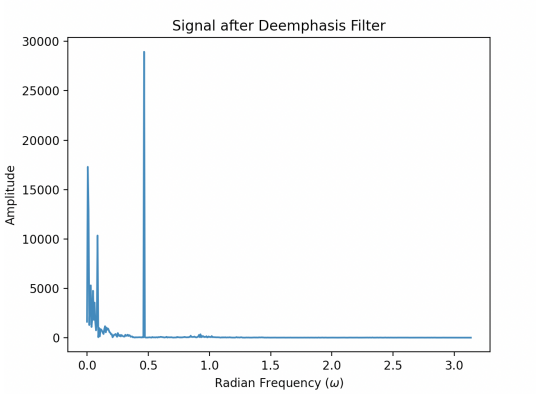

Figure 13: Magnitude Response of Signal After Deemphasis Filter

3.3 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 by using a low pass filter with $f_{\text{pass}} = 15$ kHz and $f_{\text{stop}} = 18$ kHz. First I needed to convert these frequencies as follows:

$$\omega_p = \frac{2\pi f_{\text{pass}}}{f_s} = \frac{2\pi(15000)}{256000}$$

$$\omega_s = \frac{2\pi f_{\text{stop}}}{f_s} = \frac{2\pi(18000)}{256000}$$

I then used the same δ values as before:

$$\delta_p = 0.0575$$

$$\delta_s = 0.0033$$

This led to the following lowpass filter:

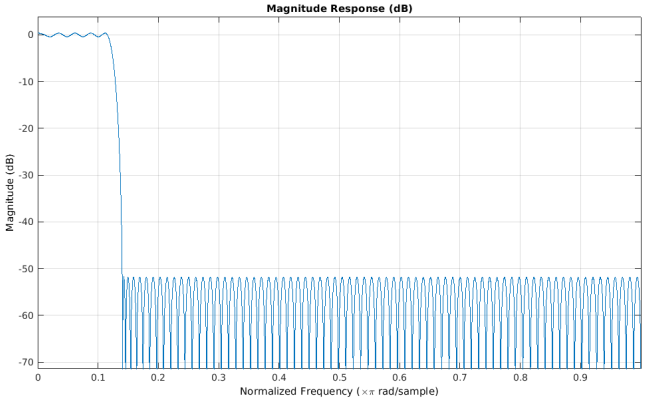

Figure 14: Magnitude Response of Low Pass Filter for $|f| \leq 15$ kHz

Below is the output of this low pass filter, where we have successfully been able to extract the mono audio signal, in the band $|f| \leq 15$ kHz or $\omega \leq 0.368$:

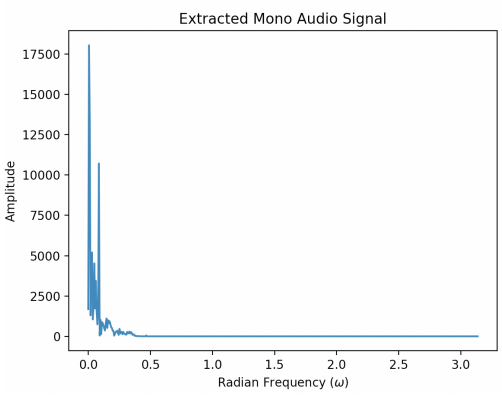

Figure 15: Magnitude Response of Signal After Mono Audio Extraction

3.4 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.

Below is the magnitude response of the signal after this decimation, now the magnitude response of our final mono audio signal. As we can see, the frequency domain has been expanded by 4 since the time-domain signal was compressed by 4:

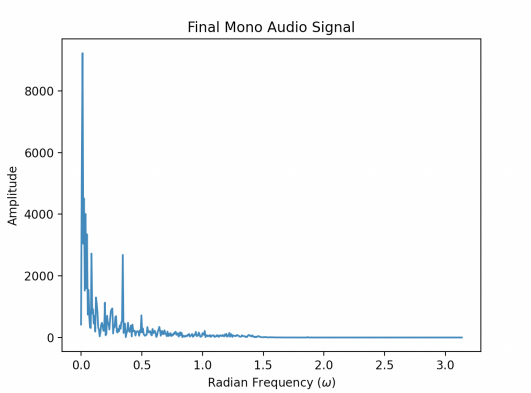

Figure 16: Magnitude Response of Final Mono Audio Signal

From here I was able to hear clear audio!

4 Blind Test

To test my mono audio extraction process, I tried both datasets 1 and 2.

Dataset 1 yields Don't Stop Believing by Journey – the audio I extracted can be found here or here: https://drive.google.com/file/d/1d9mEH587vU8Td_IjthkB2LpUzaGABf9C/view?usp=sharing.

Dataset 2 yields a commercial that starts with "...truly understand your individual needs, and working with you to find solutions that set you up for success. See how Cambridge Savings Bank can help you get home, at cambridgesavings.com." The audio I extracted can be found here or here: https://drive.google.com/file/d/1JngJKiPSLzvQ2fn32tE1vDP3mmQFVqe2/view?usp=sharing.

5 Code

Below I've included the code I used to extract the audio signal.

```
import scipy
import scipy.signal
import matplotlib.pyplot as plt
from math import e, sin, pi, cos
import math
import dtsp
import numpy as np
from lib6003.audio import wav_read, wav_write

class AudioExtractor:
   def __init__(self, fc, fs, hardwareFreq, fileName):
      self.fc = fc
      self.fs = fs
      self.fsnew = 256000 #Hz
      self.fh = hardwareFreq
      self.fileName = fileName
      self.signal = self.loadData()
      self.M = int(self.fs/self.fsnew)

   def process(self):
      """
      Does entire filtering process to extract mono audio signal
      """
      self.modulateToBase()
      self.downsample()
      self.FMDemodulation()

   def loadData(self):
      """
      Load raw data from file
      """
      signal = scipy.fromfile(open(self.fileName + '.raw'), dtype=scipy.complex64)
      signal = np.array(signal)
      return signal

   def modulateToBase(self):
      """
      Part 2a: Modulate to baseband
      """
      errorFreq = self.fc - self.fh
      w0 = errorFreq*2*pi/self.fs
      basebandShift = np.array([e**(-1j*w0*i) for i in range(len(self.signal))]) #shift to modulate
          down
      shiftedToBase = self.signal*basebandShift #multiply signal with shift
      self.signal = shiftedToBase

   def downsample(self):
      """
      Part 2b: Downsample by fs/fsNew
      """
      #LPF Parameters
      transitionBandWidth = 0.06
      wp = 1/self.M
      ws = wp+transitionBandWidth
      dpass = 0.0575
```

```
    dstop = 0.0033

    LPF = self.createLPF(wp, ws, dpass, dstop)
    self.convolve(LPF)
    self.decimate(self.M)

def createLPF(self, wp, ws, dpass, dstop):
    """
    Returns low pass filter given the wp, ws, dpass, and dstop parameters
    """
    numtaps, bands, amps, weights = dtsp.remezord([wp/2.0, ws/2.0], [1, 0], [dpass,dstop], Hz=1.0)
    bands *= 2.0  # above function outputs frequencies normalized from 0.0 to 0.5
    b = scipy.signal.remez(numtaps, bands, amps, weights, Hz=2.0)
    return b

def decimate(self, M):
    """
    Returns a signal of every Mth sample of the signal
    """
    self.signal = np.array([self.signal[i] for i in range(0, len(self.signal), M)])
    self.N = len(self.signal)

def convolve(self, filterList):
    """
    Filters signal using convolution
    """
    self.signal = scipy.signal.convolve(self.signal, filterList)

def FMDemodulation(self):
    """
    Part 3
    """
    self.frequencyDiscriminator()
    self.deemphasisFilter()

    #LPF Parameters
    CT_PB = 15000 #Hz
    CT_SB = 18000 #Hz
    DT_PB = CT_PB/self.fsnew*2
    DT_SB = CT_SB/self.fsnew*2
    dpass = 0.0575
    dstop = 0.0033

    LPF = self.createLPF(DT_PB, DT_SB, dpass, dstop)
    self.convolve(LPF)
    self.decimate(4) #Final decimation to 64 kHz

def frequencyDiscriminator(self):
    self.limiter()
    self.DTDifferentiator()
    self.toImag()

def limiter(self):
    """
    Normalizes all signal sample magnitudes to either 1 or -1
    """
    self.signal = np.array([self.signal[i]/abs(self.signal[i]) for i in range(len(self.signal))])

def DTDifferentiator(self):
```

```
    """
    Returns limited signal after DT differentiation multiplied by a
    shifted conjugated version of the limited signal
    """
    M_filter = 15
    diff = self.generateDifferentiator(M_filter)
    shiftedConj = self.shiftedConj(M_filter)
    shiftedConj = np.concatenate((shiftedConj, np.array([0 for i in range(M_filter)]))) #extend
        shifted version with zeros for convolution
    self.convolve(diff)
    self.signal = self.signal*shiftedConj

def generateDifferentiator(self, M_filter):
    """
    Generates DT differentiator filter of length M_filter and windowed
    with a Kaiser window having alpha = M_filter+1 and beta = 2.4
    """
    beta = 2.4
    hdiff_truncated = np.array([cos(pi*(n-M_filter/2))/(n-M_filter/2) -
        sin(pi*(n-M_filter/2))/(pi*(n-M_filter/2)**2) if n != M_filter/2 else 0 for n in
        range(M_filter+1)])
    kaiser = scipy.signal.kaiser(M_filter+1, beta)
    windowed = hdiff_truncated*kaiser
    return windowed

def shiftedConj(self, M_filter):
    """
    Creates shifted conjugate of signal by expanding the signal by 2,
    interpolating, shifting by M, and downsampling and taking the conjugate
    which results in a shift by M/2
    """
    L = 2
    signalToExpand = self.signal.copy()
    signalExpanded = self.expand(signalToExpand, L)
    for i in range(1, len(signalExpanded)-1):
       if i%2 == 1:
          signalExpanded[i] = 1/2*(signalExpanded[i-1]+signalExpanded[i+1])
    shiftedConj = np.array([0 for i in range(int(M_filter))] +
        [signalExpanded[i-int(M_filter)].conjugate() for i in range(int(M_filter),
        len(signalExpanded))])
    return np.array([shiftedConj[i] for i in range(0, len(shiftedConj), L)])

def expand(self, filterList, L):
    """
    Returns signal expanded by a factor of L with zeros between samples
    """
    expanded = []
    for i in filterList:
       expanded.append(i)
       expanded.append(0)
    return expanded

def toImag(self):
    """
    Takes imaginary part of signal
    """
    self.signal = np.array([self.signal[i].imag for i in range(len(self.signal))])

def deemphasisFilter(self):
```

```
        """
        Returns signal after deemphasis filter
        """
        tau = 7.5e-5 #seconds
        num = [1] #H = 1/(1+s*tau)
        den = [tau, 1]

        filtz = scipy.signal.dlti(*scipy.signal.bilinear(num, den, self.fsnew))

        a = filtz.num[0]
        b = filtz.num[1]
        c = filtz.den[0]
        d = filtz.den[1]
        result = [a*self.signal[0]/c] #start new signal with a/c*signal[0]
        for n in range(1, len(self.signal)):
            result.append(1/c*(a*self.signal[n] + b*self.signal[n-1] - d*result[n-1]))
        self.signal = np.array(result)

    def plotFFT(self, title):
        """
        Plots magnitude response of signal
        """
        w, H = scipy.signal.freqz(self.signal)
        plt.plot(w, np.abs(H))
        plt.title(title)
        plt.xlabel("Radian Frequency ($\omega$)")
        plt.ylabel("Amplitude")
        plt.show()

#Variables
fs = 2048000 #Hz
fc = 1.007e8 #Hz
hardwareFreq = 1.003e+8 #Hz 88.810400 MHz
fileName = "data1"
fsNew = 256000 #Hz



audioextract = AudioExtractor(fc, fs, hardwareFreq, fileName)
audioextract.process()
wav_write(audioextract.signal, 64000, "data2result.wav")
```