Sing us a song, you're a piano pi!

Marco Acea, Angela Chang, John Martins

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—For centuries, pianos have been used to play music. "Player pianos", containing mechanisms that allow them to sound without human control, were first introduced in the late 19th century. This mechanism allows the instrument to be freed from its traditional constraints: the number of frequencies produced is no longer limited to a meager number of ten fingers.**

***Index Terms*—Signals and Systems, Embedded Systems, Internet of Things**

I. Introduction

This project seeks to take advantage of player pianos to simulate human speech. Human speech is made up of many frequencies, and enough keys - and their frequencies - combined can reproduce speech with high enough fidelity that a player piano can speak and be understood. Different modules of this project will come together to translate the phonemes of human speech into timed key presses on a piano. The audio processing module will receive audio input. This input information will be sampled for different frequencies and amplitudes. A smooth average will be taken at each frequency that corresponds to a piano key. We will repeat this at each time period. The key scheduling module will thus take the output of the audio processing module and use the time and amplitude information to determine the keys that need to be played at different times.

We were inspired by the work of Mark Rober, who showcased on his Youtube channel a talking piano implemented on an Edelweiss self-playing piano. Outside of simulating speech, this project allows musicians more opportunities for expression by allowing them to overcome the physical constraints of ten human fingers. Such arrangements are useful for playing music not written for the piano, as well as exploring options in avant-garde genres.

We were inspired by the pianolizer project created by Stanislaw Pusep for our virtual piano implementation, on an aesthetic and logistic level. Our virtual piano took inspiration from this project on several fronts. Firstly, the keys light up as they're actuated, to help the audience visualize what is being played. Secondly, we also implemented the ability to select between several recordings, ie. the past inputs to allow the user to compare different audios.

II. Use-Case Requirements

There are four major requirements for our use-case scenario. The first thing we need access to is recordings of our users' voices. To achieve this, we'll build a user interface into our web application that allows users to record their voice and send it to the backend. For the user experience to be pleasurable, our UI needs to have a ~200ms end-to-end latency. This means that for any user interaction, there should be at most 200ms before any new actions can be taken, for example, listening to the post-processed audio produced from a user's voice recording. We derived this number based on conventional advice regarding user response. Usually, 100ms is a limit for the user's flow of thought to stay uninterrupted and for the user to feel that the system is instantaneous. Since our recording system is not real-time like a website, we had leeway in this regard; we decided that 200ms is a reasonable time for a non-real-time response to feel smooth and acceptable.

Next, with the recording audio of a user's voice, we need to extract the frequencies that make a user's speech. We need to gather information on how these frequencies change throughout time to generate a sequence of keys to be played. To achieve this, we divide the incoming audio into 'windows' which we can use to see how the frequencies of a user's voice change with time. A metric of success for this requirement involves being able to accurately estimate the frequencies and amplitude of each frequency within a user's speech for each of those time "windows". We decided that an 80% estimation accuracy for these frequencies and amplitudes would ensure our audio processing transformation introduces as little error as possible. Once we have a description of the keys we need to play on the piano, those notes need to be scheduled over time, onto a physical device so that the piano keys are played correctly. Errors introduced in our implementation of this requirement can affect the intelligibility of our piano's output. Inaccurate timing in our scheduler will affect the timing of syllables the piano needs to produce, by delaying, elongating, or speeding up syllables. To mitigate these errors, we've decided that our scheduler should miss less than 5% of timed syllables. This will ensure that any errors in timing will not affect the entirety of our system's output.

Lastly, we need to implement a virtual piano that will actuate the keys and produce the sounds we aim to recreate. The success and accuracy of this requirement encapsulate the overall performance of our system's pipeline. Since there is a much smaller range of notes possible with a piano, we understand that the output of our piano will never be an exact copy of a user's speech. Therefore, we're requiring our physical device to have an 80% fidelity rate. In other words, the output speech of our piano should be intelligible to a user 80% of the time.

III. Architecture and/or Principle of Operation

While planning our overall system architecture, we realized the importance of modularity within the different subsystems. Allowing each subsystem to treat the others as black-boxes creates an opportunity to isolate subsystem tests and debugging with simulated inputs/outputs. We've grouped our subsystems into 4 main areas: user interface, audio processing, note scheduling, and physical performance. Figure 1 illustrates how the subsystems interact with each other. Our system begins in the user interface, where users will interact with our web server via a browser on their personal devices. The user will be able to record audio files in-browser or choose amongst previously recorded files to play on the piano. The user interface handles obtaining the audio recording file, as a .wav file, and passes it along to the audio processing subsystem hosted on our backend server. The audio processing subsystem converts the input audio into the collection of frequencies that make up these sounds across time. The audio processor then maps the frequencies at each timestamp seen in the audio recording to the frequencies playable by each key of the piano and encodes this data in a text file to be sent to the note scheduler hosted on the backend server. The note scheduler then processes the data and creates a two-dimensional list of keys at different timestamps, where each entry denotes one of: non-movement (0), pressing (a number representing the volume), or lifting (-1). This is used to inform the virtual piano of the movement of the keys and how they will play and lift to create the performance.

IV. Design Requirements

As noted in our use-case requirements, we have 4 main requirements that each influenced the choices we made when designing our system.

A. Web App Latency

We are requiring that there is a <200ms latency period between interacting with our web application and noticing a change in the playing of audio on the piano. To ensure this, we are hosting our audio processing subsystem on the backend of our web application on AWS. This will allow us to use the powerful computers of the AWS EC2 instance to process the audio very rapidly. This leaves most of our 200 millisecond latency time for the transmission of data from AWS to the Raspberry Pi via sockets. We are estimating that this should be plenty of time since we are condensing the scheduled note information being sent into simple text files of bits describing which keys need to be played. To improve our transmission speeds, we will be testing an optimal balance between the size and frequency of data packets to minimize sending times. Lastly, the Raspberry Pi will immediately begin sending the received stream of bits to the shift registers to get played by the solenoids.

B. Frequency and Amplitude Extraction Accuracy

We understand that there will be some inevitable loss when translating the frequencies of the human voice to the finite and relatively small number of keys on a piano. To make sure we are maximizing the ability to play the voice as close to the original recording as possible, we need to ensure that we are extracting the frequencies and amplitudes from the recording as correctly as possible. We have chosen an 80% accuracy rating for the extraction of frequencies and their amplitudes as described in section N. This information is extracted from the voice recording by first splitting the recording into several windows of time and then processing each window through a Fourier transform, which converts the recorded data from amplitude across time to amplitude across frequency. We will be running tests to figure out the most optimal time window for our Fourier transform such that the accuracy of the frequencies and amplitudes extracted is maximized.

C. Note Scheduling Accuracy

Next, we want to design our system to schedule notes to miss or incorrectly play less than 5% of the syllables spoken in the voice recording. Human speech, in English, is typically spoken at 10 to 15 phonemes (distinct sounds), or 4 to 7 syllables, per second [3][4]. As our system seeks to emulate human speech, our rate of producing sounds should not differ much from this rate. Typically, a piano key on an analog upright piano can be pressed up to 15 times per second. This limitation comes from the mechanism that moves the piano's hammers to its strings, known as the action. We will be using a digital piano, which does not have an action. However, digital pianos are programmed to resemble analog pianos as much as possible, and thus we will still use 15 times per second as a limitation for our piano. We have tested the maximum actuation rates of some test 5N solenoids and found that they can actuate at rates up to 16 Hz. Since the solenoids can fire up to 16 times per second, we will be able to create up to 16 distinct sounds in 1 second, allowing us to capture the upper bound speeds of typical English speech.

D. Fidelity Rate of Final Piano Playback

Lastly, we are aiming to achieve an 80% fidelity rate, meaning that 80% of the words spoken in the original voice recording will be able to be identified when listening to the piano playback. To ensure this, we must be able to play the correct frequencies at the right volume levels at the right times. To ensure correct frequencies can be played in the first place, we start with the frequency range of human voices. We found that the fundamental frequency for human voice can be as low as 85Hz and that telephone communication captures at frequencies up to 4000Hz. This gives us a good approximation of the range of human voices. By applying the equation on figure 2. We found that to capture the ranges of the human voice, we need all the keys from 20 (82Hz) to 88 (4186Hz).

$$n = 12\,\log_2\left(\frac{f}{440\,\mathrm{Hz}}\right) + 49$$

Figure 1: n is the nth key of the piano starting from the leftmost key of an 88-key piano

Next, we need to make sure that we can most accurately map the frequencies obtained from our Fourier transform to the frequencies playable by the piano keys so that we can best determine which piano keys are needed to play a particular sample of the voice recording. To do this, we will be implementing audio blurring. This involves taking surround frequencies around the frequencies of each piano key, averaging their amplitudes and weighting them by how distant they are from the frequency of the piano key, and determining if this average passes a threshold determining that the key should be played. For example, in the case where the voice audio file contains weak amplitudes at exactly the frequency of note A4 of the piano (440Hz), but contains strong amplitudes at frequencies 437, 439, and 445, we will be able to capture these surrounding frequencies and determine that the note A4 should indeed be played to capture the slightly different frequencies. On top of that, humans are only able to discern a > 1% frequency difference between two tones.
To best reproduce the amplitude of each frequency that makes up any given sample of the voice recording, we will be encoding the bits sent to the solenoids with a PWM signal, allowing us to control the volume at which the keys are pressed.

V. Design Trade Studies

A. Virtual vs. Physical Piano Performance

One of the biggest challenges when initially designing the scope and specifications of our project was determining whether we would be recreating the processed speech virtually through a virtual piano interface or physically with solenoids pressing keys on a real piano. We understood that building a physical key pressing system would introduce a lot of complexity and potentially cause us to focus on mechanical systems that were out of the scope of our ECE-focused design requirements. After deliberate research and preliminary design of the circuitry and hardware needed to create the physical system, we decided that we would attempt to create the physical performance interface, and fall back to implementing the virtual performance interface if the physical system presented too many issues or troubles focusing on mechanical areas. This choice influenced several other design choices (as in the following section), and to plan around this fallback, we also derived a proof-of-concept design that we would build and test to help determine the feasibility of building the entire physical interface. Our main drive for developing this physical interface lay in providing the user with the most engaging experience possible in seeing how the piano can be used, outside the possibility of human players, to recreate speech and extend the realm of the typical use of pianos. We also coordinated efforts with Benjamin Opie, an electronic music professor here at CMU, to be able to have access to the digital piano practice room for the testing of our designs.
Just recently, we implemented our proof-of-concept design consisting of wiring up 3 5N solenoids connected to shift registers and MOSFETs for actuation. We were successfully able to actuate the solenoids in customized patterns based on serial bit streams inputted to the shift registers and outputted, in parallel, to the solenoids. This test helped us determine our power consumption and strategy when scaling this up to the full, 69-key system.

B. Parallelization of Audio Processing

The first major design trade study that we analyzed in our system pipeline was with respect to the parallelization of the audio processing. With our methodology of performing the Sliding Discrete Fourier Transform (SDFT), we are processing the amplitude of frequencies for each individual note of each time stamp across the time of the whole audio file. What this means is that we need to go key by key on the piano and process the amplitude data for each, thus resulting in a higher latency computation that runs through each note one by one. In order to combat this, we considered parallelizing the computation across multiple processors such that each one could process fewer notes and ultimately finish the computation faster, however this would also result in much higher memory usage and latency is transferring the large arrays of data between each processor and memory. This also proved to be very complex to implement given how the programs were stored in the web app backend and it was much more expensive to implement through Amazon Web Services (AWS) since we needed an instance that had many CPU cores. On the other hand, implementing our program sequentially, meaning it runs on one CPU locally on our laptops, was much simpler to program and ensure the correctness of, but ended up skyrocketing our latency when computing. In the end, we chose to implement the sequential version because we wanted to ensure processing correctness and ease of implementation over potentially erroneous and complex code and faster processing computation. Based on our baseline user requirements, producing a more correct output was more ideal.

C. Web-Hosted vs. Hardware

When we were deciding where we would host and store our programs, we identified we needed 2 main components: a strong computational unit capable of performing complex signal processing on incoming voice recording quickly, and a way to host a fast-responding user interface that could be used to record audio and interact with the system like pausing/playing and loading past recordings.

We first looked at using a Raspberry Pi for both the audio processing and notes scheduling portion of the system. We found that what we gained in low data transfer times and portability, we lost heavily in computing power. We quickly determined this would not work.

We next looked at using a Jetson Nano to do the audio processing, while keeping the Raspberry Pi for the notes scheduling. In this scenario, we could probably achieve performance with the stronger computer on the Jetson, however, we needed to take into account our virtual piano backup plan in case things didn't go as we wanted. If we needed to implement the virtual piano instead, we would need to transfer the processed audio data and key schedules to AWS, where the virtual piano would be hosted. On top of that, this would introduce uncertainty as to where to host our user interface. If we hosted it locally, users would only be able to record audio and access the piano system on our local machines, but we would be able to run commands hosted on our local Jetson much more quickly. If we hosted our user interface on AWS, this would allow any user to access the interface on their personal device for recording, however, we would need to transfer the entire audio file to the Jetson for processing, which could introduce a large lag time. These trade-offs brought us to our currently agreed-upon design choice of hosting the user interface and audio processing on an AWS EC2 instance and transferring the data to a Raspberry Pi for playback on the physical piano interface. With these choices, any user can access the web application on their personal device for recording. These recordings are seamlessly sent to the audio processing subsystem also hosted on AWS. By hosting our audio processing subsystem on AWS, we can take advantage of the powerful computers of the EC2 instance to perform the audio processing very rapidly. Lastly, if we must implement our virtual piano instead of our physical performance interface, the rest of the subsystems will be already hosted on AWS, therefore transferring data to the virtual piano would be optimal. Ideally, if we get this system toolchain working with our physical piano interface and we still have time in our project, we will work to migrate the web app and audio processing to a Jetson to remove any data transfer lag times and take advantage of the portability of having a fully hardware-based system.

D. Note Scheduling Decay Model

The next trade off we studied was how to represent the natural decay of piano notes within our note scheduling module. For context, the note scheduling module uses a model of the natural sound decay when a piano note is pressed because the scheduler needs to determine whether a note that has already been played needs to either stay pressed down (thus producing a sound that follows the natural decay), unpressed (thus indicating a let go of the note and silencing the sound), or needs to be re-pressed (thus the sound is triggered again at the specified volume). The selection of which option to choose for each note at each timestamp comes from analyzing the amplitude of each piano note frequency at adjacent time intervals and determining how to react to change in amplitude of one note frequency from one time stamp to the next. The tradeoff study here was focused on how accurately we need to model the decay of piano notes. We started with a naive approach which simply played each note again and again at each time sample at the specified volume. This approach was the simplest to implement and resulted in the most reproducible, reliable program, however, was quite jittery sounding and was susceptible to lots of noise. Next, we looked at modeling a simple exponential decay for each note in which we would keep track of each note played and predict the volume of each note in the future timestamps based on this decay to determine whether the note should continue to be held down and ring, silenced, or played again. This approach would help smooth out the jitteriness of the audio, however it produced complex code, errors, and overall unreliable test data. Lastly, we thought about modeling a much more accurate decay for the piano notes based on research papers or even recording the audio files we had for each piano note and manually measuring the decay for each. This would prove very difficult and thus we did not implement this. In the end, we ended up choosing a hybrid approach of the naive one and a basic piano decay model. We used the basic model to test out our program fidelity but ended up using the naive approach for our final project due to its reliability in processing and our priorities with making the program output something successfully.

E. Virtual Piano Note Visualization

The last major design trade-off that we analyzed was how we would visualize the notes being played by the virtual piano. In our virtual piano implementation, we planned to have a piano keyboard visualized on the screen and show which notes are currently being played that produce the sound that is being heard by the user. One idea we had was to show the notes ‘raining’ down on the screen as they are played as to show the current notes, the future notes, and provide an overall more engaging experience for visualizing the notes. The other

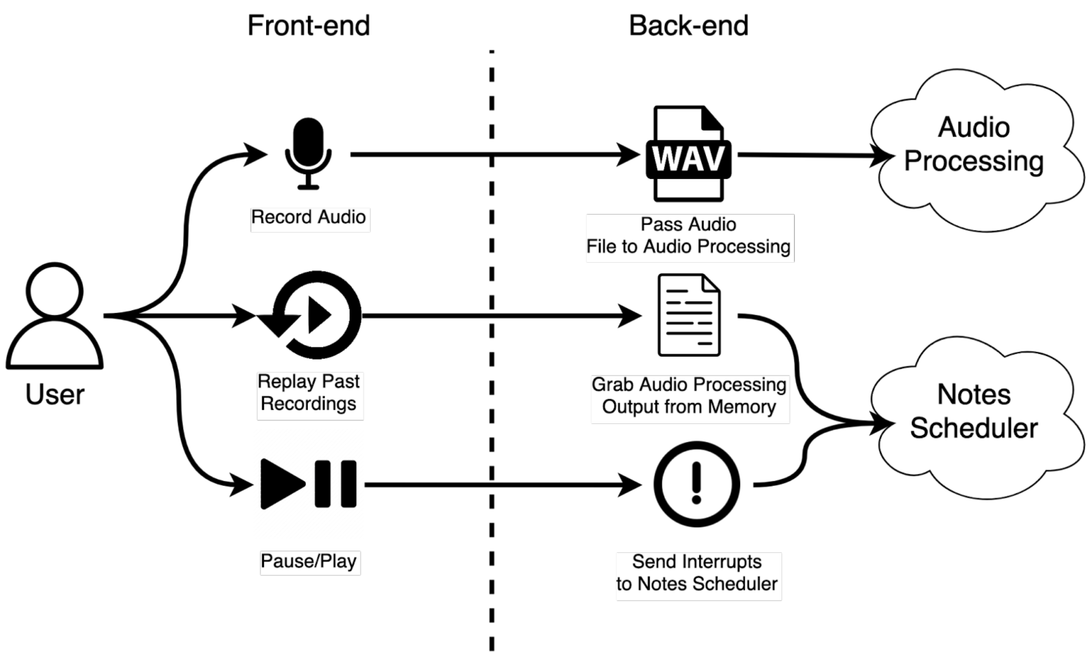
Front-end
Back-end
Record Audio
WAV
Pass Audio File to Audio Processing
Audio Processing
User
Replay Past Recordings
Grab Audio Processing Output from Memory
Notes Scheduler
Pause/Play
Send Interrupts to Notes Scheduler

Figure 2. A diagram of the web application interface design and data flow

approach we had was to simply highlight the keys currently being played on the piano (as shown in figure 3). In understanding which idea to implement, we considered our original user requirements in which we ultimately wanted to produce the representation of the sound and focus our efforts on doing that well. So between audio and visuals, the audio was much more important. For these reasons, we chose to implement the latter method in which just the notes are highlighted red to indicate they are being played. Implementing this method allowed us to focus more efforts on producing higher quality audio playback and ultimately a better product as we determined was required by our users.

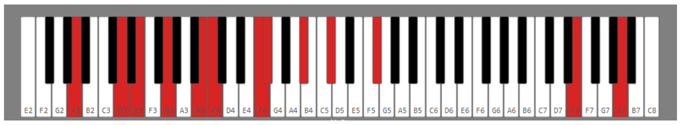
E2 F2 G2 A2 B2 C3 D3 E3 F3 G3 A3 B3 C4 D4 E4 F4 G4 A4 B4 C5 D5 E5 F5 G5 A5 B5 C6 D6 E6 F6 G6 A6 B6 C7 D7 E7 F7 G7 A7 B7 C8

Figure 3. This shows how we implemented the virtual piano visualization in our web app. The keys in red are the ones played to reproduce the sound the user would currently be hearing.

F. Comparing Different Averaging Techniques

To propagate as much information about the frequencies that make up a user's voice throughout time, we want to average those frequencies around the discrete points that correspond to piano keys. We considered four options, each proposing more added benefit than the last. The first was to filter information at the frequencies that correspond to the piano keys. This approach loses the most amount of data on what the original frequencies of a user's voice were. However, this approach offers the benefit of being able to be implemented in hardware since the number of filters needed would be small.

Our second approach was to compute a moving average of all the frequencies before the point where a piano key lies. This improves on our initial approach but lacks information about the frequencies that lie after the frequency of a piano key.

Our third potential solution was to compute an average using the neighboring points before and after the frequency for our piano key. We think this is our best bet since this averaging technique is capable of propagating all of the original features from the original audio, albeit without the same resolution.

We considered a weighted neighbors average, where again we average the neighboring points around the frequency of a piano key but weigh each neighbor's contribution to the final average by their distance from the piano key's frequency. Although a valid way of averaging our input, it would cause frequencies lying between any two piano keys to not be propagated. Therefore, we think that our third approach is our best attempt at propagating all the information regarding a user's voice onto the piano keys.

G. Solenoid Choice

Originally, we were deciding between using 5 Newton solenoids and 25 Newton solenoids for actuating the piano keys. Looking at the solenoids available to us, the 5N solenoids were typically much less expensive and used less power to run. For our proof-of-concept design of the physical interface, we ordered 5N solenoids that needed 12V and 1A to run. We found they got quite hot when left activated for extended periods of time and that 5N is just barely enough to actuate a key at an audible level. Another problem with using 5N solenoids is that since they need the full power to actuate the key, there will be no room for adjusting the volume of the key press to represent the amplitude differences of each frequency. Upon further research, we found 25N solenoids that were at a good price and used a reasonable amount of power (12V, 1.5A). We will be getting the 25N solenoids as they will be able to press the keys with ease and have much more room for adjusting the speed of the key press such that we can adjust the volume of the key to emulate the amplitude of each needed frequency.

VI. System Implementation

A. Web Application User Interface

Our web application user interface is the subsystem responsible for supporting controls the user can use to interact with our system. [Figure 3] On the web app, users will be able to record audio, upload past recordings, or pause/play the piano playback. We will be hosting our web app with Django, a python-based web framework, on an AWS EC2 instance. Using Django allows us to conveniently integrate our audio processing and note scheduling into our backend since those will be written in python as well. On the front-end user interface, we will be using Bootstrap for a pleasing and simple design and Ajax to support asynchronous calls to our audio processing and notes scheduler functions while maintaining responsive interactions. Audio recordings will be done using built-in Javascript libraries that will save the file in the .wav format, a lossless audio format that stores the audio as amplitude across time. This .wav file will be then sent to the audio processing subsystem. The pause/play functionality will be implemented by sending interrupts to the notes scheduler whenever the user toggles the pause/play of a recording. The interrupts will contain information that causes the notes scheduler to either continue or stop sending information to the Raspberry Pi.

B. Audio Processing

The following implementation details are also illustrated in figure 3. The audio processing module takes a recording of a user's voice as input. For consistency, we chose to only work with WAV (.wav) audio recordings. The incoming .wav file is converted into a *numpy* array representing the time series data of that audio, as in figure 4.

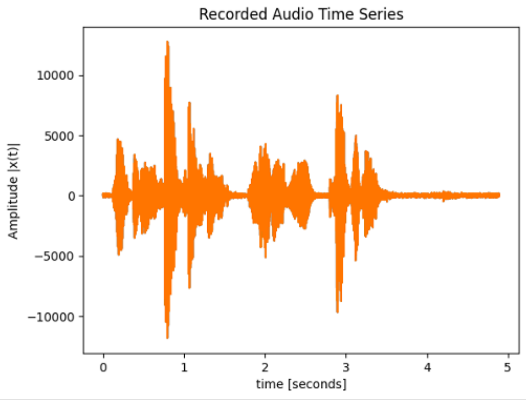

Figure 4. An example of a time-based audio signal

In order to determine what piano keys to press, we need information on how the frequencies in our user's voice change through time. To do this, we use the Sliding Discrete Fourier Transform (SDFT), which is a recursive implementation of Discrete Fourier Transform (DFT) that returns the power found at a specific frequency bin k [5].

$$X_k[n] = [X_k[n-1] - x[n-N] + x[n]]e^{j2\pi\frac{k}{N}}$$

$$x[n] = [X_k e^{-j2\pi\frac{k}{N}}] - X_k[n-1] - x[n-N]$$

Figure 5. The Sliding Discrete Fourier Transform and its inverse

Performing a single DFT on a window of time series samples would return an array of evenly spaced frequency bins. However, with a sample rate of 48 kHz and maximum window size of 3200 samples, which was determined using the play rate, we cannot extract information about the specific frequencies that correspond to piano keys. Therefore, given some frequency F, the signal processing module will find a window size N smaller than the maximum window size N_{max} such that one of its frequency bins corresponds to the frequency F.

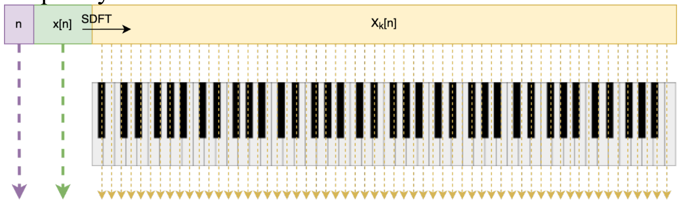

Figure 6. Computing power at each key throughout time

The audio processing module finds a window size *N* and frequency bin index *k* for each of the 88 piano keys, (N_i, k_i) With these specific values (N_i, k_i) we can create SDFT Bins (Si), not to be confused with the frequency bins returned from the DFT. These bins calculate the current power present $(X_k[n])$ at its corresponding frequency bin for some time sample x[n], using the formula for the SDFT shown in figure 5. We parse the original audio array *x[n]* through each of the 88 SDFT Bins, and then take a simple moving average of the power present at each play rate sample, which are the moments in time corresponding to the play rate of 15 Hz. The result is a two-dimensional matrix, representing the power present at each piano key throughout time. An illustration of this process is shown in figure 6 and 7.

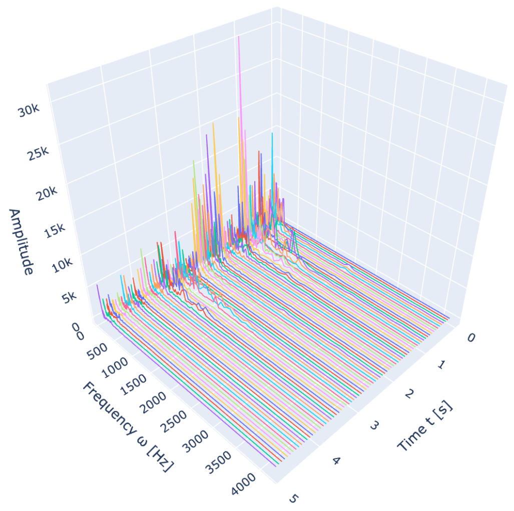

Figure 7. Change in frequency throughout time

The note scheduler, and our virtual interface have no notion of power, therefore the 2-D matrix is normalized using the largest power present throughout time, as shown in figure 8. Note how the shape of the three-dimensional plot is similar, however the z-axis corresponding to power has been replaced with note strength.

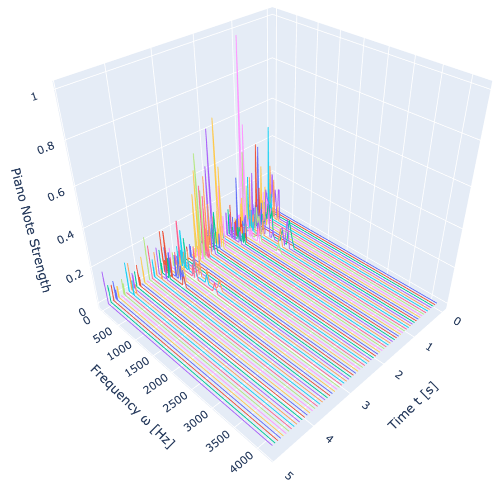

Figure 8. Normalized Frequencies through time as key strength

As an additional feature of the signal processing module, it can also return a reconstructed version of the original audio only using the frequency components represented by piano keys, as opposed to the broad spectrum of frequencies our voice can utilize. A plot of the reconstructed audio signal is shown in figure 9.

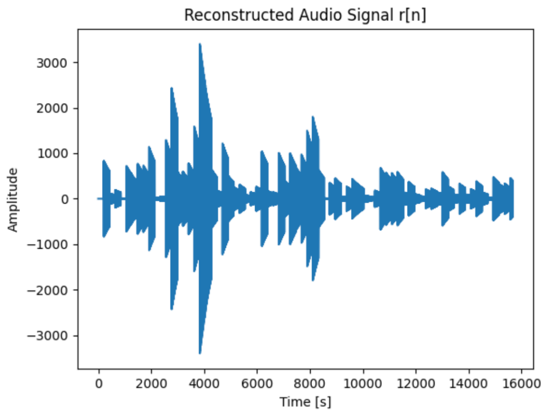

Figure 9. Reconstructed Audio

C. *Notes Scheduler*

We define the notes scheduler as the module that takes input from the audio processing module and converts it into a format our physical piano interface can use to create the replication of human speech. By definition, this requires the notes scheduler to translate time-indexed arrays of frequencies and their corresponding amplitudes into presses of piano keys at different time intervals with an appropriate level of force. In more specific terms, this module will translate the output of the audio processing system in the form of a text file with tab-separated integers. Each line of the file represents a timestamp, and each column represents a frequency. The integer at line x and column y represent the volume of frequency y at time x.

A consideration is that even though an arbitrary frequency, a, may be active at time t, the piano key does not necessarily need to be pressed at that time. It may have been pressed during a previous timestamp and still producing sound. Therefore, the note scheduler must account for which keys are already sounding while pressing new ones to produce a smooth sound that evokes human speech, as opposed to stuttering, choppy noise. The module will accomplish this task by computing, for each frequency, the difference between amplitudes for the current timestamp compared to the previous timestamp, as well as keeping track of which keys are currently pressed and for how long. If the frequency is significantly louder from one timestamp to the next, or if it has been long enough such that the sound has already faded, the key will be pressed again. Conversely, if the key is currently pressed but its amplitude is insignificant, the solenoid will lift.

D. *Physical Performance Scheduler*

For the physical performance interface, we will be implementing a circuit containing a Raspberry Pi as the signal driving unit, 9 serial-in parallel-out (SIPO) shift registers, 69 MOSFETs, and 69 cylindrical 25N solenoids. Data from the note scheduler, in the form of a 69-bit value where the nth bit represents if the nth key will be played or not. This 69-bit value will be split into 9 Bytes, where each byte will be loaded into one of the 8-bit shift registers. We will be using 9 GPIO pins of the Raspberry Pi to serially load the 69 bits into the 9 shift registers in the correct order such that the LSB is the lowest frequency note and the MSB is the highest frequency. As the data is being serially loaded into each shift register, a GPIO pin of the Raspberry Pi drives a shift clock signal to all of the shift registers such that for each bit sent to the shift register, the clock outputs a positive edge signal indicating to the shift registers to intake a bit and shift the values. Once all shift registers are filled with data (8 clock cycles), a latch signal is sent from another GPIO pin telling all the shift registers to output their values in parallel to the 69 MOSFETs. Each MOSFET acts as a transistor controlling the flow of current from our 12V power source to the 25N solenoid.

Some specifics: We need to actuate solenoids and produce a sound with a period of the time window we select for our audio processing since each of the 69-bit values represents the notes comprising a particular time window sample of the audio recording. This means we have to be able to play audio at a frequency of 1/Tw (Tw = window of time). Since we need to load 8 bits of our data into the shift registers within each of those solenoid actuation periods, our bit streams and clock signals need to be 8 times faster than the solenoid actuation frequency. This value is given by 8 * (1/Tw). Lastly, to determine the total power consumption of our system, we must calculate the power of a single solenoid and multiply it by 69 since each solenoid will be wired in parallel. For one 25N solenoid, we will need 12V and a maximum of 1.5 A of current. With this, we will need a 12V supply capable of outputting ~(1.5 * 69) = ~103 A of current. This would require a 1200 Watt power supply since power (Watts) is V*I = ~12*100.

VII. Test, Verification and Validation

In order to test the accuracy of our Fourier transform frequency extraction, we wanted to compare the original audio signal to a reconstructed audio signal using only the frequency components extracted using the SDFT. We used a qualitative approach and a quantitative approach. The qualitative approach was to write the reconstructed audio signal to a WAV file and listen to it. The reconstructed WAV file was created by multiplying the reconstructed samples with an exponential decay function so that the frequencies could be sustained for long enough like they would be on a piano. Figure 10 is an example of the original audio, Figure 11 is the reconstructed audio using the exponential function y=Ae^((-0.001)x), and Figure 12 uses the exponential decay function y=Ae^((-0.0001)x). The spectrogram of the reconstructed audio looks similar to the original spectrogram for most recordings, even going as far as to propagate the 20kHz noise band that was introduced by our microphone. Upon listening to the audio we found that we could hear the user's voice in the reconstructed audio. Our qualitative approach was to calculate the average error between two signals, and then compute the average of these errors for a collection of audio recordings and their corresponding reconstructed audio. In our results, there was a large average error, however our qualitative results gave us confidence that the frequencies we were extracting were in fact sufficient to hear what a user was saying. What was most important in being able to hear a user's voice making the decay long enough.

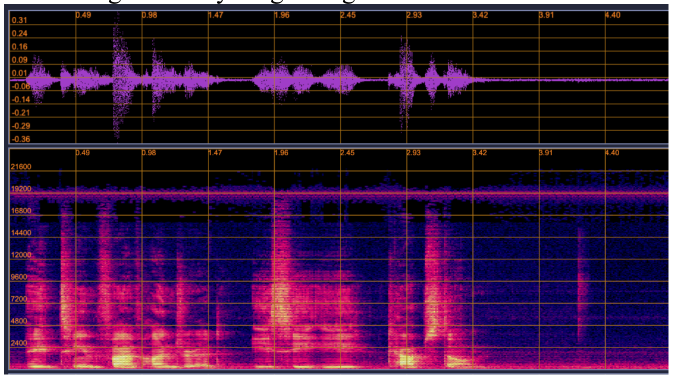
Figure 10. The original audio recording

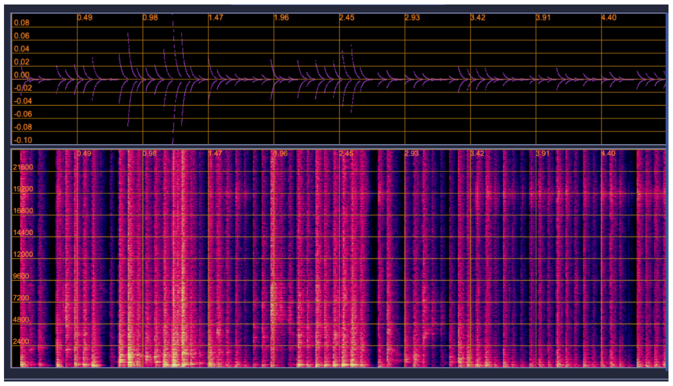
Figure 11. Reconstructed Audio using a y=e^((-0.001)x)

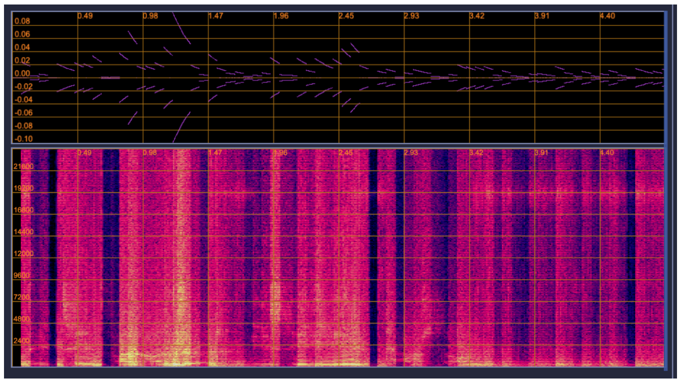
Figure 12. Reconstructed Audio using y=e^((-0.0001)x)

For the end-to-end delay on actions made on the web app, the largest delay we found was from the initial processing of a user's voice. On average, most recordings were around 5 seconds long, and took around 80 seconds to process. This is because the input audio is parsed 88 times in order to generate frequency information for each of the keys on a piano. Had we implemented parallelism into the processing algorithm, we expect the average voice recording of 5 seconds to take around 8-10 seconds to be processed, assuming the host machine has more than 8 cores.

VIII. Project Management

A. Schedule

Through the semester, we initially scheduled our time and resources with the intention of creating a physical piano interface using an array of solenoids, shift registers, and a raspberry pi. We allotted time to order parts, build a proof of concept, design, and build the final physical product. This was mostly pushed towards the end of the semester with the beginning of the semester reserved for implementing the audio processing pipeline, web app, and note scheduling interface. We pivoted to a virtual piano implementation about halfway through the semester, causing us to need to shift all our schedule and timeline around to account for needing to build the virtual piano and integrate it in the web app. Luckily, we had built in a few weeks of slack time such that all the time allotted for building/designing the physical piano interface along with this slack time was used for designing and implementing the virtual piano. We also used this time to integrate our parts together and fully test our code we could.

B. Team Member Responsibilities

1) Angela

Angela was responsible for implementing the note scheduler that refines the raw frequency data from Marco's work by implementing algorithms that use heuristics to determine what keys to press at what time.

2) John

John was responsible for creating the web app user interface, integrating the audio processing and note scheduler within the web app back end, and designing/implementing the virtual piano interface.

3) Marco

Marco was responsible for creating the audio processing module, which involved implementing the Sliding Discrete Fourier Transform library, and a library of functions that interface with the web app backend.

C. Bill of Materials and Budget

After our shift to a virtual piano interface, our cost of materials and billing drastically changed. Originally, we had many physical parts, but now our only expense was AWS credits for hosting our piano interface. The bill of materials and costs are shown here:

Please refer to Table 1 for more information.

IX. Related Work

Player pianos have been around for many years, however only recently have they begun to be built using electromechanical devices. Edelweiss Pianos sells player pianos within the range of $20,000 USD. Mark Rober, a former NASA and Apple engineer, now Youtuber, posted a video introducing his self-talking piano which inspired this project.

X. Summary

Our goal is to create a self-talking piano. In order to achieve this, we'll record a user's voice via a web-app based user interface. Once we have this data, we will extract the frequencies that make up a user's voice across time. We will average these frequencies onto the frequencies that correspond to piano keys and output that data onto a tab separate file. A notes scheduler will parse this file and control a series of solenoids that can actuate the keys on a piano. The result should be series of notes being played on the piano that mimic the sound of a human voice.

Glossary of Acronyms

- Play rate, the rate at which the keys on the piano are being pressed.

- Phoneme, the atomic unit of speech, can be typically represented by a letter in English.

References

[1] World Leaders in Research-Based User Experience. "Response Time Limits: Article by Jakob Nielsen." Nielsen Norman Group, https://www.nngroup.com/articles/response-times-3-important-limits/.

[2]"Piano Key Frequencies." Wikipedia, Wikimedia Foundation, 29 Aug. 2022, https://en.wikipedia.org/wiki/Piano_key_frequencies.

[3]Haskins Laboratories. (n.d.). Alvin M. Liberman, 82, Speech and Reading Scientist. Retrieved December 19, 2011, from http://www.haskins.yale.edu/staff/amlmsk.html

[4]Peelle, Jonathan E., and Matthew H. Davis. "Neural Oscillations Carry Speech Rhythm through to Comprehension." Frontiers in Psychology, vol. 3, 2012, https://doi.org/10.3389/fpsyg.2012.00320.

[5] Tutorial | March 28, 2005 the Sliding DFT. https://www.comm.utoronto.ca/~dimitris/ece431/slidingdft.pdf.

[6] Creaktive. "Creaktive/Pianolizer: An Easy-to-Use Toolkit for Music Exploration and Visualization, an Audio Spectrum Analyzer Helping You Turn Sounds into Piano Notes." GitHub, https://github.com/creaktive/pianolizer.

Table 1

Item	Cost	Description
AWS Credits	$5.00	Credits used to host the AWS instance of our web app used for testing and attempting to get working by the final demo.