Automatic Page-Turner for Musicians

Vanessa Yan*
vanessa.yan@yale.edu
Computer Science, Yale College
New Haven, CT

Qinying Sun*
qinying.sun@yale.edu
Computer Science, Yale College
New Haven, CT

Sally Ma*
sally.ma@yale.edu
Computer Science, Yale College
New Haven, CT

ABSTRACT

In this final project for Yale CPSC 459/559, we build an interactive end-to-end system that automatically turns the page for musicians in real-time. The user is able to upload multiple pages of music scores to a web app, after which the app parses the notes in the music scores and detects the pitches and duration of each note at the end of each music page. At the click of a button, the system begins to listen to the musician play, matching the visual and audio information to determine whether the musicians has arrived at the end of a page, at which point the web app displays the next page of the music score for the user.

KEYWORDS

interactive, real-time, music recognition, cqt

1 INTRODUCTION

A common problem faced by musicians around the world is flipping pages of the music scores. Since almost all musical instruments engage musicians' hands at a high level, it is inconvenient for them to flip the sheet music by hand as they play, especially when the music is complex and fast. In response to this problem, some musicians choose to memorize the music to eliminate their need for page turning. However, this approach is typically not realistic for orchestral musicians, who need to become performance-ready with long and difficult music in a short time span. Additionally, to practice the music to the point of being able to memorize it still requires the musicians to deal with the inconvenience of page flipping during their practices. Other musicians, typically soloists or piano accompanists, choose to have a human page turner. However, human page turners are not always available, and even if they are, they are typically only available for the actual performances rather than the many hours of practices musicians do. An existing alternative for musicians is a foot petal, also known as a foot clicker, which can be connected with a tablet such that pressing down on the foot petal flips a page. However, this alternative requires an extra device to purchase and carry around.

Our goal of the project, therefore, is to create an automatic page turner for musicians. The page turner would take in multiple electronic sheet music images, listen to the musician play, and display each sheet music image one by one for the musician. The benefits of this automatic page turner include: 1) it would eliminate the need for manual or pedal page turning, making it more convenient for musicians to play music; 2) it would be available at any time anywhere, for practice or for performance; and 3) it wouldn't require an extra device other than the tablet.

To achieve this goal, our approach consists of 5 key steps: 1) segment the last row of a sheet music; 2) read music: get an array that represents notes' duration and pitches in the last row sequentially; 3) convert real-time audio into an array of pitches and duration; 4) determine page turning by match the audio array with the sheet music array; 5) create a user interface that brings the whole system together end-to-end and allows users to interact with the integrated system in real-time.

2 RELATED WORK

Automatic page turning is intimately related to the problem of score following, an open research problem that seeks to allow the computer to track a musical performance in the form of audio, in a corresponding score representation. The most recent work on score following leverages machine learning to learn a direct mapping between sheet music and audio. Henkel et al. uses a multimodal deep learning method of creating an audio-conditioned U-Net for position estimation in full sheet images [8]. Dorfer et al. uses a deep reinforcement learning (RL) algorithm, which includes multimodal RL agents that simultaneously learn to listen to music, read the scores from images of sheet music, and follow the audio along in the sheet, in an end-to-end fashion [5]. In another paper, Dorfer et al. uses multimodal convolutional neural networks to learn joint embedding spaces for short excerpts of audio and their respective counterparts in sheet music images [6]. In the early stage of our project, we attempted to base our system on these approaches in order to leverage the state-of-the-art and allow the page turner to deal with complex music. However, we found that none of the papers' associated reference code online functions properly despite our debugging efforts (see supplementary materials for more details) [1].

Instead of relying on machine learning to learn the mapping between audio and sheet music end-to-end in a black box, we broke down the process into sub-problems and tackled each separately. One core sub-problem is sheet music parsing. Past approaches to allowing computer to parse sheet music use computer-readable representations of scores, such as MusicXML or MIDI [11]. This representation is created either manually (e.g. via the process of setting the score in a music notation program), or automatically via optical music recognition software [1, 9]. Noh et al. filed a patent for an apparatus that outputs audio data corresponding to a musical score image provided, by extracting musical signs, pitches, and duration from the sheet image [4]. However, current automatic methods are still unreliable, especially for more complex music such as orchestral pieces [15]. We specifically experimented with the online reference code of Pacha et al.'s "A Baseline For General Music Object Detection with Deep Learning," which leverages TensorFlow's Object Detection API to classify musical primitives such

*All authors contributed equally to this project.

[1]The corresponding code for the aforementioned most recent work can be found respectively at: https://github.com/CPJKU/audio_conditioned_unet/, https://github.com/CPJKU/score_following_game/, and https://github.com/CPJKU/audio_sheet_retrieval

as note heads, stems, slurs, etc. [13]. Even with the best train model, we found the results to be unsatisfactory (see more details in the appendix). We also experimented with cadenCV, which mainly uses template matching for music parsing [12]. Although the system could only handle monophonic music, we considered it a promising starting point because the system yielded the best preliminary results so far and we found it easier to improve the system due to the greater transparency behind template matching in comparison with neural networks. Our approach built and expanded upon cadenCV, as elaborated in the Method section.

Another core sub-problem is tracking a musical performance through audio input. Previous work in the area includes the Aubio system [2] and Melodia Vamp plug-in[14]. However, both methods fall short on this specific task: the pitch detection either takes too long (not real-time) or is not accurate enough to match all the midi note. In the audio processing part, we draw the Constant-Q transform method proposed by Schoerkhuber and Anssi [3]. We also compared this method to Mel Frequency Cepstral models[10], zero-crossing method, and neural networks [7].

3 METHOD

As aforementioned, there are five main segments of this project: row segmentation, read the pitches and duration of the music sheet, real-time audio processing, a comparison algorithm that determines whether to flip the page, and an end-to-end system built upon Flask.

3.1 Row Segmentation

The purpose of Row Segmentation is to bring down the image size so that the next step of template-matching takes much less computational time. A main premise of the row segmentation process is that we only care about the last row/ staff of each page of the music score.

To segment out the last row of any music score, we perform two main steps. First, we crop out all the white space beneath the last row in the following way: (i) inverting all black text to white (ii) finding non-zero points (the black text) and (iii) finding a minimum spanning bounding box around the area with text. Then, we further segment out the last staff by (i) dividing the image into horizontal bands and summing over pixel values in each band and (ii) comparing across bands to find the first band that has a lower pixel sum (more black) than its neighbors. We iterate through different band height and step sizes until we arrive at a valid output.

3.2 Read Music

Our approach to detecting the pitches and duration of a music staff is inspired by cadenCV and mainly uses template matching. The original system takes the following key steps to parse music:

(1) Convert the input image into a matrix of binary-valued pixels;
(2) Detect staff location by accumulating the number of black pixels in each row and mapping a binary image into a histogram, and a staff corresponds to a distinct peak (see Figure 1 for an example of the resulting histogram);
(3) Segment and classify primitives by matching to labelled primitive templates; note pitch values are determined by observing which row index the center of a note's bounding box lies, relative to the staff line or space sharing the row index;
(4) Correct eighth notes from misclassification as quarter notes by determining if an eighth or sixteenth flag is in the vicinity of the note head or whether it is beamed with adjacent notes (beaming is determined by if the column central to adjacent notes contains more pixels than expected);
(5) Detect key signatures by counting the number of accidentals at the start of a given staff and apply the key's accidentals to note primitives;
(6) Sort primitives by horizontal position on their staff and obtain an array of music semantics sequentially: the rests' duration as well as notes' pitches and duration;

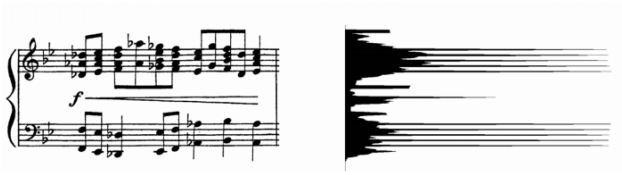

Figure 1: Histogram of the binary image for detecting staff location, where a staff corresponds to a distinct peak

We built and expanded upon the system by first transforming the output such that instead of outputting a MIDI file, it outputs an array denoting the duration of rest as well as the duration and pitches of notes sequentially. We also added recognition support for sixteenth notes and additional time signatures such as 6/4 and 3/2 time. We consider sixteenth note or rest the smallest unit, which takes up one index in the output array, and eighth note or rest takes up two indices, etc. Additional improvements we made to the system include:

(1) Testing and extending the original code to cover edge cases such as not finding beginning and ending column indices for staff;
(2) Fixing the bugs in the original code to correct its use of iterating index in order to account for the deletion of eighth flags from an array variable while correcting for misclassified eighth notes;
(3) Rewriting the code to achieve greater efficiency, reducing the time of parsing a three-staff image from 1 minute 43 seconds to 37 seconds;
(4) Increasing its accuracy in staff detection by (i) directly using template matching on staff images instead of binarizing the image then mapping the binary image into a histogram (ii) adding new templates for staff detection that extend the system's generalizability (iii) tuning the thresholds for template matching against staffs
(5) Debugging the code to account for edge cases for number of staffs and notes detected.

An example output corresponding to Figure 2, which displays an example last row of sheet music image, is:

['A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'B4', 'B4', 'B4', 'B4', 'D5', 'D5', 'D5', 'D5', 'D5',

'D5', 'D5', 'D5', 'D5', 'D5', 'D5', 'D5', 'B4', 'B4', 'B4', 'B4', 'A4', 'A4', 'A4', 'A4', 'G4', 'G4', 'G4', 'G4', 'A4', 'A4', 'A4', 'A4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'B4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'A4', 'B4', 'B4', 'B4', 'B4', 'A4', 'A4', 'A4', 'A4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4', 'G4']

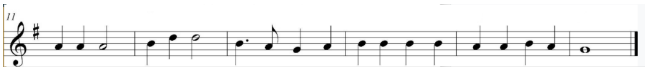

Figure 2: Example last row of a sheet music

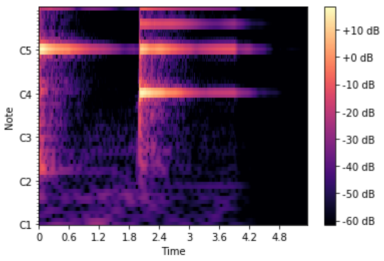

Figure 4: Q value of C5 followed by C4

3.3 Real-time Audio Processing

The primary goal is to take a real-time audio stream and output accurate pitch detection immediately. While there is previous work done in the area, none of them can match the pitch of musical instruments accurately without much noise. Most signals turned out to be fluctuating and lost the real pitch information. Therefore, we started from taking raw signals from the microphone, processing it with constant Q transform, using a buffer to smooth out noises.

Audio input Audio input is taken from the microphone of devices using pyAudio, which stems from PortAudio v18 API. The default setting is 1 channel (considering that we are only taking the performance of musical instruments), with frequency of 44100Hz, 16 bits, buffer size 1024, in the format of paFloat32 (better accuracy compared to paInt16). Note that a quiet environment is more than crucial to the audio input. With audible noises in the background (for example, the noise of a heater, a car driving by, someone talking), the pitch detection below could be skewed by a lot.

Constant Q Transform (CQT) Taking a time-domain signal, CQT is a time-frequency representation where the frequency bins are geometrically spaced and the Q-factors (ratios of the center frequencies to band-widths) of all bins are equal. In other words, it uses a series of logarithmically spaced filter f_k, with the k-th filter having a spectral width δf_k equal to a multiple of the previous filter's width. We used the package *librosa* for this purpose.

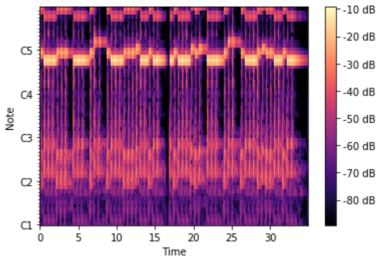

Figure 3: the CQT spectrum for Mary Has a Little Lamb

Note that CQT, though fast and accurate in most cases, does have its own constraints. It would misunderstands the octave a note is in. For example, if we play C5 followed by C4, while the CQT would find the Q-value of C5 highest at the beginning, as C4 starts, since they are an octave apart, the Q-value of C5 would remain high.

Considering the fact we are only trying to decide whether or not to flip a page, and our visual input from the last row is usually quite long and is composed of various notes, the octaves a note is in should not matter too much in terms of deciding the location. In music composition, it is rare to see two measures differ from each other simply by octaves they are in. Therefore, we still decide to use CQT and acknowledge this limitation of accurate octave detection. Further details could be find in the comparison algorithm section and test evaluation section.

Stream buffer and processing We maintain a stream buffer of size 1024. This buffer size allows us to take in 3 signals per time. We always turn the signals within the buffer to a numpy array first. We then use the CQT algorithm, with 12 bins per octave (each pitch as a bin) and 60 overall bins (a total of 5 octaves which should cover the range of most frequently used notes for all musical instruments). We then convert the CQT matrix from amplitude to db, and find the pitch (frequency bin) with the highest Q value (in db) as the most likely pitch. We take the middle signal out of the three in the buffer, smoothing out the onset and offset noise. In fact, the 0th and the 2nd pitch detected constantly fluctuated a lot while the middle pitch remained constant in most of the time.

Comparison with other feasible methods A sets of methods were being tested here to find the most accurate and reliable method. Those include:

- Mel-Frequency Cepstrum in the frequency domain, which runs more slowly compared to CQT, and cannot be directly converted to a note pitch.
- Zero-crossing in the time domain, which has lots of false, random detections that cannot be smoothed.
- Neural network, which asks for extensive training data. Most open source trained neural network models were trained on speech data, rather than music pieces. Those models tend to mix two neighboring pitches together since each pitch has a frequency bin of around 50 Hz.

All of those methods also have the problem of misunderstanding the octave a note is in (as the C5 followed by C4 example). Therefore, we still ended up using the CQT algorithm.

An additional version of processing and smoothing a music audio file rather than audio stream is provided in the repository with code and tutorials. The CQT algorithm is even more accurate and further smoothing and concatenating of the noises could be performed.

3.4 Comparison Algorithm

With both the visual and the audio detections, we perform comparison algorithms to see if there is a match and would need to flip the page. We take the music sheet visual input as the correct version. The audio has to perfectly match the pitch of the visual input of the last row in order to flip the pitch. In addition, the relative length of notes matters. If the ratio is not within a reasonable range, the algorithm will determine the audio to be not matching the sheet, therefore, not flipping the page. Considering that musicians generally play slower or off the speed proportion when they get to the end of a page in practice, we did not try to come up with the best range. We use 0.5 as the default value, but this could be subject to users' preferences.

We take an extra step in the comparison algorithm to make the visual sheet data easier to compare with: we turn them from a 1-D array to a 2-D array, the first column being the value of the pitch, and the second column being the length of that pitch. Note here that in our data representation, there is no difference between two eighth note and a quarter note, though there will be a difference in the actual music performance (there will be a short break between the two eighth note but not in the quarter note). We will discuss how our algorithm deal with this feature in the paragraph below.

The Real-time comparison starts as the stream first hits the first note of the visual input. For each pitch, we listen to the audio and get the length of the pitch. We also include a buffer here. The buffer is set to be a third of the length of the pitch. Within this buffer, if we hear the pitch of the next note, we will assume that this note has ended, and then compare the ratio of the length of this note in audio and in the sheet to see if they are proportional. If yes, we move on to the comparison of next note; if no, we start over again. Back to the buffer, if we hear the pitch of the current note again, we know that there might be a short break (might be the one between the two eighth notes). Since we do not account for their differences, we would assume we are back to the current note, restart the buffer, and add the length of the the buffer loop to the length of the current note. If we exhaust the length of the buffer before hearing the current note or the next note again, we know we are at the wrong place. Therefore, we will start over from finding the match of the first note again.

We perform special processing for the rest note. We would ignore the audio signal for the expected length of the rest note in the audio (the ratio we have seen times the length of the visual signal) and proceed to the comparison of the next note. If the rest is at the end of the row (input array), we would ignore it and return whatever result we had from previous comparisons.

The last note is also taken specially. Considering the fact that people tend to play the last note on the last row short or long as they are thinking about flipping the page, we do ask the length ratio of the last ratio to match the previous ones. As long as the pitch is hit, we consider a matching. However, we will flip the page one the duration of the pitch ends. This way, the performer gets to decide how long they want to play the last note before flipping the page.

Comparison for music files rather than real-time is easier and could be more accurate. Besides the matching of pitches, two methods were performed to compared to length ratio. One is for each pitch, we find the corresponding ratio between the audio and the sheet, and see if the ratios match. Another method is for each pitch, we would have an expected length of that pitch in the audio (ratio times the length of the sheet). We could then read the audio of that length, and get the percentage of that pitch in that length. If the percentage is higher than certain barrier, there is a match. A detailed analysis of those methods and some more complicated ones could be find in the supplementary material and the tutorial.

Our real-time comparison draws ideas from both approach. For each pitch, we do the ratio comparison. But then, we also use the idea of expected length in the design of buffer, letting the algorithm adjust to short breaks/noises in the audio input.

3.5 End-to-end Flask System

We created a Flask-based web app in order to integrate the whole system and allow users to have real-time interaction with our automatic page turner.

On the index page, a user could upload multiple sheet music images in JPG and PNG format through drag and drop. Upon receiving the user's uploaded images, the server reads and processes each image. To deal with the technical limitations of the Flask framework, we constructed the server such that it calls file.seek(0) on each FileStorage object representing each uploaded image, reads the object as a byte string, converts the string into a numpy array, and passes it to cv2.imdecode, in order to successfully read each uploaded image. To process each image, it calls functions that segment the last row and reads the music to obtain the pitch-duration array, and stores the resulting array in a global variable. It also stores the image urls for the uploaded images in a global variable.

Once the processing finishes, the web interface redirects the user to a page that displays the first sheet music. Whenever the user is ready to play music, they could click the "Start Recording" Button, which sends a POST request to the server to activate our function for receiving and parsing audio signals. Whenever the comparison function returns True for page flipping, the server renders the next sheet music image. The server increments a current page index to keep track of the proper sheet music image for display. Except for the first sheet music image, the server always performs an auto-refresh 3 seconds into displaying a new sheet music, which leads to a new GET request to the server with a nonzero current page index, such that the audio processing and comparison function could be activated again.

Whenever the user wishes to start over, they could click "Back" to return to the main page of image uploading, and the previous results will be cleared for a fresh start.

4 EXPERIMENTS

Since there is no available dataset, We used MuseScore to manually construct music sheets as well as corresponding audios with different BPM (beats per minute). Since we don't have a correct labeling dataset, we also manually label the correct output of the visual information, and test the correctness of overall pitch detection and page flipping. The test dataset we constructed could be found **here**, where the directory /wav includes all the wav files we constructed using MuseScore, the directory /samples includes all the sheet music as sample input to the system, and the directory /lastrow_output

includes all the golden outputs of the system corresponding to each sheet music input.

4.1 Sheet music processing

To test the note detection functionality, we compare the output from note_detection.py with the manually labeled golden output. Among all the test results, the staff detection comes out perfectly. Then, in the step of note detection within each staff, the test file Mary has the best performance because of its simple structure: all its notes are monotonic, situated among the staff lines, and are either quarter, half, or whole notes. For scores like Bluebird, dona, and hush, and Kookaburra, the detected pitches match the golden output, but distinguishing between quarter and eighth notes is difficult because of the diverse number of ways in which eighth notes could be grouped together. For files like fire, telemann, and teapot, the system correctly detects notes a bit above, a bit below, and all within the detected staff, but the notes that are further away from the top or bottom of the staff are not detected. For mammy, all the notes are detected except for the rest note, which is currently not supported by the system. Similarly, winter has rest notes that are not supported. The system also currently performs better on flats than sharps, as illustrated by the performance of ringo and telemann versus races. Given the difficulty of the note detection problem, we can consider relaxing the comparison algorithm (between notes detected audibly and visually) to allow room for error. A relaxation would further support the case that a user plays a music slightly off pitch.

4.2 Audio Processing

It is hard to test the accuracy with real-time audio processing as we would not have the correct labeling. Instead, we tested our Audio Processing algorithm manually by playing different pieces of musics through MuseScore and see if our audio processing algorithm outputs the right note. This is performed on the sets of notes below:

(1) individual notes from C3 to B5
(2) arpeggio of different majors and minors
(3) same note on different octaves
(4) different note on different octaves

As mentioned earlier, while other parts of the detection have really high accuracy rate (almost always correct as long as the key is playing), same or different note on different octaves becomes tricky. The later note is always influenced by the previous note. For the example we mentioned: playing a C4 after a C5, we approximately get C4 half of the time and get C5 the other half of the time. After playing a G5, even if we play a F4, it is classified as F5 most of the time. Most of those distortion is one octave apart. Fortunately, if we allow a difference of 12 (one octave) when we are comparing the pitches, we would be able to catch the right pitch all the time.

4.3 Comparison Algorithm

The comparison algorithm works almost all the time. We ran the comparison on jupyter lab. The python notebook could be find in the supplementary material. Besides the piece "Mary," which is the one we used to come up with the initial algorithm, there are ten short pieces, each having different music features (some of them have rest signs, some of them have sound conjunction, some of them have split notes, some of them have note of all four beats while others have 16th notes, etc.). For each test, each piece was run ten times. And the number below the success rate out of those ten times, in percentage. Note that those evaluations were run in quite environments. Noisy environment might end up with 0% success rate.

Tests with Different Starting Time We first test how the algorithm performs page flipping at different starting time. Since the algorithm assumes that the whole last row would be played, if the starting time is after the exact time, we define success as no page flipping happens. If the starting time is the exact time or before, we define success as page flipping happens, and only happens when it gets to the end of the page. The starting time of before or after the exact time is randomly chosen.

Evaluation of Starting Time (Success rate, in percentage)			
Music Piece	Exact	Before	After
Mary	100	100	100
Hush	100	100	100
Fire	100	100	100
Teapot	100	100	100
Winter	100	100	100
Bluebird	100	100	100
Mammy	80	70	100
Ringo	100	90	100
Races	100	100	100
Dona	100	100	100
Telemann	100	100	100

From the chart, we can conclude that the algorithm generally performs really well, especially at not flipping the page if started playing after the last row. For playing at the exact time, the only case it fails is with piece Mammy, where there are a sets of 16th note. A closer inspection of the outcome suggests that the algorithm has trouble detecting it since those notes are short in time and are also neighboring notes of the one before them. The algorithm also failed once with Ringo. That failure was not reproducible so it could be the problem of that one-time real-time recording.

Tests with Different Playing Speed We then try to test the robustness of the algorithm with different playing speed, as we would hope that no matter how fast or slow the musician play the piece, the algorithm can still accurately flip the page. The tests are performed starting at the exact time. The success rate is based on flipping the page at the right time, in percentage.

Evaluation of Playing speed (Success rate, in percentage)			
Music Piece	80 bpm	40 bpm	120bpm
Mary	100	100	100
Hush	100	100	100
Fire	100	100	100
Teapot	100	100	100
Winter	100	100	100
Bluebird	100	100	100
Mammy	80	100	70
Ringo	100	100	100
Races	100	100	100
Dona	100	100	100
Telemann	100	100	100

We note that playing the music more slowly generally helps the algorithm to better determine the flipping. Especially for the piece Mammy, as the playing speed gets slowed down, the 16th note can be clearly detected, therefore, achieving a success rate of 100 percent. Meanwhile, playing it faster hurts the result even more.

Overall speaking, given the high success rate, the algorithm is robust and trustworthy.

4.4 Overall System

To test the integrated system, we interacted with the web app 10 times to simulate real-world use cases. Each time, we uploaded a set of sheet music image samples; the set includes different number of images, from 2 to 6, in order to test the system's ability to handle multiple image. In all 10 trials, the system correctly uploads users' images, parses them, stores the result, redirects the user to the page that displays the first sheet music, and activates the audio processing and comparison function. The parsed outputs that the server prints to the terminal are the same as the results of running the music sheet parsing algorithm separately, showing that the integration of sheet music processing works as expected. Additionally, whenever the parsed result is the same as the golden output, and the surrounding environment is noise-free, the audio comparison function also returns the flip signal at the correct time, showing that the integration of the audio signal processing and comparison into our system works as expected. Clicking "Back" and re-uploading images also shows expected behavior where previous results are cleared for a fresh start. Overall, the testing shows that the web app succeeds in integrating our system, achieving real-time user-interaction, and demonstrating the functionalities we implemented.

The testing also shows limitations of our system. Firstly, when the parsed result contains errors, the page flipping signal is not obtained properly; secondly, the page flipping signal is not obtained properly whenever there is noise in the surrounding environment. The testing result suggests that future improvement should include better sheet music parsing as well as better fault and noise tolerance in our audio processing and comparison algorithm.

5 CONCLUSION AND FUTURE WORK

This project is a proof of concept and there is still a lot could be done in the future. For example, we could expand from the monophonic scenario to a much more complex, real-life music sheet.

One crucial takeaway is that neural networks or machine learning methods in general are not always the best approach to tackle down a real-life problem. In this project, we have experienced machine learning techniques at all parts. However, none of those attempts worked. Through some more basic, intuitive, and physical ways, we were able to reach our goals.

In the Audio Processing part, we could work on detecting the pitch of all octaves accurately (i.e. eliminating the influence of other notes of different octaves), potentially through mel-frequency method. We could also examine the case where we have to tell multiple pitches happened at the same time. One potential idea would be taking the ones with highest Q values (and above a certain barrier). But we might need ample data set to find out the barrier and the most efficient way.

In the comparison part, we could further work on different scenarios. For example, what if some portions of the audio playing is wrong? Can we flip the page with some mistake tolerance? Another way of improvement is to shorten the reaction time. For example, before we actually hit the last note of the last row, we should probably already flip the page. Those details have to be determined with actual user cases and large enough data set.

Additional future work includes making the web app more user-friendly and multi-functional. For example, we could implement user login and account, such that multiple users can use our system at the same time and enjoy custom functions such as saving their previously uploaded images for re-use if needed. We could improve the UI of the app for a better user experience, and host our web app on a cloud server so that the app is more accessible.

REFERENCES

[1] Stefan Balke, Sanu Pulimootil Achankunju, and Meinard Muller. 2015. Matching musical themes based on noisy OCR and OMR input. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. Brisbane, Australia, 703–707.

[2] Paul Brossier, Juan Pablo Bello, and Mark D. Plumbley. 2004. Real-time temporal segmentation of note objects in music signals. In *International Computer Music Conference*. Miami, Florida.

[3] Schoerkhuber Christian and Anssi Klapuri. 2010. Constant-Q transform toolbox for music processing. In *7th Sound and Music Computing Conference*. Barcelona, Spain.

[4] Joo-Sub Kim Dong-Hoon Noh. U.S. Patent 7 547 840 B2, Jun. 2009. Method and Apparatus for Outputting Audio Data And Musical Score Image.

[5] Matthias Dorfer, Florian Henkel, and Gerhard Widmer. 2018. Learn to Listen, Read, and Follow: Score Following As a Reinforcement Learning Game. In *Proceedings of 19th International Society for Music Information Retrieval Conference*. The Austrian Research Institute for Artificial Intelligence, Austria.

[6] Matthias Dorfer, Jan Hajič jr., Andreas Arzt, Harald Frostel, and Gerhard Widmer. 2018. Learning Audio–Sheet Music Correspondences for Cross-Modal Retrieval and Piece Identification. In *Transactions of the International Society for Music Information Retrieval*. The Austrian Research Institute for Artificial Intelligence, Austria, Article 1(1), pp.22–33. pages.

[7] Etienne Barnard et al. 1989. Pitch deteciton with a neural-net classifier. CSETech.

[8] Florian Henkel, Rainer Kelz, and Gerhard Widmer. 2019. Audio-Conditioned U-Net for Position Estimation in Full Sheet Images. The Austrian Research Institute for Artificial Intelligence, Austria.

[9] Jan Hajic jr and Pavel Pecina. 2017. The MUSCIMA++ Dataset for Handwritten Optical Music Recognition. In *Proceedings of 14th International Conference on Document Analysis and Recognition (ICDAR)*. New York, United States, 39–46.

[10] Beth Logan. 2000. Mel Frequency Cepstral Coefficients for Music Modeling. In *International Symposium on Music Information Retrieval*.

[11] Marius Miron, Julio Jose Carabias-Orti, and Jordi Janer. 2014. Audio-to-score alignment at the note level for orchestral recordings. In *Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR)*. Taipei, Taiwan, 125–130.

[12] Afika Nyati. 2017. cadenCV: An Optical Music Recognition System with Audible Playback. Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, United States.

[13] Alexander Pacha, Jan Hajič Jr., and Jorge Calvo-Zaragoza. 2018. A Baseline for General Music Object Detection with Deep Learning *(Digital Audio and Image Processing with Focus on Music Research)*. Institute for Visual Computing and Human-Centered Technology, Austria.

[14] J. Salamon and E. Gomez. 2012. Melody Extraction from Polyphonic Music Signals using Pitch Contour Characteristics. In *IEEE Transactions on Audio, Speech and Language Processing*.

[15] Verena Thomas, Christian Fremerey, Meinard Muller, and Michael Clausen. 2012. Linking Sheet Music and Audio - Challenges and New Approaches. In *Multimodal Music Processing, volume 3 of Dagstuhl Follow-Ups*, Masataka Goto Meinard Muller and Markus Schedl (Eds.). Schloss Dagstuhl–LeibnizZentrum fuer Informatik, Dagstuhl, Germany, 1–22.

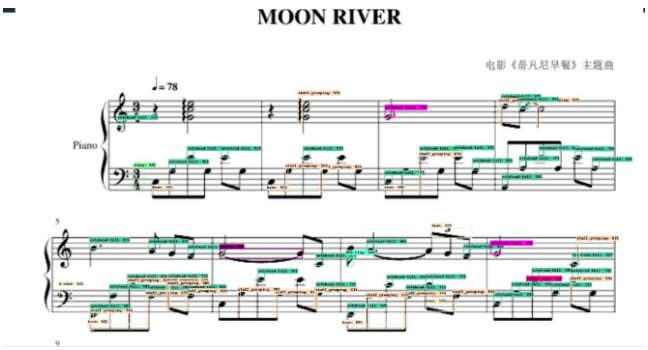

Figure 5: Example output, with minimum threshold score set to 0.5

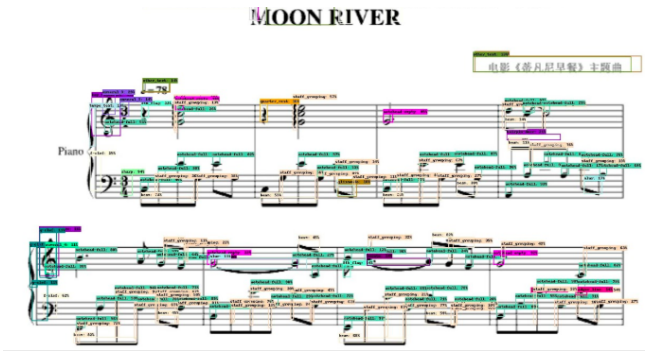

Figure 6: Example output, with minimum threshold score set to 0.1

A APPENDIX

A.1 Alternative Sheet Music Parsing Using TensorFlow

The section documents our experimental results of parsing sheet music by leveraging TensorFlow's Object Detection API. The dataset used for training the neural network is MUSCIMA++, which consists of 140 images of annotated sheet music images; the advantage of the dataset is that it has a graphical structure where the vertices of the graph are music primitives (e.g. stem, notehead-empty, notehead-full, beam, slur, etc.), and the edges of the graph connect music primitives to each other, from which information such as pitch and duration could be inferred.

We built upon the online reference code by fixing setup issues neglected by the original code's instructions and testing on new sheet music images with different minimum threshold scores (which determines the minimum confidence score associated with a bounding box that allows the classification result to be included in the the final result). The original plan was that once we reached a relatively good detection result, we would implement a search through the result; based on the music primitives and locations of their corresponding bounding boxes, we could parse these primitives into sequential notes and rests along with their duration and pitch changes based on their types and relative heights; a future improvement would be altering the model such that it infers not only vertices of the graph (the music primitives), but also the edges (how these primitives are connected).

However, we were surprised by the unsatisfactory performance of the best train model. As an example, Figure 5 shows the output of the model with min threshold score set to 0.5, and the model misses the classification of many music primitives; Figure 6 shows the output of the model with min threshold score set to 0.1, and the model still misses certain classification, and for some music primitives such as the first treble clef, there are multiple overlapping classifications with almost identical confidence score. The result demonstrates that the model is not robust enough to handle complex electronic sheet music. Another problem with this system is the speed: to process a three-staff score "Mary Has a Little Lamb", the system takes over 3 minutes, whereas cadenCV takes half as long and our final improved system takes only 37 seconds. It's not desirable for users to wait a long time while our system processes their uploaded images.

Our experimental results demonstrate that deep learning doesn't work well for our use case at the moment. To improve the result, future efforts could include having a data set much larger than just 140 images to deal with the variance in how music primitives look like in sheet music.

More experimental results could be found in the supplementary materials.