Vowel Recognition: 6.205 Final Report

Aloysius Ng
Department of Mathematics
MIT
aloy_ng@mit.edu

Jonathan Huang
Department of EECS, Department of Biology
MIT
jhuang25@mit.edu

***Abstract*—We describe and implement a method to recognize vowels from real-time speech using an FPGA board. The method of vowel recognition will be via tracking formants, or the resonant frequencies of a given signal; formants usually specific to vowels spoken by a given person. The FPGA will take in audio output via a microphone and calculate the formants of the given speech signal over time. While there are many methods to calculate formants, we chose to implement a dynamic programming version that avoids complicated matrix operations but provides its own design challenges. By comparing the formant values of the speech signal to stored formant values for each vowel, the FPGA can output the vowel it believes it is hearing. We pipe out formant frequency information out via UART and display it as a scrolling graph via HDMI. Our circuit does not use much memory, but uses a large percentage of the logic available.**

***Index Terms*—digital systems, field programmable gate arrays, transcription, audio processing**

I. Introduction

Speech recognition refers to a method of converting speech from an auditory signal into text. Automatic speech recognition (ASR), or speech recognition done via a computer, remains an active field of research in terms of improving speed and accuracy. We restrict ourselves to a subset of general ASR: can we recognize which vowel sound is being spoken at a given moment, if a vowel is being spoken? We use phonemes, or the distinct units of speech, to represent sounds spoken at a given moment: orthographically, there are only five (or six) vowels in English, but the English language is considered to have at least 15 distinct vowel sounds; these vowel sounds are often represented using the IPA (International Phonetic Alphabet).

Vowel recognition has been a line of research pursued for decades, often based around formant-based strategies [1] [2]. Formants are regions of broad spectral maxima in human speech, which roughly correspond to the resonant frequencies of the sound signal. Because vowel sounds are produced from an open vocal tract, vowels have clear formants, which can often be identified via visually looking at a spectrogram. Vowels spoken by a particular person are known to be distinguished via their first few formants, with two or three often being sufficient. Formant values of the same vowel spoken by different people are often not too dissimilar.

Calculating (or more accurately, estimating) formants from a speech signal is thus the heart of this project. Simply taking the maxima in the spectrogram (i.e. from a Fourier transform of the given data) results in noisy and inaccurate formant values. The most common strategy of formant calculation is via linear predictive coding, which is used in commonly available speech analysis softwares such as Praat. However, linear predictive coding requires extensive matrix manipulation. We follow instead an algorithm developed by Welling and Ney (1998) [3], which segments the spectrum of a speech signal into different regions corresponding to a formant. Each region is estimated by a second-order resonators with resonant frequency at the formant. The algorithm uses a dynamic programming approach takes considerable time on a normal processor. Our Python implementation of their algorithm, for instance, takes 44ms to calculate the formants when *given* the Fourier transform of a 25ms window of audio sampled at 8000Hz (windows are normally taken every 10ms to encourage overlap, as well): calculating the formants themselves cannot be done live, let alone calculating the Fourier transform and then the formants.

The goal is to leverage the parallel processing power of the FPGA and to efficiently use the DSPs on board to compute the formants live. We display our calculated formants on an HDMI display along with the vowel that best corresponds to the formants.

II. Design Details

The block diagram of the code we have implemented is in Figure 1.

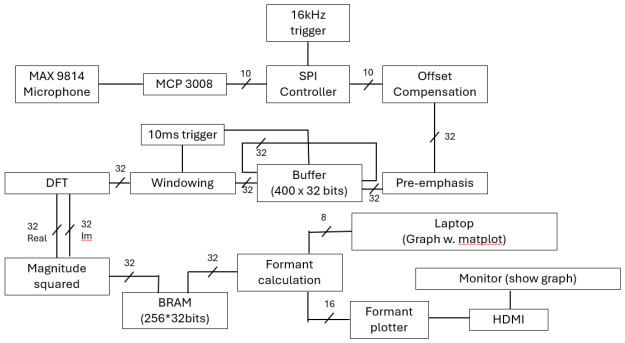

Fig. 1. Block diagram of written code. Dashed red box represents the spectrogram module.

A. Audio Input

We use the microphone used earlier in the class, the MAX9814. With the MAX9814, we can use the MCP3008 as an ADC and use the SPI protocol (completed in Lab 3) to

interface with it. We are sampling our audio at 16kHz (one sample every 6250 clock cycles) using 10 bits of precision from the MCP3008. This sample rate is sufficient to identify all frequencies below the Nyquist frequency of 8kHz, which is enough to capture all the relevant information of human speech.

B. Audio Processing

The offset is estimated as a constant value that the output of the ADC is offset by. We remove this by using a filter similar to the ETSI standard set out in [4], but use division by a power of 2 for easier calculation. Our processed audio sample is given by below where s_i, s_o are the input and output samples respectively and n be the index of the sample:

$$s_o(n) = s_i(n) - s_i(n-1) + s_o(n-1)\left(1 - \frac{1}{1024}\right). \quad (1)$$

We then use a pre-emphasis filter which boosts higher frequencies in the audio input to improve the signal-to-noise ratio. We use a filter similar to that in [4], but again elect for a power of 2 instead. Where s_p is the output of this filter,

$$s_p(n) = s_o(n) - \left(1 - \frac{1}{32}\right) s_o(n-1). \quad (2)$$

A maximum of 400 processed samples are stored at anytime, corresponding to 25ms of speech. These 400 32-bit samples are stored in flip-flops.

C. Fourier Transform

Every 10ms, this module takes the previous 25ms of audio as input from the buffer. These input values are sequentially given to the windowing module when a full window of 25ms is available in the buffer. We denote such a grouping of samples as a frame.

Note that two consecutive frames of 25ms have a 15ms overlap, corresponding to 240 audio samples. The buffer re-inserts these audio samples into the start of the buffer to preserve them for the next window 10ms later. This does not collide with a new audio sample as this reinsertion happens in 400 clock cycles, much less than the 6250 cycles that is needed for a new sample to come in.

To window each packet of 400 samples, we multiply our values with those of a Hamming window, which varies from 0.04 at the edges of the frame to 1 in the center of the frame. These are zero-padded to form a frame of 512 samples, which is fed sequentially into a DFT module that implements the Cooley-Tukey FFT algorithm.

The outputs of this module are 512 complex Fourier coefficients, returned in reversed-bit order. These are sequentially fed into a magnitude module, which computes the squared magnitude of each complex coefficient. We only keep the 256 magnitudes that correspond to one half of the Fourier coefficients.

D. Vowel Determination

For every frame of audio, we apply the algorithm outlined by Welling and Ney to output the four formants based on segmenting the interval from 0Hz to 5000Hz. Thus of our 256 FFT magnitudes, we need only care about the $I = 160$ that cover all the data below 5000 Hz. We let these magnitudes be $|S(i)|^2$ for $i \in [I]$, using the notation $[I] = \{0, \ldots, I-1\}$.

We first outline the theory of the algorithm, which relies on the following functions:

$$\begin{aligned}
T(\nu, i) &= \frac{1}{I}\sum_{j=0}^{i} |S(j)|^2 \cos\left(\frac{2\pi\nu j}{2I}\right), \\
r(\nu, j, i) &= T(\nu, i) - T(\nu, j-1), \\
\alpha(j, i) &= \frac{r(0,j,i)r(1,j,i) - r(1,j,i)r(2,j,i)}{(r(0,j,i))^2 - (r(1,j,i))^2}, \\
\beta(j, i) &= \frac{r(0,j,i)r(2,j,i) - (r(1,j,i))^2}{(r(0,j,i))^2 - (r(1,j,i))^2}, \\
Emin(j, i) &= r(0,j,i) - \alpha(j,i)r(1,j,i) - \beta(j,i)r(1,j,i), \\
F(k, i) &= \min_j [F(k-1, j) + Emin(j+1, i)], \\
B(k, i) &= \arg\min_j [F(k-1, j) + Emin(j+1, i)].
\end{aligned}$$

where

- $T(\nu, i)$ represent prefix sums for efficient calculation of $r(\nu, j, i)$
- $r(\nu, j, i)$ represent autocorrelation coefficients between sample j and sample i
- $\alpha(j, i)$ and $\beta(j, i)$ represent the coefficients of the second-order resonator on the segment of frequencies from $[j, i]$
- $Emin(j, i)$ represents the prediction error of the second-order resonator defined by coefficients $\alpha(j, i)$ and $\beta(j, i)$ to the given data on the segment $[j, i]$
- $F(k, i)$ represents the minimum cumulative error over all segmentations of $[0, i]$ into k segments
- $B(k, i)$ is the starting point of the final segment in the minimum-cumulative-error segmentation of $[0, i]$ into k segments

In terms of the above notation, the four calculated formants are derived from the segmentation of $[I]$ that minimizes the cumulative error $F(4, I-1)$. We can find the segment endpoints via repeatedly applying the function B: $i(4) = I-1, i(3) = B(4, i(4)), i(2) = B(3, i(3)), i(1) = B(2, i(2)), i(0) = B(1, i(1)) = 0$ represent the five endpoints of the four segments in decreasing order. Using the segment endpoints, we can recalculate $\alpha(j, i)$ and $\beta(j, i)$ to calculate the resonant frequency, which is given by

$$\varphi(j, i) = \arccos\left(-\frac{\alpha(j,i)(1-\beta(j,i))}{4\beta(j,i)}\right).$$

For a more detailed derivation and explanation, see [3].

The outline of how we implemented this algorithm on the FPGA is as follows:

Algorithm 1 Formant Calculations

Input: Magnitude of Fourier coefficients, $|S(i)|^2$ for $i \in [I]$
Output: $K = 4$ calculated formant frequencies

Compute $T(\nu, i)$ over $\nu \in [3], i \in [I]$, store in BRAM.
for $i \in [I]$ **do**
  for $j \in [i-1]$ **do**
    Calculate $Emin(j, i)$ and store in BRAM.
  end for
  for $k \in [1, \min(i+1, K)]$ **do**
    $F(k, i) = \infty$
    for $j \in [k-2, i-1]$ **do**
      if $F(k-1, j) + Emin(j+1, i) < F(k, i)$ **then**
        $F(k, i) = F(k-1, j) + Emin(j+1, i)$
        $B(k, i) = j$.
      end if
    end for
    Store the final value of $F(k, i)$ and $B(k, i)$ in BRAM.
  end for
end for
Find segment endpoints with $i(K) = I - 1$.
for $k = K, K-1, \ldots, 1$ **do**
  $i(k-1) = B(i, (i(k))$
  Calculate the formant frequencies $\varphi(i(k-1), i(k)$.
end for
return Formant frequencies $\varphi(0, 1) \ldots, \varphi(k-1, k)$.

There were two big points of consideration for the design of this algorithm. One was how to do arithmetic with a fixed number of bits. We chose to restrict our number representation to 24-bits in order to perform multiplication within a single cycle, done via multiplying two 24-bit numbers to a 48-bit number then truncating as necessary. Division, which is necessary to compute $\alpha(j, i)$, $\beta(j, i)$, and $\phi(j, i)$ require parallel division modules, which we tuned to run within timing contraints. A rough back of the envelope calculation shows that if we perform all arithmetic calculations sequentially we easily exceed our 10 ms window timing, or 1 million cycle computation constraint. Thorough pipelining was necessary to increase the throughput of many arithmetic calculations.

The second point of consideration was how to manage our BRAM correctly, which $75 \times 36kB$ memory. Computing and writing all of the values of $Emin(j, i)$ into BRAM (as the algorithm [3] describes) would take over a third of the BRAM by itself. From the perspective of memory conservation, we chose instead to only compute $Emin(j, i)$ as-needed, and instead use a state maching to manage when we were computing $Emin$ (steps 3-5) and computing F and B (steps 6-15). Memory conservation was another motivation for not pre-computing $r(\nu, j, i)$ and storing it in BRAM as well.

E. Data output

We output our audio processing data via UART, combining the processed audio, the Fourier coefficients, and formant frequencies in a single package. We store these data values in a stored buffer. Since there are 100 frames every second, these buffered values are transmitted at the same rate of 100 Hz (Fig. 2).

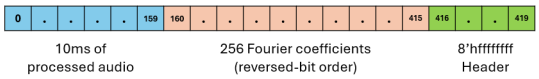

Fig. 2. UART packet diagram

The header ensures that all data is received and informs of the separation between the audio and the coefficients. This is crucial as being off-by-one on the reversed bit order gives very different results. The benefit of sending outputs via these packets is that we can record long recordings. However it also requires a higher BAUD rate. It needed to be more than 420000, so we used the standard rate of 460800 bits per second.

The output of this data is then processed off-board. The order of Fourier coefficients returned to normal and plotted with the matplotlib library on Python.

F. Data Display

We also display the formant values via HDMI on a screen, similar to a live spectrogram. Like a spectrogram, our display has time as its horizontal axis and frequency as its vertical axis. We color a pixel white if the frequency at the corresponding time is a formant of the sample at that (approximate) time value; otherwise, the pixel is left black. To create the scrolling nature of the graph, we in essence decompose our frame into 160 sections with an index into the starting section; this is equivalent to creating 160 BRAM frame buffers. We draw each frame with the pixels from left to right being stored in the frame buffers starting at the index counting upwards. At the end of each frame of the display, we write the new data to the frame buffer at the index, then increment the index by one. Because we start drawing our frame based on the data at the beginning at the index, this has the effect of shifting all of the data left one section, then updating the rightmost section, creating the desired scrolling nature of the display. To draw a given row, we fetch its data during the previous row's blanking section, which requires fetching data from BRAM. To fit our BRAM fetches into the previous row's blanking section, we make our BRAM width 8 bits; this means the 1280 bits of information in each row can fit into 160 fetches, which can fit into half of the time of the blanking section.

III. Block Diagram

IV. Results

As illustrated in Figure 3, the computed magnitudes of the Fourier coefficients mostly match with that generated by the specgram function in the matplotlib library. Differences between the two can be attributed to outputs being truncated to 8-bits for UART transmission and different settings and windowing methods. We also plotted the formant values we got, but these are not accurate yet. However, it shows that it is similar as these formants are detected when clear lines like vowels are said.

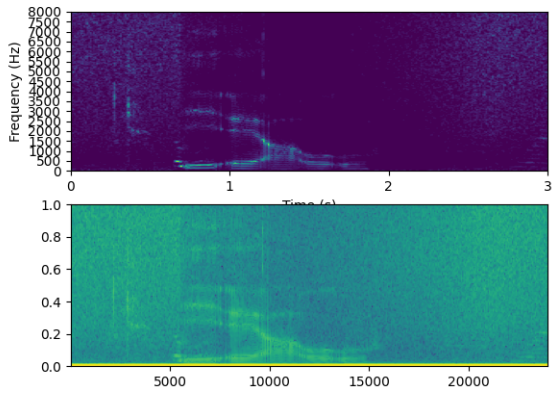

Fig. 3. Two spectrograms of the same filtered audio input. Above: Fourier coefficients computed on the board with intensity corresponding to logarithm of the value. Below: Output of matplotlib.specgram with mode set to magnitude.

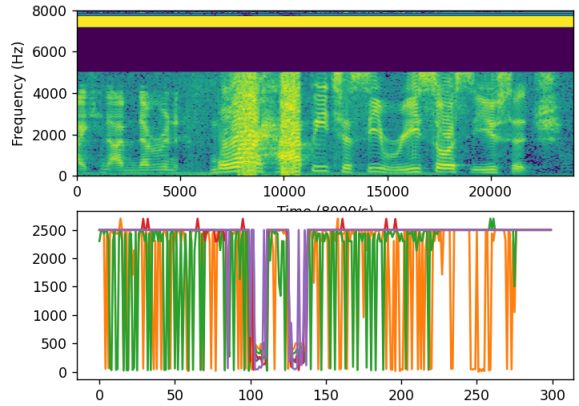

Fig. 4. Calculated values are similar but do not match correct values.

In terms of FPGA resource usage, we use approximately one third of the BRAM to store our values of our functions T, E, F, B, φ. We use 68 of 120 DSPs and 58 percent of LUTS available on the board, reflective of the amount of arithmetic that we are attempting to compute, but we also used extensive pipelining to reuse our multiplication circuits as often as possible. Our circuit was also close to the timing constraint, having a positive slack of only +0.158ns.

In terms of our project goals, this implementation was able to identify formants, which is close to achieving minimum viable product of vowel recognition, but we recognize that general speech recognition would have been an incredibly complex task. By making the calculation of the formants more accurate and comparing formant values by Euclidean distance, we would be able to finish.

V. Contributions

Aloysius wrote the sections for initial audio processing, adapted an existing FFT implementation [5] for our project, and performed most of the testing of function modules via comparing his Python implementation to our modules. Jonathan wrote the sections for most function modules with correct pipelines and BRAM memory management.

VI. Retrospection

We learned quite a few lessons from this project:

- We initially had a much bolder goal of recognizing all phonemes in the English language using a mixed Gaussian model, but we quickly realized that the magnitude of work required to label data for such a model would have been a lot of work.
- Writing the algorithm in Python first was a good sanity check on the viability of our approach. It also served as a good baseline of the amount of computation time required. Aloysius had an excellent approach of debugging our SystemVerilog by comparing it to the exact outputs of the Python program.
- Dealing with binary integers was complicated and led to some precision issues, as we had to re-scale numbers many times to avoid over-flowing our 32-bit format. While floating-point arithmetic is generally frowned upon in this class, we wonder if it could have been useful for our project.
- We spent a considerable amount of time optimizing our design before writing anything in SystemVerilog, which paid off. While we still did underestimate the extent of timing issues (multiplication of two 32-bit numbers to a 64-bit number doesn't fit in a single cycle...), this guided us on how to design our (extensive) pipelining that avoided the worst timing issues.
- Start earlier.

Acknowledgment

We thank Joe Steinmeyer and Jan Park for their advice and help throughout this project.

References

[1] M. Stanek and L. Polak, "Algorithms for vowel recognition in fluent speech based on formant positions," in *2013 36th International Conference on Telecommunications and Signal Processing (TSP)*, 2013, pp. 521–525.

[2] S. K. Pal, A. Datta, and D. D. Majumder, "A self-supervised vowel recognition system," *Pattern Recognition*, vol. 12, no. 1, pp. 27–34, 1980.

[3] L. Welling and H. Ney, "Formant estimation for speech recognition," *IEEE Transactions on Speech and Audio Processing*, vol. 6, no. 1, pp. 36–48, 1998.

[4] ETSI, "Speech processing, transmission and quality aspects (stq); distributed speech recognition; front-end feature extraction algorithm; compression algorithms." [Online]. Available: https://www.etsi.org/deliver/etsi_es/201100_201199/201108/01.01.03_60/es_201108v010103p.pdf

[5] nanamake, "Pipeline FFT Implmentation in Verilog HDL," 2024, accessed: 2024-11-26. [Online]. Available: https://github.com/nanamake/r22sdf