PSET π: Speak Spectrogram To Me

Ishira Fernando, Emily Huang

June 7, 2022

1 Part 1: Building the Fast Fourier Transform

(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 $M^{j,k} = (e^{\frac{-2\pi i}{n}})^{jk}$.

(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.

(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 (FFT), 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)
```

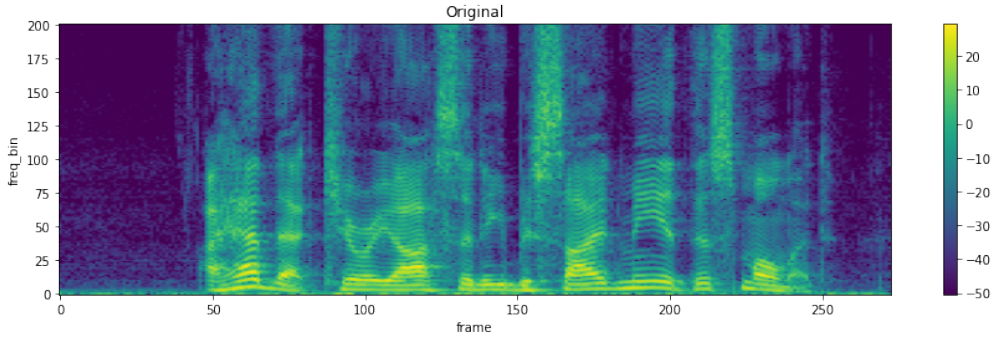
(D) **Time to Time:** Run both the Algorithms from Part B and Part B on randomly generated arrays of lengths of powers of 2 between $[2, 2^{14}]$ and record how long it takes for each one to complete. Plot the execution time vs array length for each and comment on the trends.

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 Mel-Spectrogram. Due to the innate difficulty in handling vectors with lengths that are not powers of 2, we will use numpy/scipys implementation of the FFT for this portion.

Fourier Transformations 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 Fourier constructed 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 Mel-Spectrogram of a person speaking. Note the axis, and the varying intensity of the heatmap.

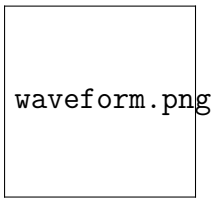


The x-axis as usual is just the time step of the audio, but the y-axis is in the frequency domain (sound familiar?), rather than being an amplitude. Instead the intensity of the audio at each frequency is encoded by the intensity of the color in the heatmap. Now instead of having just amplitude and time 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 an 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.*]

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

(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 as an input 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.

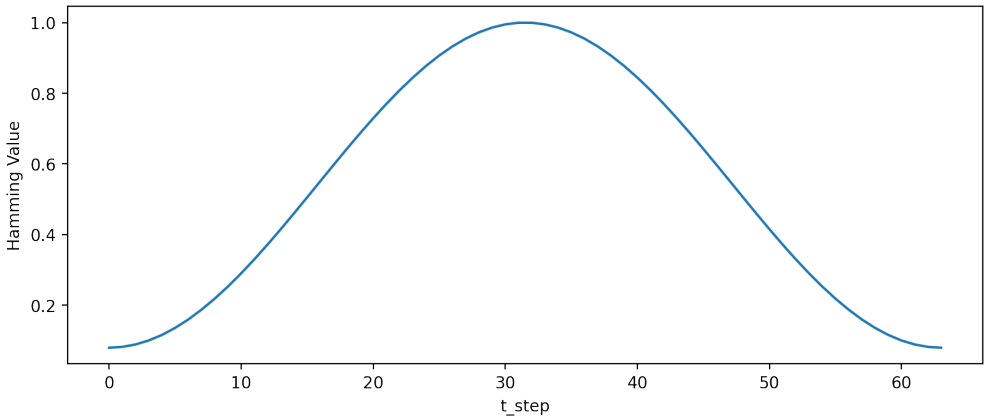
Note that we haven't converted the frequency bins output by the Fourier Transform to an actual frequency (Hz). This is okay as for this problem set we won't cares as much about the actual frequencies as we do the shape of the spectrogram

(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?

(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. To mitigate this we use what is known as 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 simply 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.*]

We've now arrived at a fairly clean spectrogram of our audio! Congratulations! If you are so inclined 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: What's That Bird?

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.

(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?

(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?

(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). What do you see? What similarities/differences do you notice between the three known species and the mystery recording? Is it what you expect?

(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?

(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).

(F) **Ground Truth:** Read more about the correct identification here. What is the mystery species? Do your answers from previous parts make sense?