Musician's Scribe

Authors: Aditya Agarwal, Alejandro Ruiz, Kumar Darsh
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—Musician's Scribe is a free-to-use web application that efficiently transforms an audio recording provided by the user into simple, comprehensible sheet music which can be downloaded as a PDF.**

***Index Terms*—Audio, Beat, Clef, Noise, Note, Octave, Pitch, Scale, Signal, Transcription**

1 INTRODUCTION

Many musicians don't have the skills to efficiently chart their musical compositions. For example, bands that perform original music will want to share their pieces with bandmates, and teachers will want to share exercises with their students quickly and easily. We aim to build a web application that takes a monophonic audio recording input and transcribes it into easy-to-read sheet music. We thought this would be a great project to do because currently there are applications being offered with required subscriptions that transcribe audio files, however we could not find any free versions. Our software would be useful for people that want to transcribe simple, monophonic audios and do not necessarily want to spend money on a more complex transcriber system.

2 USE-CASE REQUIREMENTS

There are five requirements for our system to work - frequency reference, pitch accuracy, time signatures, tempo range, and transcribed note length accuracy.

We also need the input audio signal to be monophonic with a reference A4, which has a frequency of around 440 Hz. The instrument that plays this audio file should also always be a piano. The reason we have this requirement is that different instruments may have different onsets when being played. Onset is defined as the beginning of a note. For example, with a violin, it will have a different, more gradual onset in reaching a volume where the note can be identified, as compared to a piano which has a quicker onset when a note is being played. By focusing on just one instrument which we have easy access to, we aim to make the transcription as accurate as possible.

We are also targeting a 95% onset accuracy. This metric will track the rhythm of the piece and determine its rhythmic accuracy. Secondly, we will run our frequency processor on the played recording of the sheet music we generated on an input file to see if we accomplish 95% accuracy. How this will work is that we will compare our sheet music output to that of already established sheet music for the same audio, with the audio being a common tune such as Happy Birthday. We believe it is important for the user to hear the right pitch most of the time for it to be accurate, hence our high bar for success.

Another requirement is that our system will only allow a subset of time signatures. We will provide the following commonly-used time signatures: $\frac{3}{4}, \frac{4}{4}, \frac{2}{2}, \frac{3}{8}, \frac{6}{8}$.

The tempo range required for our system is 60 to 100 beats per minute. This is because these are standard tempos that we think an average person might use when playing, which is what our system is targeting. Obviously the time signatures and tempos offered by our system might not suit the need of all professional music players, but we aim to target audience that just want to transcribe simple, monophonic audio of not very high complexity, and can produce it quickly for them.

Finally, in terms of rhythm, we aim to ensure that every transcribed note is accurate within one-half a beat of it's actual length. We think this is a good requirement because our target use case is entry-level music, which usually doesn't involve notes shorter than a half beat.

3 ARCHITECTURE AND/OR PRINCIPLE OF OPERATION

Our whole music transcription system can be split into different smaller subsystems that when combined together perform the function of a music transcription system. The reason we decided to create smaller sub-systems and integrate them together later on, is because it allows us to have multiple people work on different systems concurrently. It is also easier to test a smaller system than to test a bigger system all at once, so our validation step can easily determine where the problems in our system are. We decided to split our system into 5 sub-systems. These include the user interface, pre-processing system, pitch processor system, rhythm processor system, and transcriber system.

First, we have the user interface. This is the first point of contact that the user will have with our system. They will connect to our web application by accessing our site through any web browser of their choice. Then, they will be able to upload an audio that is a .wav file of length less than or equal to 5 minutes, containing a piano monophonic melody. The user then should select the time signature and type of clef for their transcription. After all these inputs have been selected, the user will hit a "Submit" button which will send a HTTP request to the pre-processing system.

Once the pre-processing system receives the request, it will compute the Signal-To-Noise Ratio (SNR) of the au-

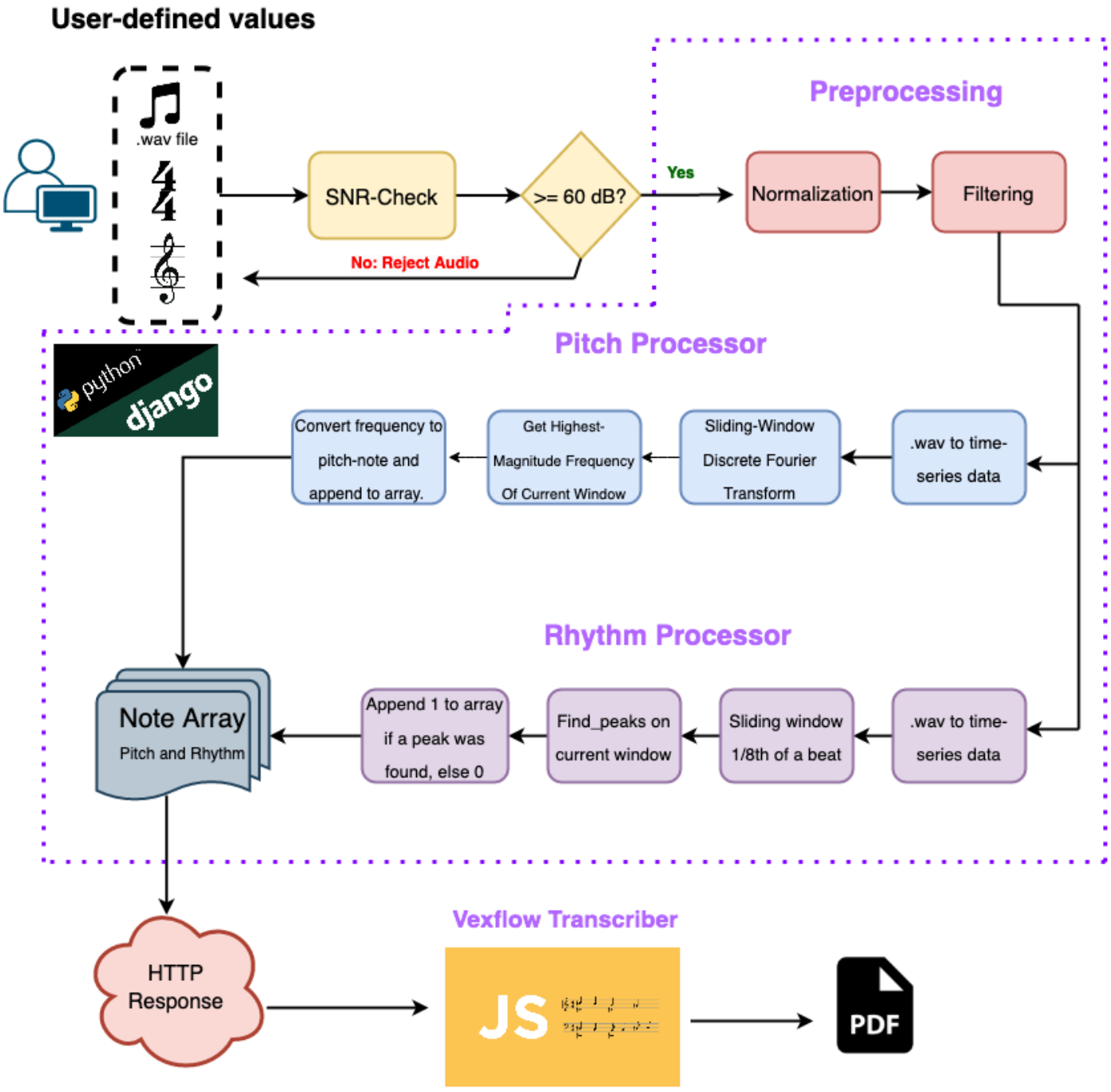


Figure 1: Overall Architecture

dio file. If the SNR is less than 60 decibels, this audio file will not be transcribed by our system and thus the pre-processing system will send a failure HTTP response back to the user. In case of a successful SNR, the audio will be normalized and filtered. The filtered audio will be sent for processing to the pitch and rhythm processor now.

The pitch processor will receive the filtered audio and will have to analyze the signal in the frequency domain in order to detect the adequate notes. We decided to use a Short-Time Fourier Transform, which iterates through our system at intervals of one-eight of a beat, applies a rectangular window to the signal, and applies a Discrete Fourier Transform to it. We find the maximum magnitude value of the Discrete Fourier transform and use the associated frequency value to determine the key played. This ensures harmonic frequencies of the note are not included, as they will have smaller magnitude than the frequency of the actual key played. Then we append to the detected key to an array, which is the output of our pitch processor.

The rhythm processor works similarly to the pitch processor because it also uses the sliding window processor, except it does not need to Fourier transform the signal. This processor will iterate through the signal and only look at smaller parts of the signal with length of $\frac{1}{8}$th of a beat. Then, it will use SciPy's find_peaks function to detect if there is a peak during that subset of the signal. If it detects a peak, that means there has been an onset and thus it will append a 1 to an array, otherwise if no peaks are found it will append a 0. The output of the rhythm processor should be a binary array indicating whether there is a new note being played at that time interval or not. This system with 1s and 0s will be used to determine the length of the note.

Finally, we have the transcription system which takes the output of the rhythm processor and the pitch processor to create a note array. Since we iterated from left to right of the signal when analyzing it using the same sliding window length when computing frequencies and whether an onset occurred or not, we know that the indexes of the rhythm and pitch processor output arrays align. Therefore, we can get the frequency of a note by looking at the pitch processor array and the length of that note by looking at whether a new note has been played or not from the pitch processor output. Once the note array is done, it will be put through Vexflow, a JavaScript library that allows us to make a music sheet PDF. We can give Vexflow the notes array, as well as the clef and the time signature specified by the user input from front-end. Vexflow should be able to generate the PDF and return it to the user.

4 DESIGN REQUIREMENTS

A. Signal-To-Noise Ratio

Standard music analysis and distribution systems, such as speakers, mp3 players, and turntables, have a minimum Signal-to-Noise ratio of 60 decibels. An audio with this SNR will be readily processed by our software, while more noisy signals will have too much variance to be accurately represented in our data structure. To ensure the input audios have the necessary SNR, we will be using an algorithm we found from SciPy pre-processing library.

B. Pitch Accuracy

At least 95% of the notes detected in the audio must be accurate in pitch, with accuracy defined as the frequency measured at any arbitrary time t will corresponding to the note played in the input signal. The remaining five percent will be at most 2-3 notes given the time limits on the audio length, and such errors do not heavily detract from the purpose of a basic composition. We want the song to sound almost equivalent to the original audio and we believe that near-perfect accuracy in terms of frequency is required for this to be satisfied.

C. Length-Note Duration

We require our design to measure each note and rest's duration accurately to the nearest one-eighth of a beat. For example, a quarter-beat measured as three-eights of a beat is acceptable, but a quarter-beat measured as a half-beat is unacceptable. This is because standard musical exercises and basic compositions will not require notes of any shorter duration of a sixteenth note. We assign this requirement this way so that it can be applied to audios of various tempos and time signatures, which wouldn't be the case if we measured accuracy in, say, milliseconds. We target a 95% rate of accuracy in regards to detecting note onsets and rests.

D. Front-End User Interface

The final technical requirement is that the user should be able to easily interact with this application through the front-facing web app. This means a user can easily upload their recording, select basic requirements, and receive a free-to-download PDF of the returned sheet music. The application will be entirely accessible via the web, not requiring any hardware or software from the user's side of things besides their own method of recording themselves. The intent is for the application to be accessible to those of limited resources, as well as those with little experience in the music industry. Anyone can record themselves playing a piano and upload it to the web, and anyone can download and share a PDF.

5 DESIGN TRADE STUDIES

5.1 Short-time Fourier Transform

To process our audio signal in our pitch and rhythm processors, we decided to use a Short-Time Fourier transform, which applies a sliding window to the time-domain audio signal and performs a Discrete Fourier Transform on each section. This allows for our spectral data to be divided into time indexes, which is necessary for determining which key is played at what point in the audio.

We considered using Hann, triangular, and Gaussian windows, but as our pulses are transient waves this resulted in a lot of attenuation at important segments of the signal.

We settled on rectangular windows, which have zero attenuation of the signal. This window design also removes the need for overlapping the segments, because there is no attenuation to account for.

Currently, we have decided on using a window size of $\frac{1}{8}$th of a beat. We decided on this value because we want to detect notes to the nearest $\frac{1}{8}$th of a beat. This means that for the rhythm processor, it would split the signal into smaller signals of length of $\frac{1}{8}$th of a beat and any pulses of shorter length will be overlooked. This is an acceptable margin of error because most basic musical pieces will not involve any notes of shorter length.

The size of the window will be dependent on the tempo of the given audio piece. Based on the user's desired beats per minute, we can determine how long a single beat is and design a window with a length of one-eighth of a beat. To get the best window-size value we plan on testing different window sizes and seeing what overall note accuracy we get. We plan on plotting figures representing on the x-axis the window size and on the y-axis the rhythm and pitch accuracy. This way we can observe which window sizes might work best.

5.2 Signal-To-Noise Ratio of Input Audio File

Our system will need to be able to process signals with decent audio quality. We do not expect it to be able to handle extremely noisy signals where the signal quality present is very low. Therefore, we decided to impose a restriction on the minimum signal-to-noise ratio required for audio to have for our system to accept and process it.

We realize a trade-off in this is that this might be frustrating to some users whose microphones do not offer a good enough quality for them to upload their audio to be transcribed by our system. Therefore, ideally, we would want to accept the lowest possible acceptable SNR audio files.

We did some research and found out that the lowest SNR that can be handled by a phono-turntable is of around 60 decibels and around 90 decibels for an amplifier or a CD player. Therefore, we will be trying to accept audio files with an SNR closer to the 60 decibels value.

We plan on testing this by inputting audio files with no noise and increasingly introducing noise. This way, we will see how much worse our music sheets generated get when more noise is being introduced. This will help us determine what might be the best SNR for us to accept.

One way we plan on selecting the best SNR, is by plotting on the x-axis the SNR and on the y-axis the pitch and rhythm accuracy that our system had. The accuracy for both of rhythm and pitch processor should be greater than 95% , therefore this plotting should allow us to visualize where the best SNR value lies at.

5.3 Preprocessing of the Signal

We first attempted to apply a bandpass filter, but we found that the windowing techniques used by most filter design systems resulted in attenuation and distortion of the initial signal. Instead, we simply analyzed the frequency-domain of the signal only within the standard range of musical notes, 16 Hz to 8000 Hz. Any pitches detected outside this range are ignored.

We also determined how we needed to remove extraneous samples from our signal, so only the user's performance is relevant in the system. We normalized the data based on it's maximum magnitude, then applied a threshold to all values; any samples of magnitude smaller than the threshold were assigned a value of 0, meaning the first non-zero sample would be the user's first played note. We initially attempted a very small threshold on the scale of 10^{-3} to minimize loss of data within the relevant portions, but this proved ineffective as very few samples had this small magnitude. We settled on a threshold equal to one-tenth the maximum magnitude of the signal; the trade-off was that each pulse was slightly truncated at it's end, just before the next pulse began. We considered this an acceptable loss because the samples' small magnitude means they have little impact on the spectral analysis compared to the samples that passed the threshold.

5.4 Finding onsets in the input signal

We decided to find the onsets in our signal by looking at intervals of length of our window size and appending a 1 if an onset is found in that interval otherwise append a 0 if no onset is found to our array. This will return an array of length $\frac{\text{Length of Signal}}{\text{Window Size}}$ which will match the length of the array returned by our pitch processor. We apply this sliding window approach on both the pitch and rhythm processors so that the elements in the array at a specific index will match to the same time interval at that index for the pitch processor. If we look at the values located at, for example, index 2 in both the output of rhythm processor and pitch processor we will know whether the note at that same time interval is being held or a new one is being played and what pitch it has. It allows us to have the rhythm and pitch processors output aligned in time, and greatly simplifying the integration process.

The other alternative we had was to try to find peaks across the entire signal at once instead of using the sliding window approach. We realized this approach might lead to the rhythm processor running a little bit faster, for example for a 10 seconds audio file it was about 0.000355 seconds faster. However, the issue is that if we analyze the signal entirely at once SciPy would return to us an array containing all the time locations of the onsets that were found. We would now need to proceed to analyze each time interval of length of the window size and determine if a peak is found there, so it would not really make sense to do it this way. Therefore, we think it is better to just call our find_peaks function while analyzing the signal at a specific

window so that we only have to iterate through the signal once and get an output that is easy to integrate with our rhythm processor. We also think that the difference in time of 0.000355 seconds is not significant enough, since at the end of the day this function call is local and not something like a RPC function call where it might be more concerning in terms of time.

6 SYSTEM IMPLEMENTATION

6.1 Preprocessing

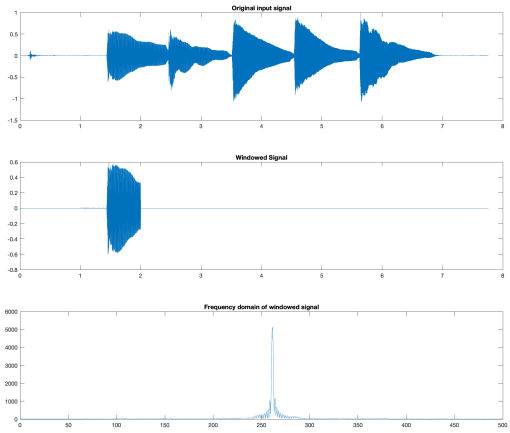
After the initial audio signal is read by the application, it is represented by an N-array in the time domain. In order to allow for varying input volume, quality, frequency range, etc. we first normalize the data, dividing each sample by the size of the largest sample in the array; the result has a magnitude range of 0 to 1. We can then apply a volume threshold of 0.1; any samples that don't meet the threshold are set to 0 and are considered rests by the system. This is the stage where the backend records the user input of clef, time signature,and tempo. If the user does not include the tempo, it is automatically detected by the backend using the library Librosa, a powerful library for audio and music processing in Python.

6.2 Pitch Processor

The Pitch Processor analyzes the audio in the frequency domain to determine which notes to transcribe. We implement it by applying the Sliding Discrete Fourier Transform algorithm. The result is a KxN array where K is $\lfloor \frac{N/8}{f_s} \rfloor$ and row k represents the N-point Discrete Fourier Transform of each segment of the audio.

$$\sum_{n=0}^{n=N} \Pi(\frac{n-k}{f_s}) x(n) e^{\frac{j2\pi kn}{N}} \quad (1)$$

From this we generate our K-point array of notes, where $K(k)$ is the index of the maximum value of the kth row of the DFT array.



Pitch Processing of Input Signal

We then iterate through the array and determine the frequency at which $K(k)$ has it's maximum value. This frequency is the detected pitch of the note, f. We then calculate the note's distance in semitones from the reference note, $f_A = 440$ Hz:

$$n = 12 \log_2(\frac{f}{f_A}) \quad (2)$$

The number of semitones determines the note played; for example, n=2 semitones corresponds to a B note.

6.3 Rhythm Processor

The Rhythm Processor analyzes the audio in the time domain to determine whether an onset has occurred at a specified time interval. It is implemented by using a sliding window approach. Our sliding window will be of length of $\frac{1}{8}$th of a beat. The algorithm will look at the part of the signal that lies in the sliding window and call the function *nd_peaks* from SciPy to try to detect any onsets. The function *nd_peaks* finds peaks inside a signal, with peaks defined as samples that are larger than their surrounding samples, having a minimum distance of one-eight beat between each other and a non-zero magnitude. The properties that we set are a distance of $\frac{1}{8}$th between peaks detected. If the *nd_peaks* function finds a peak at the current window time interval it will append a 1 to our rhythm processor output list, or a 0 if it did not find a peak. After the algorithm is done applying the sliding window throughout the whole signal we should have a binary array indicating whether onsets where found at a specific time interval, where every time interval is of length $\frac{1}{8}$th of a beat. This will be the binary array returned by our processor.

6.4 Integration System

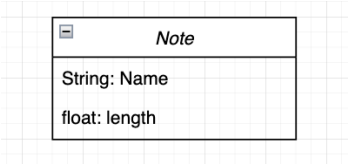
This system will combine the output of the rhythm and the pitch processors to form an array of notes.

Below is an example of the outputs of the two subprocessors based on a short input signal.

output of pitch processor noteList = ['A', 'A', 'B', 'B', 'C']; output of rhythm processor beatList = [1, 0, 1, 1, 1];

The *noteList* array represents the notes played at each one-eight beat of the audio signal, and the *beatList* array represents whether or not a new pulse at each one-eigth beat. In the above example, we have an A note held for two-eights of a beat, then two consecutive B's held for 1/8 of a beat.

The integration step outputs a list of Note objects, where each note has a corresponding pitch and length field. The Notes were originally implemented as a Python Object class, but we found that this data type was difficult to read in the front-end, so we decided to implement it as a Python dictionary with the class fields being replaced by key-value pairs. Below is the design structure of the Note class.



6.5 Vexflow Formatting

This sub-section of our design was added late in the process when we found an issue with sending our data from back-end to front-end. Our back-end determines the duration of notes in terms of seconds, but the Vexflow library used in the front-end writes notes in terms of the duration in beats. This requires an intermediate step between integration and transcription, which converts the durations to beats.

It does this based on the tempo of the audio, which is either manually chosen by the user or auto-detected in the pre-processing section. The tempo defines how many beats are in one minute of audio, so the inverse of the tempo represents the duration of a single beat. We iterate through the output of the Integration System and divide the duration field of each Note by the tempo. This value is rounded to the nearest one-half beat because our target is to achieve rhythm accuracy to the nearest one-half beat. If this rounded duration was 0, the Note was removed from the output list entirely, as it would not need to be transcribed.

Vexflow also requires us to determine how the notes will be divided into bars. After we determined the duration of each note in beats, we determined which bar they would be assigned to. The number of beats per measure is part of the time signature selected by the user; for example, a $\frac{3}{4}$ time signature means each bar will be three beats long. Instead of sending the front-end a list of note objects like before, we now send a dictionary where each value represents a bar of music. This simplifies the transcription process by essentially creating a series of smaller equally-sized datasets instead of requiring the front-end to process one very large dataset of indeterminate size.

6.6 Web App Interface

We have a Web App for the user to access our system. We plan on making it accessible through laptops. The web app will be built using Django, with React.js as the framework for the front-end. These frameworks heavily facilitate the transfer of data between the front and back end, allowing for repetition and variety on the part of the user. We will use the React.js tools to provide HTML forms for the user to upload an audio file and select the previously stated parameters of the song. Then, the front-end will send a HTTP Request to our backend system. The request will be in the format of a Django Form object, containing the audio file and the user-selected parameters. The Form calls the functions that implement the Preprocessor, Pitch Processor, and Rhythm Processor. Each Form sent by the user will correspond to a new instance of the Note Model. The response to the form is then passed to the Vexflow library.

6.7 Vexflow

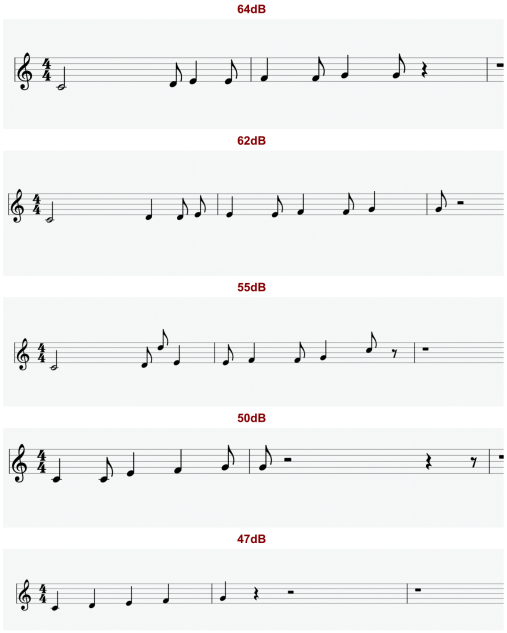
Vexflow is a JavaScript library that parses the Response from the back-end of the application, which contains a list of Note models, and generates the PDF transcription of our notes. It is implemented entirely within the front-end of the code.

Vexflow iterates through the output of the Vexflow Formatting system and draws each bar in the piece; first the musical staff of the bar, then each note in the order it reads them. The Note data structure is designed such that each field can be copied over to the Vexflow API's Note method without modification. These fields are the key, duration, and type.

7 TEST & VALIDATION

7.1 Results for Signal-to-Noise-Ratio Filter

Our system will reject audios with an SNR that is less than or equal to 60dB. We decided on this value by looking at the generated music-sheets with different SNRs. We modified the value of the SNR by adding different amounts of White-Gaussian noise to the signal to increase the amount of noise. After looking at the transcripts, the system seemed to be innacurately detect notes when the SNR was less than 60dB.



SNR Comparison

7.2 Results for Rhythm Processor

The most important value to decide on for the rhythm processor was the height of a peak in a signal. This height

represents the value at which a sound starts to be considered a note, instead of a rest. All sounds with an amplitude that is less than this height are considered a rest, otherwise they are considered a note. Before processing the signal through the rhythm processor, it is put through a normalizer which normalizes its values from the range 0 to 1. It seems like most sounds tend to get relatively low and fall to a rest quickly when they reach a value of 0.1 or less. We also tested trying to increase the height value to be 0.2, 0.3, etc and it seemed to be the most accurate when set to 0.1 or less. This was striking to us since we expected the height value to be somewhere around the 0.2-0.3 range when we had not tested it.

To test the rhythm processor as whole we decided to count the number of notes and rests that it was producing. For example, given the array [1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1] we would have at least 3 notes and two rests. One note is seen as a sequence of 1s, while a rest is a sequence of 0s. We can determine the rhythm accuracy by counting the number of notes detected in our system and the number of rests. Therefore we can divide the number of detected notes over the number of expected notes, and do the same thing for rests. Finally, we can average the value out to get the accuracy. If the number of detected notes or detected rests is greater than the expected amount, then we would just do the inverse division.

Onset Accuracy

7.3 Results for Pitch Processor

Below are the results for a variety of user input audios. The Pitch Accuracy was detected by inspecting the keys detected in the output of the pitch processor and comparing them to the keys we knew were in the input. If a key was skipped over or measured inaccurately, the score is decreased.

Pitch Accuracy

Observe that the C-scale where the user manually selected a tempo and used a metronome to ensure that tempo was met, resulted in 100% accuracy. This is the ideal input, as each note is of equal duration and the slow speed means errors in the windowing process are less notable compared to the overall size of the audio.

Major errors come up when dealing with inputs of quick, varying tempos. The Ascending C-scale and Descending C-scale inputs started at a slow tempo and sped up as time went on. The system defaults to the slowest tempo detected, meaning these faster notes took up less than one-half a beat and were discarded in the Vexflow Formatting subsystem.

The Happy Birthday input receives 100% accuracy, demonstrating that longer audio lengths don't affect the pitch detection sub-system, and that it can handle pieces where the rhythm varies between notes shorter and longer than a single beat.

8 PROJECT MANAGEMENT

8.1 Schedule

Our project was able to stay fairly on schedule throughout the course of the Spring 2023 semester. We began by working on the signal processing of the rhythm and pitch processors in Matlab while front-end development was underway. The project then moved into converting the Matlab progress in Python, and tweaking the functionality and accuracy of the pitch processors. Finally, sending the information to the front-end and displaying it visually was completed, followed by testing.

8.2 Team Member Responsibilities

Alejandro was the primary team member working on the rhythm processor, while Aditya was the primary team member for the pitch processor. Kumar was the point person for the front-end Django web application. Once this was completed, all three team members worked on converting the backend data into a displayable format through

the Javascript library Vexflow as this was a harder than anticipated task. All team members were involved in recording and testing various audio files. Written documentation and presentations were distributed evenly throughout the team. All team members were involved in debugging of the rhythm and frequency processors as well as any frontend issues.

8.3 Bill of Materials and Budget

No materials were used or purchased during the course of this project besides a piano owned by one of the team members.

8.4 AWS Usage

No AWS credits were requested as our project runs on a private account of one of the team members using free credits for a T2 micro EC2 instance.

8.5 Risk Management

Our primary risk was that we wouldn't know enough about the audio signal to accurately analyze it. Our solution was to assign key values that were a part of our detection algorithm, such as the size of the sliding windows of analysis, to be based on the tempo of the piece. This solved the problem of our algorithm potentially only working well on pieces in the middle of the ideal tempo range of 60-100 bpm, and can now handle tempos across this realistic spectrum.

In regards to scheduling througout the semester, we mitigated falling behind by starting work on the rhythm processor, pitch processor, and front end web app immediately, and split this between each of the three group members. In our schedule we accounted for a couple weeks for integration and debugging for issues such as incorrect file format, and more. Besides this we did not face any major risks or concerns in regards to budget and personnel.

9 ETHICAL ISSUES

While no one is particularly vulnerable to failure of this project,a person with less music knowledge is more reliant on the product's accuracy while others can cross verify anything that seems wrong.

Public health and safety are not at risk for our project. Public welfare is the only aspect that is under consideration in the context of our project. This is due to data privacy. With millions of individuals putting out content and uploading it to the Internet, protecting intellectual property is a concern. This is a potential issue for our project if we were to go commercial, as musicians would need assurances that their uploaded audio file and its corresponding transcription would not be stolen or misused. This would hypothetically involve encryption from the developer's end to ensure nothing is directly visible to those on the backend. However, given the scope and time frame of this project, and that we are not deploying it into production anywhere, we will not be able to and are not planning to address these concerns.

Economic and social factors were under consideration in our project, as the products already out there in the market for music transcription are behind a heavy paywall. This is why our product is free to use. Socially, those who have extensive resources of time and money often find it easier to learn a new instrument, and we aim to reduce this gap by making the process of generating sheet music free, quick, and non-stressful. Those who have the time and money for music lessons, or attend a school with a well-funded music program are the only ones who will know how to write accurate sheet music quickly. Therefore, we believe our project, ethically, has no concerns in its current scope and is helping aspiring musicians by reducing their barriers to access to become better skilled.

10 RELATED WORK

There are several tools online that do a similar job of converting audio files into sheet music. Some are listed below. It is important to note that these products are not available for free, and are behind expensive pay walls for subscriptions or expensive one time purchases.

AnthemScore is probably the most commonly known software to transcribe audio files to sheet music. It also allows the user to edit the sheet music generated and export it in various formats. Another tool is Transcribe!, which helps a user transcribe their own sheet music by slowing down the audio, altering the pitch, and having a spectogram that helps the user visualize the piece. It is a tool to assist musicians in transcribing. ScoreCloud uses machine learning to transcribe, edit, and arrange various music files. Musitek Smartscore is the final well known product, as it uses Optical Character Recognition technology to generate sheet music which can be edited. None of these products are very successful and accurate, and are behind immense pay walls, causing our project to have a use case and target audience of helping those who cannot afford the time and money for such expensive tools.

11 SUMMARY

To summarize, our project is relatively successful given the use case and design requirements outlined at the beginning of the project. Musician's Scribe is able to take in audio files of high and medium sound quality, and generate sheet music which can be downloaded for the user. As it is still only roughly 75% accurate in rhythm and pitch accuracy on some pieces, further fine tuning will be required to meet the high standards of success we have self-defined and anticipate musicians would also have.

11.1 Future work

The system still requires optimization in order to meet the testing metrics for all audio files. Our first step will be to continue fine-tuning certain elements of the program (Window size, Fourier transform parameters, onset thresholds, etc.) until we achieve our desired metrics.

After this, the project can be expanded upon greatly by adding user options and improving the specifications. We are currently constrained to detecting notes of 1/2-beat duration or more, but the same techniques can be applied in order to achieve smaller orders, such as 1/4- or 1/8-beat. This allows for more complex musical pieces to be transcribed.

To make the system usable by a broader range of instruments, such as piano and guitar, we can include options for polyphonic audio.

If this project were to be put into production and launched, we would have to address privacy concerns. As discussed in the ethics section, it is important that uploaded files cannot be stolen from the backend, meaning encryption would be necessary to protect intellectual property rights.

11.2 Lessons Learned

Over the course of the project, our group learned many lessons. As outlined in our design trade studies, we learned a lot about the merits and applications of using Short Time Fourier Transforms, Discrete Fourier Transforms, and using various sliding windows. In addition this, we had to figure out how to use SciPy and Librosa as libraries which helped us in our signal processing and tempo detection. We learned the hard way that not all online tools and libraries have great documentation, as the Javascript library Vexflow didn't have complete end-to-end documentation, and we had to alternatives, so it took longer than anticipated. Lastly, in our testing, we realized that musicians weren't sure how to rank and score our system on a holistic 100 scale, so we altered our testing have only quantitative results for onset and note detection. It has been an extremely enriching experience.

Glossary of Acronyms

Any acronyms used are defined in-line.

References

[1] Francois, https://blog.son-video.com/en/2021/03/what-is-the-signal-to-noise-ratio -can-we-improve-it-and-if-so-how/: :text=It

[2] SciPy Documentation `https://docs.scipy.org/doc/scipy/index.html`

[3] Vexflow JavaScript Music Notation and Guitar Tablature : By Mohit Muthanna Cheppudira `https://www.vexflow.com/`

[4] Librosa 0.10.0 Documentation `https://librosa.org/doc/latest/index.html`

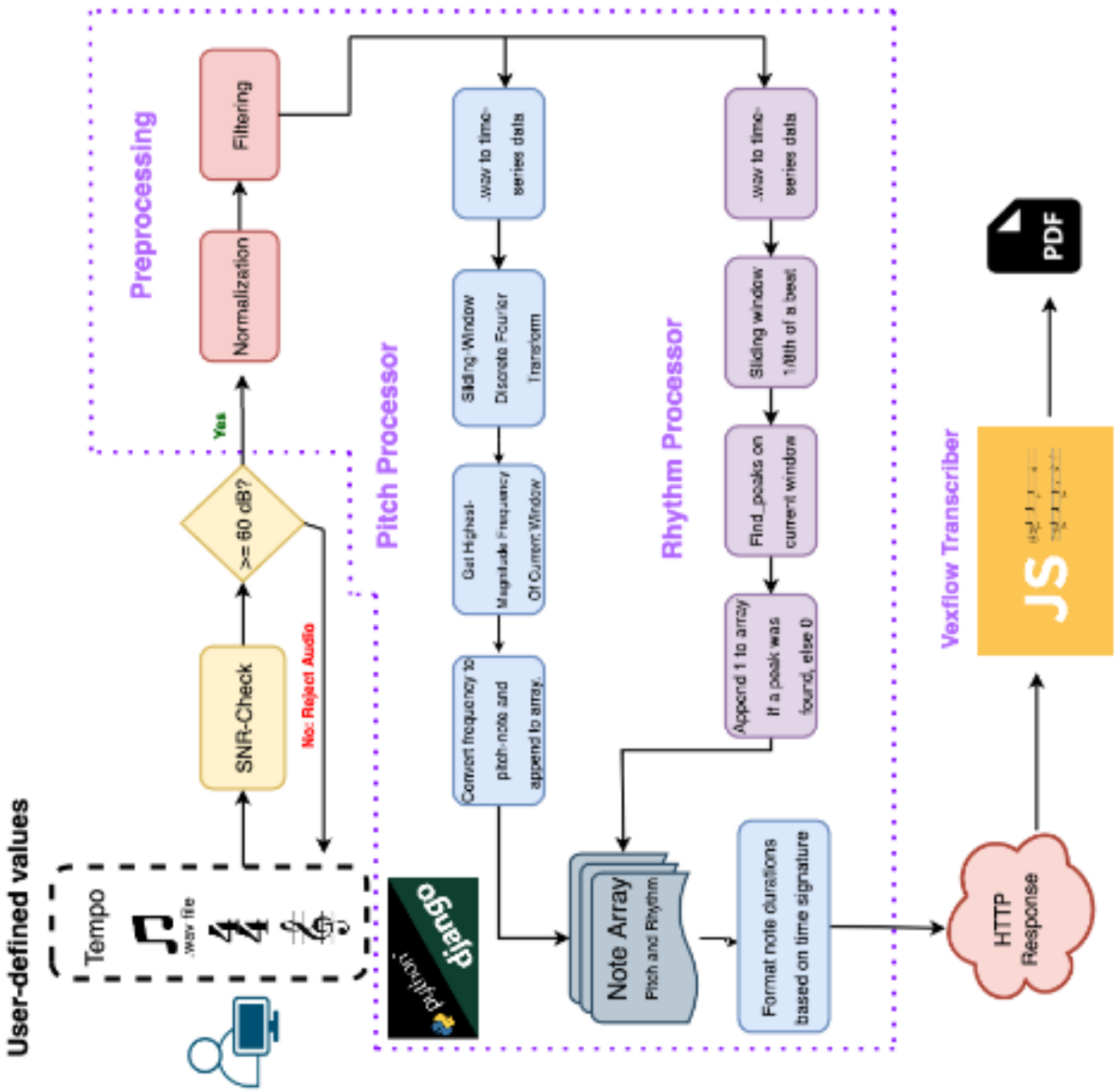


Figure 2: A full-page version of the same system block diagram as depicted earlier.

Figure 3: Rhythm Testing Results

Figure 4: Pitch Testing Results