Musician's Scribe

Authors: Aditya Agarwal, Kumar Darsh, Alejandro Ruiz
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 in a monophonic audio recording's signal as an input and transcribes the sound into easy-to-read sheet music for artists. We thought this would be a good 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 in 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 an relatively more instant onset when a note is being played. Therefore, 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 90% pitch accuracy based on user feedback. This will work in two parts which will be explained in detail later on. We will have experienced musicians determine whether they believe the sheet music was over 90% accurate as compared to the input audio file. 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 show to some people with experience in the music field, an audio file and the music sheet generated by our transcriber. Then, the users will give our system a rating out of 100, based on how good they think the pitch accuracy of our transcriber was. We believe it is important for the user to hear the right pitch most of the time for it to be accurate.

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 audios 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 $\frac{1}{8}$th of a beat. We think this is a good requirement because we are not trying to cover very short-duration notes. The least duration of a note we are trying to cover is $\frac{1}{8}$th of a beat, therefore every note should be accurate within $\frac{1}{8}$ th of a note.

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 allowed 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, therefore by splitting our system into sub-systems we can thoroughly test each subsystem in an easier way and therefore our whole system will be thoroughly tested, and more accurate. 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

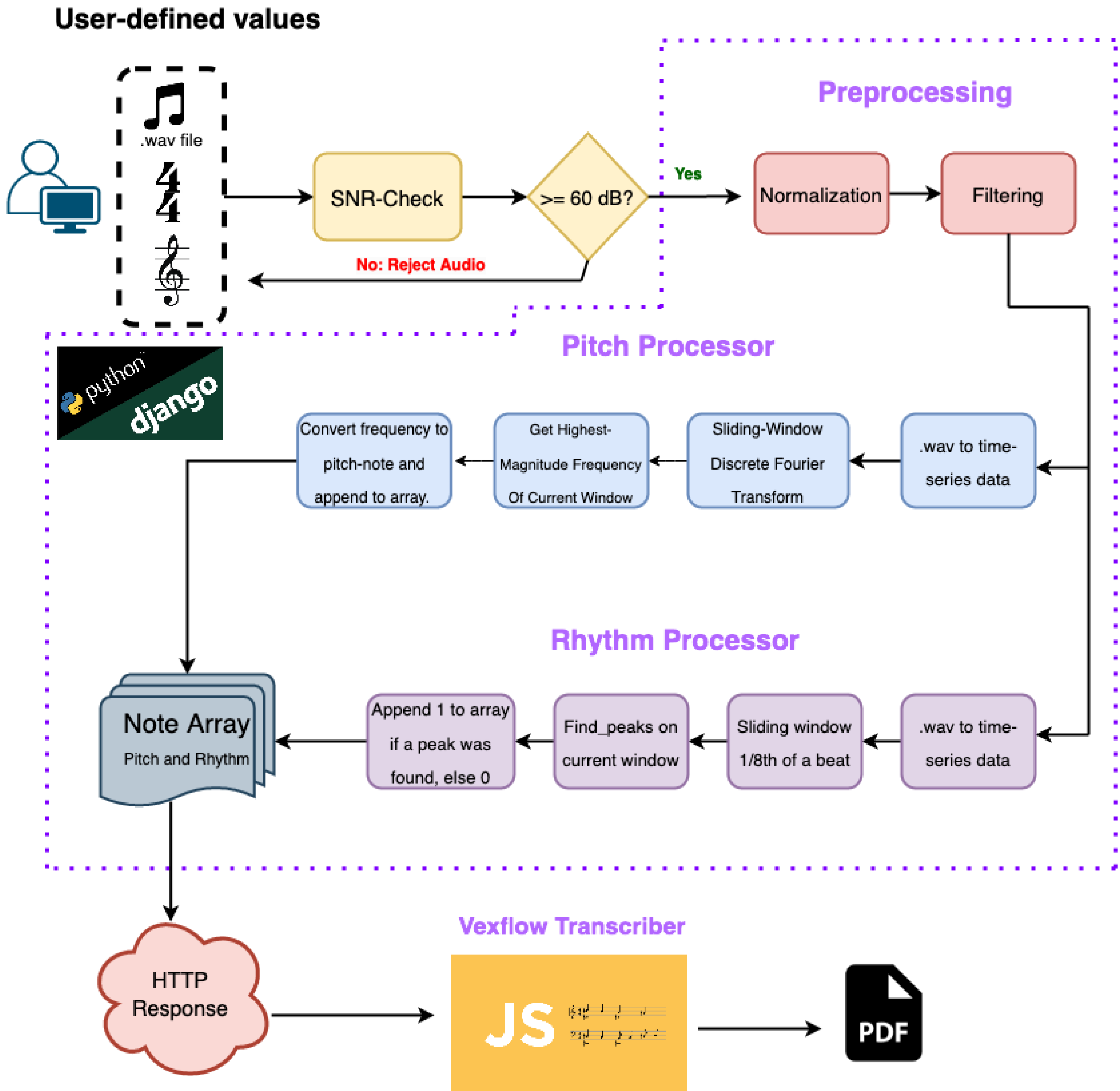


Figure 1: Overall Architecture

be able to upload an audio that is a .wav or .mp3 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 audio 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 t use a Short-Time Fourier Transform, which will transform in a sliding method our signal in intervals of length of $\frac{1}{8}$th of a beat. Each time we Fourier transform the signal, we can compute the average frequency over this interval and match it to a note whose frequency range contains the average frequency we just computed. Then we append to the note to an array. The pitch processor will output this array of notes.

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 with parameter width set to $\frac{1}{8}$th of a beat too, 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.

```
def signaltonoise(a, axis=0, ddof=0):
    a = np.asanyarray(a)
    m = a.mean(axis)
    sd = a.std(axis=axis, ddof=ddof)
    return np.where(sd == 0, 0, m/sd)
```

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.

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 Sliding Window of Discrete Fourier Transform

To process our audio signal in our pitch and rhythm processors, we decided to use a sliding window approach. This way, we can analyze the signal from beginning to end using this sliding window algorithm and calculate the average pitch and whether a new onset has occurred during that time frame. This way we can know what type of note is being played during a certain instant in terms of length and pitch. The challenge is to select the most optimal window to process the signal.

Initially, we were going to use the Short-Time Fourier Transform method. However, we decided to use a sliding Discrete Fourier Transform instead because it allowed for more control and clarity of the window and overlap sizes. The STFT would be more applicable if we needed to analyze an entire signal at once, instead of bit-by-bit.

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 90% , 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

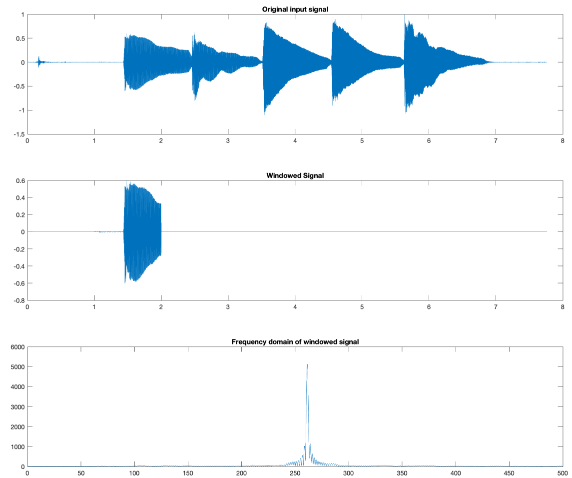
After the initial audio signal is read by the application, it is represented by an N-array in the time domain. In order to allow for varying input volume, quality, frequency range, etc. we first normalize the data, dividing each sample by the size of the largest sample in the array; the result has a 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.

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.

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

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.



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 *find_peaks* from SciPy to try to detect any onsets. The function *find_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 *find_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

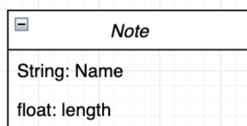
This system will combine the output of the rhythm and the pitch processors to form an array of notes.

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 are implemented as a Django Model, which will be relevant in the Front-end subsystem. Below is the design structure of the Note class.



6.5 WebApp Interface

We will need a WebApp for the user to access our system. We plan on making it accessible through laptops. The webapp 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.

6.6 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.

The following code shows how we would transcribe the notes detected in the same audio signal from Section 6.4.

```
1 system.addStave({
2   voices: [
3     // Top voice has 4 quarter notes with
        stems up.
4     score.voice(score.notes('A/4, B/8, B/8, C
        /16', { stem: 'up' })),
5   ]
6 }).addClef('treble').addTimeSignature('4/4');
```

7 TEST & VALIDATION

We will be testing starting simple and then trying to get more and more complex. We will test based on the length of the notes, and frequency of the notes. We need the accuracy of our pitch processor to be >= 90%. We can test this by entering simple songs into the pitch processor system and calculating the number of correct notes that our frequency processor got.

We use a third-party app to automatically play our transcription and generate an audio signal, which will be compare to the user-input signal.

7.1 Tests for Pitch Processor

For testing the pitch accuracy, we apply our sliding DFT algorithm to both the test and reference signals. At each segment, we compare the notes detected. If the algorithm detects the same note in both signals, it is a success. If 90% of the intervals register a success, the signal passes the test.

7.2 Tests for Rhythm Processor

Similar to the test for the Pitch Processor, we analyze both signals at sliding intervals and apply the find_peaks method to each signal. At each interval we ensure that the peaks detected in one signal match those detected in the other. In this test, the window is of the size 1/4 of a beat, as we aim for accuracy to the nearest 1/8 of a beat, meaning an error of any magnitude less than or equal to one-quarter beat is permissible. For example, if a segment has two 1/8-beat notes but only one 1/8-beat notes are detected, the test passes, but if three 1/8-beat notes are detected, the test fails.

8 PROJECT MANAGEMENT

8.1 Schedule

Our project's hardest part is the developing an accurate rhythm and pitch processor. Therefore, we put the main focus of our attention during the first 4 weeks into researching and starting to implement our frequency and rhythm processors, as well as developing plans for testing our systems. A large part of this is fine-tuning the parameters, such as the normalization steps, window sizes, and integration process. In the coming weeks, we will focus on building the web app as well as integrating the rhythm and frequency processors.

So far, Aditya has focused on developing the pitch processor, Alejandro has focused more on developing the rhythm processor, and Kumar has focused more on developing the front-end, designed the user-interface and researching the React and Django components required to build it. Alejandro and Aditya will next work on collaborating together to integrate the rhythm and pitch processor. Kumar will begin the implementation of the front-end. Our

goal is that after 2-3 weeks we will be able to have a working project, and start focusing on testing and optimizing our system. The full schedule is shown in Fig. 2 on page 8.

8.2 Team Member Responsibilities

We will all be working collaboratively on the project, however each of us will focus more of our time on a subsystem of the project. Aditya will be focusing more on developing the pitch processor, Alejandro will be focusing more on developing the rhythm processor and Kumar will focus more on developing the web-app. Then, Alejandro and Aditya will work together on integrating both processors so that an array of notes can be developed that is sent to the front-end built by Kumar, which will take the array of notes and generate a music sheet using Vexflow and return it in PDF form to the user.

8.3 Bill of Materials and Budget

Given that our project is entirely software and signal based, our bill of materials is none. All of the required libraries which would need to be installed and any associated technologies and programming languages will be outlined in our .README file which will be available to download.

8.4 Risk Mitigation Plans

A primary risk is that we won't know enough about the audio signal to accurately analyze it. For example, our window size may work much better for a slower piece than for a faster piece. We account for this by assigning the key values of our detection algorithm such as window sizes, based on the tempo of the piece.

9 RELATED WORK

There are multiple music sheet generators out there, that are more complex than ours and can handle polyphonic audios of different types of instruments. These tend to be applications that require a monthly or yearly subscription and might be targeted for people that are professionals in the field of music. Our application addressed a different demographic, those focused on instruction, amateur players, or new students.

10 SUMMARY

Our goal is to create a freely accessible online music sheet generator. The user will record themselves performing a simple song that they wish to share with others, submit the recording, and receive a transcription of their performance for both personal use and distribution to similarly interested musicians.

Our application will focus on accuracy regarding the notes and rhythm of the piece, as these are the core aspects of early musical instruction. This means the primary challenges are determining a near-perfect accurate analysis of pitch and rhythm from audios that can vary greatly in length, tempo, musical range, etc. In order for the project to be both applicable for all potential users, the application works with the users' chosen parameters and applies a one-size-fits-all method to determine how to get the most accurate transcription.

Glossary of Acronyms

Any acronyms used are defined in-line.

References

[1] Francois, What is the signal-to-noise ratio? Can we improve it and if so, how?

[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/`

TASK NUMBER	TASK TITLE
	2 or more - Gold
KEY	Aditya -- Kumar -- Alejandro
1	**Signal Processing**
1.1	Data structure w/ audio & transforms
1.2	Data structure for Fourier Transform
1.4	Frequency processor research/design in MATLAB
	Integrate Frequency processor in Python back-end
1.5	Rhythm processor research/design in MATLAB
1.6	Integrate rhythm processor in Python back-end
1.7	Integrate Freq and RhythTm processors
2	**Front-end Web App w/ React**
2.1	Design interface for pre-selections
2.2	Upload input for specific file type
2.3	Option to download backend's ouput
3	**Software Design Tasks**
3.1	Design node data structure in Django
3.2	Implement request format between vexflow and python code
3.3	Determine how vexflow parses requests in Django
3.4	Implement vexflow interface using request data - generate final pdf
4	**Project Performance & Deployment**
4.1	Freq processor accuracy manual testing
4.2	Rhythm processor accuracy manual testing
4.3	Optimization Based on User Feedback
4.4	Deploy app into prod. env. w/ Flask
5	**User Testing**
5.1	Write form for user feedback
5.2	Distribute feedback form to testers

Spring Semester 2023: 2/6, 2/13, 2/20, 2/27, 3/6, 3/13, 3/20, 3/27, 4/3, 4/10, 4/17, 4/24, 5/1 (M T W R F)

Figure 2: Gantt Chart