PSET ∞: Speak Spectrogram To Me

Ishira Fernando, Emily Huang

June 7, 2022

1 Part 1: Building the Fast Fourier Transform

We already saw some basic application of the Fourier Transform in PSET 8. However, we did take the easy route by using numpy and MATLABs native implementation of the Fast Fourier Transform (FFT). What do these algorithms actually do? What benefit does the FFT actually give us?

(A) **First Steps:** We already learnt how to construct the Fourier Matrix in lecture, and how it relates the vector we wish to transform into the Fourier domain. To begin let's consider a small 2D dimensional vector $v = [v_1, v_2]$. Write out the Fourier Matrix and calculate the transformed vector. Feel free to leave both in terms of complex exponentials.

Here are a few important things to remember.

(a) The Fourier Transform of a vector v, $\mathcal{F}(v)$ is defined as $\mathcal{F}(v) = Mv$.

(b) Here M is the Fourier Matrix and the j, k'th entry is defined as $M^{j,k} = (e^{\frac{-2\pi i}{n}})^{jk}$.

Answer: The Fourier Matrix is:

$$\begin{pmatrix} 1 & 1 \\ 1 & e^{-2\pi i/n} \end{pmatrix}$$

Then, the transformed vector is:

$$\begin{pmatrix} v_1 + v_2 \\ v_1 + e^{-2\pi i/n} v_2 \end{pmatrix}$$

(B) **Autobots Roll Out:** Let's begin the process by implementing a function that builds the Fourier Transform of a n dimensional vector v and returns it.

We recommend using numpy and MATLAB's vectorized functionality to build the Fourier Matrix rather than using for loops to make your implementation faster.

Answer:

```
def my_FT(x):
    x = np.asarray(x)
    n = x.shape[0]
    j = np.arange(n)
    k = j.reshape((n, 1))
    M = (np.exp(-2j * np.pi/ n))**(j*k)
    return np.dot(M, x)
```

(C) **Gotta Go Fast:** Next let's look at speeding up the Fourier Transform. For the purposes of this mini project we will stick with implementing the Fast Fourier Transform, for vectors of length 2^n where n is a positive integer. The non-power-of-2 case is a little too messy for the purposes of the miniproject.

The Pseudocode for the FFT is given below:

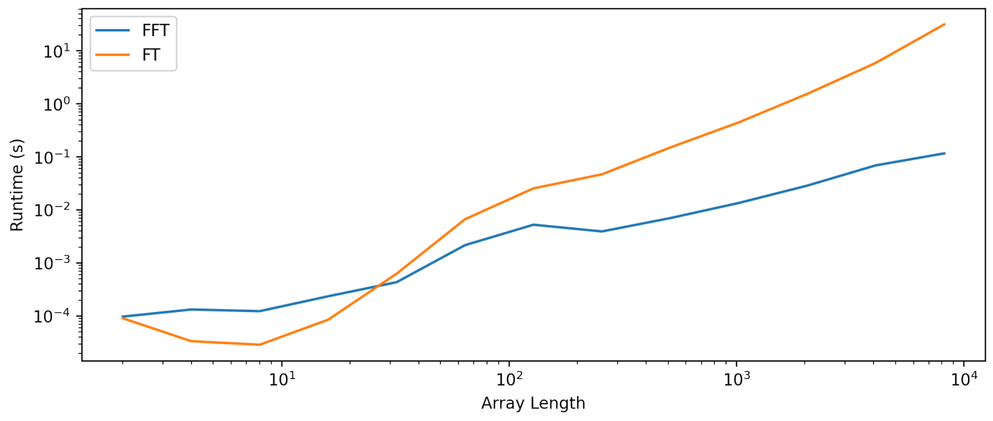
```
func FFT <- (x):
    N = length(x)
    if N <= 2:
        return slow_fourier_transform(x)
    else:
        X_even <- FFT(all even indexed values of x)
        X_odd <- FFT(all odd indexed values of x)
        values = range(N)
        factor = np.exp(-2j * np.pi * values / N)
        first_half = X_even + factor[:(N // 2)] * X_odd
        second_half = X_even + factor[(N // 2):] * X_odd
        return concatenate(first_half, second_half)
```

Answer:

```
def my_FFT(x):
    N = int(x.shape[0])
    if N <= 2:  # switch to DFT
        return my_FT(x)
    else:
        X_even = my_FFT(x[::2])
        X_odd = my_FFT(x[1::2])
        factor = np.exp(-2j * np.pi * np.arange(N) / N)
        return np.concatenate([X_even + factor[:(N // 2)] * X_odd,
                               X_even + factor[(N // 2):] * X_odd])
```

(D) **Time to Time:** Run both the Algorithms from Part B and Part B on randomly generated arrays of lengths of powers of 2 between $[2, 2^{14}]$ and record how long it takes for each one to complete. Plot the execution time vs array length for each and comment on the trends.

Answer:



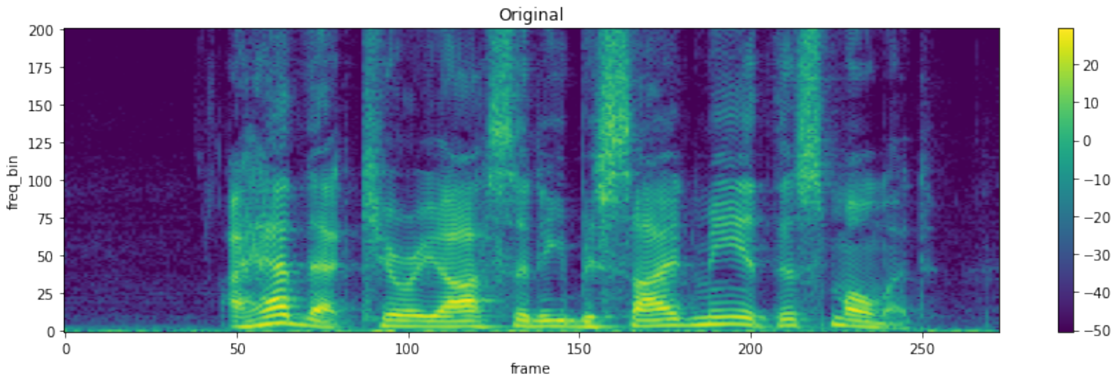
The Fast Fourier Transform outperforms the Discrete Fourier Transform for arrays with length greater than about 32 (OfM: 10). Thus for arrays shorter than 32, it makes sense to switch back to the slow DFT, as the overhead involved in constructing the separate halves and adding multiple stack frames is larger than that of just building the entire transformation matrix itself.

2 Part 2: The *Spectrogram*

Now that we have in hand the Fourier transform: a tool to rapidly transition from the time domain to the frequency domain, we can now begin the process of constructing a spectrogram. Due to the difficulty in handling audio vectors with lengths that are not powers of 2, we have pre-trimmed the audio you will use in this section to a power of 2. However, feel free to use the numpy/MATLAB FFT implementations if you would like.

Fourier Transforms enable many an interesting analysis of signals, but by far one of the most important uses of it the last few decades has been the Spectrogram. The spectrogram of an audio enables us to visualize it in a manner that is substantially more useful than waveforms, or even just the Fourier transform of a waveform.

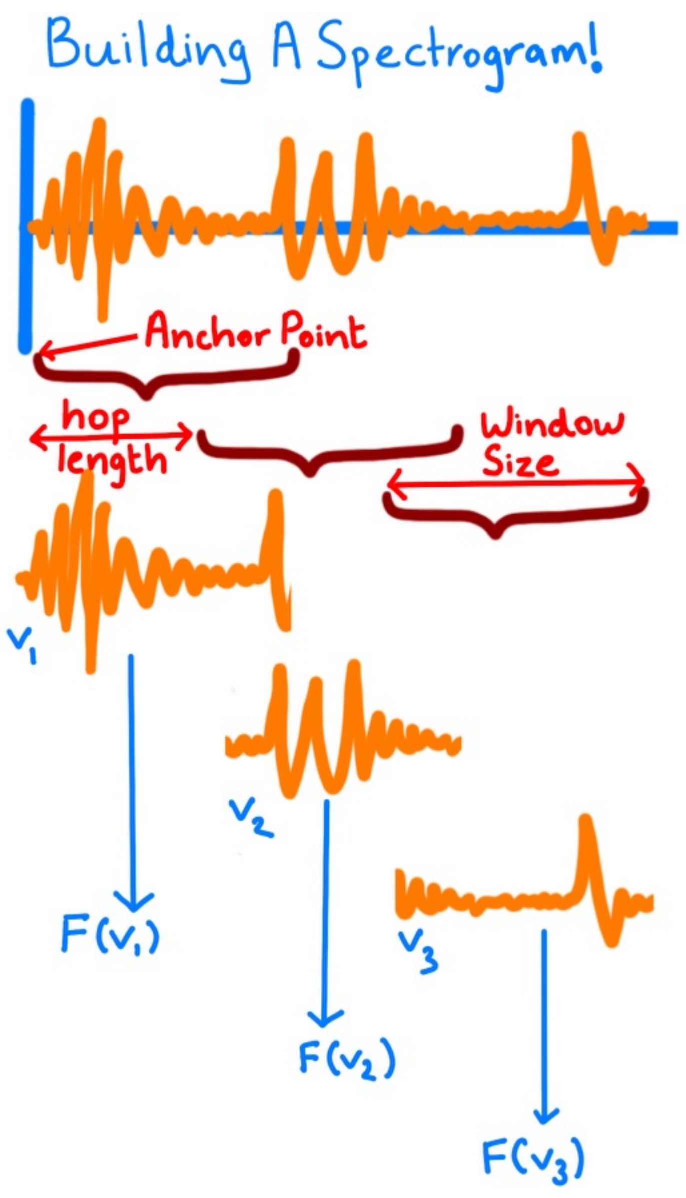
Here is the Spectrogram of a person speaking. Note the axes, and the varying intensity of the heatmap.



The x-axis as usual is just the time frame of the audio, but the y-axis is in the frequency domain (sound familiar?), rather than being an amplitude. Instead the intensity of the audio at each frequency is encoded by the intensity of the color in the heatmap. Now instead of having just amplitude and time encoded, the visualization has time, frequency and intensity of frequency encoded! You'll find out why this kind of visualization can be useful in Part 3. For now, let's figure out how to get there!

(A) **Building Intuition:** The Fourier Transform already provides us with a means to convert raw audio, which is just a 1D array of amplitudes to the frequency domain. However, simply applying the DFT/FFT to the entire audio will tell us which frequencies are present and at what intensities, but without any concept of time. How do we resolve this?

A simple insight is that we could potentially just apply the Fourier transform to a sliding window along the audio signal. This means that at each time-step, we can find the frequencies present and their intensities, and overlay the time steps to gain information about the frequencies along the time domain. The figure below provides some intuition on the process.



When using a sliding window to extract frequency components, we have to decide what window size to use and whether or not there should be overlap between the windows. The choice is not always clear cut and there are some tradeoffs involved.

i What is the benefit of using a shorter window over a larger window or vice versa? [*Hint: the length of the transformed vector given by the Fourier Transform is equal to the length of the vector to be transformed.*]

Answer: A shorter window will result in quicker transforms (less time taken to get the transform of each window). It will also provide less frequency smearing as the frequencies per time step will be generated from a sample that is very close to the starting point of the time step, so the frequencies calculated will be a closer representation of the true frequencies present. A longer window will result in higher frequency resolution as the Fourier transform of a larger vector has more frequency bins.

ii Why would it make sense to use an overlapping window over distinct windows when applying the Fourier Transform?

Answer: It enables us to use larger windows (for more frequency resolution) without reducing the temporal resolution of the spectrogram as you can have fewer windows when using distinct slicing over overlapping slicing. Furthermore, overlapping time windows smooth out the frequency transitions from time step to time-step. Distinct windows may result in spectral lines (lines in the spectrogram) that are not smooth and are harder to interpret.

(B) **Baby's First Spectrogram:** Having built some intuition on what is at play here we can go ahead and build our very first spectrogram. Write a function that takes in an input audio signal (1D Array) and returns the spectrogram of that audio signal. Use that function to plot the spectrogram of `sample.wav`. Pseudocode for building the spectrogram is below.

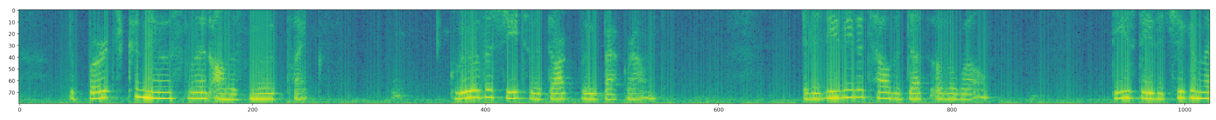
```
func sliding_fourier <- audio_array, hop_size, window_size, freq_cutoff:
    windows <- get_windows(audio_array, hop_size, window_size)
    spectrogram <- zero_matrix((num_windows, freq_cutoff))
    for window in windows:
        fourier_coefs <- FFT(window)
        # getting the one side fourier spectrum
        fourier_coefs <- 2 * fourier_coefs[-freq_cutoff:]
        spectrogram[window_number, :] <- real_value(fourier_coefs)
    return log(spectrogram)
```

Use a `window_size` of 512, a `freq_cutoff` of 80 and try varying hop sizes (powers of 2, between 2 and 1024). Which hop size seems to provide you with a fair trade off between temporal resolution and runtime? Plot the final spectrogram.

Note that we haven't converted the frequency bins output by the Fourier Transform to an actual frequency (Hz). This is okay as for this problem set we won't cares as much about the actual frequencies as we do the shape of the spectrogram.

Answer:

With our homemade implementation of the FFT we found that a hop size of 64 provided the highest temporal resolution, while also having a tolerable runtime. If we used a library implementation this could possibly have been lower.



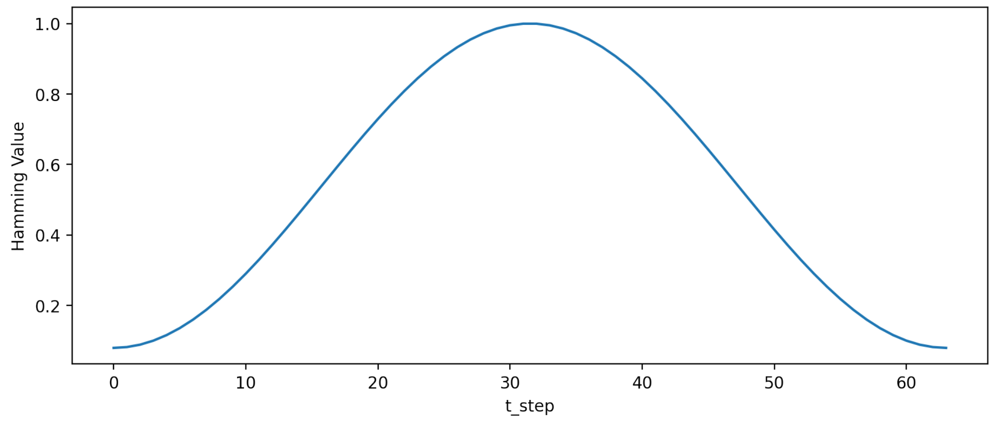
(C) **Speaking Artefacts:** Unfortunately this spectrogram is imperfect. It doesn't take a keen eye to notice one particular type of visual artefact jumps out from the spectrogram and makes it difficult to interpret. What is it?

Answer: The most obvious artefact is the presence of vertical lines wherever there is a lot of signal. In some cases it can be quite extreme, and can mask the fundamental signal as well as all the associated sub-signal (harmonic frequencies). A notably bad region is around the 200'th time step.

(D) **Hamming to the Rescue:** One of the causes of the artefact you discovered in the previous subpart is the use of the overlapping time window. Strong signals that at the ends of the window result in somewhat flat frequency noise which appear as vertical lines. To mitigate this we use a **window function**. A window function simply scales all the values in a certain time window by some function. One particular window function is the Hamming Window and is defined as follows:

$$w(n) = 0.54 - 0.46 \cdot \cos\left(\frac{2\pi n}{M-1}\right)$$

Where n is the time-step within the window and M is the size of the window. For a window of length 64, the Hamming values are distributed as follows:

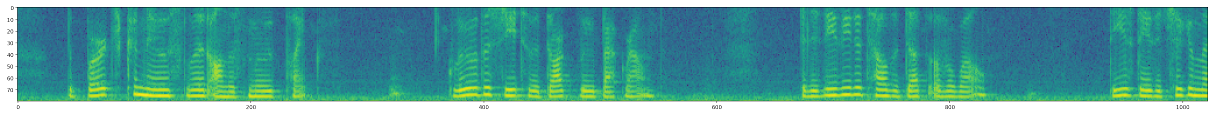


Use the hamming window when building your spectrogram, by element-wise multiplying the amplitudes in each window of the audio array by the hamming coefficient, and plot the resulting spectrogram for `sample.wav`. What's different this time around?

[*Hint 1: both numpy and MATLAB have implementations of the Hamming window by default. Don't waste time reimplementing it.*]

Answer:

The vertical line noise is greatly reduced by the window function and as a result the resonant frequencies (parallel lines to the brightest signal frequencies) are much more visibly defined. This enables easier analysis of the data and provides a much cleaner visual.



We've now arrived at a fairly clean spectrogram of our audio! Congratulations! If you enjoyed this, try generating some spectrograms of your own audio. You might need to trim the audio to a length that matches up with our power of 2 scheme. In the next part we will take a look at how useful spectrograms can be.

3 Part 3: Birds aren't Real?

Let's take a break from CS168 for a few minutes and get some much needed vitamin D. Stand outside, take a few deep breaths - what do you see? Close your eyes - what do you hear?

Chances are, you'll hear a bird call or two. We see and hear birds every day, but few people know what specific species they're listening to. In this section, you'll get a step closer to identifying these mysterious beings by using the tools you've built so far.

We'll be using audio recordings of bird calls that you can download a zip file of here, or access on our Github.

(A) **A Mystery Afoot:** In your zip file, you'll find 4 different recordings of bird calls. One is labeled "mystery". Play this one and make a note of what you hear. Listen to the other three recordings and do a rough comparison of them. Are there any that sound particularly similar? Either to each other, or to the mystery call?

Answer: (subjective) The mystery recording is a little fuzzy, but there is a bird call that occurs four times. The call is high frequency and clear.

The green warbler, greenish warbler sound nearly identical to each other, as well as the mystery call. The American redstart call is very quick and high frequency, and you can hear another bird call in the background of its recording.

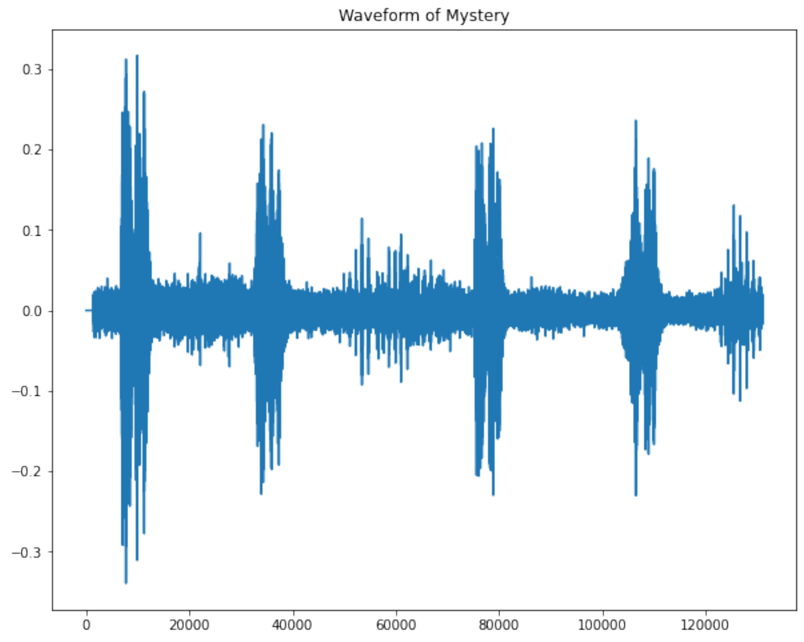
(B) **Getting Waveforms:** Load each call into an array and plot the audio waveforms. What can you tell about each based on their plot? What is hard to tell?

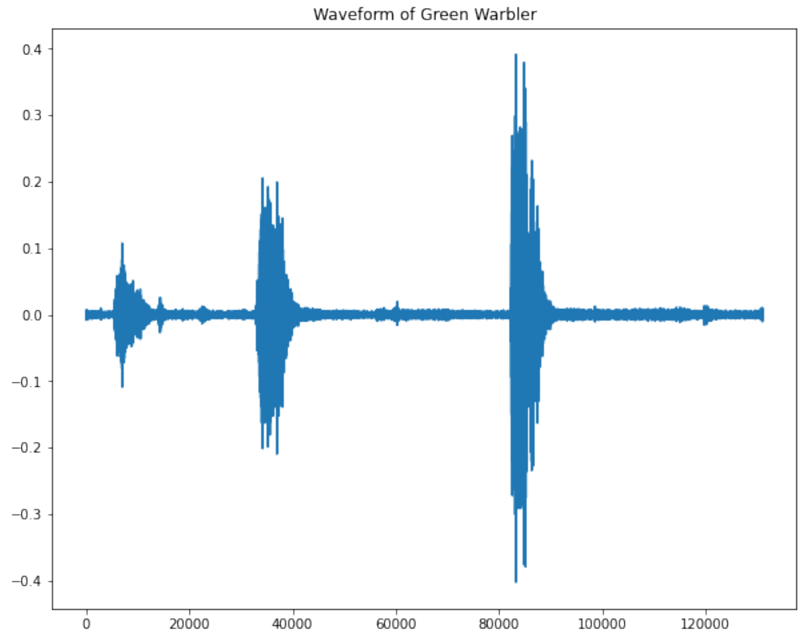
Answer: The mystery one has a considerable more amount of noise (high enough amplitudes at timepoints when the call is not occurring). Each call is defined by high amplitude at short intervals.

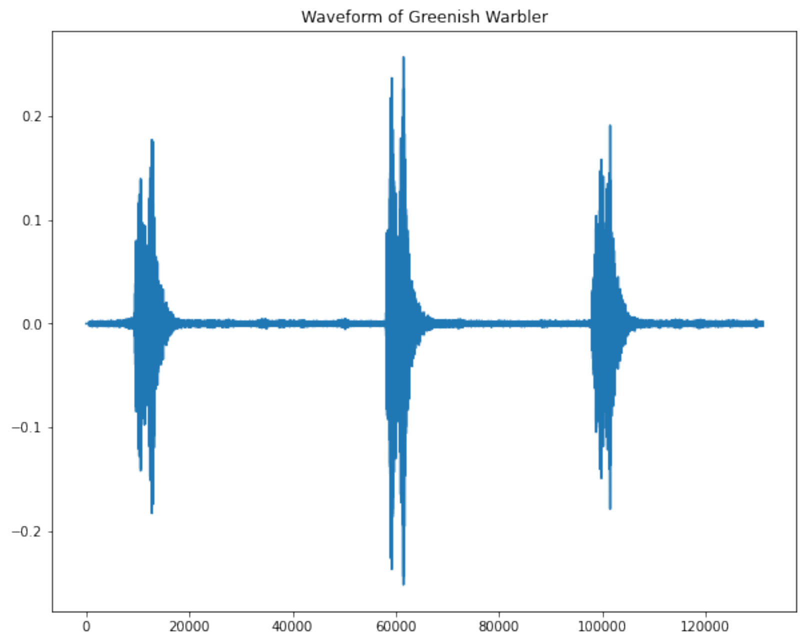
The green warbler call is similar to the mystery call, with less noise. There seems to be more of a dropoff at the end compared to the mystery one.

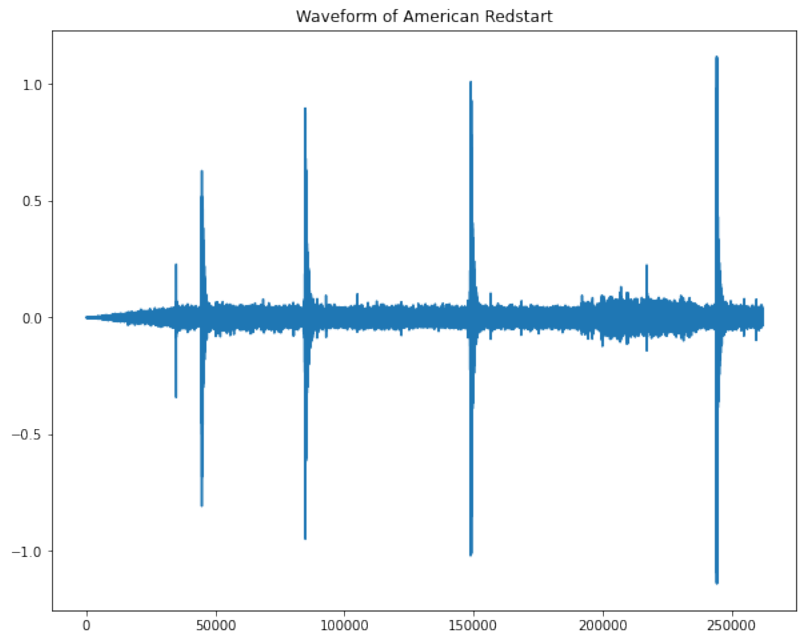
The greenish warbler is also similar to both the mystery recording and green warbler. It's difficult to distinguish between each, though greenish warbler also has less noise, and peaks may be more defined.

The American redstart is slightly more noisy, compared to the green and greenish warbler. The call is also more abrupt than the other two, which makes sense since in the recording it sounds more rapid, but keep in mind it is around twice the length of the other recordings.





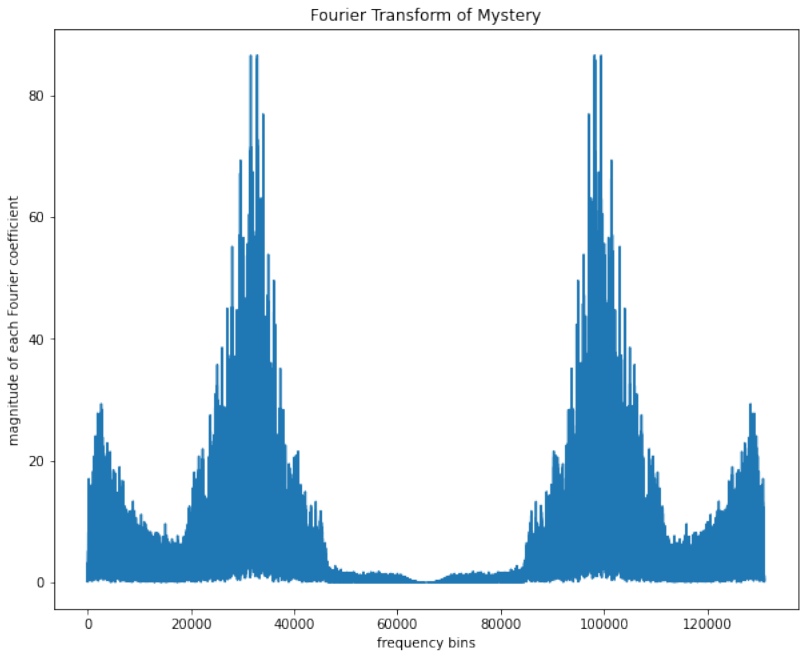


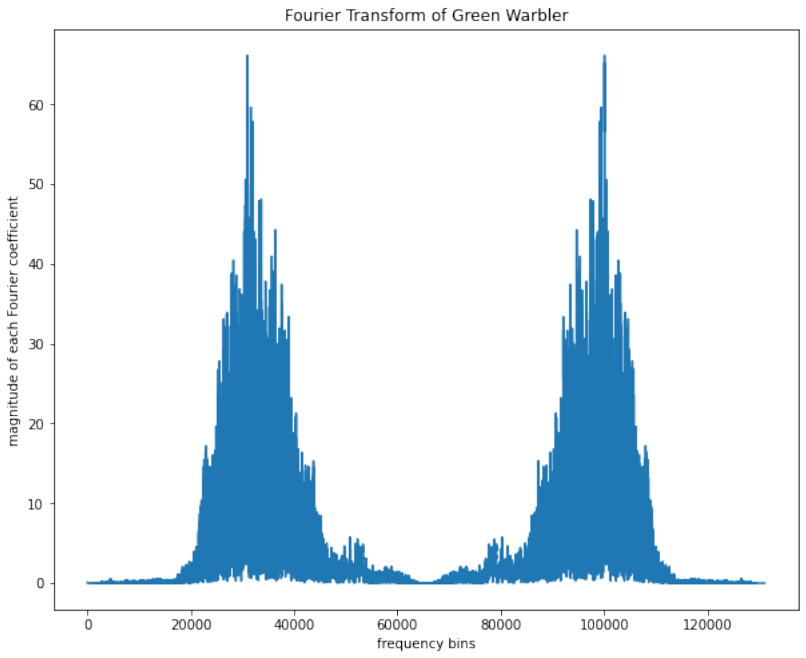


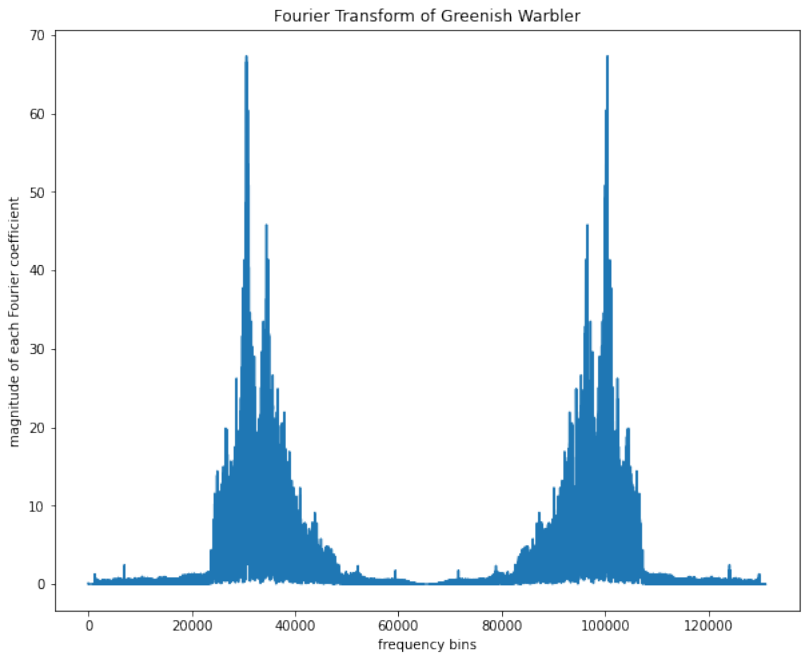
(C) **FFT Strikes Again:** Using your FFT implementation from Part 1, take the Discrete Fourier Transform of each and plot the magnitude of coefficients for each (4 plots total). (You may need to crop the audio array so it is a power of two). What do you see? What similarities/differences do you notice between the three known species and the mystery recording? Is it what you expect?

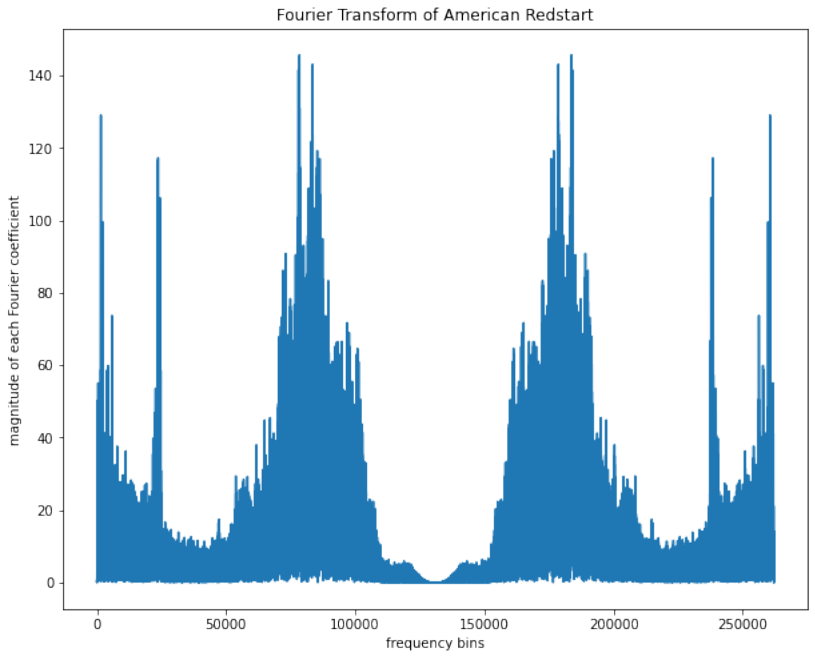
Answer: For the mystery Fourier transform, magnitude of low frequencies is higher, likely corresponding to the noise in the recording. The American redstart Fourier transform is sort of similar, but with a larger peak at low frequency.

The greenish warbler has very little noise, and has two peaks of magnitude, unlike the green recording. However, the shape is not entirely clear when comparing them with the mystery one.









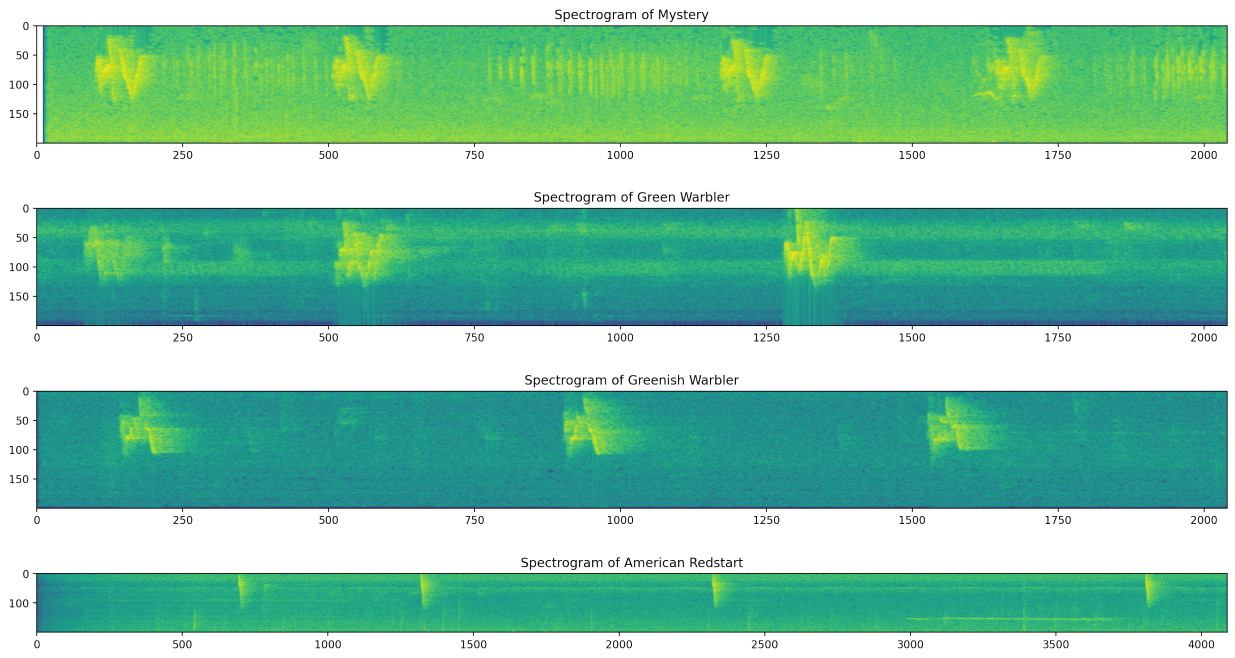
(D) **Spectrograms to the Rescue:** Then, using your spectrogram implementation from Part 2, plot the spectrogram of each. What do you see? What similarities/differences are there?

Answer: We can see the “footprint” of each call in the spectrograms.

The mystery recording has three sort of parts - the slope up, the slope down, and the flick at the tail end. This is very similar in shape to the green warbler recording, though the mystery has a bit more noise which can be seen by the vertical lines and lighter green background.

The greenish warbler has two main parts - the slope up and down. The two parts are similar to the first two parts of the green warbler/mystery recording, but it is missing the tail flick.

Finally, the American redstart is only one part, which is a quick slope down.



(E) **Solving the Mystery:** What species does the mystery recording belong to? Use what you've garnered from the previous parts of this question and/or try any other methods that might be helpful for narrowing it down (changing playback speed, reducing noise).

Answer: It is a green warbler's call. It is most clear from the spectrogram, as there is a flick up at the end of a call, while this flick isn't present in the greenish warbler's. Playing around with different filters also makes the call more defined.

(F) **Ground Truth:** Read more about the correct identification here. It turns out that the Greenish and Green Warblers are very difficult to separate by eye, and using a spectrogram is one of the easiest ways to separate them in the field. What is the mystery species? Do your answers from previous parts make sense? Have you been convinced of the utility of spectrograms?

Answer: The mystery species is indeed a green warbler. This aligns with what we wrote in part D and E. The flick up is like the third syllable that the article talks about.

4 Reflection

In this miniproject, we have implemented both the Fourier Transform and the Fast Fourier Transform from scractch, learnt to build a clean, windowed spectrogram, and then applied both these methods to the difficult problem of analyzing similar bird calls. This miniproject was intended as a supplement to week 8's content. The first parts aims to provide more scaffolding and intuition around the FFT. This was introduced in lecture, but we did not delve deeper into it in the miniproject, hence we thought it would be useful to include here.

The second part dives into spectrograms and how Fourier Tranforms are useful for building them. Spectrograms are critical to modern speech processing in tasks such as Automated Speech Recognition and understanding how the Fourier Transfrom helps create them is a useful extension of what we learn in lecture. Also the process is very fun and seeing the result, especially after cleaning up the spectrogram with the Hamming filter is very cool.

Then, the third part uses these techniques in an interesting real world application (birders actually rely on spectrograms to identify similar bird calls!). In the subparts we show that the original waveforms and even the raw the Fourier transforms are not enough to definitively identify the mystery bird call - the spectrogram is needed for this. From this, students can better understand which situations may require different techniques.

5 Code and Links

Our Github with relevant files and code is located here.
(https://github.com/emily2h/spectrogram)

Audio files for part two and three are also available for download here.
(https://drive.google.com/drive/folders/13l000Lsabd1My_WC0arsGWlikmZ0X7yt?usp=sharing)

Code here.
(https://colab.research.google.com/drive/1kfUhFzG24xKHZ5QHeoLtfY7POA4szsJI?usp=sharing)