Problem Set 1: Speech Signals

The University of Texas at Austin
CS 378: Intro to Speech and Audio Processing
Instructor: David Harwath

This problem set consists of 3 problems (each with multiple parts), totaling 100 points.

You will explore:

1. Computing the resonant frequencies (formants) of vocal tract configurations corresponding to different vowels **(20 points)**
2. Reading a spectrogram **(20 points)**
3. Analyzing a real speech signal and implementing a MFCC pipeline in Python. **(60 points)**

To turn in: You should submit a single .pdf writeup to Canvas, containing all of your deliverables for all parts of the problem set. For the written parts of the assignment, you must show your work. For the coding parts of the assignment, you must turn in all of the plots and code fragments that are specified as deliverables in the corresponding exercises.

Collaboration Policy: You may collaborate in groups of no larger than 3 students, and each student must turn in their individual writeup written in their own words. **You must clearly indicate all of your collaborators in the title block of your writeup.**

Late policy: Sometimes we have bad days, bad weeks, and bad semesters. In an effort to accommodate any unexpected, unfortunate personal crisis, per the course syllabus I have built 4 "slip days" into our course. You do not have to utilize this policy, but if you find yourself struggling with unexpected personal events, I encourage you to e-mail me as soon as possible to notify me that you are using our grace policy. Late assignments will otherwise not be accepted.

Exercise 1 (20 pts)

For the following problems, assume the speed of sound through air is $c = 34,000$ cm/sec.

1.1 Vowel 1

Consider the vocal tract approximation in Figure 1.

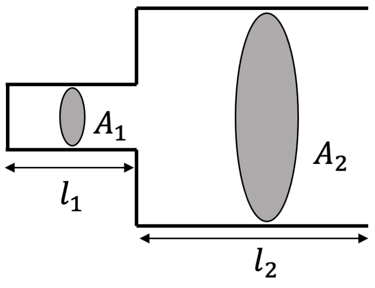

Figure 1: Acoustic tube approximation for vowel 1

Assuming the area and length measurements $A_1 = 1$ cm^2 $A_2 = 10$ cm^2, $l_1 = 9$ cm and $l_2 = 7$ cm, compute the first three formants (sorted by increasing frequency) of the acoustic tube configuration shown in Figure 1. **(8 points)**

Based upon the formant values you computed above, list 1 American English vowel that would be a good match for this vocal tract configuration. **(2 points)**

1.2 Vowel 2

Consider the vocal tract approximation in Figure 2.

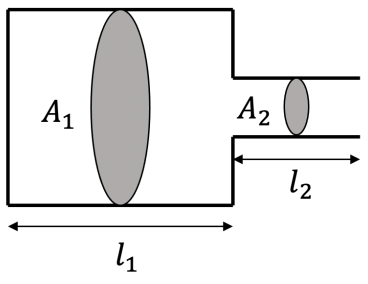

Figure 2: Acoustic tube approximation for vowel 2

Assuming the area and length measurements $A_1 = 9$ cm^2 $A_2 = 1$ cm^2, $l_1 = 9$ cm and $l_2 = 5$ cm, compute the first three formants (sorted by increasing frequency) of the acoustic tube configuration shown in Figure 2. **(8 points)**

Based upon the formant values you computed above, list 1 American English vowel that would be a good match for this vocal tract configuration. **(2 points)**

Exercise 2 (20 pts)

Consider the spectrogram in Figure 3. Best viewed on a computer; you might need to zoom in to help you answer the following questions.

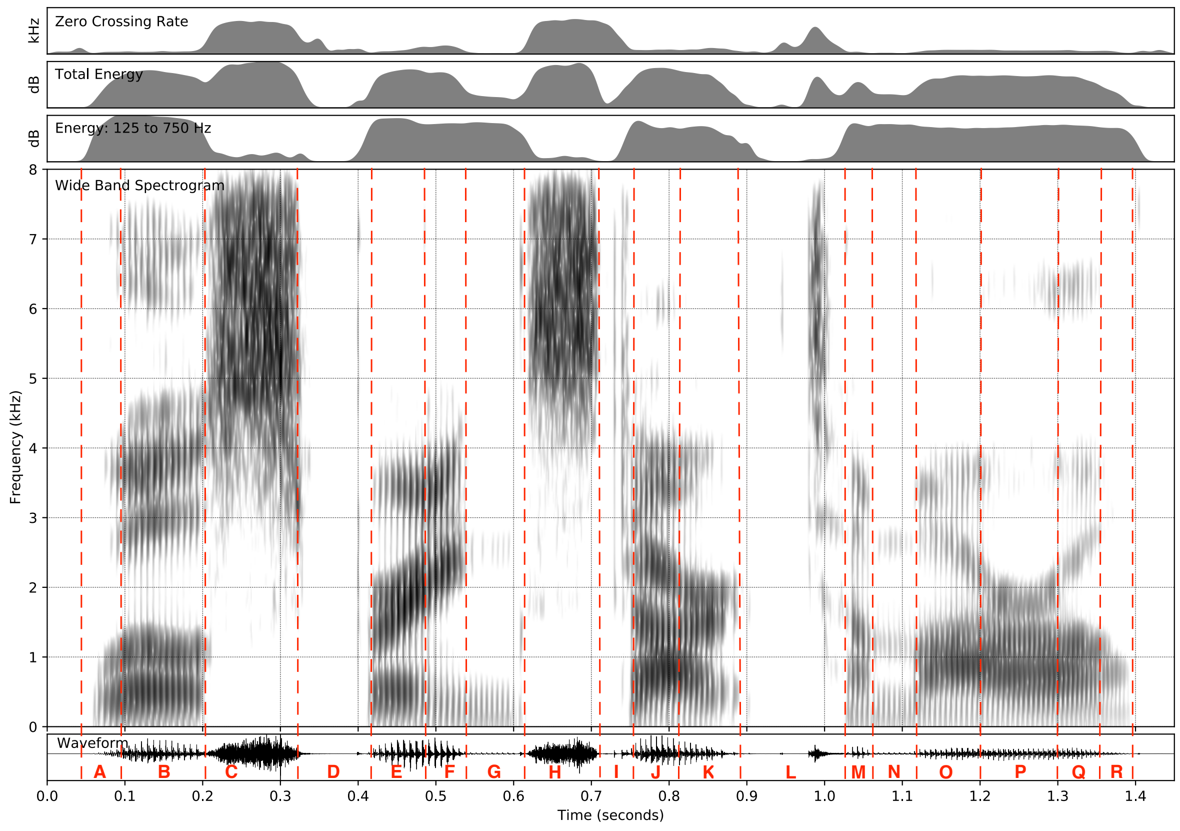

Figure 3: Mystery Spectrogram

2.1 Spectrogram Reading

I have indicated all of the phonetic boundaries in the spectrogram with the vertical, dashed, red lines. Each segment of the spectrogram that sits between two of these boundaries is one phone, and I have assigned them the letters A through R (red capital letters at the bottom of the waveform beneath the spectrogram). For each of the 8 phonetic categories below, list one letter (A through R) that refers to a segment in the spectrogram above that matches the phonetic category: **(16 points)**

1) Fricative	2) Stop	3) Semivowel	4) Nasal
5) Front Vowel	6) Back Vowel	7) Alveolar Consonant	8) Retroflex

2.2 Fundamental frequency estimation

What is the fundamental frequency of the voiced excitation (F_0) at 0.55 seconds? **(4 points)**

Exercise 3 (60 pts)

In this exercise, you will implement a Mel-Frequency Cepstral Coefficient (MFCC) acoustic feature extraction pipeline. You will need to download three files from Canvas:

1. `signal.wav` : The speech recording you will use to test your pipeline.
2. `mel_filters.npy` : A set of pre-computed Mel filter banks.
3. `reference_mfcc.npy` : A reference MFCC feature sequence for `signal.wav` that you can compare against your implementation.

You will also need a Python environment with the `math`, `numpy`, `scipy`, and `matplotlib` packages available. These are the **only** packages you should use for this exercise (in other words, you should not use `librosa` or similar libraries).

Any of the UTCS lab machines should already have everything you need installed, but if you find it more convenient to use a different machine (such as a personal laptop) you should feel free to do so. I recommend using a Jupyter notebook for this exercise.

If you don't know how to use a particular function, you can find all the documentation you should need at the links below:

- `https://numpy.org/doc/`
- `https://docs.scipy.org/doc/`
- `https://matplotlib.org/3.3.3/contents.html`

To recap, the MFCC extraction pipeline we covered in lecture consists of the following steps:

1. Load an audio waveform.
2. DC subtraction (i.e. remove the mean of the signal)
3. Pre-emphasis filtering
4. Transforming the audio into a sequence of frames
5. For each frame:
   (a) Multiplying the frame with a window function
   (b) Computing the Fourier transform
   (c) Computing the magnitude spectrum from the Fourier transform
   (d) Computing the power spectrum from the magnitude spectrum
   (e) Binning the power spectrum with Mel filterbanks
   (f) Taking the logarithm of the Mel-filterbank power spectrum
   (g) Computing Discrete Cosine Transform (DCT) of the log-Mel-power spectrum
   (h) Truncating the DCT to keep only the first C elements

For an input audio waveform that occupies F frames, the resulting MFCC representation of the audio will be a F by C matrix. We will now implement each of the above steps. You should strive to keep your code as clean and easy to modify as possible, as we'll be examining the impact of various tweaks to the pipeline.

Loading the audio file: Using `scipy.io.wavfile.read()`, load the `signal.wav` file from disk. What is the sampling rate of the audio, and how many samples are in the recording? How many seconds worth of audio does this correspond to?

Mean subtraction: Let $x_1[n]$ be the raw samples of the audio file that you just loaded. The first step is to perform mean subtraction:

$$x_2[n] = x_1[n] - x_0$$

where x_0 is the average value of all the samples in the waveform.

Pre-emphasis: Recall from lecture that the periodic excitation from the glottis in voiced speech is not an ideal delta train, but is "smoothed." To be more concrete, the harmonics of the excitation fall off in magnitude as a function of frequency at a rate of $1/f^2$. The radiation characteristics at the lips partially counteract this by a factor of f, but in order to flatten the spectrum completely we need to apply another filter whose magnitude response is proportional to f. In ASR lingo, this is called "pre-emphasis" filtering and is achieved with a first order finite impulse response (FIR) filter:

$$x_3[n] = x_2[n] - bx_2[n-1] \qquad [3.1]$$

where $b = 0.97$ is conventionally used. When computing $x_3[0]$, you can ignore the previous sample (since it does not exist) and assume $x_3[0] = x_2[0]$.

Computing frames: The next step is to compute a series of N_f frames, where each frame captures a small, fixed-length piece of the signal. The two parameters that control the nature of the frames we will extract are the frame *length* (sometimes also called the width) and frame *shift* (sometimes also called the "hop"). Assuming a window length L and shift S (both specified in samples), The k^{th} frame will be:

$$x_k[n] = x_3[kS + n], n = 0, 1, 2, \dots, L-1$$

Unless $(length(x_3) - L)$ is exactly divisible by S, the very last frame won't have enough samples to fill an entire window length L. In this case, we can just pad the final window with zeros up to length L. Implement a function that takes a signal x, a window length L, and a window shift S as parameters and returns the framed representation of the signal as a matrix of size N_f by L, where N_f is the number of frames.

Applying the window function The next step is to element-wise multiply each frame $x_k[n]$ with a window function $w[n]$:

$$x_k^w[n] = x_k[n] \cdot w[n]$$

The Hamming window is most commonly used in ASR, and can be created simply by making an appropriate call to the `scipy.signal.hamming()` function (`scipy.signal` also has factory functions for a large number of other window types). Write a function that takes a single frame $x_k[n]$ (i.e. one row of the matrix you computed in the last step) as input, and returns $x_k^w[n]$ as output.

Computing the Fourier transform Next, we will compute the Fourier transform of each windowed frame. Recall the so-called "direct" implementation of the DFT we covered in class:

$$X[m] = \sum_{n=0}^{N} x[n] e^{-2\pi j \frac{m}{N} n}$$

The `scipy.fft.fft()` function implements the above equation extremely efficiently. The two parameters to `scipy.fft.fft()` you should pay attention to are `x`, which you can pass $x_k^w[n]$ into, and `n`, representing the number of DFT points N we want to compute. In order to use the Mel filters later on in this lab, you'll want to use $N = 512$ by default.

You are more than welcome to implement the "direct" DFT function above instead of calling `scipy.fft.fft()` if you feel that it would help you better understand what the DFT is doing. Be aware that the direct implementation will run slower than the FFT (on my laptop, it takes about 2 minutes to compute the DFT of all frames in the utterance, whereas the FFT is nearly instant). If you opt to implement your own DFT, there are a few things to keep in mind:

- In Python, the complex number $a + bj$ can be created by making a call to `complex(a, b)`. You can also write a complex literal as `2+1j`, `0+1j`, etc. Just make sure you always prefix `j` with a number, otherwise Python will try to interpret the expression as a variable named `j`.
- When you instantiate a `numpy.array` using any of the common factory functions like `numpy.zeros()` or `numpy.ones()`, you can specify the data type with the `dtype` keyword argument. When you instantiate an `array` that will hold complex numbers, be sure to specify `dtype=complex`, otherwise Python will throw an error when you try to fill a float array with complex numbers.

Computing the magnitude and power spectra: The magnitude spectrum is defined as:

$$X_{mag}[m] = |X[m]| = \sqrt{\text{Real}(X[m])^2 + \text{Imaginary}(X[m])^2}$$

To compute the magnitude of a complex number in Python, simply use `numpy.abs()`. Once you've done that, square each element to compute the power spectrum:

$$X_{pow}[m] = X_{mag}[m]^2 \qquad [3.2]$$

Mel-filterbank application: Recall from lecture that we use a set of Mel-scale filterbanks to warp the frequency axis to better reflect human perception, as well as to lump nearby frequencies together and reduce the overall dimensionality of the spectrum. Load the `mel_filters.npy` file into a variable called `mel_filters`, which will contain a 23 by 257 matrix of floats.

To get an idea of what the filterbanks look like, plot them with `plot(mel_filters.T))`. We transpose the matrix because when given a 2D matrix as input, `plot()` plots each column as a separate series, and we want to plot the 23 different Mel-filter shapes. **(Include this plot in your writeup.)**

You might be wondering why `mel_filters` has 257 columns. Recall from lecture that the Fourier transform of a signal contains both *positive and negative* frequencies, but for a real-valued signal such as ours, the magnitude spectrum will be symmetric. Therefore, when computing the Mel-filterbank energies, we are only use the positive frequencies, which live between 0 and π along the digital frequency axis, corresponding to the first $(N/2) + 1$ elements of $X[m]$. Because we used $N = 512$ when computing our DFT, our positive frequencies will appear in the first 257 elements of $X[m]$, and the Mel-filters are sized accordingly.

The energy contained within the i^{th} filter is simply a weighted sum of the filter response *mel* times the power spectrum:

$$X_{mel}[k] = \sum_{m=0}^{m=257} mel_k[m] X_{pow}[m], k = 0, 1, 2, \ldots, 22$$

You should be able to implement the above equation for all filters simultaneously using a matrix multiplication between $mel\_filters$ and an appropriate slice of X_{pow}.

Taking the log: This is a simple step:

$$X_{logmel}[k] = \max(-50, \log(X_{mel}[k]))$$

We clip the log energies at -50 because log tends to negative infinity when the energy of a bin approaches zero.

Side note: many recent deep neural network acoustic models will stop their feature extraction pipeline at this step, feeding the log-Mel power spectrum as input to the network (in research papers you will sometimes see these called log-Mel filterbank energies, or Mel-Frequency Spectral Coefficients (MFSCs)). In this case, it's common to use a larger number of Mel filters - 40, 80, and 128 are commonly used values. When using a larger number of filters, the same digital frequency interval is spanned (0 to π), but the individual filters are narrower and spaced closer together in order to achieve a higher resolution spectrum. The reason why neural nets can get away with this is because they can easily handle high dimensional, highly correlated input features (which MFSCs are). Classical ASR acoustic models based on diagonal covariance GMMs struggle to model high dimensional features, and also make the assumption that their input features are uncorrelated. The DCT does a good job at decorrelating MFSCs and concentrating the variance in the first few dimensions (the intuition behind Principal Component Analysis carries over to here), which is one reason why GMMs work much better with MFCCs than MFSCs.

Computing the DCT and "liftering": Now we'll compute the Discrete Cosine Transform (DCT) of the log-Mel power spectrum. Typically, we only want to keep the first 13 DCT coefficients, $C[0], C[1], C[2], ..., C[12]$, which are commonly referred to as the MFCCs:

$$C[i] = \sum_{k=0}^{22} X_{logmel}[k] \cos\left(\frac{\pi i}{23}(k + \frac{1}{2})\right)$$

Putting it all together: Compute the entire sequence of MFCC frames for the `signal.wav` file using your pipeline and the following configuration:

- Window length L set to 25 milliseconds (what is this in samples?)
- Window shift S set to 10 milliseconds (what is this in samples?)
- Hamming window
- Number of DFT points $N = 512$
- MFCC coefficients to retain: $C[0]$ through $C[12]$ (13 in total).

Visualize your MFCC features **for only C_1 through C_{12}, because the dynamic range of C_0 is much larger than the rest of the coefficients** using `imshow()` and include the plot in your writeup. Also compute the difference between your MFCCs and the reference MFCCs (all 13 coefficients). What is the Mean Squared Error (MSE) between the two matrices?

Next, plot the MFSCs X_{logmel} for the entire utterance (i.e. omit the DCT and liftering steps) again using `imshow()`. Alongside it, plot the log power spectrum, i.e. $max(-50, \log(X_{pow}[m]))$, which corresponds to skipping the application of the Mel-filterbanks. Comment on how these two spectra look different, and include both plots in your writeup.

Hints:

- `imshow()` by default places the origin in the upper left hand corner. In our case we want low frequencies to appear at the bottom, so we'll need to set `origin='lower'`.
- We've been computing our feature matrices so that frames are indexed along the rows, but in order to get the time dimension on the horizontal axis you'll need to transpose your matrices before plotting them with `imshow()`
- Finally, your spectra might look a little bit "squished" when plotting with `imshow()`. You can try using the `aspect` keyword argument to stretch the plots out to make them more legible.

Next, change your frame extraction configuration to use a window length L of 4 milliseconds, and a window shift S of 1 millisecond (don't forget to also change the length of the Hamming window to match). Using this configuration, plot X_{logmel} and $max(-50, \log(X_{pow}[m]))$ alongside one another again. How does the spectrum look different when using 25 millisecond windows as opposed to 4 millisecond windows? Why does this happen? Include both of these plots in your writeup.

Finally, make a plot of C_1 as a function of time and include this plot in your writeup. What do you notice about the value that C_1 takes on during a vowel vs. during a fricative? Why is this the case? Hint: closely examine the DCT equation for $i = 1$. You might find it useful to plot the cosine term in the equation for $i = 1$.

Congratulations - you just implemented the signal processing front end for a typical speech recognition system! Don't forget to include the code for your MFCC extraction pipeline in your writeup.