Lab 1: Working with IQ data in Python

Aims

After completing this lab assignment, you should:

- Know how to write basic Python code.
- Know how to use libraries for signal processing and visualization in Python, including scipy and matplotlib, to work with IQ signals.
- Have a basic implementation of an IQ demodulator for FM radio, which you may use to complete future homework assignments.

What to submit

For this lab, you should submit the following:

- A written report in PDF format.
- A single Python script that reads in a data capture file and writes an audio file.

The written report

You should submit a brief report explaining the signal processing chain used to demodulate the FM radio signal. At each stage of the signal processing chain at which the data undergoes a significant transformation (i.e. stages in the block diagram where a new label is assigned to the signal, e.g. x1, x2, etc.), explain what the transformation is and how the signal is affected by it. Make sure to note instances where the sample rate of the signal changes (length of array changes in Python) and when the signal changes from complex to real, for example.

Use relevant plots and visualizations - spectograms, PSD plots, scatter plots of the complex IQ plane, images of filter frequency response, etc. - to support your explanation. You should also include relevant plots showing the characteristics of important external signals you use, such as your digital oscillator and filters.

At each stage, choose the appropriate plots to highlight important features of the signal. For example, you should not show a scatter plot of the complex IQ plane at all of the stages where this plot does not show any meaningful pattern, but you should show this plot at a stage where the scatter plot looks like an FM signal. You should use appropriate scales and axis limits to highlight the signal characteristics of interest.

Your report should demonstrate a good understanding of the signal processing involved in demodulating the FM signal. Please do not just create every kind of plot for every data processing stage and dump it into a report, as this just demonstrates that you know how to glue code snippets together and make it work - an important skill, but not what I am grading these reports on.

Submit this report as a PDF document.

The Python script

To check your work, I will run your script on an FM data capture file, changing only the path to the data file to reflect its location on my system. Then I will listen to the audio file that it writes. Your script should run non-interactively (i.e., I can "run as script" and do not have to run one section at a time.)

Although you will need to write code to create plots and visualizations for your report, you don't have to include this code in the script you submit to me.

Submit this report as a Python script (.py extension). Do not copy and paste your code into a Word document or other format - I want a script that I can execute as-is.

Other information and policies

This lab is an individual effort. You may not share your code with another student. You are strongly encouraged to use existing answers on sites like Stack Overflow to help you debug your work, but you may not **post** a question asking people on the Internet to do your work for you.

If you have a question or problem, please post it on the designated forum on NYU Classes for Lab 1. When you post a question,

- Do not attach all of your code - just the relevant snippet that you think is the source of the problem.
- Don't say "It doesn't work" - explain what you expect to happen, what happens instead, and why you think this is not the correct behavior. If there is an error, give the text of the error message.
- Explain what you have already tried doing to fix the problem.

Some relevant details on grading criteria for lab reports are here.

Important information about academic integrity and avoiding plagiarism is available here.

Getting up and running with Python

For this lab, you'll need a Python installation that includes the major scientific computing libraries: scipy, numpy, and matplotlib. I highly recommend Pyzo. It's available for Windows, Mac, and Linux, includes the necessary modules, and comes with a nice IDE.

Once you are comfortable with your environment, it's time to learn the basics of Python. If you have used any other scripting language, including Matlab, you will find it very easy.

If you don't have previous experience with Python, run through these tutorials until you are comfortable with the basic usage of the language. (Don't spend a lot of time learning every single detail.)

- First steps
- Basic types
- Control flow
- Defining functions
- Reusing code: scripts and modules

Using signal processing libraries with IQ data

Before we start working with our actual FM radio signal, let's practice reading in data stored in a file and working with it in Python as complex samples. We'll use some standard scientific computing modules for this.

If you are using Pyzo, you should already have the modules we need - numpy, matplotlib, and scipy - installed. To use these modules in your code, run the lines:

```
# includes core parts of numpy, matplotlib
import matplotlib.pyplot as plt
import numpy as np
# include scipy's signal processing functions
import scipy.signal as signal
```

Note the "as" - this changes the way we call functions from the library. If we would just run "import numpy", for example, then to run a numpy function - for example fromfile - we would have to use "numpy.fromfile". When we "import numpy as np", then we can use the shorthand "np.fromfile". This is mostly useful when we want to use multiple libraries that have defined the same function. However, it can be confusing when you try looking for answers and the code you find online won't run verbatim in your implementation because of differences in namespaces. Be aware of this issue when working on this lab.

Now, we'll load in some IQ data from a binary data file. Binary data files don't contain any information about the data in them (the size of a sample of data, whether it is real or complex, and what the sampling rate of the data was, for example). To use data from a file properly, you need to know all this in advance and use this knowledge when you load in the data.

For example, here is a file containing data stored as 32-bit float values, with the real and imaginary parts of each sample interleaved (i.e. if a sample is 3.8+j2.1, it will be stored as a 32-bit 3.8 followed by a 32-bit 2.1). Download and try loading in this data and viewing its spectogram:

```
# practice reading in complex values stored in a file
# Read in data that has been stored as raw I/Q interleaved 32-bit fl
oat samples
dat = np.fromfile("iqsamples.float32", dtype="float32")
# Look at the data. Is it complex?
dat

# Turn the interleaved I and Q samples into complex values
# the syntax "dat[0::2]" means "every 2nd value in
# array dat starting from the 0th until the end"
dat = dat[0::2] + 1j*dat[1::2]

# Note: a quicker way to turn the interleaved I and Q samples  into
 complex values
# (courtesy of http://stackoverflow.com/a/5658446/) would be:
# dat = dat.astype(np.float32).view(np.complex64)

# Now look at the data again. Verify that it is complex:
dat

# Plot the spectogram of this data
plt.specgram(dat, NFFT=1024, Fs=1000000)
plt.title("PSD of 'signal' loaded from file")
plt.xlabel("Time")
plt.ylabel("Frequency")
plt.show()  # if you've done this right, you should see a fun surpri
se here!

# Let's try a PSD plot of the same data
plt.psd(dat, NFFT=1024, Fs=1000000)
plt.title("PSD of 'signal' loaded from file")
plt.show()


# And let's look at it on the complex plan
# Note that showing *every* data point would be time- and processing
-intensive
# so we'll just show a few
plt.scatter(np.real(dat[0:100000]), np.imag(dat[0:100000]))
plt.title("Constellation of the 'signal' loaded from file")
plt.show()
```

Note that if you load in this file with the wrong data type, or without turning the interleaved data into complex values, you'll see a different spectogram. Try it yourself and see what happens.

Now let's try some other signal processing operations.

```
Fs = 1000000 # define sampling rate

# Let's try a frequency translation. For a complex signal,
# frequency translation is achieved with multiplication by a complex
 exponential

# To mix the data down, generate a complex exponential
# with phase -f_shift/Fs
fc = np.exp(-1.0j*2.0*np.pi* 50000/Fs*np.arange(len(dat)))
# Try plotting this complex exponential with a scatter plot of the c
omplex plan -
# what do you expect it to look like?
y = dat * fc

# How has our PSD changed?

plt.psd(dat, NFFT=1024, Fs=1000000, color="blue")  # original
plt.psd(y, NFFT=1024, Fs=1000000, color="green")  # translated
plt.title("PSD of 'signal' loaded from file")
plt.show()


# What happens when you filter your data with a lowpass filter?
f_bw = 150000
Fs  = 1000000
```

```
n_taps = 64
lpf = signal.remez(n_taps, [0, f_bw, f_bw+(Fs/2-f_bw)/4, Fs/2], [1,0
], Hz=Fs)

# Plot your filter's frequency response:
w, h = signal.freqz(lpf)
plt.plot(w, 20 * np.log10(abs(h)))
plt.xscale('log')
plt.title('Filter frequency response')
plt.xlabel('Frequency')
plt.ylabel('Amplitude')
plt.margins(0, 0.1)
plt.grid(which='both', axis='both')
plt.show()

y = signal.lfilter(lpf, 1.0, dat)

# How has our PSD changed?

plt.psd(dat, NFFT=1024, Fs=1000000, color="blue")  # original
plt.psd(y, NFFT=1024, Fs=1000000, color="green")  # filtered
plt.title("PSD of 'signal' loaded from file")
plt.show()

# Let's try decimating following a lowpass filter

# Figure out our best decimation rate
dec_rate = int(Fs / f_bw)
z = signal.decimate(y, dec_rate)
Fs_z = Fs/dec_rate

# New PSD - now with new Fs
plt.psd(z, NFFT=1024, Fs=Fs_z, color="blue")
plt.show()
```

Implementing an FM demodulator

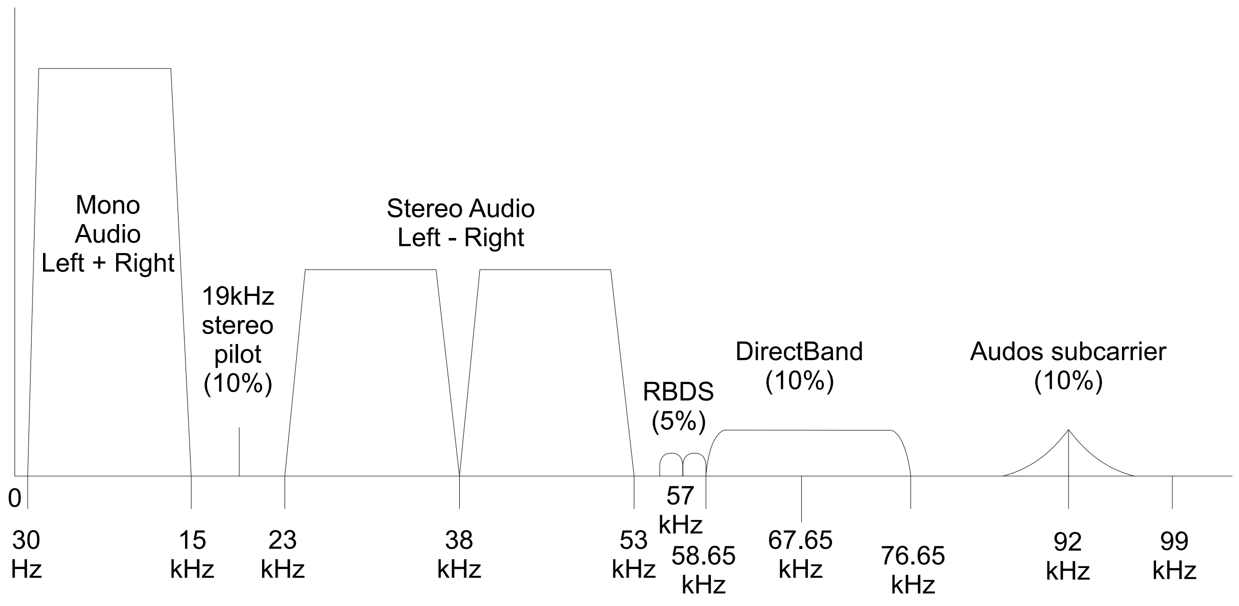
We know that an FM signal modulates the message signal $m(t)$ to the carrier signal such that the derivative of the phase deviation, $\frac{d\phi(t)}{dt}$, is proportional to the message:

$$x_{FM}(t) = A_c \cos\left(2\pi f_c t + \phi(t)\right) = A_c \cos\left(2\pi f_c t + 2\pi f_\Delta \int_0^t m(\alpha)\, d\alpha\right)$$

So recovering the message from the FM signal should be a simple matter of calculating the rate of change of the phase of the received signal. (For this stage, we'll use a kind of frequency discriminator, which we'll talk about in a minute.)

In practice, things are a little more complicated. First of all, the captured IQ data we will be working with was sampled at a rate of 1140000 Hz, at a center frequency offset from the signal of interest by 250000 Hz. So we will need to move that radio channel down to baseband (center it at 0 Hz) and then filter and decimate it to focus on just the FM broadcast signal (which has a 200 kHz bandwidth.) That 200 kHz baseband signal is what we will pass to the frequency discriminator.

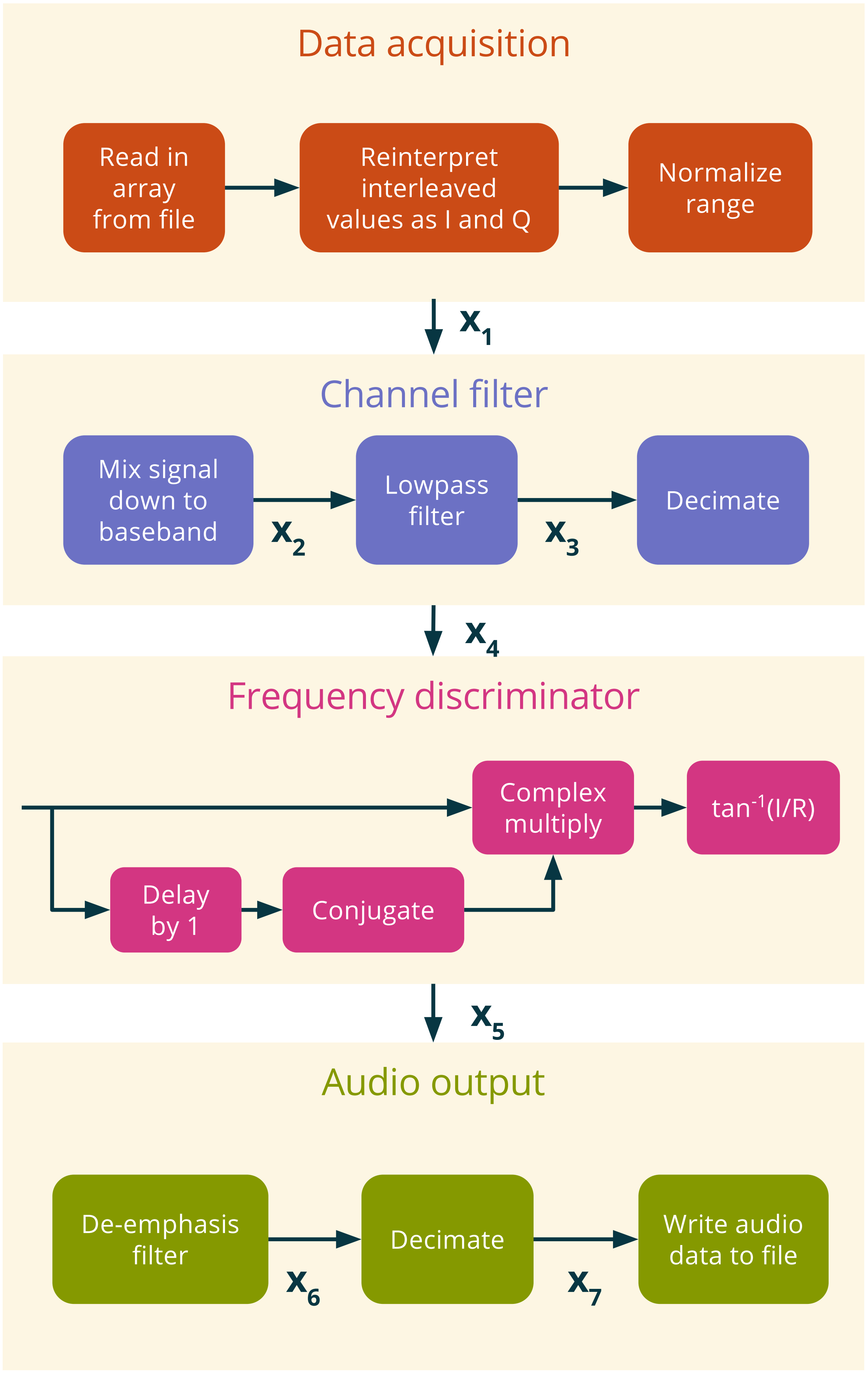
FM radio broadcast signals contain several sub-signals: mono audio, stereo audio, digital data, and more. After the frequency discrimination stage, we should be able to create an image of our 200 kHz broadcast signal just like the one below, and identify which carriers are present in your signal.)



Typical baseband spectrum of an FM broadcast signal. Image from Wikimedia Commons, is in the public domain.

We're only going to work with the mono audio channel. We need to pass it through a de-emphasis filter (this is to compensate for an analogous emphasis filter that was applied to the data at the transmitter.) We'll also need to decimate the signal down to the standard audio sampling rate (roughly 44.1-48 kHz). Finally, we are ready to write our audio data to a file and play it back.

A block diagram of the implementation you are going to create follows:


Data acquisition
Read in array from file
Reinterpret interleaved values as I and Q
Normalize range
x_1
Channel filter
Mix signal down to baseband
x_2
Lowpass filter
x_3
Decimate
x_4
Frequency discriminator
Complex multiply
$\tan^{-1}(I/R)$
Delay by 1
Conjugate
x_5
Audio output
De-emphasis filter
x_6
Decimate
x_7
Write audio data to file

Block diagram of our software FM demodulator.

My implementation is roughly 30 lines of (non-whitespace, non-comment) Python code.

For your convenience, here are some snippets of code for special subsystems as well as information about the data files.

Data files

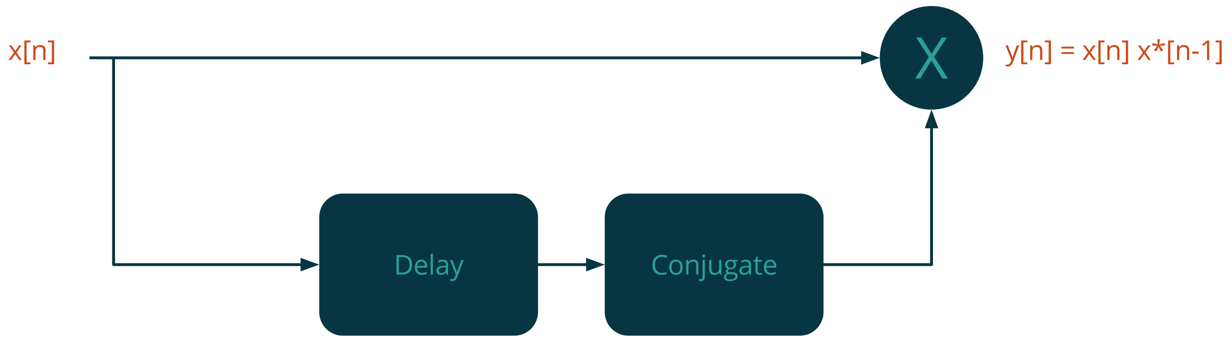
You can download the FM data capture files here.

Each of these files contain 8192000 samples of a slice of FM radio spectrum, sampled at 1140000 Hz. The 200 kHz FM radio channel we are trying to demodulate is located at an offset of 250000 Hz from the center frequency.

The data is stored as interleaved unsigned 8-bit samples. (That's the **uint8** data type.)

Frequency discriminator

We'll use a kind of frequency discriminator called a *polar discriminator*. A polar discriminator measures the phase difference between consecutive samples of a complex-sampled FM signal. More specifically, it takes successive complex-valued samples and multiplies the new sample by the conjugate of the old sample. Then it takes the angle of this complex value. This turns out to be the instantaneous frequency of the sampled FM signal.



Block diagram of the frequency discriminator.

```
# Given a signal x (in a numpy array)
y = x[1:] * np.conj(x[:-1])
z = np.angle(y)
```

De-emphasis filter

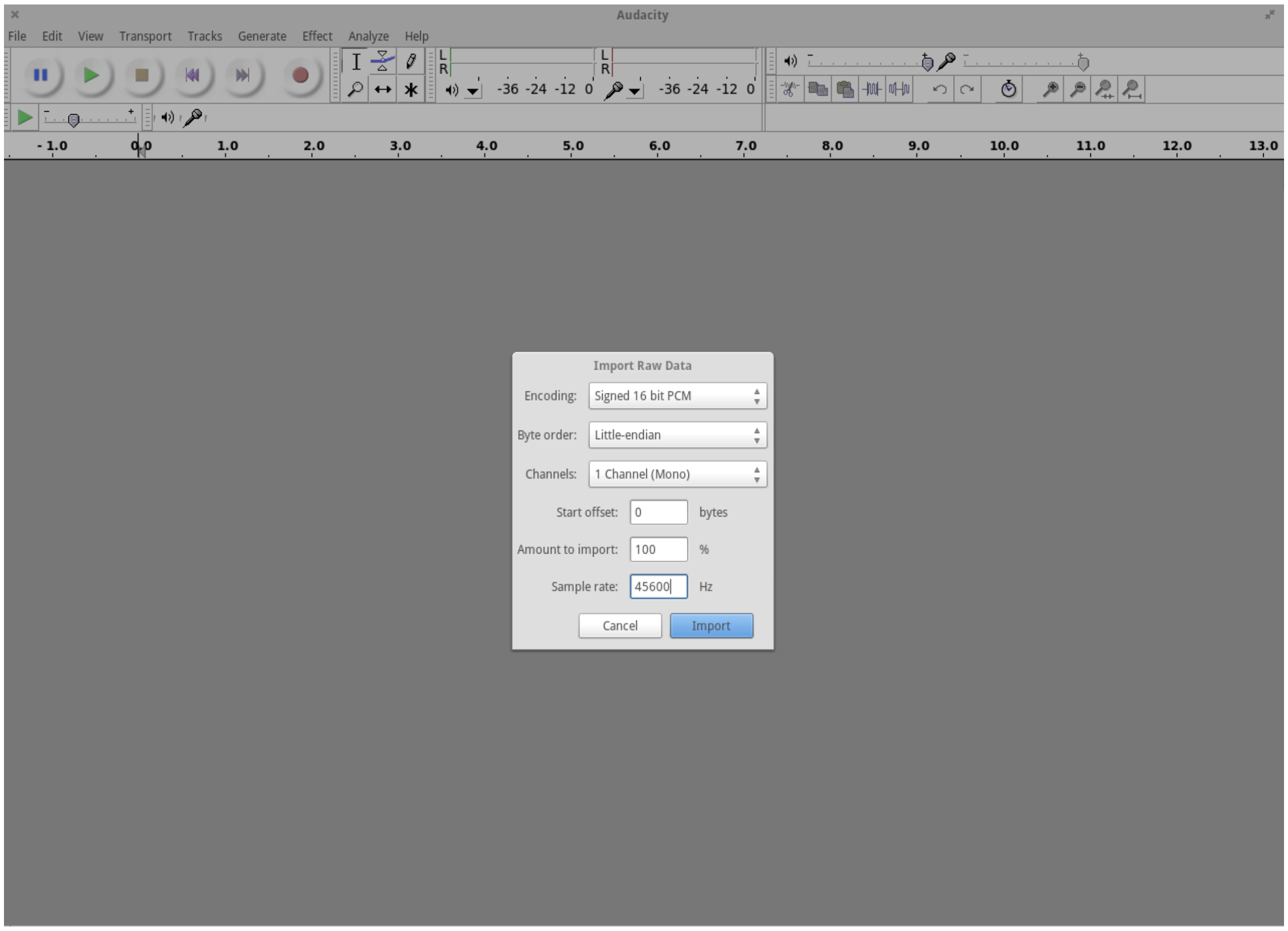
```
# The de-emphasis filter
# Given a signal 'x5' (in a numpy array) with sampling rate Fs_y
d = Fs_y * 75e-6   # Calculate the # of samples to hit the -3dB poin
t
x = np.exp(-1/d)   # Calculate the decay between each sample
b = [1-x]          # Create the filter coefficients
a = [1,-x]
x6 = signal.lfilter(b,a,x5)
```

Writing audio to file

Note that raw audio should be sampled at around 44.1-48 kHz.

```
# Given a signal x (in a numpy array)
x *= 10000 / np.max(np.abs(x))              # scale so it's audible
x.astype("int16").tofile("wbfm-mono.raw")    # write to file
```

You can use Audacity (available for Windows, Linux, and Mac) to play back your decoded audio file. From the "File" menu, choose "Import > Raw Data" and then make sure to use the appropriate settings (set the sample rate to whatever rate *your* audio data is sampled at):


Audacity
File Edit View Transport Tracks Generate Effect Analyze Help
Import Raw Data
Cancel
Import

Encoding: Signed 16 bit PCM
Byte order: Little-endian
Channels: 1 Channel (Mono)
Start offset: 0 bytes
Amount to import: 100 %
Sample rate: 45600 Hz