Handwritten Music Reader

Ishwarya Ananthabhotla, Kelly Liu, Aaron Nojima
ishwarya@mit.edu, kellyliu@mit.edu, aanojima@mit.edu
Massachusetts Institute of Technology

Abstract

Currently most applications require usage of a special editor. We propose a simple solution that does not require such an editor and handles an image of handwritten music scores as input. While this solution is linked to optical character recognition it requires different techniques applicable to music notation and semantics. We designed a three-stage system architecture that will separate stave lines, segment symbols, and classify before outputting a data model that can be used by other applications. We only support the fundamental musical symbols but overall we were able to generate a near-perfect comprehension of some handwritten scores.

1. Introduction

For musicians and composers, handwriting music is much more intuitive than the usage of special editors. However, everyone has different handwriting styles which means that eventually their sheet music needs to be converted into a digital format. In order to convert to digital, most musicians use editors to input their music. However, these editors are often quite different in usage compared to writing by hand.

Currently, desktop and tablet/mobile music editor applications can currently handle the conversion of some user-input into digital sheet music. While both types of applications are advantageous in some aspects, they fail to re-create the native comfort of composing with pen and paper.

For desktop and web applications, user-input usually depends on using the mouse and keyboard to create or plot musical symbols. For even just the fundamental building block, the note, the user must either always enter a note's pitch and duration via keyboard or drag and drop a note object onto stave lines. Not only are these slightly more difficult and tedious tasks (as they probably take a longer time to enter in each note and will require the user to learn new shortcuts), but they also feel less natural as opposed to handwriting.

Tablet applications are arguably better than desktop ones since the mouse and keyboard are now replaced by touch capabilities. With such features, users can now emulate drawing on capable devices. With regards to music recognition software, users can draw all sorts of musical symbols and ideally the program should be able to convert the strokes into a digital output. Some tablet devices also support the usage of electronic pens for higher precision and accuracy for every stroke and also giving a near-similar experience to using a pen and paper. However, this is only available for users with the aforementioned hardware. Furthermore, even though the electronic pen is a good simulation of handwriting, many experienced composers would still prefer to not write on a glass screen.

2. Related Work

Our application falls under optical character recognition (OCR). The goal of OCR is to recognize an image of characters as something a computer or machine can understand. The characters in OCR can be the characters of a language's writing system, digits, and symbols to name a few.

Optical music recognition (OMR) is tightly linked to OCR but commonly used OCR techniques cannot be used in this specific type of recognition since the reading of letters and words is very one-dimensional. Music on the other hand is two-dimensional (pitch on the vertical axis and time on the horizontal axis) but in reality it can be even more complex than that when other specific symbols are included as recognizable characters. Another issue specific to music recognition is the overlapping and combination of symbols such as with the five stave lines. This adds an extra processing step before common OCR algorithms can be applied. If we take into account parallel parts in a music score (different clefs), symbols

for dynamics, articulations,etc., OMR becomes quite a challenging topic.

Within OMR specifically, other researchers have done a lot of work designing systems and algorithms that would be best for an OMR application. We consulted several other publications regarding the design and implementation of our program.

One paper written by a group from Brown University describes an application titled MusicHand [1]. They provide a lot of information regarding various musical symbols and even provide mathematical equations that characterize the different symbols. However, they described their system for use on an editor as opposed to using pen and paper. Although this system is very accurate and provides insight based on stroke patterns, we want to stick to our native handwritten application since it is more intuitive and requires less symbol-specific algorithms.

In another paper written by a group from the Northwestern Computer Music School of Music, the authors describe the challenges of physically scanning handwritten music scores [2]. They mention issues such as viewing the image at a tilted angle (as opposed to directly above) and describe some methods for image adjustment. While we most likely will not be concerned with image correction for our first iteration of our application, we will reference their methods in the future.

Finally, we consulted the work done by a group from the the COMSATS Institute of Information Technology [3]. In their publication, they describe a simple work-flow behind OMR. They also provide many alternative algorithms and methods to extend their system to all types of constraints (e..g: performance, prioritization). We decided to closely follow their work-flow and implement algorithms and other techniques found in these papers.

3. Approach and Algorithm

Our program has three main modules that take the input image and eventually generate a data model for each symbol (and one for the overall sheet music). Given an image of handwritten music we first extract the image into symbols and stave lines. Then using the image with only symbols, we find the bounding boxes for every symbol. For each symbol image, we pass it into the symbol classifier which will identify the type of symbol. Once we know the symbol type, we use further identification methods to determine more specific information and data. For accidentals (including key signatures), and notes we need to reference the stave lines image to determine which pitch the line is associated with. After all symbols have been fully classified, the constructor creates an appropriate data model for each symbol and groups them together into one sheet music data model.

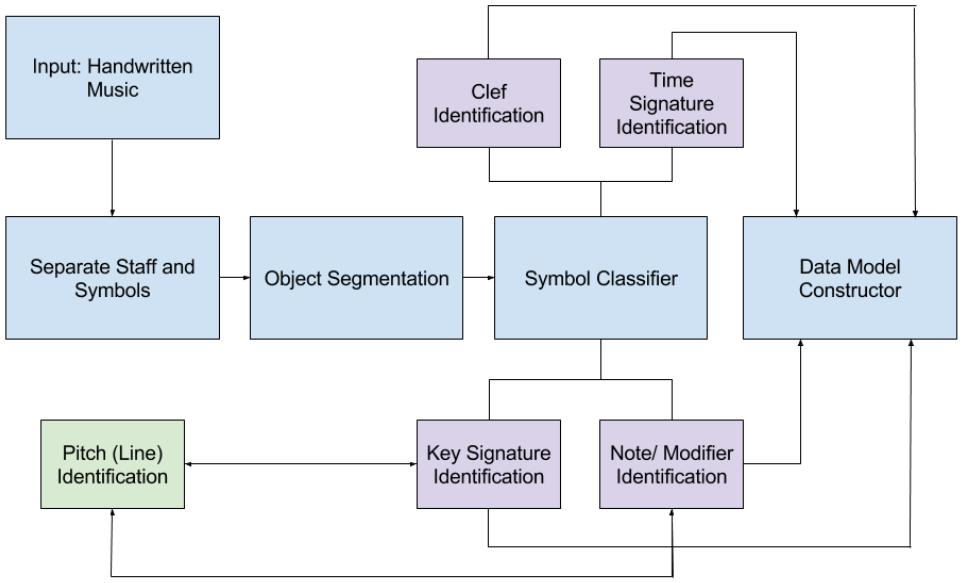


Figure: Architecture and basic work-flow of our program

3.1. Input

Our application takes in handwritten images of sheet music. We handle some number of key symbols including clefs, accidentals, time signatures, and notes (eighth, quarter, half, whole). We recommend using printed stave lines for better results. The paper should be white computer paper (blank) and all symbols should be written with black ink or at least a very dark color. When the user takes a picture of the sheet music, he or she should set the position of the camera directly above, or at least near above, the plane of the paper. The user should avoid casting any shadows when taking the picture although since we use a binary threshold on the input image, we can handle small degrees of different lighting. In addition the user should focus on the row of interest and try to avoid any unnecessary white padding from the unused computer paper. For now we only support sheet music on one row (as opposed to having multiple rows spanning the whole page). This specified image is the input to our program.

3.2. Stave Processing

The first step of our program is to process the image by converting any image to gray-scale and applying a binary adaptive threshold on the image to set all pixels to have a value of either 0 (black) or 255 (white). From here, we separate the modified input into two different images: one containing only the musical symbols and another containing only the stave lines. To do this, we used the fundamental morphological operators erosion and dilation.

For extracting the images, we need to create an appropriate filter. We used a horizontal row with only ones (a 1 by n matrix with all 1's). For extracting the vertical image, we used a vertical column with only ones (a n by 1 matrix with all 1's). By first performing erosion, we eliminate all any horizontal group of pixels less than a certain width (based on the size of the horizontal filter). We then perform dilation to restore all horizontal lines that passed through the first stage back to their original shape. Similarly for the vertical image we first eliminate any vertical group of pixels less than a certain height (based on the size of the vertical filter) and restore the intermediate image.

Since this module essentially contains a pixel mapping for the stave lines, it also handles pitch (line) identification for musical symbols given its absolute bounding box coordinates and dimensions. We will use line indices to represent stave lines with '0' being the bottom line, '1' being the space in between the bottom line and the one directly above. The top line will have an index of '8'. We currently do not support notes for lines going off the base stave lines.

Clefs and the numbers in the time signature are ignored since their vertical position does not affect its semantic value. However, we need to identify the correct line for all notes and accidentals (and therefore covering the key signature). The key idea here is to find a point for each symbol that will be used to identify which line or space it should fall under. For symmetrical and centered objects such as sharps, naturals, and whole notes, we can simply take the center of the bounding box and find the closest line or space. However, for symbols such as all other notes and flats, this may not necessarily be the case. We do know however, that the point of interest is in the center of the note's head (the circular object). Therefore, we use a Hough circle transform to find circles within the symbol image. We used the center of this circle to identify the line this note or flat falls under.

3.3. Object Segmentation

The next module segments the symbol image to find individual bounding boxes for each musical object. First, we apply a binary threshold on the image containing only symbols. Next we find the contours on the image, the contours being a set of points such that all continuous or adjacent points have the same color or intensity. Since we've applied a binary threshold to our symbols image, this will find all groups of isolated pixels which should correspond to all the musical objects.

Once we have all of the contours we need to find a bounding box for each one. When using the bounding box coordinates and dimensions alongside the image inputs, we can create small images for each symbol which will prove useful in line identification and symbol classification. However, some musical objects with very thinly drawn lines may become separated after performing contour finding. To counter this issue, we merged iterated through all of the bounding boxes and checked for any significant overlap. Should any two overlap significantly, we would combine the two into one bounding box. We would often times need to perform this step for complex drawings such as the treble clef. Finally, since music is read from left to right, we wanted to return a list of the bounding boxes in the same manner and sorted the boxes in ascending x-coordinates.

3.4. Symbol Classifier

The symbol classifier does a preliminary classification of a given musical symbol by checking its order. For example, since we know that the objects are given from left to right, we can safely assume that the first object will be the clef followed by the key signature, time signature, and then the notes (possibly preceded by an accidental). These assumptions are based on actual composition styles. The symbol classifier will assign a 'type' to each bounding box which varies from 'CLEF', 'KEY_SIGNATURE', 'TIME_SIGNATURE', 'NOTE', and 'ACCIDENTAL'.

Once we know the 'type', the symbol classifier will use a KNN algorithm to output a specific label for each musical symbol. We have obtained training data on the Internet for classifying digits and generated our own handwritten training data for eight, quarter, half, and whole notes as well as flats, sharps, and naturals. Prior to iterating through the musical symbols, we will train our each KNN model (specifying k as 3).

For clefs we only support treble at the moment but can easily extend this to bass and alto clef later on once we obtain enough training data. For accidentals (as modifiers and key signature components), digits (in the time signature), and notes, we first extract the hog features from the symbol image. Using these features as the input vector and the pre-trained KNN models, we label each musical symbol. Below are the following labels (as integers) we reference in our application.

NoteLabel:
  WHOLE, HALF, QUARTER, EIGHTH = 1,2,4,8
ClefLabel:
  TREBLE, BASS = 11, 12
TimeSignatureLabel:
  COUNT, TYPE = 21, 22
AccidentalLabel:
  FLAT, NATURAL, SHARP =31,32, 33

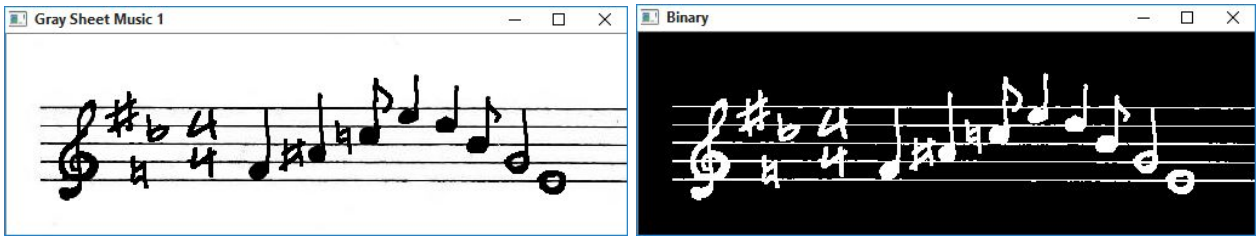
3.5. Output

The output for our of our program is a sheet music data model. The sheet music contains a clef data model, key signature data model, time signature data model, and a list of note data models. For the clef data model, we currently only support treble clef. For key signature, the user must specify the number of accidentals since we currently have not yet implemented any clear method for clarifying when the key signature ends and the time signature begins. The key signature simply keeps a list of accidental data models, each of which further contain a type (flat, natural, sharp) and pitch (in this case just the line index). The time signature data model contains two values: one for the type of note and another for the count per measure, both of which can be represented as integers. Finally, each note data model contains a pitch (in this case line index), type (length, e.g.: eight, quarter, half, whole), and a modifier if any should apply (this is an accidental that overrides the key signature). For the overall sheet music data model, we currently support display functionality which prints out all necessary information for the component data models.

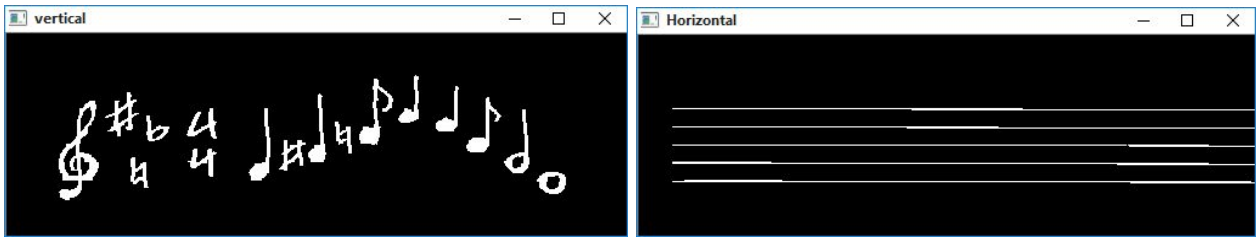
4. Experimental Results

In this section we will show the outputs from each module

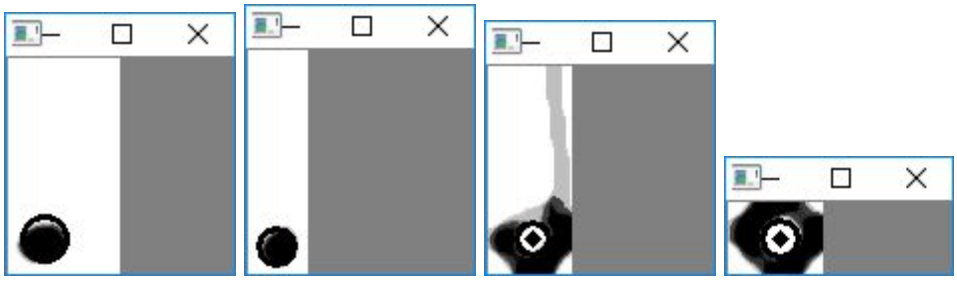
4.1. Processing/Staves Module Results



Figures show the original gray-scale image (left) and the output of an binary adaptive threshold (right)



Figures show the extracted symbols or “vertical” image (left) and extracted stave lines or “horizontal” image (right)



Figures show (from left to right) the Hough circle transform identifying eighth, quarter, half, and whole notes

4.2. Object Segmentation Results

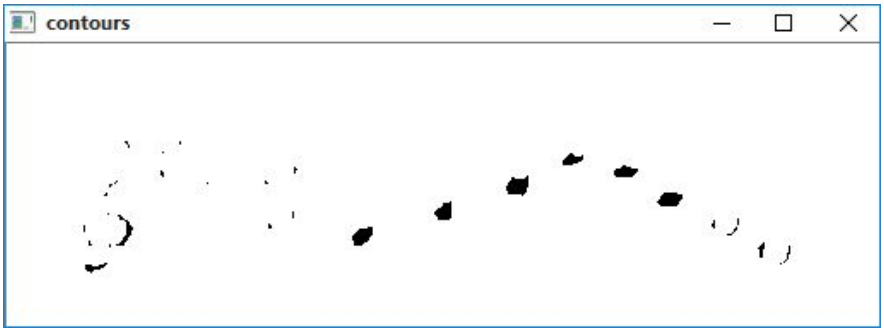


Figure shows contours found in the symbols “vertical” image

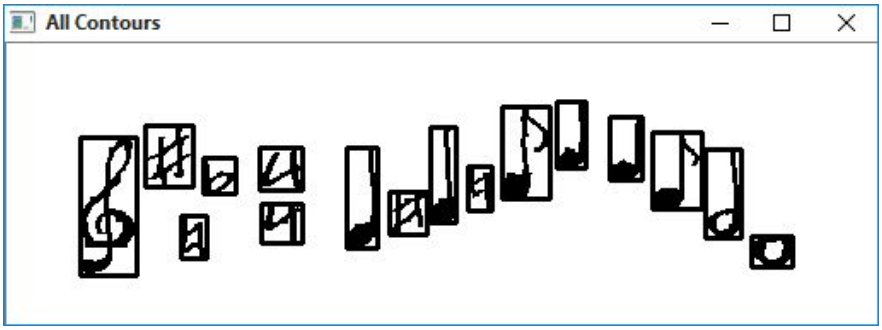


Figure shows bounding box formed for all separate symbols
Note: this is after performing flood-filling and merging of overlapping bounding boxes

Successes: In general there hasn’t been an object that hasn’t been fully assigned a bounding box
Errors: However, we occasionally had some modifiers merged with notes or submerged bounding boxes

4.3. Symbol Classifier Results

Since the symbol classifier assigns labels, it is more appropriate to just show the final result (the data model) from the output of our program:

CLEF - TREBLE
KEY SIGNATURE - [TYPE: 33, PITCH: 8; TYPE: 32, PITCH: 2; TYPE: 31, PITCH: 5;]
TIME SIGNATURE - COUNT: 4.0, TYPE: 4.0
NOTE - LENGTH: 4, PITCH: 1
NOTE - LENGTH: 4, PITCH: 3, ACCIDENTAL: 33
NOTE - LENGTH: 8, PITCH: 5, ACCIDENTAL: 32
NOTE - LENGTH: 4, PITCH: 7
NOTE - LENGTH: 4, PITCH: 6
NOTE - LENGTH: 8, PITCH: 4
NOTE - LENGTH: 2, PITCH: 2
NOTE - LENGTH: 1, PITCH: 0
[Finished in 6.1s]

The only errors in this test result are in the key signature (two of the pitches are off by 1 index) out of 26 key characteristics (92.3% correctness)

5. Conclusion

Our system is capable of handling the core basics of music score notation. We wanted to start off with a small collection of symbols and test our program's robustness before moving on to more complicated tasks. Overall, the results proved fairly successful for a variety of symbols, of course under our given constraints, but there were also issues that we need to work on before moving on. We have not done thorough testing of our program but merely came up with a few scores that included various characteristics we wanted to test for. In the future, we need to compose more handwritten sheets and see how our program can handle all of them. This should account for even more test cases and different handwriting styles.

Once we finish any issues that may arise from this testing, the next phase is to add more symbols. In musical composition, we have barely scratched the surface with what symbols we support. Given that music composition should be able to capture the creativity of its artist, we need to extend our classification algorithms to recognize other unique symbols. This also means that we need to enhance our data model output to include complex music semantics. Finally, we would like to add more functionality to our data models in general. Our current data model structure contains all the necessary information but we would like to be able to render neatly computer-drawn scores or even play the song back by compiling some audio file based on an initial image input.

6. References

[1] G. Taubman. MusicHand: A Handwritten Music Recognition System.
[2] A. Wolman, J. Choi, S. Asgharzadeh, and J. Kahana. Recognition of Handwritten Music Notation.
[3] Q. Arshad, W. Khan, Z. Ihsan. Overview of Algorithms and Techniques for Optical Music Recognition.