

FPDJ Final Report

1st Sean Mintz

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
mintzsea@mit.edu

3rd Brian Shi

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
brianshi@mit.edu

2nd Pravalika Putalapattu

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
prava241@mit.edu

Abstract—We present a design for FPDJ: An FPGA-controlled DJ setup that uses voice commands to alter audio output. In order to achieve this, FPDJ processes the incoming audio using a Fourier transform and continuously constructs a Mel Frequency Cepstral Coefficient representation, which is then compared with pre-computed spectrograms corresponding to keywords using a dynamic programming algorithm. The result is an interactive system that alters its state in response to spotted keywords.

Index Terms—FPGA, keyword-spotting, MFCC, dynamic time warping

I. CONCEPTUAL BACKGROUND

A Mel Spectrogram is a time–frequency representation of audio data that shows how the spectral content of a signal changes over time. To construct it, discrete overlapping windows of audio input are recorded, and a Fast Fourier Transform (FFT) is computed for each window. The resulting frequency data from each window capture the energy levels at different frequency bins and when displayed side by side, reveal the time evolution of the signal’s spectral content. To match human auditory perception, mel spectrograms map the linear FFT frequency bins onto a logarithmic Mel frequency scale, and the energy values in each band are typically converted to a logarithmic scale as well.

For comparison, these mel spectrograms are often converted to Mel Frequency Cepstral Coefficients (MFCC), which involves converting the frequency domain vectors for each time point back into the time domain using the Discrete Cosine Transform. MFCC is a commonly used representation for audio processing due to high levels of accuracy when used with classification.

II. AUDIO PROCESSING PIPELINE

This section discusses the details of how the audio input is compared to the template MFCC data and is processed. Although the implementation is our own and specific to our use case, this follows a well-researched procedure that bears many similarities to [1], [2].

The major challenges present in this project largely have to do with the complexity of this processing pipeline and

the fact that the practical ranges of the data values must be experimentally validated to ensure that enough data is retained at each step to do classification at the end without the values overflowing or using an exorbitant number of bits.

Our overall system diagram is seen in Figure 1.

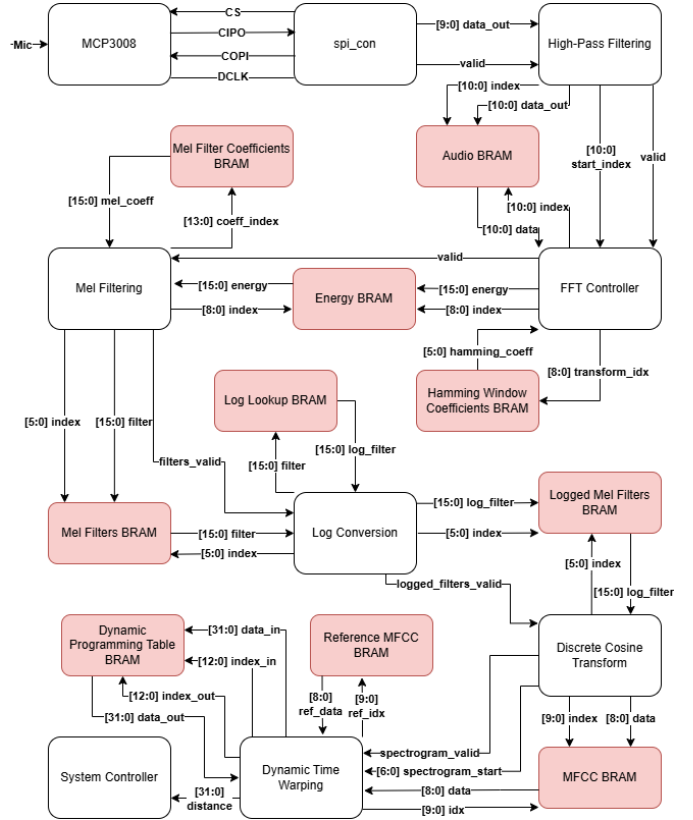


Fig. 1. System Diagram for Audio Processing Pipeline

A. Audio Input

Audio is sampled using a MCP3008 analog-to-digital converter (ADC) at 16 kHz. Each sample produces a 10-bit value representing the discrete amplitude of the microphone signal.

These samples are then transmitted over the SPI interface to the rest of the processing pipeline.

B. Shifting and High-Pass Filtering

Each raw audio sample is re-centered around zero to remove the ADC's DC offset of the unsigned representation. A high-pass filter is then applied to the shifted audio, implemented as a fixed-point multiply and shift. With $\alpha = 0.97$, we seek to apply

$$\text{out}[t] = \text{in}[t] - 0.97 \cdot \text{in}[t - 1]$$

Since $0.97 \cdot 2^7 \approx 124$, we can approximate the above expression with

$$\text{out}[t] = \text{in}[t] - ((124 \cdot \text{in}[t - 1]) \gg 7)$$

The motivation of the high-pass filter is to boost high frequencies relative to low ones since there tends to be less energy in higher frequency bands, and these high frequencies are important for distinguishing human speech patterns [3]. The high-pass filtered data is saved in a BRAM for the next step, and the module signals when a full window is complete.

C. Hamming Windowing

With the high-pass filtered output, we create windows of 512 samples to be a single time point in our spectrogram. These windows have an overlap of (1/2), so the window processing pipeline begins every 256 new samples.

We compute the hamming window of our 512 data points, which de-emphasizes the edges in order to avoid noise. This is done as a dot product of each audio data item with the pre-computed coefficient corresponding to its index.

D. Fast Fourier Transform

The Discrete Fourier Transform is computed using the Fast Fourier Transform algorithm. This is implemented using the Xilinx® LogiCORE™ IP Fast Fourier Transform v 9.1. We write a controller that communicates with this IP via AXIS, sending the 512 points of input data (hamming-windowed audio) and receiving the 512 frequency outputs. Note that, since all inputs are real numbers, the frequency data is symmetric around the Nyquist Frequency, and only the first 257 data points have useful, non-redundant information.

The Hamming windowing from section C. is actually done by our FFT Controller, which communicates with a BRAM holding the coefficients and computes the product of the data and its coefficient before inputting into the FFT IP. This product is a 16 bit number for each data point, which gives a high level of precision without making the FFT IP's computations needlessly expensive.

The output of the FFT IP for each frequency band is actually 2, 16 bit components, one for the real valued energy and another for the imaginary. We square both of these numbers and add them together to get a final energy value for each frequency. With the range of values that are output in practice from the FFT IP, we are able to store this final output as a 16 bit number with very little loss of precision. These 512 numbers are saved in the Energy BRAM for the next step.

E. Mel Filtering

The next step in the generation of a mel spectrogram is to convert the linearly spaced frequency bands from the FFT to be logarithmically spaced (mel scale). This is done via convolution with a series of triangular filters that get wider and more spaced out for larger numbers in a logarithmic fashion. A visualization of this is seen in Figure 2.

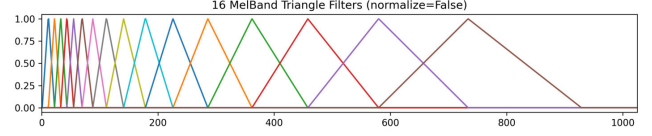


Fig. 2. Unnormalized Mel Filter Functions

In reality, these triangular filters toward the right cover much larger numbers of points, so they need to be area normalized so that frequencies are equally emphasized. This is seen in Figure 3.

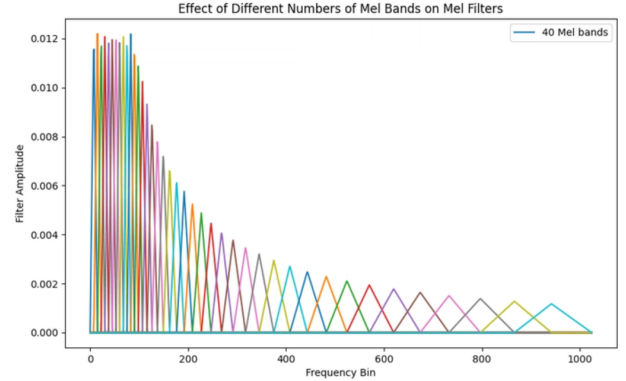


Fig. 3. Area Normalized Mel Filter Functions

We compute 40 mel bands from our original 257 non-redundant frequency bands. Precomputed 16-bit coefficients for each convolution are stored in a BRAM with depth $40 \times 257 = 10280$. These are all normalized such that a convolution multiplies the data by 256, such that the upper 16 bits of the product can be taken as the output value. Each of these 40, 16 bit numbers is written to the Mel Filters BRAM for use in the next step.

Our System Verilog implementation of this computes each of these convolutions sequentially to save resources. The number of cycles this takes is heavily optimized in our implementation by recognizing that the values that filter n uses begin where the values that filter $n - 2$ ended, and that once the first 0 coefficient is seen after seeing a non-zero, the convolution value is final.

F. Log Conversion

Just as we convert to a mel scale so that our frequency axis is perceptually relevant, it is also helpful to convert the actual energy values at each frequency to be a log of their energy values. This is done using a log lookup table stored in BRAM.

16 bit values are used as indices, and an 16 bit pre-computed logarithm is output. These values are saved in another BRAM for the next step.

G. Discrete Cosine Transform

Our final step to obtain an output in the form of Mel Frequency Cepstral Coefficients is to convert the vector of frequency data back into the time domain using the Discrete Cosine Transform (DCT). This is done as a 13×40 matrix multiplication of our length 40 vector of mel bins. The coefficients of the DCT are computed using the following formula:

$$M[k-1][n] = \cos\left(\frac{\pi}{40}(n+0.5)k\right)$$

Where n ranges from 0 to 39 (over the columns) and k ranges from 1 to 13 (over the rows). Note that we start k at 1, since $k=0$ just corresponds to overall signal energy and is often disregarded.

The overall result of this matrix multiplication is a length 13 acoustic vector. The DCT module writes its 13 data points to the MFCC BRAM. This BRAM contains these length 13 vectors for 70 different time points. The DCT module keeps track of the most recent row of this table it has written to and moves this over whenever it finishes writing a new batch of data so that the MFCC always contains the most recent 70 data points. The number 70 is picked so that the MFCC represents a good length of time for comparing single words (with 16 kHz sample rate, 512 window size, and $\frac{1}{2}$ overlap between windows, 70 rows gives $\frac{(70+1) \times 256}{16000} = 1.136$ seconds). The DCT module also passes on this start point to the next module so that it is known what row of the MFCC data should be interpreted as the first.

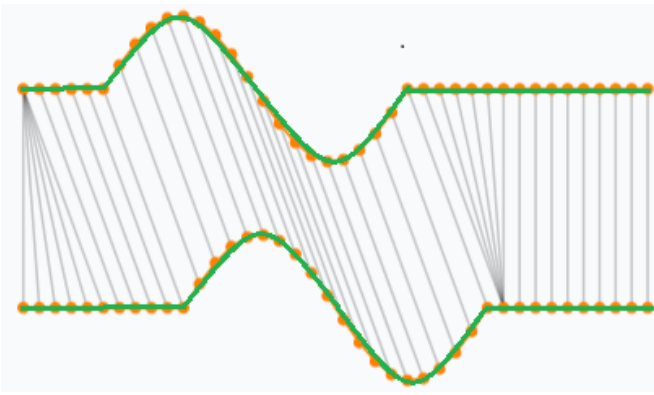


Fig. 4. Dynamic Time Warping Visualization

H. Dynamic Time Warping

Finally, we must compare the the generated values in the MFCC BRAM with two different template MFCCs, each one corresponding to the spoken commands “stop” and “next”, respectively. We perform the comparison using Dynamic Time Warping (DTW), a dynamic programming algorithm that finds the closest matching of 2 different input strings in a way that

is allowed to skip indices of either string [4], [5]. This allows the matching to be translation and speed invariant, which is useful since speech varies in speed, timing, and pronunciation. A visualization of this is given in Figure 4.

The DTW update is the following recurrence:

$$D(i, j) = d(i, j) + \min(D(i-1, j), D(i, j-1), D(i-1, j-1)),$$

where $D(i, j)$ is the total accumulated DTW cost and $d(i, j)$ is the local per-frame distance between MFCC vectors.

Our vector distance is implemented as squared Euclidean distance, which simplifies the computation by avoiding the use of a square root while still having roughly the same properties:

$$d(i, j) = \sum_{k=0}^{12} (data[i, k] - ref[j, k])^2$$

$data$ denotes the incoming MFCC data and ref denotes the reference MFCC. This formula is just the Euclidean distance of two vectors $data[i]$ and $ref[j]$ without taking the square root of the result.

The base case of DTW is $D(0, 0) = d(0, 0)$, as this just considers then first item. Other special cases are $D(i, 0)$ and $D(0, j)$, which only consider $D(i-1, 0)$ and $D(0, j-1)$, respectively. Our final result, returned as a 32 bit distance score, is $D(69, 69)$, the minimum possible matching of the MFCC data with the reference.

In our DTW implementation, we complete the DP table from left to right, top to bottom. As we calculate the cumulative distance for each (i, j) , we optimize by reading input and reference mel vector elements two at a time from the respective BRAMs to reduce the number of required cycles. Moreover, while $D(i, j)$ depends on three adjacent elements, we only query for the top entry $D(i-1, j)$ from the DTW results BRAM since the new leftmost element was the previously computed element, and the new diagonal element was the previous top element.

I. System Controller

A system controller sits downstream of the DTW module and decides how to modify the system state. It receives the DTW costs for a “stop” command and a “next” command. If the DTW cost for a command drops below a threshold, the system controller interprets the audio as a match and modifies the state machine appropriately. Given two streamed music samples from external music BRAMs, the controller decides whether to keep playing, switch tracks, or stop.

If we receive a positive read for “next”, our system switches the audio output track and enters a cooldown period of 200,000,000 cycles (2s) to avoid double-counting subsequent positive “next” signals from the DTW that are tied to the same spoken command. For the “stop” command, we simply zero out the speaker output and transition to the stopped state since the “stop” command is idempotent.

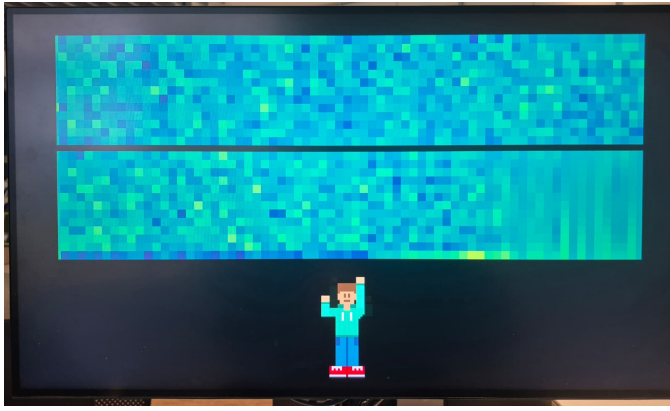


Fig. 5. Live visual display, including input audio MFCC, reference MFCC, and dancing sprite

J. HDMI Display

We also display a live visualization of the audio via HDMI while the system is running, shown in Figure 5. The upper band of the display is a rolling window of the MFCC calculations on live audio. The lower band is the reference MFCC data for the “stop” command. We use two BRAM lookups for this display, one to fetch the appropriate MFCC value for the pixel, and one to convert the MFCC value to the appropriate color. We have also implemented a dancing sprite, using a sprite selection variable to switch between the two frames of the image. The upper band stops rolling and the figure stops dancing upon use of the “stop” command.

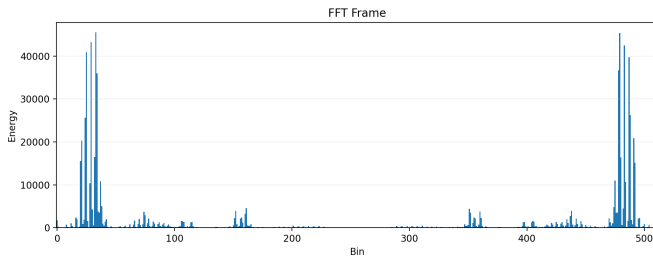


Fig. 6. FFT Live Audio Visualization

III. DATA COLLECTION AND EXPERIMENTS

We implement a multi-byte UART transmitter to send data samples from LSB to MSB to the computer; and a BRAM UART reader to inspect BRAM data directly. To ensure that bytes are grouped and interpreted correctly on the receiving side, we use switches to control transmission: we start transmission on the FPGA by flipping the switch only when the sampling script is already running on the computer.

The system requires pre-loaded audio and template data. To build out the MFCC reference templates, we live record the “stop” and “next” commands using the multi-byte UART transmitter, which then temporarily serves as the input data source instead of live microphone input when collecting data. We feed these audio samples through our entire data pipeline

to generate a FFT output for a given window and a MFCC output for the entire audio sample. These results serve as the dataset for visualizations and the references for the DTW algorithm. Figure 6 highlights the FFT outputs for an arbitrary given window of the dataset. We can observe that in the first half of bins, the energy is largely clustered around smaller frequency bins. We can disregard the upper half of bins.

Figure 7 is a diagram of the different MFCC/DCT outputs for three different recordings (“stop”, “stop” alternate, and “duck”) and their respective DTW DP matrix heatmaps when compared to “stop”. Qualitatively, the “stop” and “stop” alternate recordings generate reasonably similar MFCC heat maps, and they both differ significantly from the “duck” MFCC heatmap. The DTW DP matrices also display expected results: for “stop” vs “stop”, the matrix is symmetric and zero-diagonal. Moreover, the final cost for the “stop” alternate recording was smaller compared to the “duck” command.

The final DTW costs of the recordings when compared to the “stop” reference were A) 0, B) 2,017,101, and C) 2,607,327. Using these results and further live testing, we set a DTW cost threshold for positive detection at 2,200,000. We observed that in normal conversation, none of our commands were accidentally triggered. To minimize conflicts, we selected two very distinctly stressed words (“stop” and “next”) as our system words.

IV. PERFORMANCE WITHOUT DTW

We evaluated the performance of the system without using dynamic time warping and instead just directly matching each index of the template MFCC with the incoming MFCC. In order to do this, another module called DTW simple was implemented with the same API as the DTW module.

The motivation behind this was to evaluate the conventional wisdom that DTW is the best way to check if a keyword was said. Based on the fact that DTW on completely different words is able to get a distance that’s only slightly higher than DTW on the same word, one may wonder if DTW is not restrictive enough and allows low distance matchings even for completely different signals. Since we will compare every spectrogram at each time point with the reference anyways, it was worth evaluating how well keyword spotting can work without any help from DTW.

We observed that saying the matching word without DTW does not result in a reduced distance compared to other noise, and suggests that DTW is in fact necessary to prevent small changes in intonation and speed from preventing words being recognized as the same. All of the distance values hovered above 3 million in this case and didn’t change noticeably in response to the key word. We conclude that DTW is necessary in order to recognize speech using MFCC. However, there are certainly extensions to DTW that can restrict the algorithm’s ability to distort the data and prevent the algorithm from matching wildly different inputs and would help better separate our matches from our noise [6].

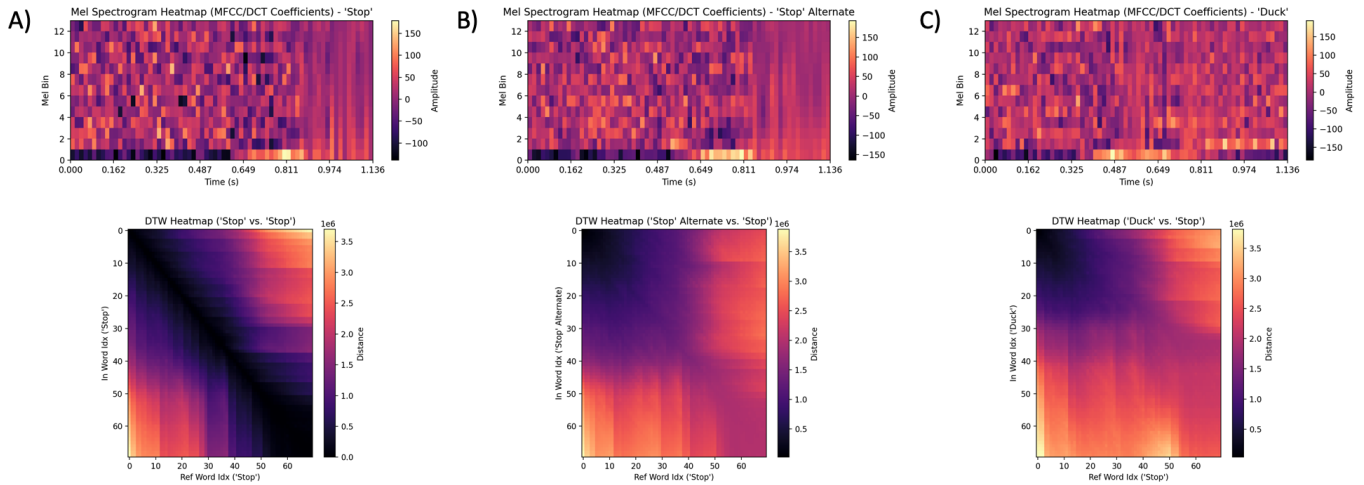


Fig. 7. MFCC/DCT Coeffs and DTW DP Matrix Heatmap: (A) "Stop", (B) "Stop" (alternate), (C) "Duck"

V. OVERALL SYSTEM EVALUATION

Our total utilization when running this system with two keywords and the full display pipeline included 62 RAM36 blocks, 11 RAM18 blocks, 14 DSP blocks, 3069 LUTs, and 2279 Flip Flops. The BRAM usage is largely split up over the following major components:

- 1) Stored songs, which are each 5 second segments that each fit in 10 RAM36 blocks
- 2) The log lookup table, which uses 16 RAM36 blocks to ensure high precision
- 3) The mel filterbank, which takes 8 RAM36 blocks to store the coefficients for these dot-products.
- 4) The DTW tables, which take 8 RAM36 blocks for every keyword we compare against.

Around half of the Flip Flop usage (999 FFs) is due to the Xilinx FFT, as it is configured to preserve BRAM when possible. Another quarter (452) is attributed to the two instances of DTW. The rest of the usage is spread out quite evenly through the other components.

Similarly, for LUT usage, the FFT and the DTW utilize the most resources. This time, the DTW uses the most (1262) followed by the FFT (560), with the contributions from other components being much smaller.

We were able to synthesize the entire system with a worst slack value of 0.883 ns, meeting the timing constraints of the FPGA by a safe margin.

The latency of our pipeline is such that, from the perspective of the end user, the commands are processed without noticeable delay. The process of running the FFT takes under 7,000 clock cycles. Mel Filtering takes under 1,700. Log conversion takes around 50 cycles. DCT takes under 1,600 cycles. Finally, DTW takes 177,000 cycles. Overall, the latency of our pipeline from getting a new set of samples that we want to process to modifying our state machine is under 190,000 clock cycles, which is less than 2 ms.

As a note for future implementers, our implementations do not heavily optimize for latency, and it is certainly possible to reduce many of these values at the expense of more resources and still fit this on the Spartan-7 FPGA. Particularly for DTW, this is a potential area of extension.

Moreover, we observed that our memory constraints limited the number of keywords we could support. To add additional keywords, further work can also reduce the memory footprint of the DTW algorithm. For example, rather than tracking the entire 70×70 DTW DP table, we simply need to track two rows that alternate as the top and current row.

VI. IMPLEMENTATION INSIGHTS

We extensively used Icarus for python testbenching, and additionally utilized the Vicoco simulation tool for testing integration with Xilinx IPs [7]. All modules are unit tested to observe that their behavior is correct and that their state machine is automatically reset after each use (running twice does not effect correctness). In addition to unit testing, it was also helpful to perform integration tests using the UART reader to ensure that we were getting expected outputs.

Often, when running integration tests and data processing with UART, we would receive junk or uninterpretable data. To address these issues, we needed to take a step back to understand the possible root causes and bugs that would be leading to the incorrect results. The source of errors were often incorrect receiver settings and bug errors with the code. To differentiate between the two, writing high quality test benches and separate Python implementations to sanity check our FPGA logic helped immensely. At a high level, we found the procedure of unit testing, then isolated integration tests with pre-loaded data, then finally live testing to be the most effective workflow.

VII. CONCLUSION

We introduce FPDJ, an end-to-end FPGA keyword-spotting system that enables voice control of audio playback. FPDJ

compares MFCC data using the DTW algorithm to match live audio data against command templates. Our implementation successfully achieved the initial commitment and stretch goals set out in the project checklist. We built a functional audio processing pipeline and accurately recognized the “stop” command. Moreover, we met our stretch goals by extending FPDJ to further recognize the “next” command and designed an engaging visual representation that is responsive to the voice commands.

Further work could expand the system’s vocabulary set and refine the DTW algorithm for more confident thresholding and command recognition. Moreover, while we demonstrate a specific use case in audio control, our voice command recognition architecture lays the foundation for further applications in other domains ranging from robotics to assistive technologies. Ultimately, FPDJ demonstrates efficient, hardware-centric voice control for a diverse potential set of systems.

VIII. CODE REPOSITORY

Code: <https://github.mit.edu/6205F25/fa25-6205-team00.git>

IX. INDIVIDUAL CONTRIBUTIONS

A. Code

Brian worked on audio I/O, the pre-emphasis filter, DTW, system controller, and data integration.

Sean worked on the FFT controller, the mel filtering module, the log conversion module, and the discrete cosine transform module.

Pravi worked on Euclidean distance calculations, DTW, MFCC visualizations, and sprite visualizations.

B. Report

Brian wrote the descriptions for Audio Input, Shifting and High-Pass Filtering, Dynamic Time Warping, System Controller, and Data Collection and Experiments.

Sean wrote the descriptions for the Hamming Windowing, FFT, Mel Filtering, Log Conversion, and DCT, and Performance Without DTW. Additionally, he drew the system diagram.

Pravi wrote the description for HDMI Display and worked on the Conceptual Background and the description for Dynamic Time Warping.

REFERENCES

- [1] D. -F. Syu, S. -W. Syu, S. -J. Ruan, Y. -C. Huang and C. -K. Yang, “FPGA implementation of automatic speech recognition system in a car environment,” 2015 IEEE 4th Global Conference on Consumer Electronics (GCCE), Osaka, Japan, 2015, pp. 485-486, doi: 10.1109/GCCE.2015.7398519. 1
- [2] M. Bahoura and H. Ezzaidi, “Hardware implementation of MFCC feature extraction for respiratory sounds analysis,” 2013 8th International Workshop on Systems, Signal Processing and their Applications (WoSSPA), Algiers, Algeria, 2013, pp. 226-229, doi: 10.1109/WoSSPA.2013.6602366. 1
- [3] H. Fayek, “Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What’s In-Between,” Haytham Fayek Blog, 2016. Available: <https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html>. 2
- [4] R. Tavenard, “An introduction to Dynamic Time Warping,” Romain Tavenard Blog, 2021. Available: <https://rtavenar.github.io/blog/dtw.html>. 3
- [5] M. Picheny, B. Ramabhadran, S. Chen, “Lecture 2: Signal Processing and Dynamic Time Warping,” IBM T.J. Watson Research Center, 2012. Available: <https://www.ee.columbia.edu/~stanchen/fall12/e6870/slides/lecture2.pdf>. 3
- [6] H. Sakoe and S. Chiba, “Dynamic programming algorithm optimization for spoken word recognition,” in IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 26, no. 1, pp. 43-49, February 1978, doi: 10.1109/TASSP.1978.1163055. 4
- [7] K. Vuksanaj, “Vicoco.” Available: <https://github.com/kiran-vuksanaj/vicoco>. 5