Flat Panel Smart Speaker

Ben Kevelson, Jun Qiu, Julia Weinstock, Zilin Zeng

Faculty Advisors: Dr. Michael Heilemann, Dr. Sarah Smith, and Professor Dan Phinney
University of Rochester Department of Electrical and Computer Engineering
Audio and Music Engineering Senior Design Project

May 8th, 2022

I. Introduction

A. Problem Description

Flat panel technology has been an emerging field of research at the University of Rochester over the past few years, and recent developments have shown the promise of flat panels as acoustic surfaces. This project aims to demonstrate the multifaceted capabilities of flat panel technology by building a standalone smart speaker, which utilizes a compound speaker and microphone on a single panel. The speaker should be able to handle voice commands from the user, such as playing and pausing music, getting the weather, and setting timers.

Commercially available smart speakers, such as the Apple Homepod mini, are usually cylindrical or round in shape, and are advertised to sit on the counter or shelf in the consumer's home [1]. An advantage of a flat panel smart speaker is that it has a relatively slim profile and can be mounted on a wall to take up minimal counter space. Popular smart speakers contain speakers and an array of microphones to play and detect sound in all directions. The flat panel can be used both as a speaker and a microphone, and can exploit fixed boundary conditions to utilize only a single sensor to detect sound, rather than an array.

B. Background on Flat Panel Technology

The basis of using flat panels as a speaker hinges on the acoustic properties of flat panels themselves. Usually built out of materials such as acrylic, glass and gatorboard, flat panels can then be mounted on a rectangular frame. Under these conditions, the frame can be assumed to act as a sort of "infinite baffle", meaning that the area within the frame is the only relevant part when talking about vibrations. In order to induce vibration on the panel surface and create sound, driver devices are mounted on the rear of the panel's front-facing surface, and can then be manipulated via electric current to vibrate; the subsequent vibration of the panel's surface creates sound. Additionally, external sound sources can also induce vibrations in the panel, which can be interpreted as sound via a piezoelectric sensor which is also mounted to the rear of the panel. For our purposes, these are the main functional components of a flat panel loudspeaker. One of the main advantages of this technology is that the panel surface can act as both a speaker and a microphone due to its capability to both receive and create sound.

Although the nature of flat panel speakers gives them certain advantages over conventional speakers, there are shortcomings in other areas. Using a non-optimized driver configuration, a flat panel speaker is going to have a less consistent frequency response and loudness level compared to popular alternatives, particularly when talking about low frequencies. In order to compensate for this, researchers at the University of Rochester use an array of drivers arranged in predetermined locations on the rear of the panel. By hooking them up together such that they emit the same signal, the drivers are able to cancel certain modes of the panel's vibration, which can accentuate lower modes and bring more consistency to the panel's frequency response. Another driver can then be placed in another predetermined location to control higher frequencies. When used in tandem with a software-based crossover network to split output signals into high-frequency and low-frequency streams (see Section II-B for more details on the crossover network), this specialized driver setup is able to improve on the general frequency response of the panel, particularly for lower frequencies.

While our project aims to use flat panel technology as a vessel for a smart speaker device, the potential of the technology remains largely speculative. Recent research from the University of Rochester has shown that flat panel devices are able to effectively extrapolate the direction of arrival for incident sounds using one or more sensors, which could have massive applications within the realm of a flat panel smart speaker. Many smart assistants on the market already use "directed listening" to tune out irrelevant sound sources when listening for commands, and flat panel technology could feasibly do the same thing. However, due to the emergent nature of the research and the limited timeframe of our project, we were unable to attempt implementation of these methods in our smart speaker.

Another potential use for flat panels is as a haptic interface. While this is beyond the scope of our work specifically, research out of the University of Rochester has shown that flat panels can be effective in extrapolating touch location on a panel's surface using a driver array mounted to the rear. This is another application that could potentially have uses for a smart speaker in the future, as it could allow for users to interact with the speaker using touch without the need for traditional buttons and interfaces. Much like the previous research, however, there is much work to be done in this area and the technology was not ready for immediate use.

All in all, flat panel speakers are an older concept that has been brought back into the light by new research and breakthroughs. Due to the nature of their limited use and circulation,

we suspect there are many advancements that are yet to be made in their development and optimization. Our project attempts to highlight a few of the panel's capabilities, and showcase the potential of the technology in consumer use.

C. Proposed Solution

The flat panel smart speaker needs to be a duplex system in terms of its hardware. It needs to have drivers attached to the back of the panel to make the panel sound by pushing the membrane back and forth (just like how a loudspeaker cone operates with a vibrating diaphragm). It needs a piezo mounted to the back of the panel to receive the vibrations on the panel (just like how a microphone records sound by detecting vibrations (pressure change) on its diaphragm). The software for the speaker needs to include a music cancellation program to cancel out interfering noise in the speech and a voice transcription program to transcribe the user's speech. After speech transcription, our smart speaker should be able to respond to the user's commands by playing the user-selected music or answering simple questions related to time and weather. The hardware and the software should be bridged with an audio interface, and our software should be installed in a PC or in a Raspberry Pi.

D. Constraints

Over the course of building flat panel smart speakers, there were several limiting factors that impacted the proceeding of our project. First of all, the fact that all the matlab codes we wrote can only be tested after our actual panel is built consequenced to insufficient amount of debugging time. Although we ran simulations and made sure the code works in our simulations of an actual panel. There were plenty of unexpected problems that took us a lot of time to fix. Second of all, the use of IBM Watson voice recognition service requires a price and we only had one account that is paid by the school, which means all the code for the panel assistant also can not be tested. Hardware wise, the PBC, printed circuit board, of the piezo amplifier that we designed and ordered took weeks to arrive. For demo day, we had no choice but to use the perf board version of our piezo amplifier.

E. Impact

Before starting work on the project, we first took into consideration the potential long and

short term impacts of this project.

The global impacts of this project are mainly centered around the promise of flat-panel technology. If we assume that the flat-panel smart speaker were to be widely used globally, it would likely bring innovation to the way that speakers and audio devices are designed, as well as changing the way hardware is designed. Additionally, the other applications of flat-panel tech may also come to impact the world once it is adopted globally, such as waterproofing for electronic devices, innovations in haptics, and advancements in source detection.

The economic impacts of this project could be quite large in theory, as there is a growing multi-million dollar market for home audio devices. If the flat-panel smart speaker were to be successful commercially, it would likely have a stake in that market, and could even drive other manufacturers to pursue similar ideas and applications of flat-panel technology.

The environmental impacts of this project are mostly centered in the manufacturing, as the material choices could have large environmental impact if the speaker were to ever go into mass production. For instance, materials such as acrylic (which has been explored as a panel material for this project) is not generally considered environmentally friendly. On the other hand, glass (which has also been looked into) is much more environmentally friendly and is 100% recyclable. These considerations need to be weighed against performance and taken into account for the final model, as well as other material choices such as the frame, drivers, sensors, and electronics.

The social impacts of this project are less obvious. One could argue that the social impact would be seen in the perception of this technology, and how consumers view the functionality of their technology. The most pressing impact is the virtual assistant's interaction with consumers and how that could potentially shape how people interact with intelligent conversational agents.

II. Technical Challenges and Solutions

A. Software

1. Virtual Assistant

The software components of this project hinge largely on the use of a virtual assistant software, which handles all the voice recognition, responses, music/sound playback, etc. Many smart speakers/virtual assistant devices on the market use their own virtual assistant software, such as Amazon's Alexa and Google Assistant. These softwares are finely tuned for the

hardware parameters of their respective devices (Amazon Echo and Google Home respectively), and are highly advanced in their natural language processing and interactive capabilities. Our initial goal for this project was to use the open source version of one of these softwares, which companies like Google and Amazon release on the internet for free. A previous iteration of a flat panel smart speaker designed in the summer of 2021 used the Alexa open-source software loaded onto a Raspberry Pi 4 Model B microprocessor. The input and output was interfaced through the necessary hardware components to ensure performance and functionality (see Section II-B for more details on the hardware configuration), and the device was able to perform most of the basic functions of a virtual assistant (See Figure 1 below).

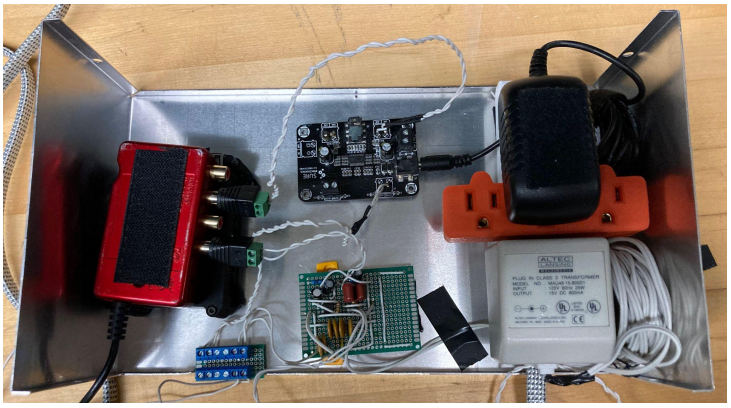

Figure 1: Hardware setup for the previous implementation of a flat panel smart speaker.

The intent of our project was to expand on the shortcomings of the previous model, and optimize the panel performance based on new research from the University of Rochester. In particular, the main shortcoming of the previous model was an inability to play music on command (as in, “Hey Alexa, play Bruno Mars”). This was due to the fact that the open-source demo of the Alexa technology does not have Spotify connectivity, which is what the real Alexa software uses as its music API. We explored various roundabouts to this issue, such as hacking the source code with our own Spotify API key, as well as loading in our own music library and creating a custom set of commands that would queue them. Ultimately, this proved unsuccessful due to the nature of the Alexa open-source demo – much of the code and functionality is locked within their cloud service, and the user end implements streaming connectivity to access that processing in such a way that casual users would be unable to edit certain operative components of the source code. We also explored the potential of the Google Assistant software for

application within the panel assistant, but encountered many of the same problems as with the Alexa software. We were ultimately unable to make any major progress using the Google Assistant, so we abandoned that software and pursued other avenues.

Another reason the open-source software was a nonviable option was that the cloud-based nature of the software prevented us from optimizing the program for panel technology specifically. One of the major hurdles of the project, the real-time cancellation of played music (see Section II-A-3 for more details on the real-time cancellation implementation), was a component that we felt was best implemented within the software of the virtual assistant. Any optimizations or changes we would have to make within the code would have to be in post-processing on the output/input streams of the Alexa software, which was difficult to intercept within our hardware and brought about many technical challenges. For these reasons, we ultimately decided to abandon the open-source virtual assistant demos, and pursue a customized avenue for our virtual assistant software needs.

This brings us to “panelassistant.m”, a MATLAB program developed in-house for the specific challenges that flat panel smart speakers present (panelassistant.m will be referred to as Panel Assistant for clarity’s sake). This was designed largely by Tre DiPassio, and we can’t express enough how pivotal his contributions have been to the progress of this project. The program uses the IBM Watson Text-to-speech API as a transcription method, and employs hard-coded response messages that are turned into voice responses via the IBM Watson API (Appendix A). Once the text from the user is transcribed, the program uses string comparison to estimate the pre-coded command that the input command is most similar to (Appendix B). Additionally, pre-processing code is implemented in order to recognize the wake word (we used “Alexa” in order to demonstrate the comparability of our product with mainstream market competitors, as well as its recognizability by the IBM Watson service) as well as format the input string for comparison. This includes methods to clip the wake word from the input command, as well as turning numerical characters into spelled-out numbers (Appendix C). A post-processing function is also present, which categorizes the expected command using confidence scores and index numbers and validates the estimated command by parsing it for keywords pertaining to each of the dictionary commands (Appendix D). Once the processing portion is finished, the recognized command is run through the general command function, which can perform various

tasks based on the command such as creating and running timers, mathematical operations, witty quips about sports and niche references to members within the sphere of the project, and more.

The program can also play music based off of input commands, which was a core focus of the project and one of the biggest hurdles we encountered. Since our own program was not subject to the software restrictions that the open-source programs were prone to, we were able to load in our own music files downloaded from the internet and adapt the dictionary commands to include play requests for individual songs and various artists. Since the functionality required for this process is unique and differs from the scope of the other general commands, we made our own play function (Appendix E) which queues songs based on the input command and initializes objects within the Panel Assistant code for real-time playback and cancellation measures (see Section II-A-3). If a play command is triggered, an internal flag is also tripped which changes the listening loop to a program that would implement active cancellation. Once the song finishes, the flag would be reset, and the listening loop would return to a normal method.

These features make up the main functionality of the Panel Assistant software. Although there is more code that does not exist in the documentation as well as additional helper functions, the referenced code and operative sections make up the majority of the functionality. Additional improvements to the software could be made, in the sense that the Panel Assistant software does not really have any user-end natural language processing capabilities. The transcription of the speech is all handled through the IBM Watson API, so the tuning of the AI itself and the transcription process is largely inaccessible by our team and we are unable to make edits to it. This presents some logical difficulties, because all of the input command processing we do is based on the text transcription, which limits our ability to tune the process for accuracy and versatility of speaking styles. For instance, the dictionary keys contain certain commands that are repeat versions of other commands, or are a variation of how one could feasibly say it (See Appendix F). For these sorts of cases, the program cannot adaptively interpret the spoken phrase and instead relies on a variety of finely-tuned text string comparisons. This makes it hard to adapt the model to a larger range of commands and uses, since all of the command phrases are hard-coded into the dictionary.

Another shortcoming of the software is due to the nature of its implementation in MATLAB. Since it is a higher level scripting language, implementation onto a smaller piece of hardware such as a microcontroller or a digital signal processor would be significantly more

difficult. The final setup of our project used MATLAB running on a Mac laptop, which is very bulky and not suited for out-of-the-box consumer use. A future version of this project could transcribe all of the code's functionality into Python or another language that can be easily implemented onto smaller hardware. Indeed, many of the function calls within the Panel Assistant software are imported from Python libraries anyway, so it makes sense that there would be some cross-compatibility between the languages. The optimal language may actually be something that runs 'closer to the metal' (such as C), specifically when talking about the processing intensity of the real-time cancellation (see Section II-A-3). However, there may be other roadblocks standing between a code translation from MATLAB into another language, so the concept went unexplored within our timeframe.

2. Crossover Network

In order to select a reliable crossover network for the flat-panel smart speaker, we did research on various types of crossover network that are available for our use, active crossovers, passive crossovers, and DSP (digital signal processing) crossovers. An active crossover splits audio signal into different frequency bands before sending each band to its dedicated amplifier, therefore it is installed between the audio source and the amplifiers. An active crossover processes line level audio signals at its input, which opens up possibilities of utilizing op-amps in their design and have less robust inductors, capacitors and resistors. However, due to its features of filtering signals before amplification, each frequency band and driver is assigned to an individual amplifier. This arrangement prevents energy waste by only amplifying frequencies that will ultimately be reproduced by the driver. At the same time, it guarantees more control over the volume and sometimes frequency of the input signal going into different amplifying stages. Another feature worth mentioning is that the use of active crossover simplifies the matching of impedance between the amplifiers and drivers. Also, an active crossover requires external power supply due to the presence of an operational amplifier.

On the contrary, a passive amplifier splits audio signals into different frequency bands after the signals are amplified, which means it is installed between the amplifiers and drivers. As a result, passive amplifiers are dealing with already-amplified signals, which greatly increases its possibility to overheat, even failing when the signal is too high. Also, since the input signal for passive crossover is so hot, it has to contain robust electronic components. However, one benefit

that passive crossover has over active crossover is the fact that a passive crossover does not require external power supply since it is mainly made of passive electrical components including inductors, capacitors, and resistors. In terms of control, passive crossovers are often designed to work at specific crossover points and are often made from specific speakers, which in our case could be beneficial due to the fact that we know exactly which exciter to use. The graph in Figure 2 is a chart of pros and cons of two types of crossover network.

Active Crossovers	**Passive Crossovers**
Require external power to function.	Do not require external power to function.
Deal with line level audio signals at their input.	Deal with speaker level audio signals.
Filter before amplification. Each driver gets its own amplifier.	Filter already-amplified signals. Rely on external amps.
Greater control over the volume of each band.	No control over the volume of each band.

Figure 2: Comparison between active and passive crossovers [2]

Using an active crossover seems like a more reliable solution to our crossover network problem compared to using a passive crossover. Although active crossover requires external power for an operational amplifier and multiple amplifiers for drivers, it is relatively more reliable and functional since it deals with line level audio signal and at the same time, allows us to manipulate volume at each band. On the contrary, a passive crossover does not require external power to function, but it needs more robust components since it deals with high voltage input, and we do not have any control over volume of each band. Although both active and passive crossover have their own specialities, the question of whether one of these two types of crossover network is actually what our team is looking for raises as we do more and more research.

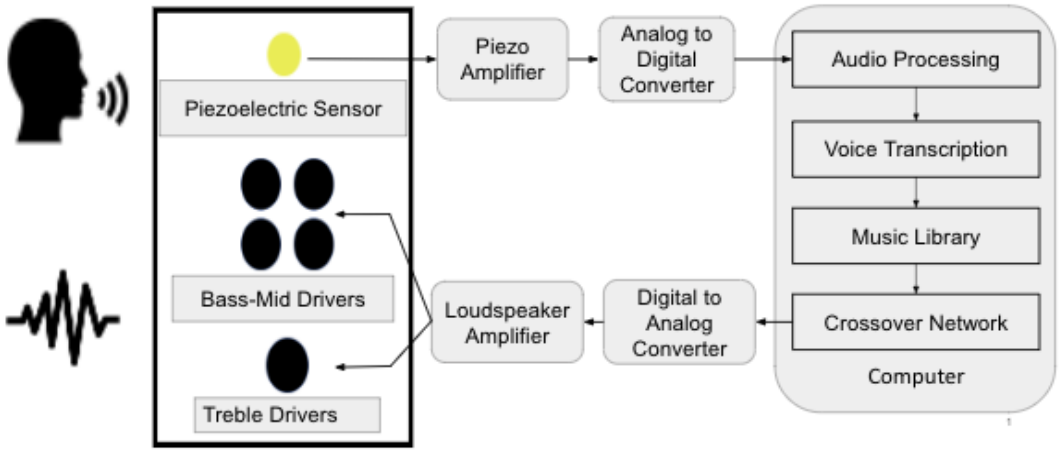

Figure 3: Signal flow for the flat panel smartspeaker

From Figure 3, we can see that there is one step that the signal flows from a piezo amplifier to an analog to digital converter, which opens up the field of digital signal processing crossover network to our team. A digital crossover divides the audio signal into multiple frequency bands on a digital level. It requires an Analog to Digital converter for it to function. Since we are using an audio interface in the first place, the problem of AD conversion is no longer within our concern. Using a digital crossover is very similar to using an active crossover. It has almost all the advantages an active crossover network has. On top of that, it does not require external power supply since we are using an interface. We also have the privilege of constimizing parameters of the crossover with ease. Due to the difficulty of knowing the exact crossover frequency before the panel is built, using a digital crossover network gives us the freedom to start with coding the crossover program using an estimated cutoff frequency first. Unlike an active or passive crossover which we have to know the exact crossover frequency before building the network.

In terms of coding the crossover network in matlab, we used the crossoverFilter System object to implement an audio crossover filter for splitting audio signal into two frequency bands with a crossover frequency of 825 Hz, which we measured to be the optimized crossover frequency of our panel using a vibrometer.

There are four parameters that we can adjust in the crossoverFilter function, ‘NumCrossovers’, ‘CrossoverFrequencies’, ‘CrossoverSlopes’, and ‘SampleRate’. In our case

we simply set ‘NumCrossovers’ as 1 because we are dividing the entire frequency range into two frequency bands, so the number of magnitude response band crossings is one. ‘CrossoverFrequencies’ in our case is 825 Hz. ‘CrossoverSlopes’ determine the rate at which the audio level increases or decreases per octave as the frequency increases or decreases. The scalar number of slopes is not one of the decisive factors that affect the overall audio quality, 12dB/octave and 24dB/octave are all good starting points. We happened to choose 48dB/octave. ‘SampleRate’ is consistent throughout the program, and in our case is 44100 Hz.

3. Music Cancellation

As mentioned previously, the main purpose for this project was to make a smart speaker that can demonstrate the technological capabilities of flat panel technology, meaning both their ability to act as a conduit for a smart assistant as well as a quality speaker. The obvious choice for doing this was to add music commands to our smart assistant, as almost all smart assistants on the market today have Spotify or Apple Music connectivity so the device can interpret commands such as “Alexa, play Bruno Mars”. Since all smart speakers generally have both a built-in speaker and microphone (which is obvious given the nature of smart devices), the microphones need to be able to interpret new commands while also playing music. While this seems like a trivial problem at first, the device actually needs to be very well tuned in order to be able to receive new commands while playing music with comparable levels of clarity to when it isn’t playing music.

To this end, modern smart speaker devices implement real-time cancellation algorithms which are specific to the acoustic parameters of the devices themselves. While the exact mechanisms of these algorithms are protected behind the cloud services of their respective companies, it logically pertains to the fact that the device knows the music signal it will be playing through its own speakers. The microphones also receive this sound, and need a way to cancel that from the input stream so it hears everything besides the music playing. Some methods for doing this include pure subtraction (replacing the input at the microphone and directly subtracting the estimated music signal from said input), spectral subtraction (a similar method which instead subtracts an estimated noise spectrum from the input spectrum, then reconstructs the subtracted signal into time domain), and machine learning-based approaches.

In the context of flat panel smart speakers, this presented a large challenge to our team, since a method of real-time cancellation has not yet been implemented onto a flat panel device. We surmised that a pure subtraction based method could be very effective in flat panel smart speakers, since the piezo sensor and the driver devices are in fixed positions. This means that the delay between the drivers playing the music and the sensor receiving the music should be a constant value, and a convolution of the panel's prerecorded impulse response with the played music signal could be an accurate representation of the signal that the sensor sees when playing music. An experiment was performed in MATLAB using an analog setup of a cone speaker and flat panel speaker. The flat panel was equipped with both drivers and sensors, and played various types of music which the sensor received. An external cone speaker then played spoken phrases, with the aim to decode a clean version of the spoken audio from the sensor input. We then designed a program that would attempt to isolate the spoken phrases from the music played through the panel by means of pure subtraction (Appendix G). The program uses a cross-correlation function in MATLAB, which compares the input at the sensor with the raw music file, to estimate the delay between the sensors and the drivers. A variety of sample values are tested to find the value that minimizes the cross-correlation between these two signals, as well as an optimization for gain values of the estimated output. Once the optimal gain and delay values are found, the delay and gain are applied to the known output signal and subtracted from the input at the piezo sensor. Figure 4 shows the result of the alignment of the signals:

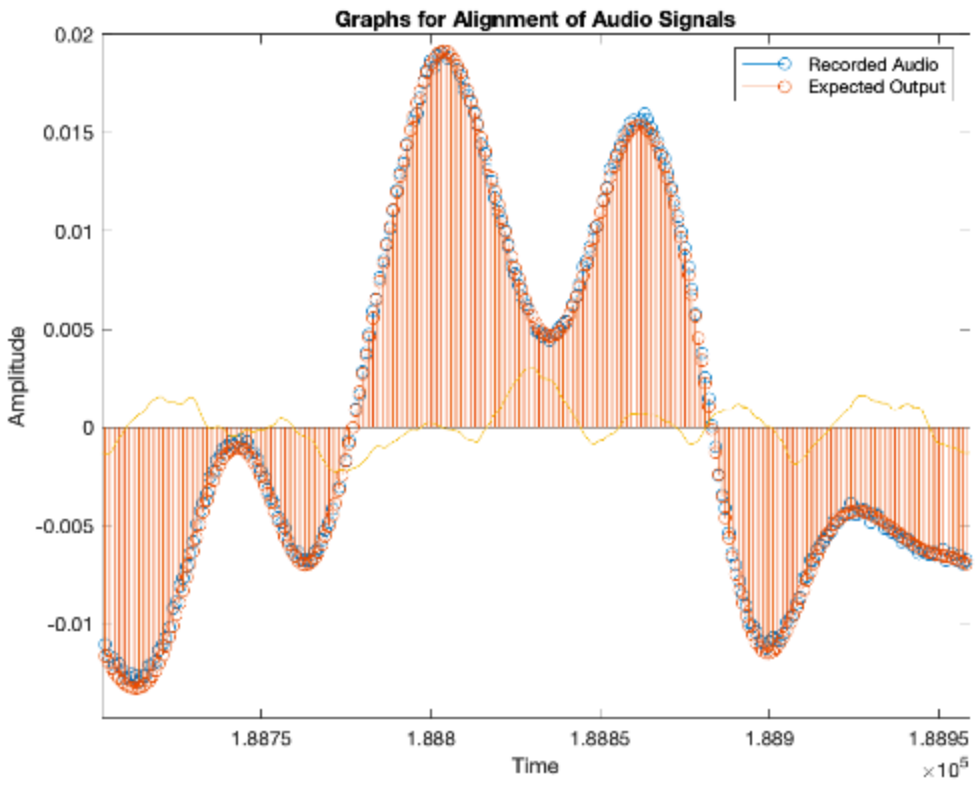

Figure 4: Graph showing the alignment of both signals.

As the graph shows, the alignment method between the recorded audio and the estimated output is quite effective. Use of the pure subtraction method yielded clearly intelligible speech which the IBM Watson API was able to consistently interpret, and this validated our decision to use pure subtraction methodology for real time cancellation.

Once we had confirmed the potential for pure subtraction in flat panel technology, we created a program that more closely resembled real time cancellation (Appendix H). As opposed to the previous experiment, this setup uses audio read in from a longer file on a frame-by-frame basis. This more closely simulated the frame-by-frame nature of the input audio stream coming from the sensor. One new concept we implement in this program is the use of a helper class, AssistantAudioHandler.m (referred to hereafter as AAH). The reason we originally designed this class was to consolidate audio functionality in Panel Assistant such that all the audio related functions were in a single, easy-to-read file. In particular, certain audio functions such as sound() and soundsc() were utilized in the first build of Panel Assistant, which presents a problem for real-time cancellation since they pause the whole program in order to run an entire audio file. In

order to fix this, we implemented an audioDeviceWriter object, which writes audio data to the output on a frame by frame basis, which would be a big structural improvement for when we attempted to implement real-time cancellation. Another major usage of the AAH class was an internal circular buffer (Appendix I). The reason we built a circular buffer within AAH is also for the cancellation, so we could store delayed frames and have a place to extract them from. Our function takes a delay value as an input, and extracts a frame from the buffer delayed by that value. We also utilize index shifting in order to extract frames that are delayed by such an increment that they are beyond the indices of the buffer length. Other functionality exists within AAH, such as writing frames to buffers and functions for playing audio, but these are not shown for sake of consolidation and the fact that they are pretty straightforward.

Once AAH was designed and tested, it was folded into the real-time cancellation simulation. The program receives audio from the pre-recorded files on a frame by frame basis, then stores the played music into a buffer. The frames are then extracted and delayed by the predetermined sample value, adjusted by the optimal gain levels, and subtracted from the dirty piezo input signal (one important thing to note that will become relevant later is that the raw music signal was pre-convolved with the impulse response for the purposes of cancellation). Each canceled frame was then written to an output array and saved. Figure 5 shows the result of the real-time cancellation simulation using spectrograms of the recorded input, the estimated music signal, and the isolated speech.

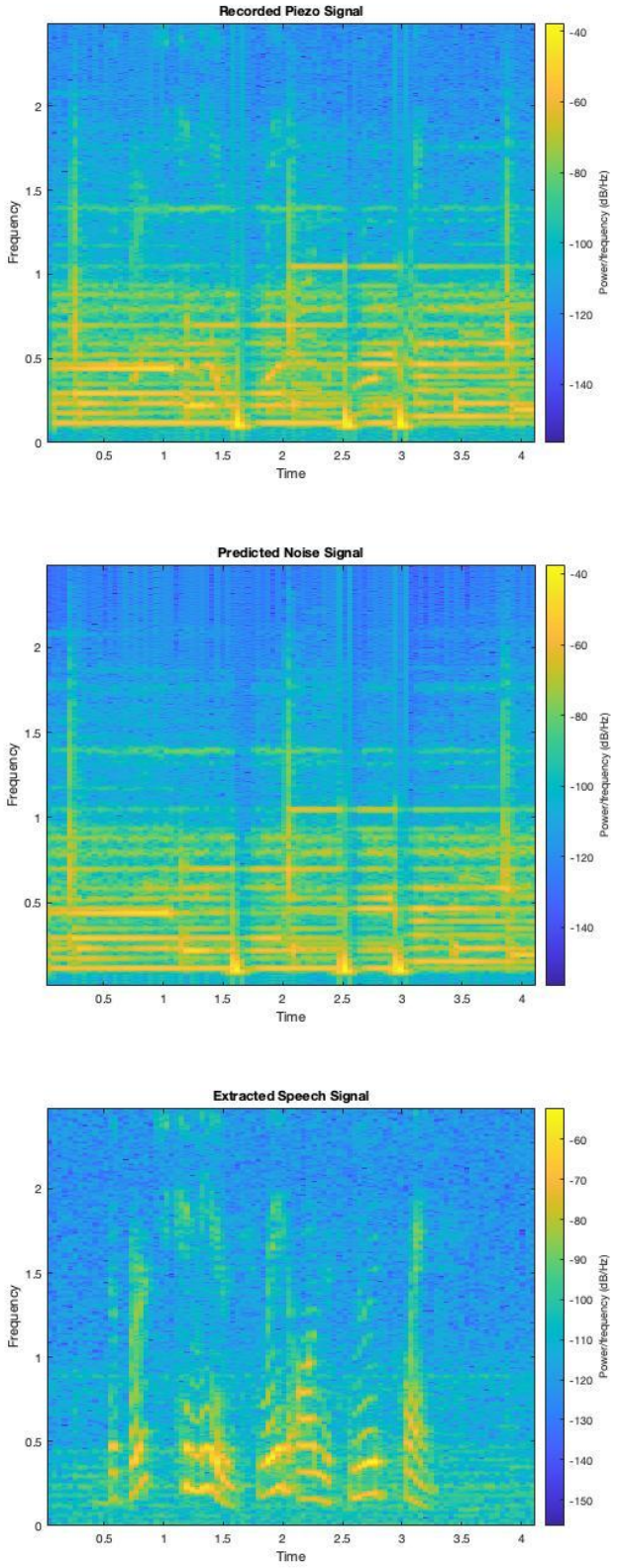

Figure 5: Spectrograms showing the result of isolation of the spoken phrases.

As the spectrograms demonstrate, the real-time simulation for subtraction of the music signal from the piezo input was highly effective. The extracted speech signal was extremely intelligible, and easily able to be interpreted by the IBM Watson API. This was a big breakthrough for our work, as it seemed the cancellation method we originally concocted held merit for our purposes after all.

The next and final step, as well as the most difficult one, was to implement real-time cancellation into Panel Assistant. This posed a lot of logistical challenges for our group – at this point in the project, we were only a few weeks out from the design day deadline. Due to the difficulty of this and the necessity of having a flat panel device fully built in order to test this, the real-time cancellation in Panel Assistant was not able to be tested at all until less than 48 hours away from the mock design day. The method was to be very similar to the real-time cancellation simulation, storing the played audio into a buffer and processing it in such a way that it can be subtracted from the piezo input on a frame by frame basis. Convolution between the played frame and the impulse response of the panel was also going to occur on a frame-by-frame basis, which eventually became a roadblock for our efforts. On the basis of an iteration of a single frame, we intended to do the aforementioned convolution, as well as crossover processing for the output channels and the subtraction method all in one go. This became very processing intensive for a single frame of size 1024 (samples), and we feared that this intensity would cause the program to either drop/distort samples of the input or output stream. We instead opted to pre-convolve the audio files with the impulse response of the panel, as well perform the crossover processing beforehand. This saved a lot of processing intensity off of the listening loop, and seemed like a much better approach for feasible real-time cancellation.

A major issue that arose in the 48 hours prior to the mock design day was a discrepancy in the fidelity of the impulse response measurement. When it seemed that the real-time cancellation was not working within Panel Assistant, the team opted to re-examine our methods to find malfunctions in our concept. After analyzing the estimated music signal overlaid with the actual piezo input, it was suspected that a faulty impulse response was a contributing factor to the lack of functionality. Figure 6 shows a graph of the current impulse response as well as a known clean impulse response.

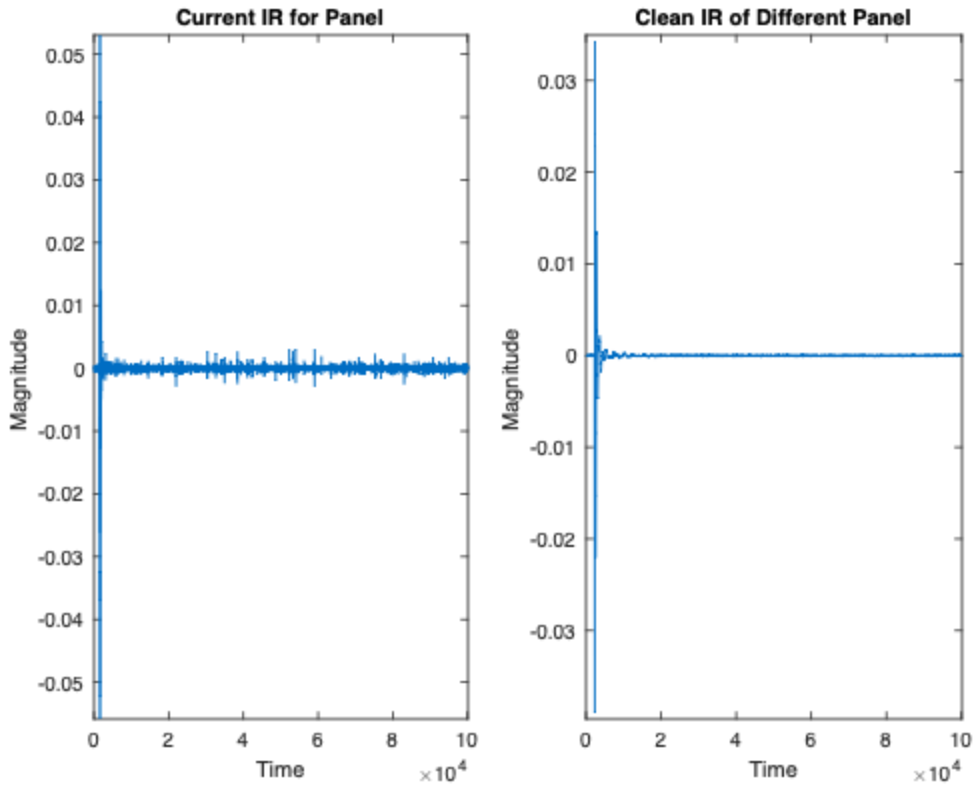

Figure 6: Our IR vs. a clean control IR

As the graph demonstrates, there is clearly visible noise in the impulse response measurement of our panel. We suspect this was due to hardware limitations, such as noisy interfaces, or some kind of other extraneous factor. Regardless, this was discovered a few hours before the mock design day, presenting a massive hurdle for the cancellation method. The pure subtraction method is very susceptible to misalignment and errors, so a noisy IR such as the aforementioned one is not suitable for real time cancellation. Thus, due to extreme time restrictions, we were unable to successfully implement real time cancellation into Panel Assistant. We instead opted for a ramshackle solution of playing audio using commands, but not receiving new commands while playing. The program instead has a button on the laptop UI that allows the user to end the song with any button press and revert to a normal listening loop. A future implementation of this model would probably be able to use real-time cancellation based on our methodology, if the impulse response is re-measured and an adequate time is available for troubleshooting. If the method were to be implemented successfully, it would signify a big leap

forward in the use of flat panel technology as an alternative to current smart assistants on the market.

B. Hardware

Because the flat panel used for the speaker was provided to us, this section will only discuss the hardware used and applied to the panel for signal processing, and not the flat panel itself. A discussion on the physical panel can be found in Section II-C. The hardware implementation for this panel was based on previous implementations and research into flat panel smart speakers performed in the vibroacoustics lab at the University of Rochester. There are six main components in the hardware signal flow: (1) Piezoelectric vibration sensor, (2) Piezo amplifier, (3) Power Amplifier, (4) Treble Driver, (5) Bass Array, and (6) Digital interface. Figure 7 shows the block diagram for signal flow and interaction and integration of the hardware with the software, and Figure 8 shows the final implementation of all of the hardware as it was installed on the back of the panel. The proceeding sections will discuss each component of the hardware in greater detail.

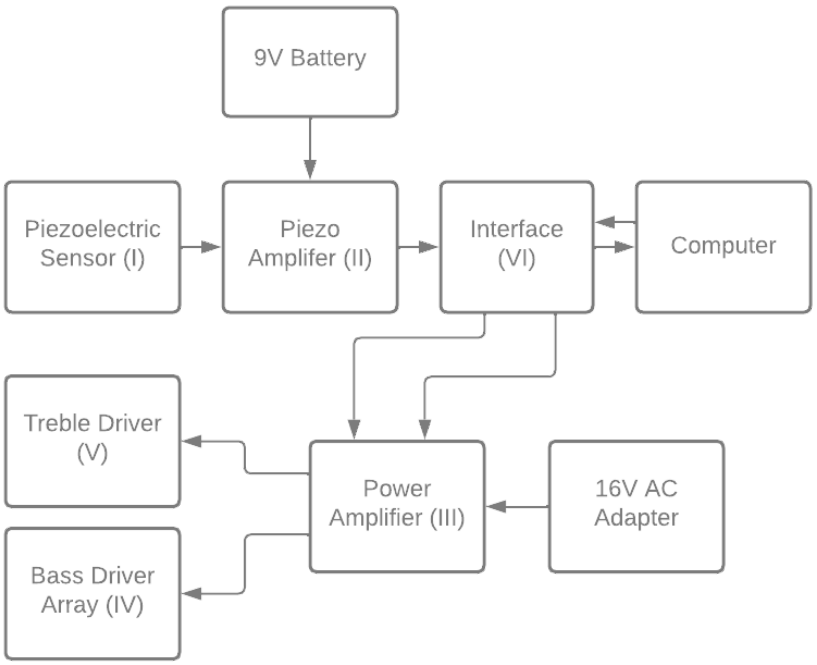

Figure 7: Block diagram of the hardware implementation and signal flow

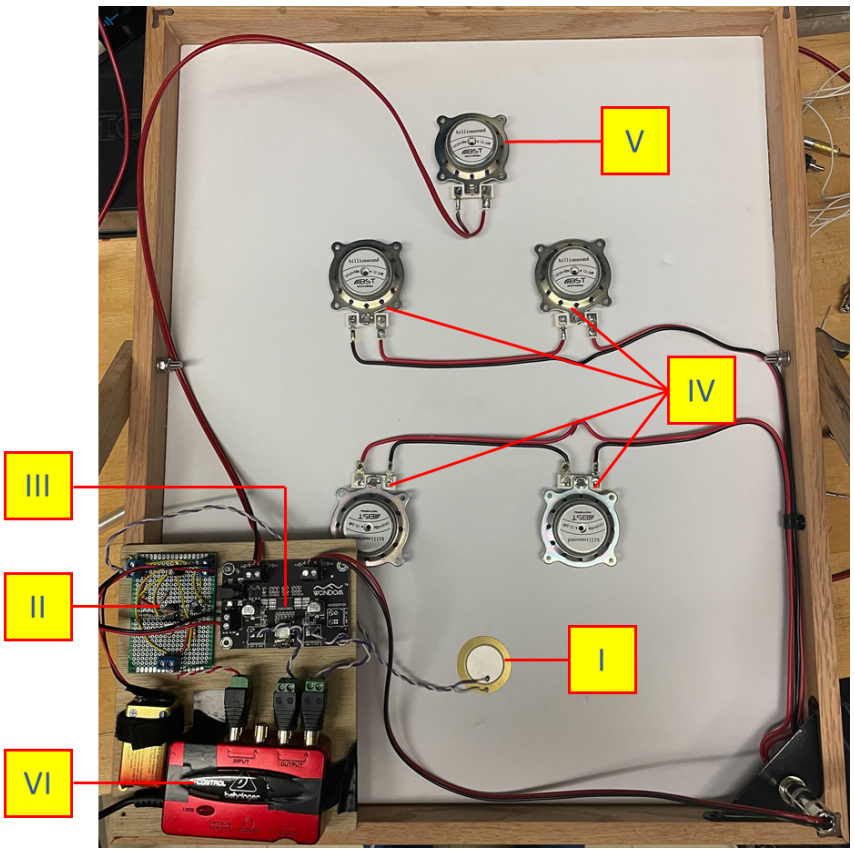

Figure 8: A picture of the hardware as implemented on the flat panel. Each component corresponds with the block diagrams shown in Figure 8. I) Piezoelectric sensor, II) Piezo amplifier, III) Power amplifier, IV) Bass Driver Array, V) Treble driver, VI) Interface

1. Piezoelectric Sensor (I)

The ceramic piezoelectric sensor attaches to the back of the panel and together they function as a microphone to detect voice commands. It does this by detecting the vibration of the panel as sound waves excite it. This is a key advantage of the flat panel technology, because it can act as both a microphone and a speaker. Initially, the sensor we used to detect speech was very small, only about 3 centimeters. The signal recorded from the small sensor was extremely noisy. The fidelity of the sensor is essential to implementing a system that can understand and execute voice commands. So, in order to increase signal-to-noise ratio (SNR), we experimented with increasing the size of the piezo sensor.

We tested three different sizes of piezo amplifiers, which can be seen in Figure 9. As in Table 1, preliminary tests showed that as size increased, preliminary tests showed that the SNR of the recordings also increased. Therefore, we chose to use the largest piezo sensor in our final

design. Recordings of each of the sensors can be found in Appendix J.

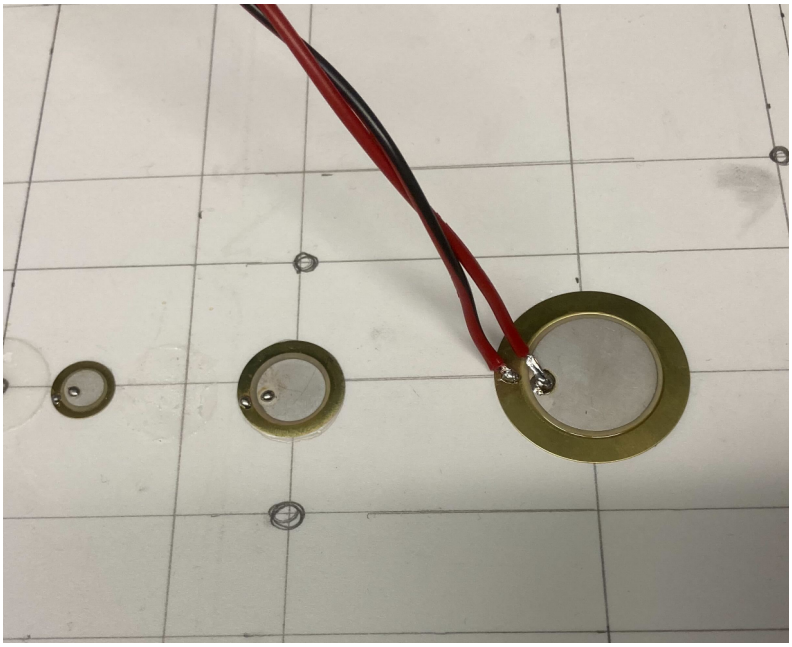

Figure 9: Three sizes of piezo sensors attached to a test panel. From left to right is the smallest piezo sensor, the middle size, and the largest piezo we tested.

Table 1: SNRs recorded from each sensor

Size (relative)	**SNR (dB)**
Small	0.49
Medium	6.03
Large	22.06

2. Piezo Amplifier (II)

Because the piezo sensor is ceramic, it has a relatively high impedance. So, it was necessary to attach the piezo sensor to an amplifier circuit to slightly increase the gain, match impedance to the interface, and filter the signal to eliminate unnecessary noise.

We first evaluated the circuit that was used in previous prototypes of the flat panel smart speaker (Figure 10). In testing this circuit, we found it was inconsistent and unreliable. It would stop working seemingly at random. Furthermore, it was complicated and required the use of a virtual ground in order to operate.

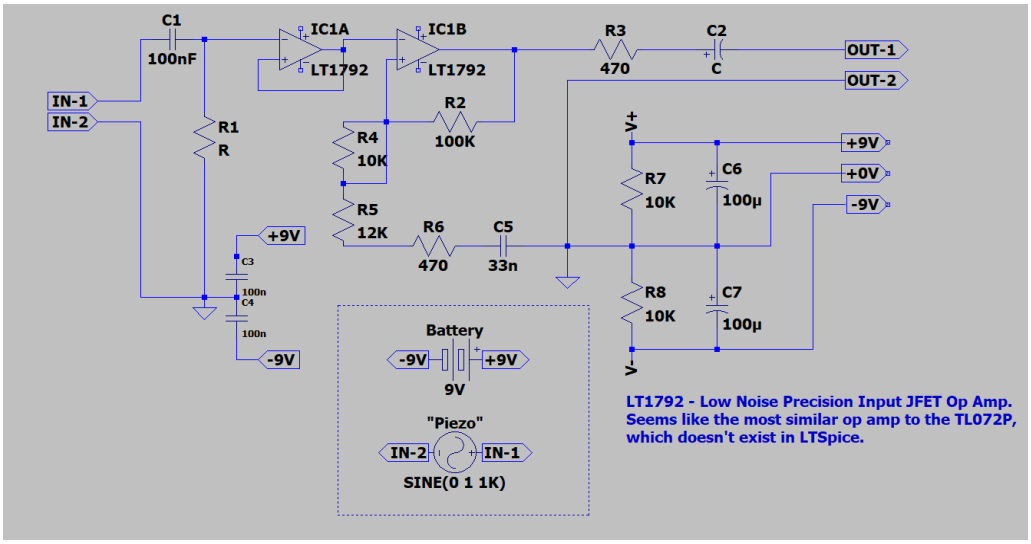

Figure 10: The piezo amplifier circuit implemented in previous prototypes.

We began looking for simpler implementations of a piezo amplifier. We decided to use the circuit proposed in [3], as it was simple, effective, and only required a 9V power source. We modified the circuit slightly. The final circuit can be seen in Figure 11.

When this circuit was first prototyped on a breadboard, it was extremely noisy. This was due to a few factors. Breadboards are inherently noisy, and the power rails were acting as antennas and picking up outside noise. Additionally, the wires that connected the piezo to the amplifier were very long and unshielded, which further increased noise. This was easily fixed by prototyping the circuit on a perfboard and reducing the length of the wires attached to the piezo as much as possible.

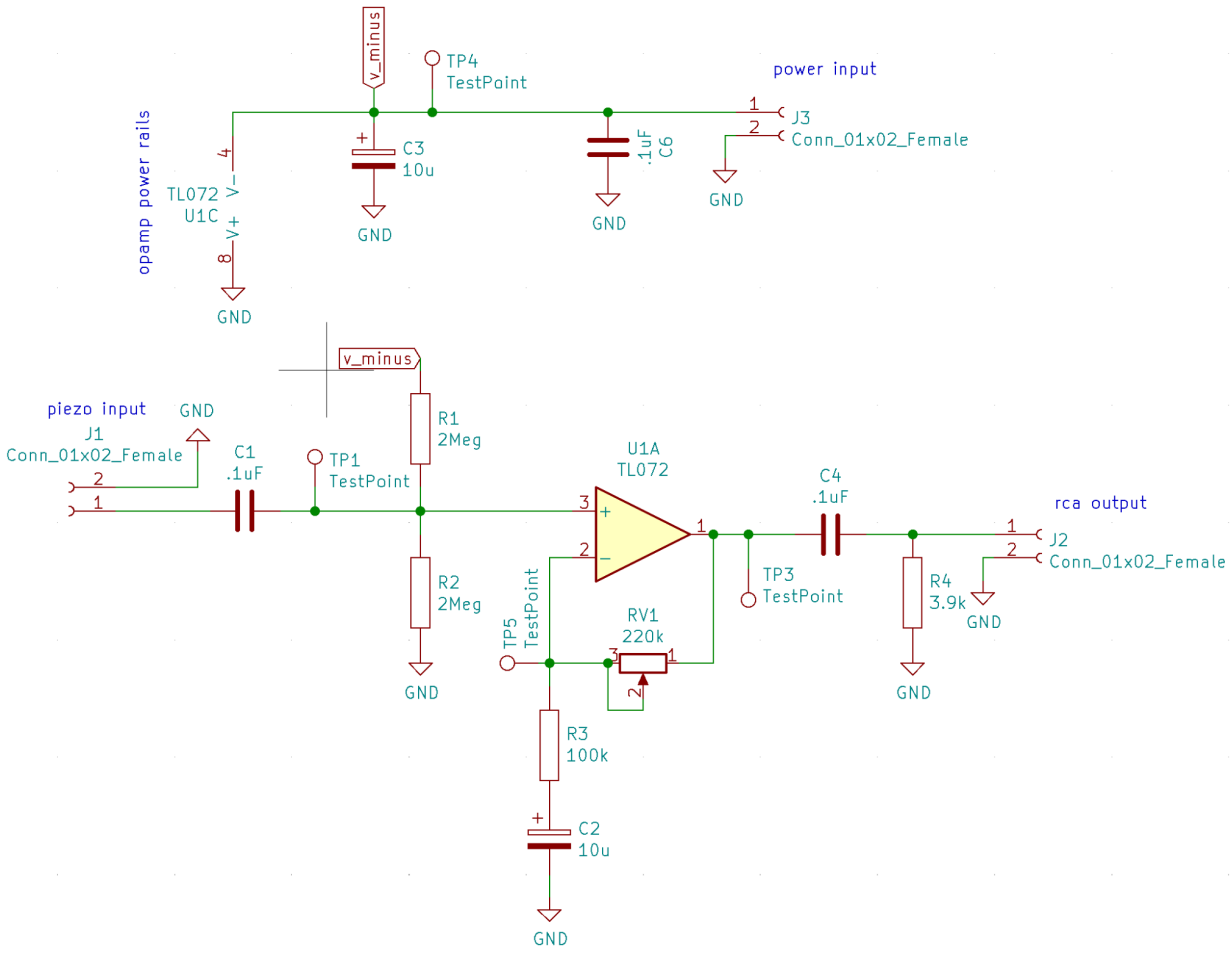

Figure 11: The final piezo amplifier design, modified from [3].

3. Power Amplifier (III)

Considering the impedance from the loudspeaker drivers, the output signal directly from the audio interface is not powerful enough to drive the speakers; thus, we need a power amplifier that converts a low-power signal to a higher power one. By boosting the output signal current to a nominal level, the speaker drivers can operate.

We first assumed that our power amp should generate 40W (general wattage for a loudspeaker), and we looked into two types of power amps for our selection: class AB power amp and class D power amp.

Class AB, as seen in Figure 12 is the most common type for audio power applications since it achieves almost no crossover distortion. If one seeks high fidelity (HiFi) quality for the loudspeaker, he(she) may go after a class AB power amp. Yet, there are the trade-offs for the amp's audio quality, the biggest of which is its low power efficiency. Class AB power amplifier has a power efficiency of 75%; that 25% power transferred to heat requires a huge heatsink to dissipate. Furthermore, one needs a larger power supply for a class AB power to make up for its low power efficiency. The bulkiness of the class AB power amp increases the cost and makes it more challenging to fit it onto the back frame of the panel.

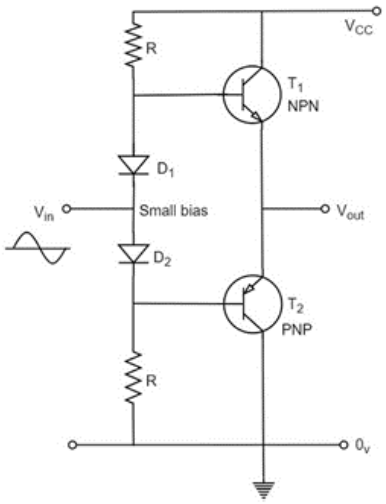

Figure 12: Circuit for a Class AB Power Amplifier

A class D power amp, as seen in Figure 13, is a more efficient in power compared to a class AB power amp. Class D power amps typically have a power efficiency of 90%, achieving outstanding heat control. Cost and space can be drastically saved by selecting a class D power amp. As long as an acceptable sound quality could be achieved, we would prioritize using a class D power amp.

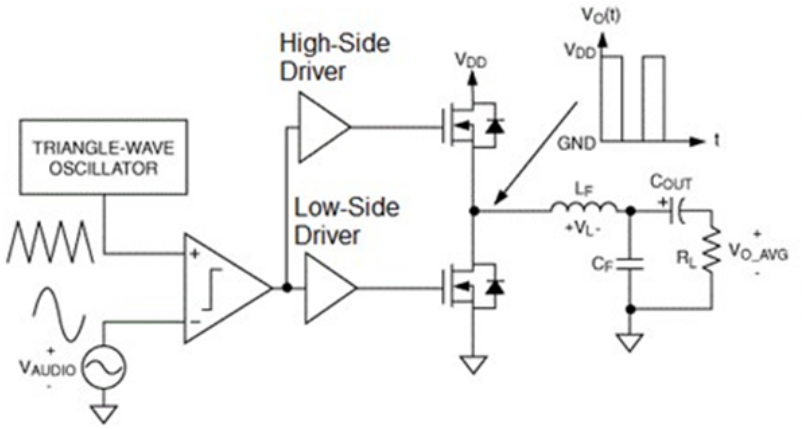

Figure 13: Circuit for a Class D Power Amplifier

We first looked into power amp boards available in the lab since it would be less time-consuming if we could find a suitable amp off the shelf than spinning a circuit board on our own.

We started with a Sure Electronics AA-AB32165 class D power amp [4].

Sure Electronics AA-AB32165 (Figure 14)

- Power: 2x25W at 6 Ohm
- Type: Class-D
- Gain: 21.6dB / 27.6dB / 31.1dB / 33.6dB depending on the switch
- Dimensions (mm): 110.2 L x 68.6 W x 16 H

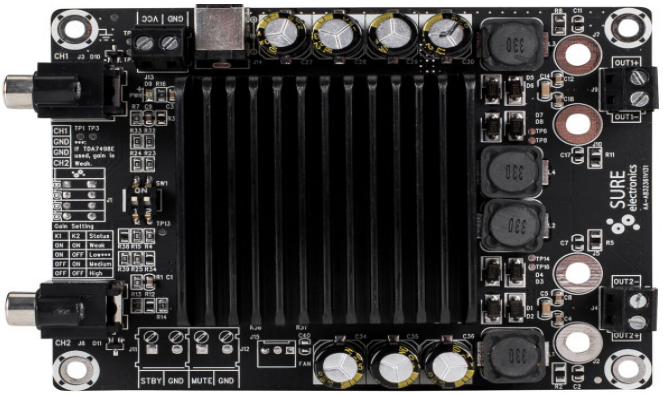

Figure 14: Sure Electronics AA-AB32165 Power Amp

Before applying it to our flat panel smart speaker, we tested the amp on the KEF LS50 loudspeaker. We connected the power amp board to a laptop using an RCA stereo to ⅛ inch cable and connected the loudspeaker to one of the two power amp outputs. We figured out that 25% volume on the laptop could drive the loudspeaker to a nominal volume if the power amp was set to a gain of 21.6dB (shown in Figure 15.) An LS50 loudspeaker's nominal impedance is 8 Ohms, which is a higher impedance compared to the flat panel smart speaker since we designed each channel (bass-mid frequency driver array and treble frequency driver) on the panel to have 4 Ohms impedance. If the power amp can drive an 8-Ohm-loudspeaker, it should be able to drive a 4-Ohm one as well.

Keeping that in mind, we then applied the power amp to a single-driver panel loudspeaker (4 Ohms in impedance) and tested the proper setup for it. The panel loudspeaker has an impedance of 4 Ohms and we used a DC power supply set to 16V-0.3A to drive the power amp up.

Parameter	Conditions	Min.	Typ.	Max.	Units
Gain	SW1 ON, SW2 ON	20.6	21.6	22.6	dB
	SW1 ON, SW2 OFF	26.6	27.6	28.6	dB
	SW1 OFF, SW2 ON	30.1	31.1	32.1	dB
	SW1 OFF, SW2 OFF	32.6	33.6	34.6	dB

Figure 15: Sure Electronics AA-AB32165 Power Amp Gain Options

We remained to use 21.6dB in power amp gain. We tested this setup using the Beatles' "Penny Lane" (video can be found in Appendix K.) This song had a loud piccolo solo in the last chorus section, where a lot of power would be drained from the power source to drive the speaker to operate. Throughout the test, we had guaranteed that under this setup, even 100% volume on the laptop would not burn the loudspeaker driver. Nevertheless, the size of this AA-AB32165 is too large to fit onto the back frame of the panel. Thus we decided to look for a power amp board smaller in size.

Sure Electronics AA-AB32261 [5]
- Power: 2x150mW at 8 Ohm
- Type: Class-AB

- Gain: 15.7 dB
- Dimensions: 3" L x 2" W x 5/8" H

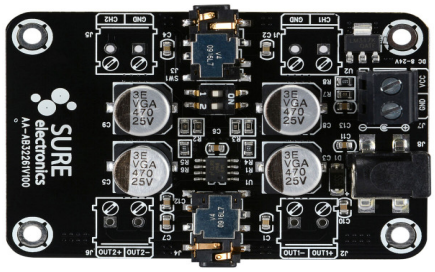

Figure 16: Sure Electronics AA-AB32261 Power Amp

We went through the same test using an AA-AB32261 power amp board (shown in Figure 16). This power amp is smaller in size but it's also low in power. We had figured out that power was enough for our drivers. Having the test data from the previous amp, we applied this power amp board to our flat panel smart speaker. Though low in power, this 150mW power amp could still drive the speaker drivers to a nominal level under the 16V-0.3A power supply. The small size of this power amp also saved up a lot of space when we mounted it to the back frame of the panel, which is why we finally decided to use AA-AB32261 for the power amplification (III).

4. Driver Positions (IV and V):

The vibration position can be critical for a membrane. A simple example would be a drum. When the drummer's drumstick hits the center of the drum surface, the drum generates a full and bassy resonant tone; whereas when the drumstick hits the edge of the drum, the drum generates a short and sharp high-pitched sound. The reason for it is that membranes have modes, and each mode possesses a unique resonant frequency based on the membrane's physical dimension. Our flat panel smart speaker acts as a rectangular membrane acoustically. It is fixed along all four edges since we had the rectangular Gatorfoam board (panel) embedded in a wooden frame. The mode shapes (standing wave vibration patterns) Ψ for a rectangular membrane of width Lx and length Ly can be calculated based on the following equation [6]:

$$\psi_{mn}(x,y) = sin\left(\frac{m\pi}{L_x}x\right) sin\left(\frac{n\pi}{L_y}y\right)$$

In this equation, (m, n) is called the mode shape identifier, which refers to the number of humps (antinodes) in the x and y directions, respectively. For example, Figure 17 shows how (1, 1) mode, (2, 2) mode, and (4, 4) mode look like on a rectangular membrane.

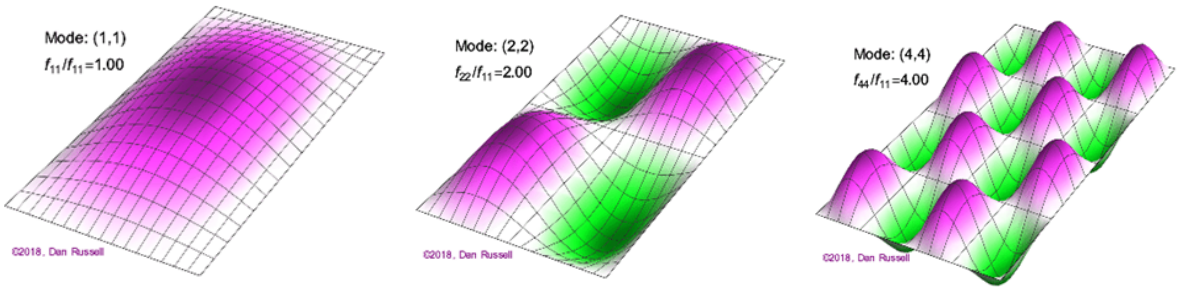

Figure 17: Modes on the Rectangular Membranes

The displacement of the membrane (width Lx and length Ly) from a specific mode can be derived by knowing the mode shape equation. Here, M and N are wave amplitudes and ω_{mn} is the wave's angular velocity:

$$z_{mn}(x,y,t) = sin\left(\frac{m\pi}{L_x}x\right) sin\left(\frac{n\pi}{L_y}y\right) [Msin(\omega_{mn}t) + Ncos(\omega_{mn}t)]$$

The overall response of the membrane at the resonant frequency of the degenerate modes is the superposition of these modes [7]:

$$z(x,y,t) = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} z_{mn}(x,y,t))$$

Building upon the mode shape equation, the resonant frequency for the (m, n) mode is:

$$f_{mn} = \frac{1}{2}\sqrt{\frac{T}{\sigma}}\left[\left(\frac{m}{L_x}\right)^2 + \left(\frac{n}{L_x}\right)^2\right]^{\frac{1}{2}}$$

where T in the function stands for the tension on the membrane, while σ stands for the membrane mass per unit membrane area. One can see from the equation that the (1, 1) mode always has the lowest resonant frequency, and its resonant frequency f_{11} is considered the fundamental frequency. Modes with large m and n are called high-order modes, which have larger resonant frequencies.

Modes and their resonance frequencies are important references as researchers at the University of Rochester (U of R) design driver positions. One can extrapolate, from the graph on modes and their antinode (hump) positions, that most low-order modes can be excited in the center of the membrane, and high-order modes, instead, can be better excited at the edge of the membrane. This pattern for mode distribution is the key to understanding how we arranged positions for inertia exciters (speaker drivers) at the back of the flat panel smart speaker.

We had four inertia exciters (IV) at the center of the panel. The center of the panel is assigned for only the bass and the mid-frequency portion of the playback signal. Lower order modes for the membrane are best excited in this area on the panel, and due to these modes' resonant frequencies, we can achieve an excellent response in playing back bass and mid-frequencies. For similar reasons, we attached another inertia exciter right above the bass and mid-frequency sector (V) that takes charge of playing back the treble(high)-frequency portion of the signal.

5. Interface (VI)

An audio interface is a hub for PC and audio electronics. In our project, the audio interface connects the hardware and software components of our flat panel smart speaker. We chose a Behringer U-Control UCA222 as our interface (VI), which is a USB-powered interface with two RCA inputs and two RCA outputs, seen in Figure 18. We powered the interface up using the laptop USB port, connected the piezo amplifier to one of the input RCA, linked the left

channel RCA output to the bass-mid frequency driver array, and linked the right channel RCA output to the treble frequency driver. With our setup, once the interface receives an amplified speech recording from the piezo amplifier, its built-in Analog to Digital Converter (ADC) can digitize the speech recording and communicate (via USB) with the virtual assistant software (panelAssistant installed on the laptop) for voice transcription. The virtual assistant then tells which song our user wants to play, delivers the music data stream to the crossover network to separate the data stream into two frequency sections, and feeds the processed data stream into the Digital to Analog Converter (DAC) in the interface. Eventually, the DAC converts binary data into acoustic waves and assigns corresponding driver channels for playing back music.

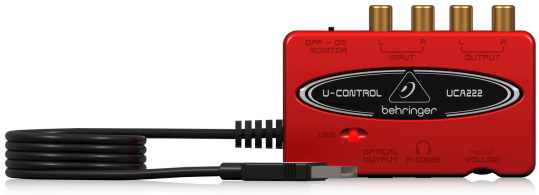

Figure 18: Behringer UCA222 Audio Interface

Aside from the Behringer UCA222, we considered other options for the audio interface, such as the Behringer UM2, seen in Figure 19.

Figure 19: Behringer UM2 Interface

The UM2 is also a USB audio interface, however, it is excessively bulky. UCA222, on the other hand, is of a similar size to a smartphone. One of our goals was to shrink the size of the hardware, and thus we took advantage of UCA222's tiny size and easily mounted the interface onto the back frame of the panel.

A sound adapter (Figure 20) was another option considered. The sound adapter is simply a USB adapter with a stereo output jack and a mono microphone input jack. The adapter is way smaller in size compared to an audio interface, yet it introduces a lot of noise. In contrast to the noisy sound adapter, a UCA222 supports 16-bit/48kHz audio that ensures high-quality and low-latency playback out of the computer.

Figure 20: Sound Adapter

Ultimately, the UCA222 was our final choice for the audio interface because it serves as a good balance between audio quality and size.

6. Power supply

To ensure the portability of our flat panel smart speaker, we used a 9V battery for the piezo amplifier and used a 16V wall mount for the power amp. We also thought about consolidating the two power sources. We could have a DC/DC converter after the wall mount and design buffers between the power amp and the piezo amp so that the two amps could be assigned with the appropriate power levels with one shared power supply. A block diagram of this design can be seen in Figure 21.

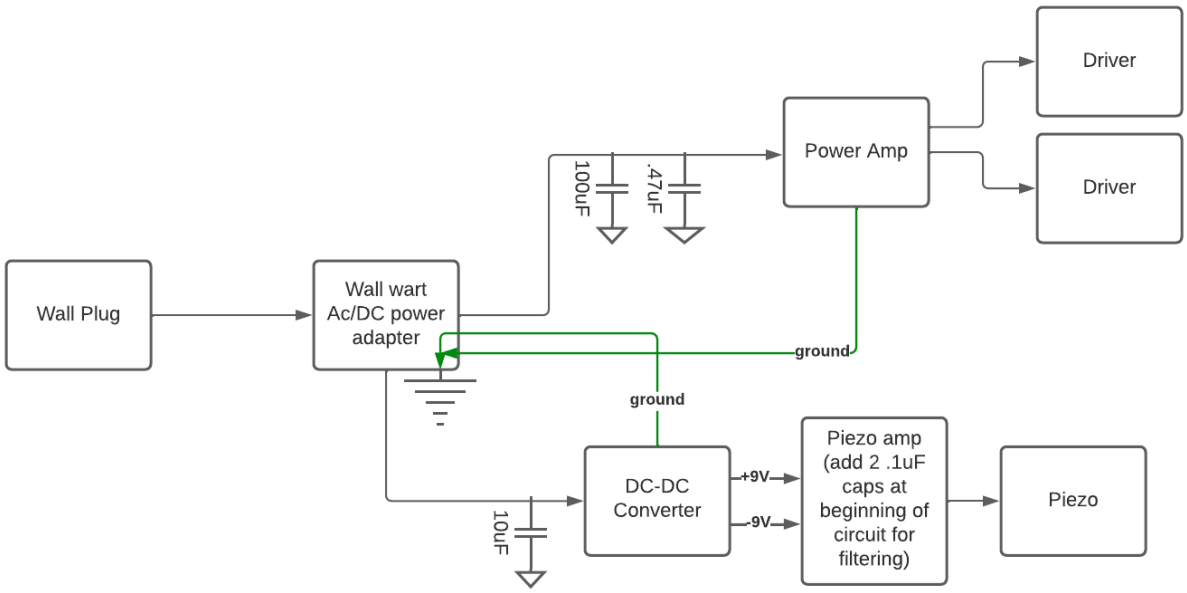

Figure 21: Hardware Block Diagrams with a Shared Power Supply

C. Panel Construction

1. Size and Material

Since the implementation of flat panel technology is not a well-explored field, there are a lot of questions to be asked pertaining to the best way to build a flat panel speaker. In particular, the material used for the panel itself is of utmost importance. Research out of the University of Rochester has yielded quantitative results on the performance of various materials as panels, including glass, acrylic, and gatorboard. Certain materials tend to perform better in the speaker capacity (producing cleaner and louder sound using drivers) and some materials perform better in the microphone capacity (receiving intelligible speech via the piezo sensor). We ended up using gatorboard in our final build, as it was able to perform sufficiently as both a speaker and microphone, and a smart speaker needs to be effective in both of these capacities. Another technique used in the construction of the panel is *constrained layer damping*, a method of partially damping the vibrations of acoustic surfaces by conjoining two or more layers of them together with some sort of binding agent [8]. Through this, we are able to create a panel surface that performs better as a speaker, and improve the performance of certain materials such as the gatorboard we used.

Another consideration in the construction of our panel was size, which is also nuanced in its application. It is important to note that our team did not actually construct our final panel, which was built by the U of R's Mark Bocko, a prolific figure in the research of flat panel technology and one of the only people with the means to build fully-fledged flat panel devices. We ended up using a larger size panel, since it would be louder and also more eye-catching. A big part of this project was designing something that may have commercial applications, so the look of the panel was definitely an influencing factor. The frame of the panel is built from a lightweight wood, which doesn't factor into the performance as much since the panel is only vibrating on the portion that isn't directly affixed to the panel. All in all, the panel design is pretty well optimized for performance, and implements the results of recent research in the field. Additional improvements to the performance could be implemented in the panel construction via crossbars that sit behind the driver placements, which gives them something to vibrate against and has been shown to improve the sound and volume of the panel as a speaker. However, this was not incorporated into our final design.

2. Mounting of hardware on panel

At this point, all the components are working properly, so the next step was to mount all the hardware components onto the frame of our panel. The diagram in Figure 22 shows the design of our flat-panel.

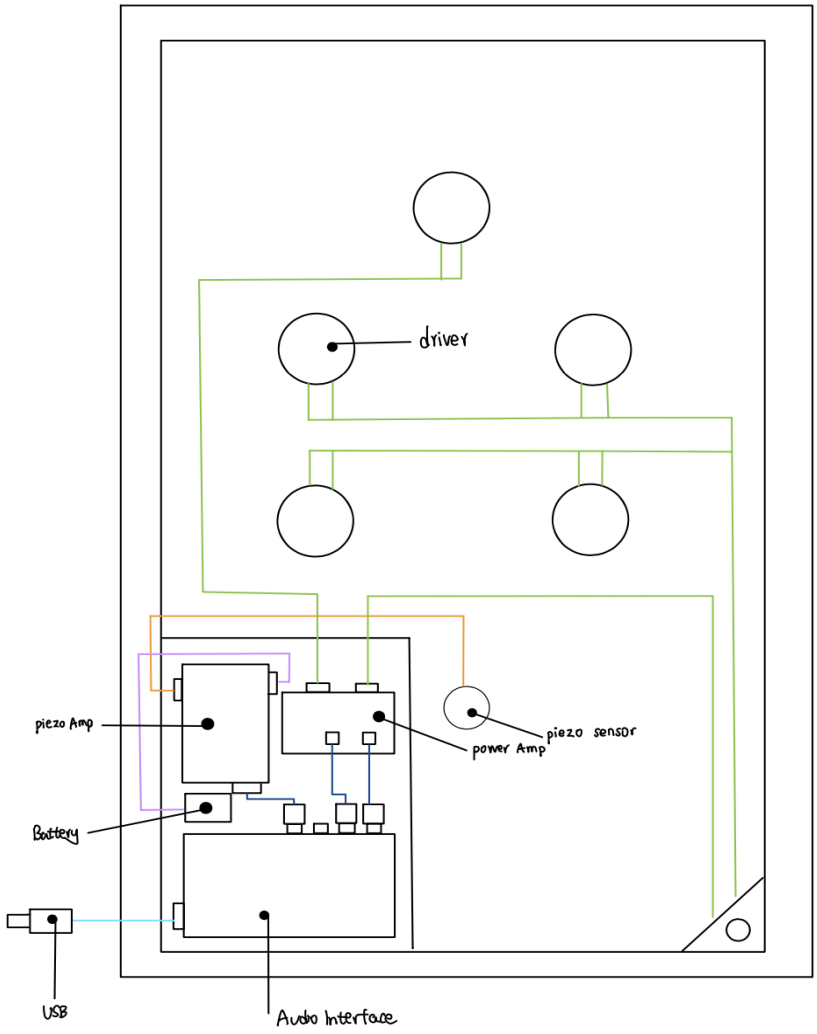

Figure 22: Diagram of flat panel smartspeaker

The only components that have physical contact with the panel are the array of drivers and the piezo sensor. The rest of the components are expected to be mounted onto the frame. In order to achieve this, we used a wood board that has sufficient space for a piezo amplifier, a power amplifier, an audio interface, and a battery. With the help from professor Paul Osborne, we successfully have the circuit boards mounted on the wood and audio interface attached onto the board. One aspect that we did not take into consideration is the placement of the components. From the diagram. We can see that we placed the piezo amplifier on the top left corner of the wood board, which consequently led to a relatively long wire length from piezo

sensor to piezo signal amplifier. On the diagram, the wire is shown in orange. From our previous research, the wire length between the two components should be as short as it can be since wire length plays one of the most important roles in introducing noise to the signal captured by a piezo sensor. We noticed this problem after the piezo amplifier was mounted on the board. Before taking down the components, we first tried to solder a longer wire and listen to the signal. Fortunately, the signal after the piezo amplifier recorded by audacity sounds relatively clean. There is a minimum difference before and after the replacement of wire. The following is our assumption: Since the components are mounted, it stabilizes the joint between wire and components. Before mounting the components, despite the fact that all parts work, they hang freely for most of the time. This instability may to some extent create a negative impact on the piezo signal. We placed the audio interface below the piezo amplifier and power amplifier above the audio interface because it matches the layout of how the output and input ports are located on these components. This concludes the mounting of hardware on the panel speaker.

III. Standards

Since our project is a smart speaker, the concern of privacy violation should be taken into careful consideration. Unlike Google Nest or Amazon Alexa whose smart speaker is usually a device that is placed and displayed on a table. Flat-panel smart speakers could potentially be designed as painting on the wall, mirror, or TV screen. Due to the fact that even smart speakers like Amazon Alexa are involved in dozens of charges regarding privacy, it is within our expectation that data protection and regulation law could be used against this product if the developers fail to protect users' personal data.

In 2018, the European Union passed a new law regarding data privacy, called the General Data Protection Regulation, it is regarded as "the toughest privacy and security law in the world." The protection regulation imposes obligations on organizations around the world as long as the devices collect or target data from users that are related to the European Union [9]. The regulation defines legal terms including Personal Data, Data Processing, Data subject, and etc. Each relates closely to our project. For instance, Personal Data is "any information that relates to an individual who can be directly or indirectly identified." In our case if our products are on the market for sale and customers buy them online, it is crucial to protect their credit card

information and personal information including names and email addresses. Data Processing means “any action performed on data, whether automated or manual. The examples cited in the text include collecting, recording, organizing, structuring, storing, using, erasing.” This term and regulation protecting this term requires more openness from tech companies about what data they have and how they share it. In our case, we need to explain to the customers that our panel assistant is constantly listening to the environment sound and once a keyword is detected, the panel starts to record automatically and transcribe the recording into words before sending it to the voice recognition API. The approach that Google takes on following privacy regulations is they notify users about exactly what information Google needs and what control users have on their personal data. Google built a Safety Center website that describes how they made commitments to privacy and security of users. They also came up with privacy and security principles that explain to the users how they respect users’ privacy. This principle includes “Never sell our users’ personal information to anyone”, “Make it easy for people to control their privacy”, “Empower people to review, move, or delete their data”, and etc [10]. We believe that the idea of protecting user’s privacy is relatively straightforward, and how we would like to approach this goal is clear due to the excessive experience that Google and Amazon has. But each step towards the goal should be meticulous and minimum mistakes should be made due to the importance of protecting users’ privacy.

IV. Conclusions

A. Hardware Conclusions

1. Results

We met our expectations at the beginning of this flat panel smart speaker project by simplifying the hardware design and shrinking the size of the hardware. For the previous hardware implementation of a flat panel smart speaker, we had three bulky power supply adapters plus one power strip as the power source for the panel loudspeaker, which was extremely space-consuming. With the reconfiguration of the hardware, we can power the panel loudspeaker up with only three connections: one USB connection for the audio interface, one 9V battery for the piezo amp, and one wall mount for the power amp. This redesign in the hardware also saves up a lot of space for the enclosure. In the previous implementation, a gigantic metal box was required for the enclosure. After the reconfiguration, we can fit all our hardware

electronics to the back frame of the panel, which drastically improves the portability of the flat panel smart speaker.

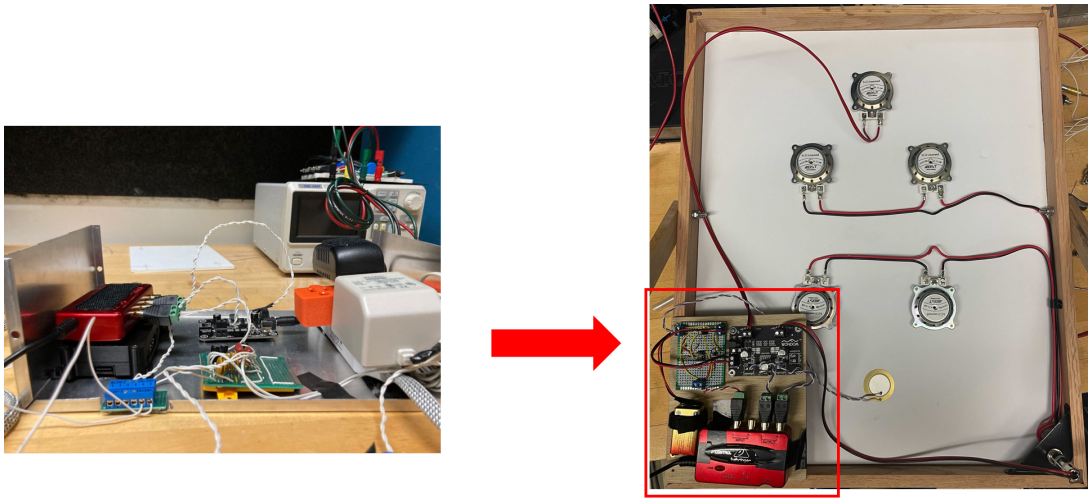

Figure 23: Better Configured Hardware

Aside from improving the overall hardware design for the project, we spent a great deal of time achieving high-quality piezo recording at the least possible cost. The previous design used PCB's Accelerometer Vibration Sensor. It was of great recording quality but it was extremely expensive. We changed that into a piezo disc that costs only one dollar, and redesigned the piezo amp to make it smaller in size and less noisy. We have redesigned the piezo and its amp so that even a cheap piezo can deliver a clear enough speech for voice transcription.

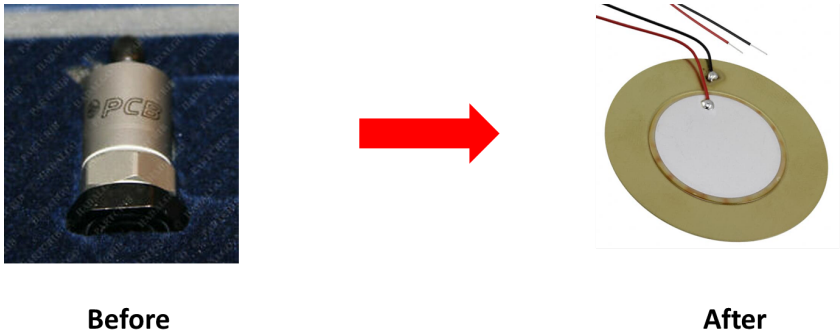

Figure 24: Cheaper Piezo

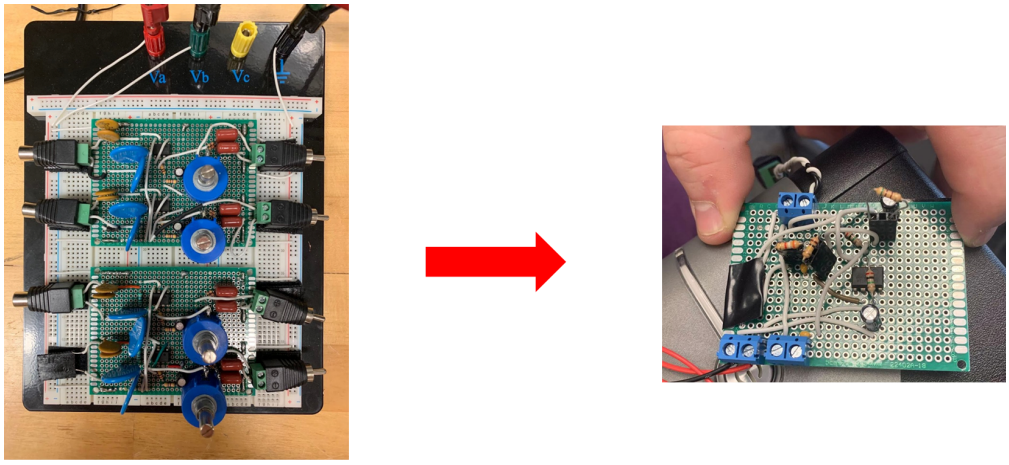

Figure 25: Smaller and Less Noisy Piezo Amp

However, looking back to the goals we set at the beginning of this project, there are two goals in the hardware section that we haven't fulfilled: we didn't spin the circuit board for the piezo and we failed to include the Raspberry Pi in our hardware configuration.

2. Shortcomings

Just as mentioned previously, there were some shortcomings of our implementation.. The first thing being we should have finished our PCB design for the piezo amplifier. Our current design for the piezo amp is built on a perforation board, which is fragile. The wiring on the perfboard can easily snap if the flat panel smart speaker falls. Substituting the perfboard with a PCB can substantially improve our hardware stability and functionality. We also could have designed a shared power supply for both the piezo amp and the power amp. Since the battery drains out quickly, taking the 9V battery out of our hardware design would further simplify our circuit configuration. One USB cord and one wall mount should be enough to power the flat panel smart speaker up. We also should have deployed software onto a small Raspberry Pi and substituted our PC with the Pi. If that was achieved, the hardware connection would be simplified further: even the USB cord would be unnecessary. If the Raspberry Pi had all those voice transcription and music cancellation programs installed, we would have turned our flat panel smart speaker into a fully stand-alone speaker. Taking the PC out of our current design would be a huge leap in improving the portability of the panel. Lastly, we should have designed a better enclosure for our flat panel smart speaker. Though convenient, it is unsafe to have exposed

electronics. A case on the back of the panel will be a better and more stable enclosure compared to mounting the electronics to the frame.

B. Software

In terms of shortcomings for software, although our Flat-panel smart speaker is able to perform many of the tasks that an ordinary smart speaker on the market right now is able to perform, there are a few places that need to be improved. First of all, the panel assistant struggles to detect voice commands when there is background noise. It is certain that this shortcoming is partially due to the property of piezo sensors. Piezo sensors exploit piezoelectricity, charge created across certain materials when a mechanical stress is applied, by measuring the voltage across a piezoelectric element generated by the applied pressure. It has a relatively low SNR, the ratio of signal power to the noise power, so it is difficult for the panel assistant to recognize the key words based on the transcription of received signal. Software might be able to help resolve this issue by constantly monitoring and adjusting the noise threshold of surrounding noise. Another solution is to train the assistant with the user's own voice so it is easilier for the assistant to recognize his or her command. In the case of Amazon Alexa, users are able to train Alexa to better recognize them by creating voice profiles. All it takes is for the user to say a couple sentences that Alexa requests. Although these methods will not solve the problem of not being able to receive commands properly in a noisy environment, they for sure will help with the problem.

Another shortcoming on the software side is about the real-time cancellation algorithm. During the daily use of a smart speaker, it is almost certain that a user will come across a situation of giving commands while the smart speaker is playing music. For a Flat-panel smart speaker, the piezo sensor and drivers are on the same panel, which means the piezo sensor will pick up all the vibration that is present on the panel including the music played. Since we know exactly what music will be played, and the distance between drivers and piezo sensors are constant, we are given all the requirements to estimate what the music will sound like when played by the drivers then received by the piezo sensor. Also, computing time will create a number of sample delays that can be calculated precisely. With all these conditions, we can take music out of the equation and leave only the user's command instead. In our ideal algorithm, the convolution of impulse response of our system and the music played should happen in real time,

that being said we do not need to do a convolution for each song before calling the real-time cancellation method. However, we found that doing convolution in real-time requires a long time to calculate so it creates click sounds. As a solution, we calculated the convolution for each song that we put in our library. Which is neither realistic nor reasonable. The reason is simply: there are millions of songs on the internet. It takes dozens of seconds to convolve one song with impulse response and the time spent convolving will be the time a user needs to wait for a response. Also, if a user requests one song and immediately asks for another, it will take even longer for the assistant to process. What's more, it is not wise to store hundreds of convolution audio files on the device.

Another shortcoming is the loss of volume control on the panel assistant. In September 2021, Amazon Alexa offered users an option to enable 'Adaptive Volume', which increases speaking volume of the assistant when loud background noise is detected. This functionality makes the smart speaker much more user-friendly since users no longer need to adjust the volume manually. In our case, adding functionality for users to be able to increase or decrease volume by giving commands is a doable and helpful path to improve the product.

Nowaday, almost every smart speaker on the market supports bluetooth or wifi connection, allowing users to be able to control it from a distance. In the concept of smart home, a flat panel smart speaker has the potential to become the core processing device that gives commands to the devices that are already interconnected in a smart home. That being said, Wifi and bluetooth connection are essential functionalities to add. Right now, we have success in programming remotely on Raspberry Pi, meaning we are able to write code on it without any external monitor or keyboard directly connected to Raspberry Pi.

C. Acknowledgements

We would like to thank Dr. Michael Heilemann and Tre DiPassio for their extensive help and guidance throughout this project. We couldn't have done it without them. We feel very fortunate to be able to contribute to the research being conducted in the Vibroacoustics laboratory, and thank them for the opportunity. We would also like to thank Professor Smith and Professor Phinney for providing valuable feedback during weekly project meetings.

V. User Manual

1. Download Matlab:

   Download Matlab. Start the download process in laptop at https://www.mathworks.com/products/matlab.html.

2. Plug in your panel:

   Plug the included USB connector into your laptop. A red light on the interface (The red device on the back of the panel) should be constantly on.

3. Put in battery:

   Place a 9V battery in the battery holder on the back of the panel and connect it with the piezo amplifier.

4. Run the code:

   Copy and paste the Matlab code in the Appendix section of this report into Matlab and run the file. When "Voice detected, Listening" appears on the command window, the assistant is ready to go.

Things to try:

Alexa, what time is it	Alexa, play poker face
Alexa, what is the weather today	Alexa, stop playing music
Alexa, what is the news today	Alexa, play kanye west
Alexa, what is the best football team	Alexa, play diamonds from sierra leone

VI. References

[1] "HomePod mini - technical specifications," *Apple*. [Online]. Available: https://www.apple.com/homepod-mini/specs/. [Accessed: 08-May-2022].

[2] "What Is A Speaker Crossover Network? (Active & Passive)" [Online]. Available: https://mynewmicrophone.com/what-is-a-speaker-crossover-network-active-passive/ [Accessed: 08-May-2022]

[3] A. Garaipoom, "Acoustic Guitar Pickup Circuit & Wireless using TL071," *ElecCircuit*, 21-Jul-2020. [Online]. Available: https://www.eleccircuit.com/acoustic-guitar-pickup-circuit-using-tl071/. [Accessed: 08-May-2022].

[4] Sure Electronics AA-AB32165 2x25W at 6 Ohm TDA7492 Class-D Audio Amplifier Board User Manual, Sure Electronics, Nanjing, Jiangsu Province, China. Accessed: May. 8, 2022. [Online]. Available: https://www.parts-express.com/pedocs/manuals/320-332--sure-aa-ab32165-2x25W-owners-manual.pdf

[5] *Parts Express*. [Online]. Available: https://www.parts-express.com/Sure-AA-AB32261-Stereo-2x150mW-Class-AB-LM4881-Headphone-Amplifier-Board-320-321. [Accessed: 08-May-2022].

[6] D. A. Russell, "Acoustics and Vibration Animations," Rectangular Membranes. [Online]. Available: https://www.acs.psu.edu/drussell/Demos/rect-membrane/rect-mem.html. [Accessed: 08-May-2022].

[7] T. D. Rossing and N. H. Fletcher, "Two-Dimensional Systems: Membranes and Plates," in *Principles of vibration and sound, 2nd Edition*, New York: Springer, 2004.

[8] B. M. Shafer, "An overview of constrained-layer damping theory and Application - Scitation." [Online]. Available: https://asa.scitation.org/doi/abs/10.1121/1.4800606. [Accessed: 08-May-2022].

[9] "What is GDPR, the EU's new data protection law?" [Online]. Available: https://gdpr.eu/what-is-gdpr/ [Accessed: 08-May-2022].

[10] "Our privacy and security principles." [Online]. Available: https://safety.google/principles/ [Accessed: 08-May-2022]

VII. Appendix

A.

```
% Initialization of IBM Watson connection for text-to-speech
self.authenticator_TTS = py.ibm_cloud_sdk_core.authenticators.IAMAuthenticator(self.APIKEY_TTS);
self.TTS = py.ibm_watson.TextToSpeechV1(self.authenticator_TTS);
self.TTS.set_service_url(self.URL_TTS);

% Initialization of IBM Watson connection for speech-to-text
self.authenticator_STT = py.ibm_cloud_sdk_core.authenticators.IAMAuthenticator(self.APIKEY_STT);
self.STT = py.ibm_watson.SpeechToTextV1(self.authenticator_STT);
self.STT.set_service_url(self.URL_STT);

% Helper function for text-to-speech transcription
function [y,fs] = IBM_TTS(self, input_string)
        % Open blank audio file to write response to
        audio_file = py.open('output.wav' ,'wb');
        % Generate response from IBM Watson connection
        response = self.TTS.synthesize(input_string, pyargs('accept', 'audio/wav', 'voice', "en-US_LisaV2Voice"));
        % Write generated response to output file
        audio_file.write(response.result.content);
        [y,fs] = audioread('output.wav');
end

% Helper function for speech-to-text transcription
function [output_transcript] = IBM_STT(self,commandAudioArray,fs)

        % Write recorded response to .wav file
        audiowrite('curr_command.wav',commandAudioArray,fs)
        % Open generated .wav file for IBM Watson transcription
        audio_file = py.open('curr_command.wav', "rb");
        % Generate transcription from IBM Watson connection
        result = self.STT.recognize(audio_file, pyargs('content_type', 'audio/wav', 'word_confidence', 'True',
'smart_formatting', 'True'));
        output_transcript = string(result.result{'results'}{1}{'alternatives'}{1}{'transcript'});

end
```

B.

```
% Helper function for comparison of input command string to dictionary of pre-generated commands
function [mostSimilarCommandNumber, mostSimilarCommand, maxSimVal] =
getMostSimilarCommand(self,input_string)

        % Force all uppercase letters in input string to lowercase:
        input_string = regexprep(lower(input_string),"","");
        % Create a tokenized document from the input string:
        input_tokens = tokenizedDocument(input_string);
        % Calculate similarity scores between input tokens and pre-compiled dictionary command tokens:
        similarityMatrix = bm25Similarity(self.dictionaryTokens,input_tokens);
        [commNums,~,simVals] = find(similarityMatrix);
        [maxSimVal,I] = max(simVals);

        % Sort previous calculations into list of most similar commands, with respective confidence scores:
        [self.currConfidenceRatings, self.currConfidenceOrder] = sortrows(simVals);
```

```
        self.currConfidenceRatings = flipud(self.currConfidenceRatings);
        self.currConfidenceOrder = flipud(self.currConfidenceOrder);

        self.currCommNums = commNums;
        mostSimilarCommandNumber = commNums(I);
        mostSimilarCommand = self.dictionary(mostSimilarCommandNumber);
end
```

C.

```
% Pre-processing command for parsing of exit phrase, wake word filtering, etc.
function [output_string] = preProcessInputString(self,input_string)

    % Force lower case for input string
    input_string = lower(input_string);

    % Look for the exit phrase (used for exiting the program with a voice command):
    split_string = split(input_string);
    num_exit_words = 0;
    if ismember("exit", split_string)
       num_exit_words = num_exit_words + 1;
    end
    if ismember("stage", split_string)
       num_exit_words = num_exit_words + 1;
    end
    if ismember("left", split_string)
       num_exit_words = num_exit_words + 1;
    end

    % If all 3 words of the exit phrase were recognized, kill the program
    if num_exit_words == 3
       self.break_state = 1;
       input_string = 'NONE';
       output_string = 'exit stage left';
       return
    end

    % Now filter out the wake word
    split_string = split(input_string);
    self.debugString = split(input_string);
    if ismember(self.wakeWord,split_string) || self.wakeWordActive || ismember(self.wakeWord2,split_string) ||
ismember(self.wakeWord3,split_string)
        self.wakeWordActive = 0;
        % Valid wake word was recognized, now do processing
        for idx = 1:length(split_string)
           elem = split_string(idx);
           if strcmp(elem,self.wakeWord) || strcmp(elem,self.wakeWord2) || strcmp(elem,self.wakeWord3)
              % Chop off excess so only things past the wake word are included
              if (idx == length(split_string)) || (idx == (length(split_string) - 1) )
                 input_string = "NONE";
              else
                 input_string = split_string(idx+1:end);
              end

              if strcmp(input_string,"NONE")
```

```
                % Condition for when only the wake word was heard, no subsequent command
                self.wakeWordActive = 1;
                 % Read out automated response
                [y,fs] = audioread('WHAT_CAN_I_DO.wav');
                sound(y,fs)
                pause(length(y)/fs + 0.5);
                input_string = 'NONE';
                output_string = 'NONE';
            end
            break
        end
    end
% Condition for no wake word recognized
else
    input_string = 'NONE';
    output_string = 'NONE';
end



% Checking for numbers in the input string
if ~strcmp(input_string,'NONE')
    self.curr_numbers = [];
    % Replace any numeric characters with spelled out letters
    num_idx = 1;
    % Set a timer flag
    timer_for_flag = 0;
    split_string = split(input_string);
    for idx = 1:length(split_string)
        elem = split_string(idx);
        % Parse for keywords involving timer commands
        if ~isempty(str2num(elem)) && ~strcmp(elem,'timer') && ~strcmp(elem,'minutes') &&
~strcmp(elem,'seconds') && ~strcmp(elem,'hours') && ~strcmp(elem,'what')
            if ~strcmp(elem,'hot')
                % Strange exception we included for a processing glitch with song names
                self.curr_numbers(num_idx) = str2num(elem);
                num_idx = num_idx + 1;
                split_string(idx) = "number";
            end
        end

        % Comparison clause for word “one”, doesn’t transcribe properly
        if strcmp(elem,'one')
            % for some reason one doesn't transcribed as '1'
            self.curr_numbers(num_idx) = 1;
            num_idx = num_idx + 1;
            split_string(idx) = "number";
        end

        % Convert all mislabeled “for” to “four” for math commands
        if ( ismember("plus",split_string) || ismember("minus",split_string) || ismember("times",split_string) ||
ismember("divided",split_string) || ismember("divide",split_string) ) && strcmp(elem,'for')
            self.curr_numbers(num_idx) = 4;
            num_idx = num_idx + 1;
            split_string(idx) = "number";
        end
```

```
            % Check for erroneous timer 'fors'
            if (ismember("timer",split_string)) && strcmp(elem,'for')
               if timer_for_flag
                  self.curr_numbers(num_idx) = 4;
                  num_idx = num_idx + 1;
                  split_string(idx) = "number";
               end
               timer_for_flag = 1;
            end
         end

         % Reconnect processed string for output
         output_string = join(split_string);
      end
   end
```

D.

```
function [output_commNum] = postProcessInputString(self,input_string)

   % Use getMostSimilarCommand to compare string to input tokens (see Appendix B)
   self.getMostSimilarCommand(input_string);

   % Parse through the indices of the confidence scores, starting at the most likely option
   for idx = 1:length(self.currConfidenceRatings)

      % Get command number for the confidence order index
      current_command_number = self.currCommNums(self.currConfidenceOrder(idx));

      % Check that current command number for important words
      validCommand = self.checkForImportantWords(current_command_number,input_string);

      if validCommand
         output_commNum = current_command_number;
         current_command_number
         return
      end

   end

   % If no valid command was detected, set command number to -1 as an exception
   output_commNum = -1;

end
```

E.

```
function [] = doPlayCommand(self, commNum)


   % Clear the song name variable
   self.song_name = '';

   % Scan for command numbers to play individual artists
   % Kanye
```

```
if ismember(commNum, 326:328)
   X = randi(5);
   self.song_name = self.kanye_songs(X);
   self.processed_song_data_index = X;
   % Play response
   [y,fs] = audioread('KANYE6.wav');
   sound(y,fs);
   pause(length(y)/fs + 0.5);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Other artist queue commands are omitted here for consolidation   %
%  Other artists are Kendrick Lamar, the Beatles, MF DOOM,          %
%  Pink Floyd, Tyler the Creator, Lady Gaga, Steely Dan,            %
%  Miles Davis, and Anderson Paak.                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Paak
elseif ismember(commNum, 345:348)
   X = randi(5);
   self.song_name = self.paak_songs(X);
   self.processed_song_data_index = X + 45;
   % Play response
   [y,fs] = audioread('PAAK6.wav');
   sound(y,fs);
   pause(length(y)/fs + 0.5);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Scan for command numbers to play individual songs

% Kanye songs
elseif ismember(commNum, 349)
   self.song_name = self.kanye_songs(1);
   self.processed_song_data_index = 1;
   [y,fs] = audioread('KANYE1.wav');
   sound(y,fs);
   pause(length(y)/fs + 0.5);
elseif ismember(commNum, 350)
   self.song_name = self.kanye_songs(2);
   self.processed_song_data_index = 2;
   [y,fs] = audioread('KANYE2.wav');
   sound(y,fs);
   pause(length(y)/fs + 0.5);
elseif ismember(commNum, 351)
   self.song_name = self.kanye_songs(3);
   self.processed_song_data_index = 3;
   [y,fs] = audioread('KANYE3.wav');
   sound(y,fs);
   pause(length(y)/fs + 0.5);
elseif ismember(commNum, 352)
   self.song_name = self.kanye_songs(4);
   self.processed_song_data_index = 4;
   [y,fs] = audioread('KANYE4.wav');
   sound(y,fs);
   pause(length(y)/fs + 0.5);
elseif ismember(commNum, 353)
```

```
        self.song_name = self.kanye_songs(5);
        self.processed_song_data_index = 5;
        [y,fs] = audioread('KANYE5.wav');
        sound(y,fs);
        pause(length(y)/fs + 0.5);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %  Other songs queue commands are omitted here for consolidation  %
    %  Other songs are by Kendrick Lamar, the Beatles, MF DOOM,      %
    %  Pink Floyd, Tyler the Creator, Lady Gaga, Steely Dan,         %
    %  Miles Davis, and Anderson Paak.                               %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Paak songs
    elseif ismember(commNum, 394)
        self.song_name = self.paak_songs(1);
        self.processed_song_data_index = 46;
        [y,fs] = audioread('PAAK1.wav');
        sound(y,fs);
        pause(length(y)/fs + 0.5);
    elseif ismember(commNum, 395)
        self.song_name = self.paak_songs(2);
        self.processed_song_data_index = 47;
        [y,fs] = audioread('PAAK2.wav');
        sound(y,fs);
        pause(length(y)/fs + 0.5);
    elseif ismember(commNum, 396)
        self.song_name = self.paak_songs(3);
        self.processed_song_data_index = 48;
        [y,fs] = audioread('PAAK3.wav');
        sound(y,fs);
        pause(length(y)/fs + 0.5);
    elseif ismember(commNum, 397)
        self.song_name = self.paak_songs(4);
        self.processed_song_data_index = 49;
        [y,fs] = audioread('PAAK4.wav');
        sound(y,fs);
        pause(length(y)/fs + 0.5);
    elseif ismember(commNum, 398)
        self.song_name = self.paak_songs(5);
        self.processed_song_data_index = 50;
        [y,fs] = audioread('PAAK5.wav');
        sound(y,fs);
        pause(length(y)/fs + 0.5);
    end

    % Show the song name
    self.song_name = append(erase(self.song_name, ".wav"), "_new_filtered.wav");

    % Initialize file, file information, writer object, conditional variables
    self.fileReader = dsp.AudioFileReader(self.song_name);
    self.fileInfo = audioinfo(self.song_name);
    self.deviceWriter = audioDeviceWriter('SampleRate', self.fileInfo.SampleRate);
    self.isPlaying = 1;

end
```

F.

% This is an excerpt from the dictionary of commands, showing the need for having multiple variations of a single command hard-coded into the program.

```
dictionary_comp1 = [
        "whats the weather today"          % 1
        "what is the weather today"        % 2
        "weather today"                  % 3
        "the weather today"               % 4
        "weather"                       % 5

        "what is the news today"           % 6
        "what is the news"                % 7
        "news today"                     % 8
        "whats going on today"             % 9
        "whats happening today"            % 10
        "what happened today"              % 11
        "what happened yesterday"          % 12
```

G.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Function to perform cancellation for flat panel devices    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Convolve the played audio with the impulse response of the panel
expected_output = conv(played_audio,IR)';

% Indices for  the relevant part of the audio files :
start_idx = 172737;
stop_idx = 197348;

% Get close to the sample delay and then iteratively arrive at the appropriate sample delay to line up the audio files:
[Z,Z_idx] = xcorr(recorded_audio,expected_output(1:length(recorded_audio)));
Z_idx;
[~,I] = max(Z);
Z_idx(I);
 % Sample delay is based off of the results of xcorr()
sample_delay = Z_idx(I);

% The value of sample_delay is a good approximation of the sample value, but the following iterative approach will optimize the value to be as close as possible, since pure subtraction is very sensitive to misalignment.

% This block of code pads the audio signals for later processing
if sample_delay <= 0
   pad = zeros(10,1);
   recorded_audio_pad = [pad; recorded_audio];
   poss_delays = 1:20;
else
   min_delay = -sample_delay + 1;
   if min_delay > -10
      poss_delays = [min_delay:10] + sample_delay;
   else
      poss_delays = [-10:10] + sample_delay;
   end
```

```
    recorded_audio_pad = recorded_audio;
end
min_sum = inf;
min_sum_idx = -1;

% Iteratively parse through the delay values such that you find the value that minimizes the optimization of xcorr()
for idx = 1:length(poss_delays)
    curr_delay = poss_delays(idx);

    RA = recorded_audio_pad(curr_delay:end);
    EO = expected_output(1:length(RA))';

    if length(RA) < 400000
        RA = RA(1:length(RA));
        EO = EO(1:length(EO));
    else
        RA = RA(1:400000);
        EO = EO(1:400000);
    end

    RA = RA(start_idx:stop_idx);
    EO = EO(start_idx:stop_idx);

    % Additional processing to calculate the optimum gain value for canceled audio:
    opt_gain_exp =@(x) sum( abs( (RA - x.*EO)));
    opt_gain = fminsearch(opt_gain_exp,0);
    EO = EO *opt_gain;

    CO_PS = RA - EO;

    curr_sum = sum(abs(CO_PS));

    if curr_sum < min_sum
        min_sum = curr_sum;
        min_sum_idx = idx;
    end

end

% Optimized sample day
sample_delay = poss_delays(min_sum_idx);

% Trim recorded audio and expected output signals
RA = recorded_audio_pad(sample_delay:end);
EO = expected_output(1:length(RA))';

if length(RA) < 400000
        RA = RA(1:length(RA));
        EO = EO(1:length(EO));
else
    RA = RA(1:400000);
    EO = EO(1:400000);
end

EO_pre = EO;
```

```
% Final optimization of gain value for canceled audio
opt_gain_exp =@(x) sum( abs( (RA(start_idx:stop_idx) - x.*EO(start_idx:stop_idx))));
opt_gain = fminsearch(opt_gain_exp,0);


% Calibration step
EO = EO * opt_gain;



% Implementation of pure subtraction
CO_PS = RA - EO;

% CO_PS is the final canceled signal

% Do normalization for listening
RA_norm = RA / max(abs(RA));                % Raw mixture
EO_norm = EO / max(abs(EO));                % Just the music signal
CO_PS_norm = CO_PS / max(abs(CO_PS));   % Enhanced
```

H.

```
% Real Time Simulation for Cancellation Technique

% Initializing AudioAssistantHandler instance and all variables
AAH = AssistantAudioHandler(1024, 10, 48000, IR_DRIVER, 1);
opt_gain = 1.0029; % 1.0989
delayed_frame = zeros(1, 1024);
cancelled_frame = zeros(1,1024);
total_output = zeros(1, floor(length(played_audio)));
total_input = zeros(1, floor(length(played_audio)));
total_delayed = zeros(1, floor(length(played_audio)));


% Smaller for loop to speed run time
for i = 1 : 200

   % Get input frame, i.e. what the sensor sees
   input_frame = recorded_audio( (1 + (frame_len*(i-1))) : frame_len*i);
   % Get output frame, i.e. what the driver is playing
   output_frame = expected_output( (1 + (frame_len*(i-1))) : frame_len*i);

   % Write samples to buffer
   AAH.write_to_buffer(output_frame)

   % Using i > 2 since the delay is 1025, which is greater than a frame.
   % If i > 2 then the delayed frame is acquired with a delay of 1025 and
   % subtracted from the raw piezo input
   if i > 2
      delayed_frame = AAH.read_from_buffer_with_delay(sample_delay-1);
      cancelled_frame = input_frame.' - (delayed_frame*opt_gain);
      % Saving the output
      total_output((1 + (frame_len*(i-1))): frame_len*i) = cancelled_frame;

      % Storing sensor input and delayed frames for graphing purposes
      total_input((1 + (frame_len*(i-1))): frame_len*i) = input_frame;
```

```
    total_delayed((1 + (frame_len*(i-1))): frame_len*i) = delayed_frame;
  end

end

% the array total_output contains the simulated canceled audio data
```

Demo Video of the Virtual Assistant Implemented on the Panel:
https://drive.google.com/file/d/1PJSb3pfJldvJ1jDDFQwYhgYOu9JW5XCS/view?usp=sharing

I.

```
function [output_frame] = read_from_buffer_with_delay(self, delay)
        % Since the write function would presumably have been run before this, set read pointer as the previous position of write pointer
        self.read_pointer = self.write_pointer - 1;

        % Exception for when write_pointer = 1
        if self.read_pointer == 0
           self.read_pointer = 10;
        end

        % The biggest problem with this method is that the delay value may push the indices of the delayed frame below the indices of the circular buffer, meaning I have to wrap it back around to the end. Thus, I have an if statement to catch any exceptions for this issue.

        % Condition 1: the higher index is lower than the first index of the circular buffer, which de facto means that both indices are lower than the first index of the circular buffer
        if (self.frame_len * self.read_pointer - delay) < 1
           output_frame = self.circular_buffer( ((self.frame_len * (self.read_pointer - 1)) + 1 - delay) + length(self.circular_buffer) : (self.frame_len * self.read_pointer - delay) + length(self.circular_buffer));

        % Condition 2: only the lower index is lower than the first index of the circular buffer, which means that the higher index is still within the bounds of the circular buffer index
        elseif ((self.frame_len * (self.read_pointer - 1)) + 1 - delay) < 1
           output_frame = [self.circular_buffer( ((self.frame_len * (self.read_pointer - 1)) + 1 - delay) + length(self.circular_buffer) : length(self.circular_buffer))   self.circular_buffer(1 : (self.frame_len * self.read_pointer - delay))];

        % Condition 3: Neither index is outside the indices of the circular buffer
        else
           output_frame = self.circular_buffer((self.frame_len * (self.read_pointer - 1)) + 1 - delay : self.frame_len * self.read_pointer - delay);
        end
      end
```

J. Piezo comparison recordings

Small:
https://drive.google.com/file/d/1inpQ-WB_wsPhJ3f2bWJunp2OjtFlL0ie/view?usp=sharing
Medium:
https://drive.google.com/file/d/1u6nTR7Ay4mTs0X1aZnSsebz5HqOlsw97/view?usp=sharing
Large:
https://drive.google.com/file/d/1v8PUJba5qZ3gto4BSVIat4YMlKPxpdjG/view?usp=sharing

K. Demo Video of the Virtual Assistant Implemented on the Panel:
https://drive.google.com/file/d/1a3LO67VSScmDhL8WzlVtUBFMsJoKw8C0/view?usp=sharing