EE/SE/CPRE 491 - Spring 2019

Student Suggested Project

Sheet Vision

Final Report

Team Number

sddec19-13

Faculty Advisor

Alexander Stoytchev

Team Members

Bryan Fung

Garrett Greenfield

Ricardo Faure

Trevin Nance

Walter Svenddal

Team Website

http://sddec19-13.sd.ece.iastate.edu/

Table of Contents

List of Abbreviations & Symbols 2

List of Definitions 2

1 Introduction 2

1.1 Problem Statement 2

1.2 Solution 3

2 Design and Analysis 3

2.1 Operating Environment 3

2.2 Use-Case Diagram 3

2.3 Intended Users 4

2.4 Design Specifications 4

2.5 Proposed Design/Method 4

2.6 Block Diagram of System 4

2.7 Design Analysis 4

2.8 Process Details 5

3 Implementation Details 5

3.1 Front-end Application 5

3.2 Back-end Stack 6

3.3 Computer Vision Algorithm 6

3.3.1 Overview 6

3.3.2 Image Sectioning 7

3.3.3 Staff/Bar Identification 9

3.3.4 Note Detection 12

3.3.5 Note Mapping 13

4 Testing 15

4.1 Testing Process Details 15

4.1.1 Front-end 15

4.1.2 Back-end 15

4.1.3 Computer Vision 16

4.1.3.1 Preprocessing 16

4.1.3.2 Note detection and mapping 16

4.2 Testing Results 16

4.2.1 Front-end 16

4.2.2 Back-end 16

4.2.3 Computer Vision 17

5 Related Products 17

5.1 Sheet Vision 17
5.2 Musescore 18
5.3 Playscore 18

6 Appendices 18
6.1 Appendix I - "Operation Manual" 18
6.2 Appendix II - "Initial Versions of Design" 19
6.2.1 Sheet Vision Desktop 19
6.2.2 AWS Previous Backend Designs 19
6.2.2.1 EC2 + Apache 19
6.2.2.2 API Gateway + Lambda 19

List of Abbreviations & Symbols

1. AWS Amazon Web Services
2. API Application programming interface
3. MIDI Musical Instrument Digital Interface
4. CA Certificate Authority

List of Definitions

1. Sheet Music - Music in its written or printed form.
2. Musical Notes - A sign or character used to represent a tone, its position, and form indicating the pitch and duration of the tone.
3. Tabs - A form of written music, but instead of being represented in the traditional sense (what tone it makes), notes are represented by the specific position they are supposed to be played in.
4. Amazon S3 Bucket - Storage location for our Amazon Web Services.
5. Ledger Lines - Lines outside of the staff used to help identify the pitch of a note

1 Introduction

1.1 Problem Statement

Reading sheet music is no easy task. With the creation of alternate ways to learn how to play music, such as tabs and youtube tutorials, there has been a decline in the number of people who can properly read sheet music. The problem with tabs and other kinds of methods of reading music is that they lack the complexity to be able to convey all of the specific nuances that a specific piece may have. The best option that captures all of the nuances that most musicians wish to convey when writing music, is sheet music. The problem with sheet music is that it can be very difficult at first, and since there is a decline in the number of people that can read it, it can be difficult to find a proper way to read it.

1.2 Solution

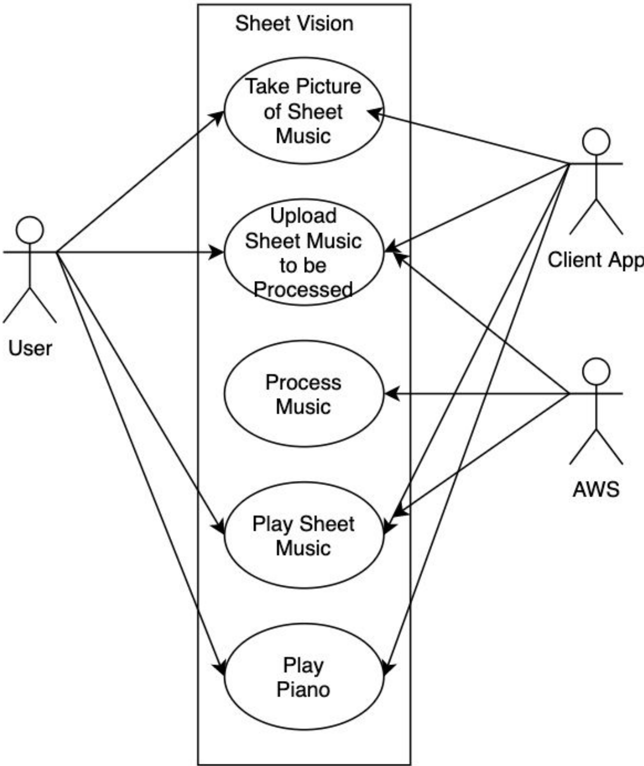
The solution for this is Sheet Vision, an application that can read and show a user how the sheet music is played, and how it is supposed to sound. This will lead to the user being able to draw parallels between what is on the sheet, and the music being played, supplementing the learning process of reading sheet music.

2 Design and Analysis

2.1 Operating Environment

The product is expected to be used with bright and uniform lighting. Lighting is important to allow the image taken to be clear and evenly colored. This is necessary so the computer vision algorithm can accurately detect the notes and rests, and where they lie on the staff.

2.2 Use-Case Diagram



This is a use case diagram for the application. It demonstrates the use case ideas, services and actors for the application.

2.3 Intended Users

This product is intended for beginner musicians. This application should provide instructions simple and clear enough for even first-time musicians can keep up with using the product, yet powerful enough to ease some of the struggles of reading more complex songs for veteran musicians.

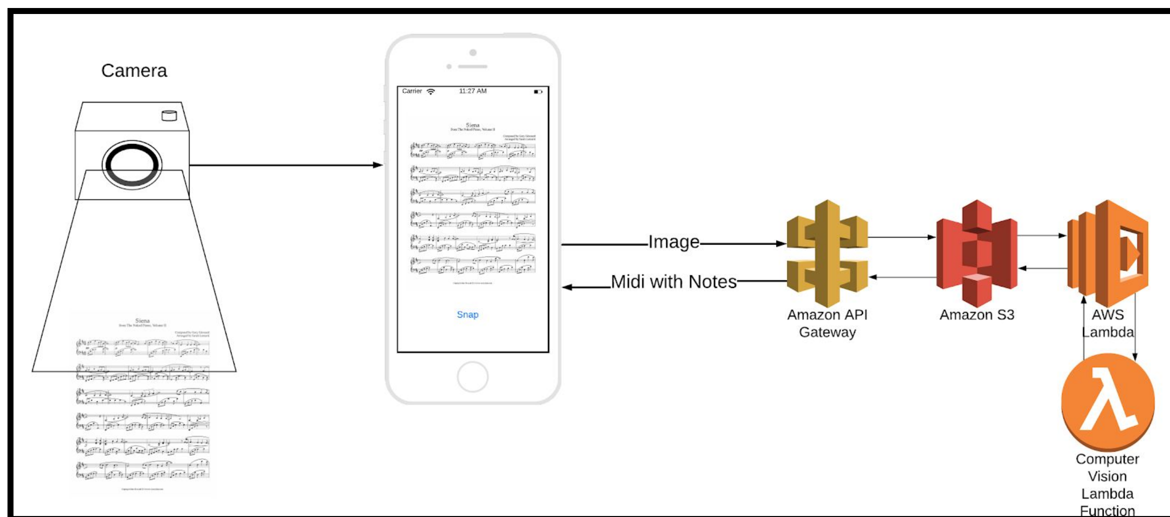
2.4 Design Specifications

The mobile client will be a standard mobile application. The application will be able to take images from the user's camera and take pictures from their gallery and upload them to the server. Then the front end application will be able to play the JSON files that are received from AWS, and then as an additional feature.

2.5 Proposed Design/Method

The design calls for the front end code to be written in React Native to provide multi-platform usability. The machine vision will be written in Python using the OpenCV library along with Numpy on an Amazon Web Services (AWS) server to take the load off the user's device, and to allow the machine vision code to be reused without having to be re-written for each platform.

2.6 Block Diagram of System



2.7 Design Analysis

The computer vision algorithms will mainly be run on a machine on an AWS web machine and be accessed with the use of API requests sent from the client application to this machine, which we will refer to as the server. The client application will be available in the form of a multi-platform mobile application. The mobile client will be written using React-Native, which allows the application to be

available to both Android, iOS. With a multi-platform client, how is it that we are maintaining the main component of the application intact and consistent through all operating systems and devices?

The way we are maintaining this consistency is by only having the Computer Vision component stored in one place, in which any kind of client can access, therefore, we decided it would be best to keep the computer vision and processing all separate from the client application and have a clear API in which we simply send data to the AWS stack to be processed and get back a data structure with data we can use to play sounds and create UI updates that act on the data returned. The data will be consistent no matter what device it gets sent to, making it easy to make multiple versions of the same app, without risking functionality and avoiding data inconsistencies when processing data on different systems.

2.8 Process Details

When the user opens the application they will be greeted with the option to upload an image of sheet music or take a picture themselves. Once the image has been selected/taken, the user will then be able to upload the images. A PUT request will be sent to the AWS Gateway, which will send the image to an Amazon S3 Bucket. The AWS machine will accept the POST which will contain the image. Upon receiving the image Amazon S3 will run the Lambda trigger, which processes the image and posts the JSON result back into the S3 Bucket, ready to be returned to the client.

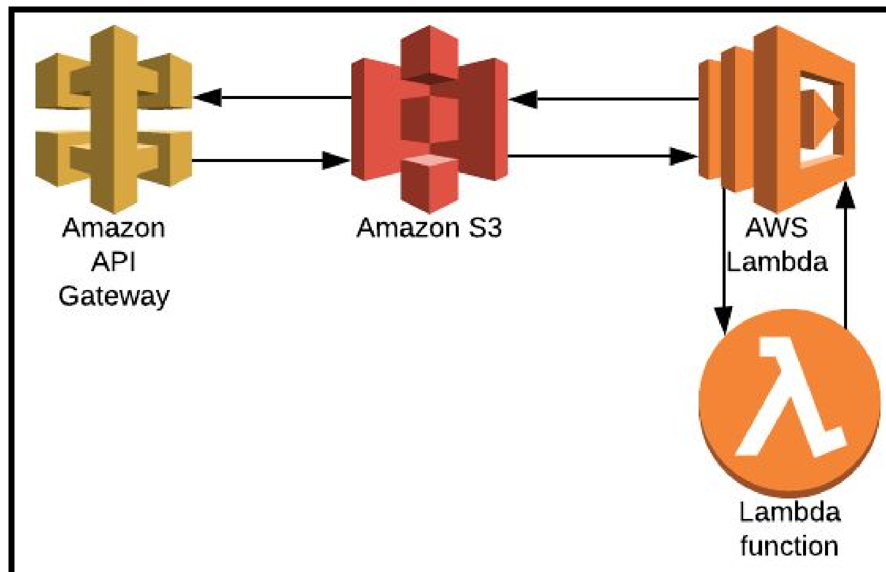
Inside the main image processing, the image goes through multiple stages to turn the sheet music into a JSON list of notes and timings, similar to a MIDI file. Details of the image processing can be found in section 3.3. The JSON string is then sent from AWS to the client application. From here the JSON string can be read by the application, which will play the music while animating the notes which are being played on an animated piano.

3 Implementation Details

3.1 Front-end Application

The front-end application was designed with three tabs to be simple to use and as user-friendly as possible. The app was built on React with Expo libraries because of its one code multi-platform development, and rapid prototyping capabilities. With this framework, the project was programmed in javascript with these key things in mind; The user can access it on either device and it be equally user-friendly whichever platform they choose, The user can easily understand the full functionality of the application with little to no questions on how to perform the applications intended purpose, and the app should be able to be used by all beginners no matter their musical background or upbringing.

3.2 Back-end Stack



3.2.1

The backend stack consists mainly of three sections: AWS API Gateway, AWS S3, and AWS Lambda. The API Gateway acts as a REST API to send data to the different Amazon Web Services and to retrieve data back from the S3 Buckets. API Gateway consists of two main functions, PUT and GET. The PUT request will require 2 parameters, the folder location of where data will be saved into the S3 bucket, and the filename of the file to be saved into the previously described location. The GET will also require 2 parameters, the folder location of where we want to retrieve the file, and the name of the file to be retrieved. Once a PUT is called, the image sent will be put into the specified location on the S3 bucket, where S3 will call a trigger function on AWS Lambda.

A trigger function is code that is called from AWS Lambda, every time an item is saved onto the S3 bucket after certain specifications have been met, such as the correct file type and contents, as well correct file saved location. The specifications for these files is a .txt file containing the image in base64 format, as well as these images being saved to the userInput folder. This trigger function will run the computer vision algorithms on the passed in image representation and post the result back into the S3 box on the processedImages folder, ready to be sent back to the client. The client will then call a GET request after enough time has elapsed for the image to be processed in which it will go through API gateway and retrieve the image through the API Gateway GET method from the S3 box.

3.3 Computer Vision Algorithm

3.3.1 Overview

This section will talk about the implementation of the computer vision algorithm, which takes an image of sheet music as an argument and returns a JSON string containing all the note information which

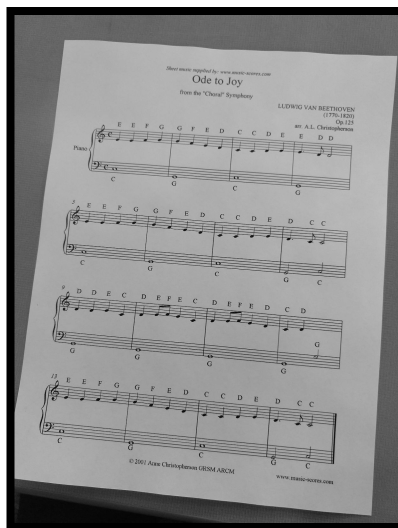
was found within the given image. This algorithm is run on AWS and was programmed in Python 3.6. The algorithm can be divided up into these four parts:

1. Image sectioning.
2. Staff/Bar identification.
3. Note detection.
4. Note mapping.

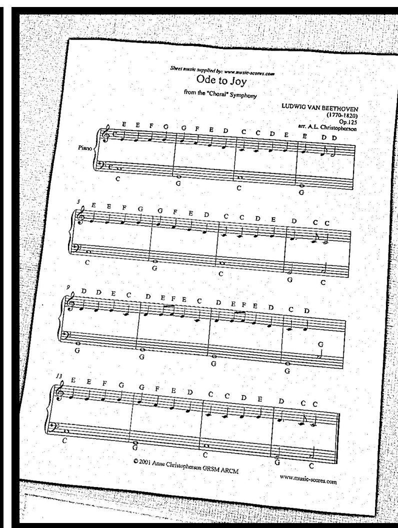
These four parts are modular, in that each can act as its own function. Each part takes its own input and produces its own output, so the development of any one part wouldn't interfere with the functionality of the other three.

3.3.2 Image Sectioning

In this first stage, the original photograph of the piece of sheet music is taken as input and outputs sections of the photo which are most likely to contain staff lines and notes. This stage reduces the image size by scaling the image an integer amount until it contains less than two million pixels to improve the runtimes of the algorithm, especially the Gaussian Threshold. The resulting sections

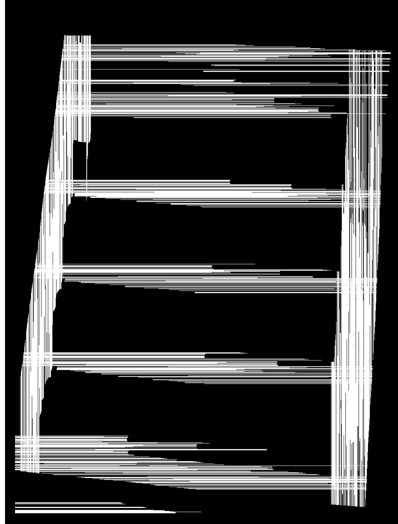


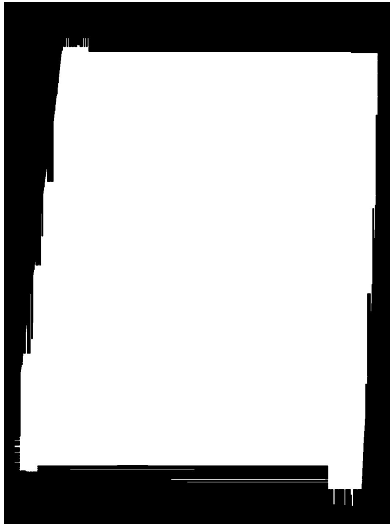
3.3.2.1



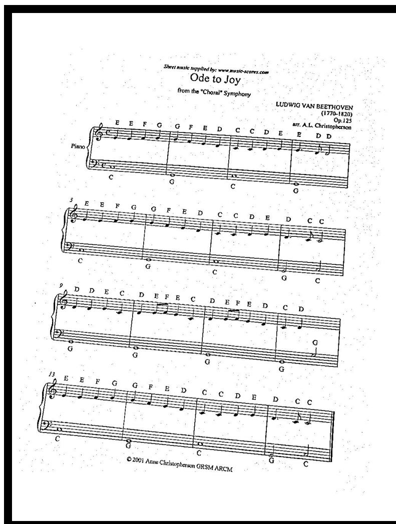
3.3.2.2

First off, the image is turned to grayscale, since the color isn't necessary to interpret the music in the image, as seen in image 3.3.2.1. Then in image 3.3.2.2, a Gaussian threshold is applied. The Gaussian threshold sets each pixel in the image to either black or white based on nearby pixels, which allows the algorithm to turn to black/white correctly even with poor lighting.

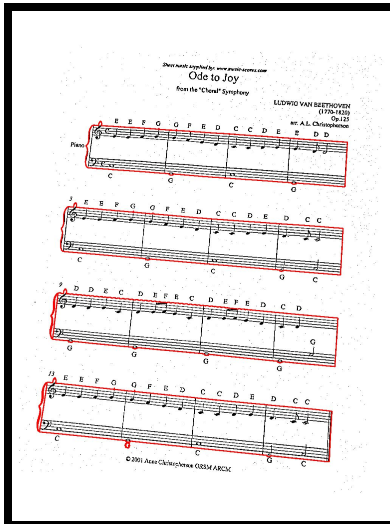

3.3.2.3


3.3.2.4

An assumption of the algorithm is that the sheet music is surrounded by the borders of the white sheet it's printed on. To better find the staff lines in the sheet, it's helpful to remove any background not related to music. Using OpenCV's `findContours` function, a white object in the image can be found, in this case, the sheet itself. First, as shown in image 3.3.2.3, two long thin structuring elements are applied to the image, one horizontal and one vertical. This will create a clear white border where the border of the sheet is. Using the `findContours` function, the points of the white border are found, and then the interior of the border is filled as in image 3.3.2.4. Image 3.3.2.4 is a mask that will be used in the next part.



3.3.2.5



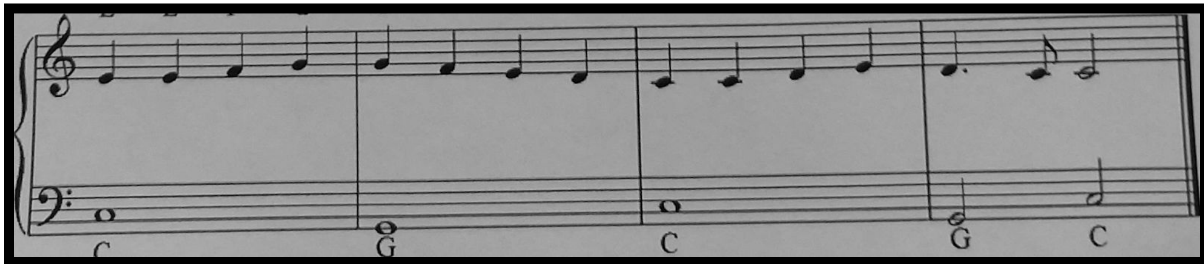
3.3.2.6

The mask from the previous image is then applied to the thresholded image in 3.3.2.2. Using OpenCV's `bitwise_not` on the mask and then `bitwise_or` on the mask and image 3.3.2.2, the borders around the sheet in the image is kept. This results in image 3.3.2.5. Finally, `findContours` can be applied again to this resultant image, except this time searching for black objects in the image. The

largest black contours are assumed to contain staff lines. Using the `minAreaRect` function in OpenCV, a rectangle around the contour is found. This rectangle is expanded to the full image's size due to the scaling down at the beginning of this part. The rectangle is used to remove that section from the original image and is then sent to the next part of the process.

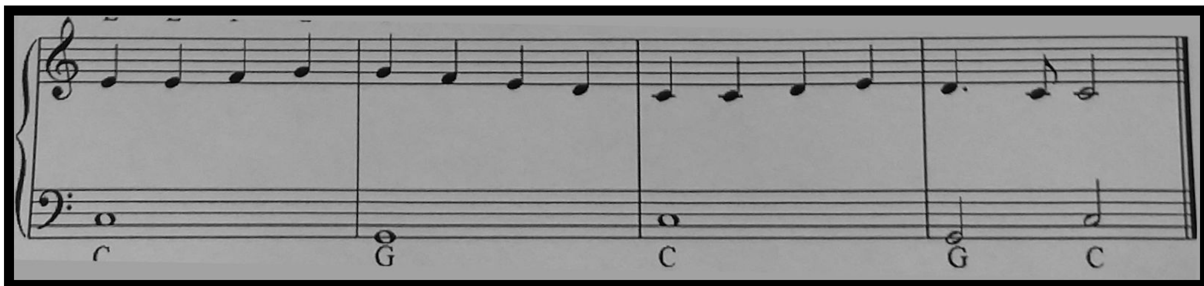
3.3.3 Staff/Bar Identification

In this second stage, sections of the image which are believed to contain music are given as input, and the locations of staff and bar lines within the image are given as output. If adequate staff/bar lines cannot be found, an exception is thrown instead.



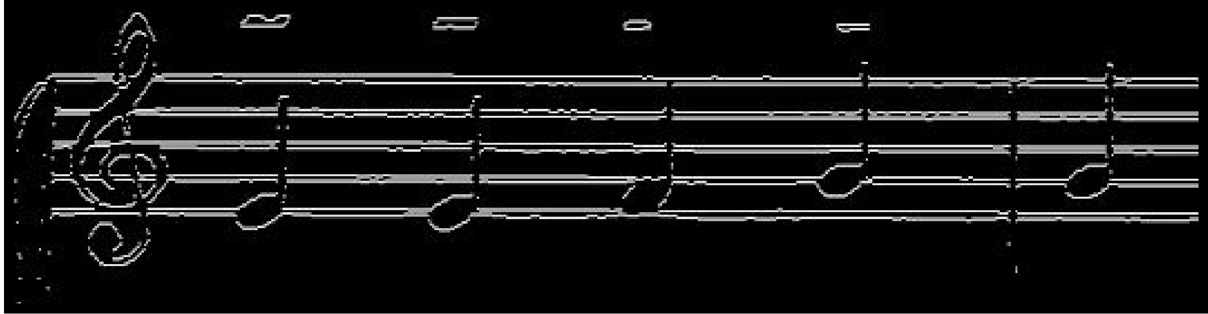
3.3.3.1

Image 3.3.3.1 is an example of the image incoming from part 3.3.2. First off, the image is rotated so the measure lines are, on average, horizontal. This is done by using OpenCV's `houghLines`, finding the average line among all the discovered horizontal lines (between -10 and 10 degrees from the horizontal). A better example of how the Hough lines function will be described below. The angle of the average line is found, then the image is rotated by that amount. The extra pixels around the border after the image's rotation are set to be the statistical mode color of the input image, ensuring the borders will be about as gray as the rest of the image so that the borders don't interfere with another Gaussian Threshold. The result is seen in image 3.3.3.2. In this image, the C on the bottom left corner which was cut off is now raised, and that space is filled with a similar gray to the rest of the image.



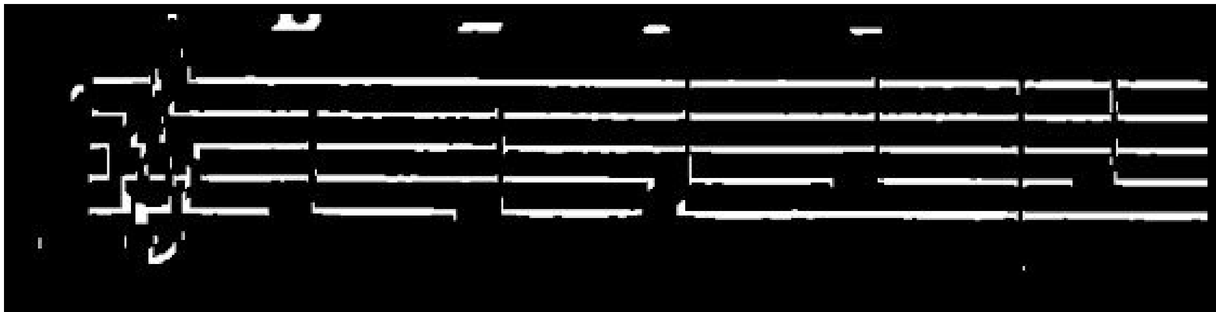
3.3.3.2

Next, using OpenCV's `morphologyEx` function, a new mask is created where a white pixel is drawn wherever there's a vertical change from white to black, or black to white. This creates thin lines above and below any black space in the image. This is seen in image 3.3.3.3.



3.3.3.3

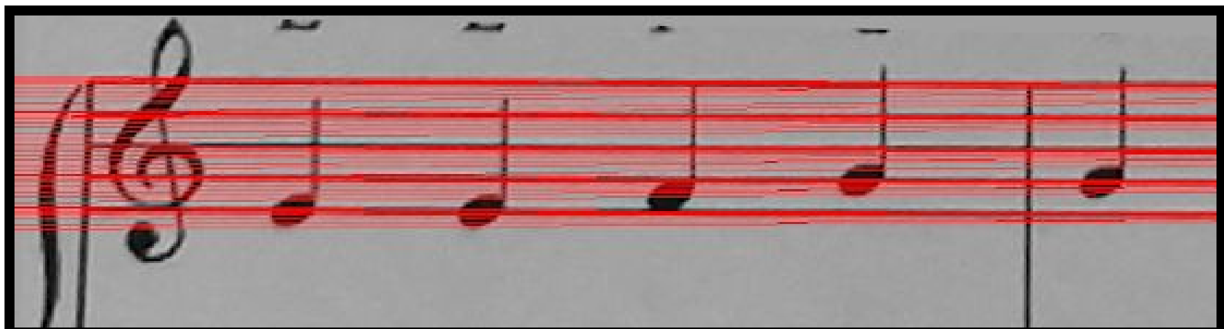
Another structuring element is used to merge the top and bottom thin lines from image 3.3.3.3, but only if the lines on top and bottom are close enough. This is why the staff lines are visible but the notes are not. These white lines completely cover the black staff lines in the original image.



3.3.3.3

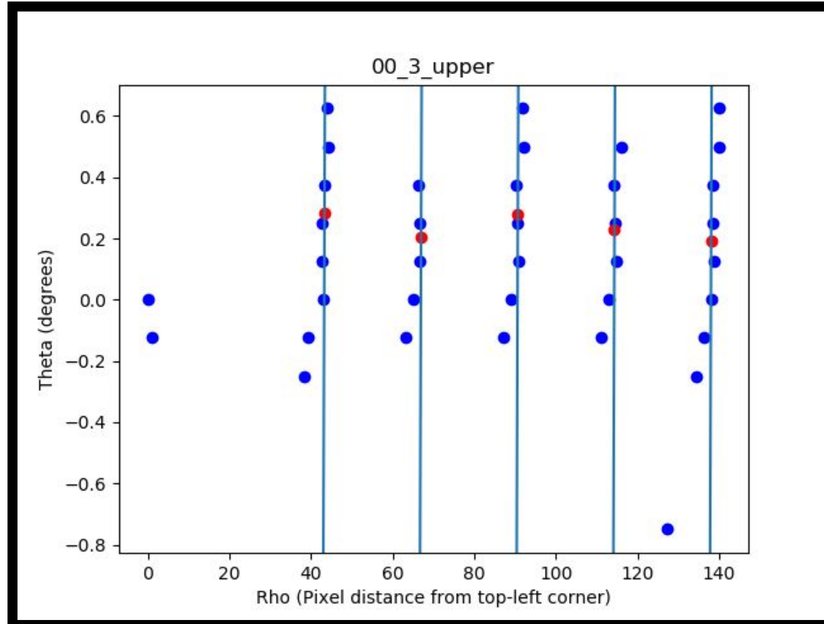
It would have been simple to do a column reduce on the image so that the values in each column are added up into a single column, and the lines can be found from detecting the highest values. However, the issue is that the lines are not necessarily straight. Due to imperfections in the original photo, mainly due to the curvature of the paper when the photo was taken, the staff lines are sometimes curved, so a different technique needs to be used.

A common way of finding lines within an image is to use `houghLines` function from OpenCV, which tries to find white lines by testing many different lines within the image. The issue with this method is that even if only a single line exists, multiple hough lines can be detected due to the nature of the algorithm. Image 3.3.3.4 is an example of this.



3.3.3.4

When there are multiple lines that represent the same line in the image, these lines need to be consolidated. One way to represent these lines is in dual space, where each line is turned into a point, and each point's coordinates relate to a line's angle to the horizontal and the line's distance from the top left corner. Image 3.3.3.5 shows an example of these dual lines being plotted as blue points.

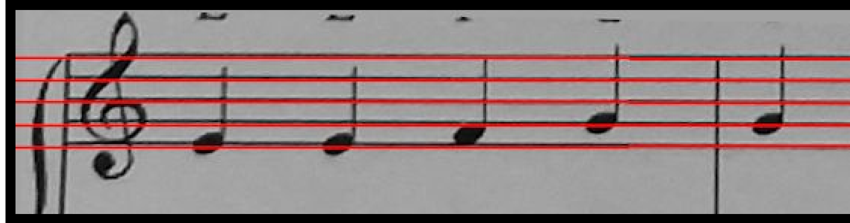


3.3.3.5

A clear pattern can be seen in the dual space where multiple nearby dual lines arrange themselves in a line in dual space. It was observed that any of the strings of dual lines in this dual space represents an actual staff line in the image. Thus, an average of these strings of dual lines needs to be turned into one average dual line, and the best five dual lines, equidistant from the top-left corner of the image need to be found.

We created an algorithm that would try different combinations of five lines of best fit through the dual space, and weight them by how many points were adjacent to them. We couldn't use classical statistics methods such as linear regression since linear regression will skew a line of best fit heavily towards outliers when what we needed for our algorithm was to be able to find a line that could be as near as many points as possible and weigh outliers less heavily. For example, we needed an algorithm that could ignore the two outliers near X of 0.

The technique of finding the five best lines by testing many different positions of five lines and weighing them in favor of how many dual lines they're close to seemed to work out well, so we stuck with it. Image 3.3.3.5 shows in red the five dual lines which would become the five lines in image 3.3.3.6.



3.3.3.6

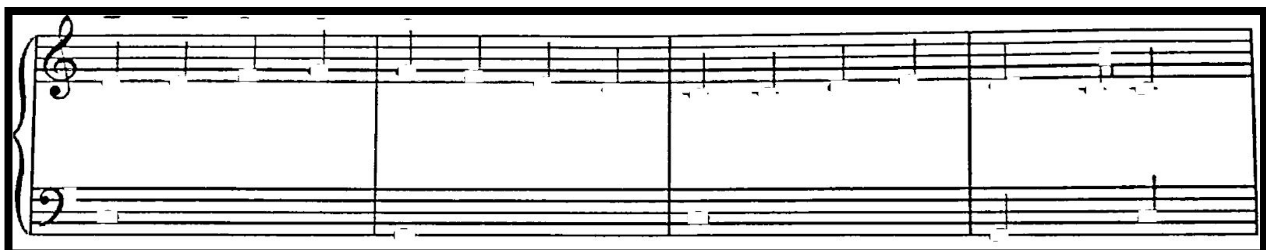
These five lines are passed to the next part of the process.

3.3.4 Note Detection

The third stage of the computer vision algorithm functions to find all of the music symbols located within a given section of sheet music. This stage takes in a list of ten lines found in the staff/bar identification phase of the computer vision algorithm. These lines are sorted by their y location, then based on the order of the lines are assigned pitch and octave, while also finding the average distance between staff lines. Once this is done, additional lines are added to accommodate the spaces between the lines and ledger lines in order to give a complete picture of what notes can be represented in the music.

3.3.4.1
This figure shows the lines passed into the note detection portion of the algorithm.

From this point, we are ready to start the search for symbols in the music. To find the symbols in the actual sheet music we use template matching. For each symbol type we use a variety of different templates to account for variations in the shape of the notes, for example, a quarter note can be between two staff lines or on a staff line cause a large difference in the overall shape of the symbol. We next search for each symbol type one at a time, this is for reasons mainly regarding actual notes which are found. Once all instances of a type are found we filter them down based on the overlap between every two notes so we only end with one instance of each note. Once we are done with a symbol type we then delete all of the found symbols from the thresholded section by simply drawing white squares over them as shown in figure 3.3.4.2.

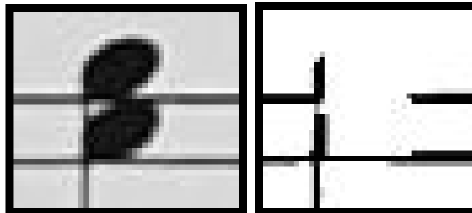


3.3.4.2

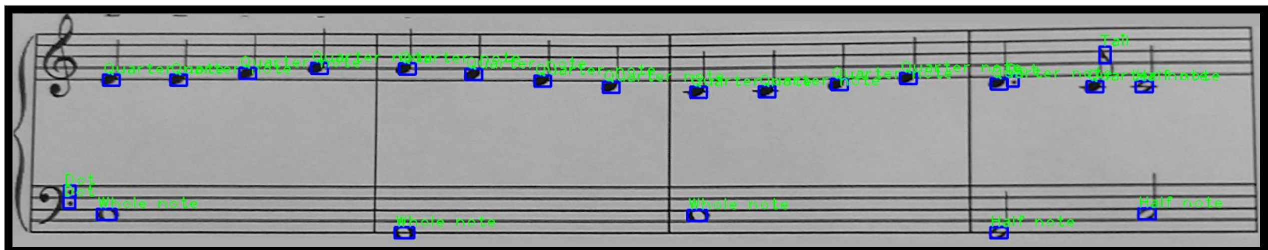
For searching each section and deleting the symbols we find there are several important considerations we must make. The first being the size of the image. This initially was a major obstacle to our computer vision algorithm. Images that were used for initial prototyping were very small in comparison to the high-resolution images taken by modern phone cameras. To mitigate this we use the already calculated average distance between the staff lines to find a scaling factor for each template.

$$scaling\ factor = (height\ multiplier \times template\ height) \div average\ space\ size.$$

We then multiply the height and width of each template by this scaling factor in order to have a properly sized template. The height multiplier is a value determined by the symbol which we are currently searching for. Note heads for quarter notes, half notes, and whole notes have a height multiplier of one since they are the exact size of each space, whereas quarter note rests for example span multiple spaces. The other factor which we must consider when searching for symbols is how deleting one symbol might affect another. When dealing with intervals of a second or a third in sheet music the notes will generally be the same value of note, and will be in contact with each other, shown in figure 3.3.4.3, this is why we must get all instances of each symbol type before deleting them from the image.



3.3.4.3 3.3.4.4



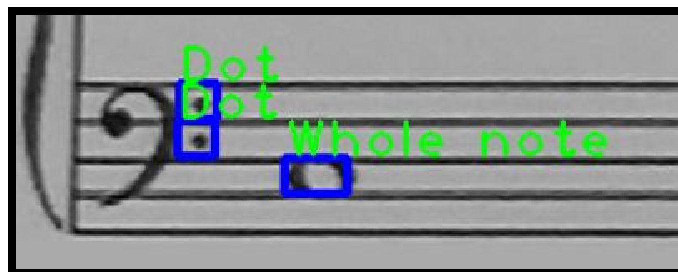
3.3.4.5

3.3.4.5 shows an example output of the note detection portion of the algorithm.

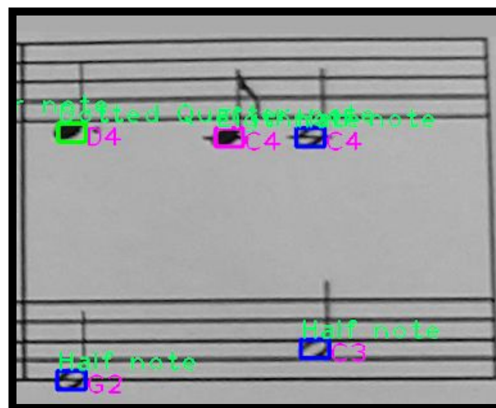
3.3.5 Note Mapping

The final step of the computer vision algorithm is to change the symbols found into a data structure which can be used by the front end application to play the given song and animate the piano accordingly. The first step in this section is to map combine any symbols which directly affect others such as tails for eighth notes, accidentals, and augmentation dots.

To do this we first go through all of the note symbols first and map them to the nearest line. The note then takes on the octave and pitch assigned to the line. Once we have all of our notes mapped, we then map our rests, when mapping half rests and whole rests we check their y location to ensure the rest was actually a rest and not some other shape that was picked up in the note detection phase that looked similar, this works since half and whole rests must always be in the second space from the top of their staff. Once we have all rests we then map the tails to change quarter notes to eighth notes, this is done by finding the closest quarter note to a tail that it is either above or below. At this point, accidentals and augmentation dots are applied to their respective notes. Accidentals are only allowed in an area before a given note, and augmentation dots are only allowed behind a given note. Figure 3.3.5.1 shows an example of points on the given image found to be augmentation dots that do not map to a note or rest, and therefore will not be included in the mapped notes.



3.3.5.1



3.3.5.2

Image 3.3.5.2 shows the output from mapping all of the notes. We can see that the quarter note head with an eighth note tail has been been switched to be an eighth note, and that the quarter note with an augmentation dot is now marked as a dotted quarter note, as well as all of the notes being marked with a pitch and an octave. Now that we have all of our found points mapped, we need to create our JSON for the front end to consume.

The JSON has the following structure shown in 3.3.5.3 is created by iterating through one clef at a time sorted by the x coordinate. The `time` field represents the starting point in terms of quarter notes when the note should be played, for example a note at time zero is the first note played in a section. The `duration` field for each note is based off of the type of note we have. A quarter note is one, so since a

half note is two quarter notes, half note's duration would be 2, and an eighth note would be ½. The volume is used to show if an object is a rest or a note, if the `volume` field is 100, then the object represents a musical note, if it is 0, then it is a rest. The overall JSON structure is a list comprised of lists of these notes, each inner list represents the music for exactly one section of music. All of these lists played in order will result in the entire song being played.

```
{
      "lists": [
            [
                  {
                        "Pitch": "F",
                        "Time": "0",
                        "Duration": "1",
                        "Volume": "100",
                        "Octave": "5"
                  }, ...
            ]
      ]
}
```

3.3.5.3

4 Testing

4.1 Testing Process Details

4.1.1 Front-end

The front-end of this project was tested with the rapid prototyping of the Expo library. This allowed us to update the code as we changed it onto the phone directly. As we began development from information coming from and to the server we used console logs in order to verify correct information flow between the two parts of the application. For the final step we did community testing to ensure that the user interface was appealing, intuitive, and fully functional.

4.1.2 Back-end

The back-end of this project was tested with the use of AWS Lambda's testing tools, embedded within AWS Lambda. A list of images is saved into the S3 server and are then read into the computer vision code on Lambda. The image processing code is run on each image and the results are posted back into a folder on the S3 bucket, complete with the output json for each one and the results in images for each song processed. These can then be viewed by us on the buckets to verify the computer vision code is being run properly on Lambda. As well as this, the REST API is tested using API Gateway's testing suite,

in which we can run each of the REST functions complete with parameters and test if requests are completing with proper responses, and the changes for each request is reflected on the S3 bucket.

4.1.3 Computer Vision

4.1.3.1 Preprocessing

The first two stages of testing were somewhat difficult to test, since there was no absolute “right” or “wrong” way to detect lines or sections, aside from obvious errors. During these stages of testing, output images were created to survey the different stages of the algorithm and how they were operating. Using these images, obvious errors could be picked apart, such as “An entire section wasn’t found” or “These 5 found staff lines aren’t horizontal”. When those obvious errors were fixed, so long as segments of the algorithm were within acceptable visual limits, it counted as passing. If the five detected staff lines looked pretty close to the five staff lines in the original image, then they were good enough. As these stages of the algorithm reached maturity, the obviously incorrect outputs were completely gone.

4.1.3.2 Note detection and mapping

To test the accuracy of the note detection and mapping portions of the algorithm we took results from the prior sections which were verified to be correct and used them as inputs. We then ran these portions of the algorithm writing all of our findings on the given images. From this point testing involved simply looking at the markings on each image to find symbols on the image that had been missed or incorrectly mapped keeping a tally for each symbol type to compare to the actual amount of that symbol present in the given image. Overall seven different pieces of sheet music resulting in 25 different sections of sheet music were used for the test.

4.2 Testing Results

4.2.1 Front-end

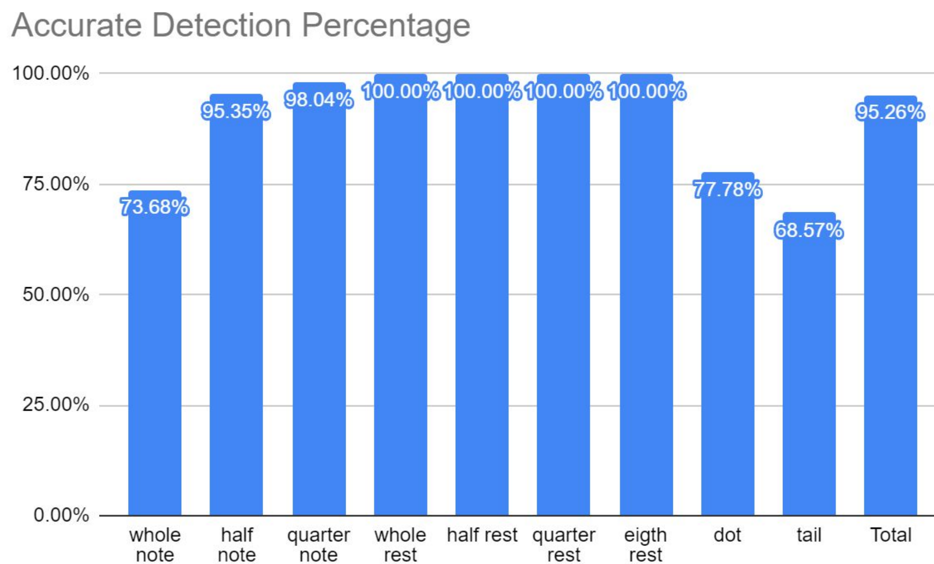
The results would show the accuracy of the notes we were pulling from the server in accordance to what we displayed and played on the piano. Other than accuracy and cleanliness of the application, we performed a community demo where we let some of our friends review and give suggestions to the layout of the application. This allowed us to have an unbiased review of how the application looked and let us make changes accordingly.

4.2.2 Back-end

The results for testing show accurate changes to the S3 bucket which reflect images being stored in the correct locations after sending properly formed PUT and GET requests on the AWS API Gateway Testing Suite. The results for the AWS Lambda testing suite show the results of the image processing algorithms on the S3 bucket as well, with completed requests with successful response codes as well as expected computer vision algorithm outputs, complete with images and output JSON files.

4.2.3 Computer Vision

In 4.2.3.1 we can see a breakdown by symbol type for how accurate our note detection and mapping is. The percentages shown are the symbols of each type accurately detected and mapped, over the total number of each type in all of the pieces of sheet music. However, there were very few whole, half and eighth rests present in the images selected for testing, resulting in a poor statistical representation of how accurate we are at detecting these symbols. It is worth noting that for false positives, or notes that were detected in places where they did not actually exist in the sheet music, we only had two through all of our testing which were located in the treble clef symbol.



4.2.3.1

5 Related Products

5.1 Sheet Vision

Sheet Vision is an existing computer vision of the same name as our project. Sheet Vision is also a computer vision project aimed at reading music from an image. However, there are several major differences in the projects.

1. The competing Sheet Vision project is an external library and not a standalone application.
2. The competing Sheet Vision works on a single staff of computer generated where our project works on an image of a printed grand staff.
3. The competing Sheet Vision project only generates a MIDI file where our project uses a specialized JSON structure to animate and play on the app piano.

5.2 Musescore

Musescore is an application used for creating and playing of sheet music. The purpose of Musescore is for creating and playing sheet music. In order to play a piece of sheet music in Musescore a user must either find an existing transcription online in Musescore's format, or create their own, where as our project allows a user to simply take a picture. Musescore's mobile application also does not instruct the user on how to play the piece on piano, where ours does.

5.3 Playscore

Playscore is a mobile application which sets out to accomplish the same goal of reading sheet music from photographs. Playscore provides a wider range of supported music but has several problems. Playscore suffers from a poor user interface which provides no instruction as to how to use it, and is not intuitive. Our application provides a more user friendly interface as well as instruction on how to use our application. Playscore also has issues with the progress bar displayed on the sheet music by the app, it also only plays the music and does not help the user to play it. Again our application provides piano animations to help instruct the user on how to play the music.

6 Appendices

6.1 Appendix I - "Operation Manual"

The setup of our project is a very quick and easy process.

1. To install the application:
    a. Open the github link "https://git.ece.iastate.edu/sd/sddec19-13" in your browser.
    b. Clone the github to a location of your choosing.
    c. Use terminal or bash to change to the project's directory.
    d. Then type "cd sheet-vision" to enter the project's folder.
    e. In terminal type "npm install" to install all necessary packages.
    f. If there are warnings with the build, run "npm audit fix".
    g. Now the project is set up and is now ready to be able to be run.
    h. On the phone device you plan to use, download the Expo application from the Play Store or the App Store.
2. To start the application:
    a. In terminal enter "expo start --tunnel".
    b. This will bring up a tab on your browser with a QR Code on it.
    c. On your mobile app, go to the Expo application and scan the QR Code.
    d. Then the application will begin loading to on your phone.
3. To use the application:
    a. After it finishes building up, you will be able to see the Home screen.
    b. Then you will go to the camera section on the bottom.

c. And from there you will be able to take a picture or choose a picture from your camera roll of a piece of sheet music.
d. After the processing is done, you will be notified that the song is ready to play.
e. Go to the Piano section and then press “Play Song”, you will be able to see and hear how the sheet music should play & sound like in the piano.
f. NOTE: You can change the speed of how the song is being played by changing the bpm and there will also be a “?” button where you can see what bpm means.

6.2 Appendix II - “Initial Versions of Design”

6.2.1 Sheet Vision Desktop

The original vision was to have the application be fully multi-platform, this would include a desktop application that would be usable in Windows, MacOS and Linux. This has been since been scrapped from the final design. This is mainly due to the need for maintenance and extensive testing for 5 different platforms that we’re being targeted. For this implementation we used a stack of ReactJS and ElectronJS to program the desktop client. This implementation was working and ready to move onto tweaking stages for development after the piano was implemented, but we decided to move on from this, since it wasn’t the best way to represent the application, we wanted to focus on the mobile implementation.

6.2.2 AWS Previous Backend Designs

6.2.2.1 EC2 + Apache

At the beginning of this project, we started up an ubuntu box and installed Apache and OpenSSL on it to host the python code. This design was revisited on multiple occasions but later removed due incompatibilities with React Native after updates that happened to React Native over the Summer 2019, which makes React-Native deny all requests that aren’t received from servers with CA’s that are on the phone’s trusted CA’s list, in which we would have to manually approve on every single phone that uses the app, which will not be possible.

6.2.2.2 API Gateway + Lambda

After scrapping EC2, we tried moving to API Gateway and AWS Lambda to host the serverless compute, this would let us host the OpenCV code without having to worry about a server, and since Amazon has a certificate that is authorized on most devices by default already, this removes this hurdle for us as well. The main problem that we had with AWS Lambda is that it would limit the sizes of requests we would be able to send to it, to 10 MB’s. The images that we were sending were constantly going over this threshold. We could compress these images before sending them, but for the Computer Vision algorithms to work properly, we need to give it the best images we possibly can, therefore, compression was not an option. To remediate this, we went with the current implementation of the serverless compute, which is API Gateway + AWS S3 + AWS Lambda.