MAGICAL MUSIC DECODER

Final Report
for
Senior Design Project

Selda Küçükbiçmen
Ali Uygar Küçükemre
Ahmet Oğuz Akyüz
Kemal Taşkın
1. INTRODUCTION 2
2. LITERATURE SURVEY 3
2.1 OMR 3
2.2 ABC 3
3. WORK DONE 3
3.1 Pre-Processing 6
   3.1.1 Deskewing 6
   3.1.2 Threshold 7
   3.1.3 Noise Removal 7
3.2 Segmentation 8
   3.2.1 Stave Line Removal 8
   3.2.2 Object Location 9
   3.2.3 Primitive Identification 9
   3.2.4 Sub-Segmentation 11
3.3 Post-Processing 12
   3.3.1 Post Identification 12
   3.3.2 Header Extraction 14
   3.3.3 Conversion 15
4. INTERFACES 16
4.1 Simple Mode 16
4.2 Advanced Mode 17
4.3 ABC Editor 18
4.4 MIDI Player 18
5. TEST RESULTS 18
6. CONCLUSION 19
7. REFERENCES 20
APPENDIX A (CONVENTIONS) 21
APPENDIX B (SOURCE FILE ABSTRACTS) 22
Abstract

For those who are interested in pattern recognition and image processing optical character recognition is a much known topic. It simply deals with the recognition of handwritten characters. There are many studies in this topic and hence many efficient algorithms have been proposed up to this moment. Optical music recognition, on the other hand, is a sibling of OCR. Unlike OCR it deals with musical scores. However there are not as much studies on optical music recognition as OCR. The reasons for this are rather economic and social and they are beyond our scope.

This study is proposed and this project is managed to investigate the features of OMR and to develop algorithms in this context. Magical Music Decoder is a product of 8 months of work. The analysis, design, progress reports and proposal, progress and final presentations are present in http://www.freewebs.com/kemaltaskin/mmd.html.

With the purpose of developing new algorithms we followed the basics of a standard OMR process and produced our own approaches to the problem.

1. INTRODUCTION

Magical Music Decoder is an image processing and pattern recognition tool for Windows family, which basically constructs sound files in MIDI format from sheet music in JPEG format. Besides its extra features such as the construction of the MIDI files or the graphical user interfaces, the process of MMD can be considered as optical music recognition.

MMD basically traces the steps of OMR and at the end it generates a list of items to be printed sequentially. The algorithms used up to this step are described in detail in the “Algorithms” section of this paper.

Since the main problem is to identify the items in the image, the rest is to interpret the items (which are stored in a list) into a well-known format. We chose ABC music notation language for this purpose. What we did was to construct an ABC file with information stored in our list. We embedded free tools from ABC homepage [1] into the code in order to construct MIDI and PS files from the ABC files.

We always thought about MMD that it should not only be a hard-to-use spaghetti program, but it should have a smart interface making it a user friendly application. In order to achieve this goal, we constructed a fancy graphical user interface which is mentioned in this report later.

The rest of this document is as follows: Second chapter describes the literature survey done at the beginning of the design time. Third chapter is the core part of the document and describes the algorithms developed with their pseudo codes and difficulties. Fourth chapter is a visual part with the screenshots of the program that describes the usage of it. In the fifth chapter we give the test results of our program in various data. The main body of the document ends with the conclusion and reference chapters. In appendix A we mentioned
about the conventions we used during our project and in appendix B we refer to the source files and their ingredients.

2. LITERATURE SURVEY

2.1 OMR

OMR (Optical Music Recognition) is the process of extracting musical information from a 2D matrix obtained from an image. It is a fact that OMR is very similar to OCR (Optical Character Recognition). The major differences of OMR is the existence of stave lines, its sensitivity against errors and the existence of some dependencies to the context which is in the nature of note interpretation standards. One of these standards is the choral effects, in other words the representation of more than one melody in a single line of note. Another one is the walkthrough of the melody according to the predefined forms of music. This kind of information is very hard even with the knowledge of the context.

There are a few studies in this area. One of the most important study (a program, though) is Gamera [2]. Gamera is an OMR program implemented in Python [3]. ROMA is another one [4].

From the studies present, one can write out the mechanism of the process. OMR has straight forward steps each dealing with the matrix one after another [4]:

- Pre-processing the image
- Removal of non-musical symbols
- Identification (and removal) of staff lines
- Object segmentation
- Object assembling

These steps are included in MMD also, and they will be described in detail later.

Another common property of these studies is that they are all dedicated to classical western music. Up to this point there are no pattern classifying tools in the area for mid-eastern music such as classical Turkish music. Unfortunately MIDI format does not also support the specific characteristics of this music. The revision of MMD for classical Turkish music will possibly be our next goal.

2.2 ABC

ABC is a very popular note interpretation language. Any type of musical note can be interpreted with its level of context independency. As mentioned before, there are also open source libraries dedicated to ABC. Some of them are ABC to ABC, ABC to MIDI, MIDI to ABC, ABC to PS routines. One may easily find more information about ABC and these libraries from the official web site of this language.
An ABC file has two parts: Header and Body. Header part consists of general information about the note. The following example describes some common fields of this part:

X: 1 % The number of the song
T: Moonlight Sonata % The title
M: ¼ % The measure
L: ¼ % Default note length
Q: 120 % Tempo
Z: Kemal Taşkın 27.05.2003 % Author
K: G % Key

There are also optional fields which can be included to have more fun.

In the body, there are notes (with their lengths, accidentals and other specific signs), rests, measure bars, loop signs (dots), choral effects, chords and other musical effects.

Notes are represented as “c d e f g a b” letters. Rests are “z”. Other symbols and different forms of notes can be achieved from the web site again. A sample note is as follows:

X: 1  
T: Gökyüzünde Yalnız Gezen Yıldızlar  
C: Music: Teoman Alpay, Lyrics: Münir Ebcioğlu  
M: 4/4  
L: 1/8  
Q:1/4=160  
K:Bb % 2 flats, In Turkish Classical Music, it is Nihavend Makamı  
% Time signature=4/4 MIDI-clocks/click=24 32nd-notes/24-MIDI-clocks=8  
% MIDI Key signature, sharp/flats=-2 minor=0  
zB2AB3/2A/2G2 | zc2=Bc3/2_B/2A2 | zbagfgef | d8 |  
zg2fg3/2f/2ge | zf2=ef3/2c/2fd | zgfedec | BcABG4

The corresponding figure is as follows (Figure 1).

![Figure 1: An output of ABC formatted song.](image)

3. **WORK DONE**

The algorithms proposed in this document are the body of MMD. They are all implemented in C language and tested by over 50 images. The source codes are also available for those who are interested in improving MMD.

MMD core process consists of three major steps:
- Pre-processing
- Segmentation
- Post-processing

While the most important part is the segmentation, the most critical part is the pre-processing. Pre-processing is the process, in which the image (2D matrix, formal speaking) is prepared to be segmented. This process has three steps: deskewing, threshold, and noise-removal.

After the pre-processing we still have a 2D matrix at hand, however at this point it is a deskewed noise-free binary matrix. So begins the segmentation part. This part consists of four steps: stave line removal, object location, primitive segmentation, sub-segmentation. Stave line removal is the step that the stave lines removed without touching the objects. This is a crucial part that if the objects are cut during this process, the retrieval becomes very hard. The coordinates of these lines are also stored at this point. This is also important since this information will be used in “note identification” step later.

After the stave line removal; the objects are located in a trivial algorithm, the located objects (which are now in a list) are identified to be a primitive or a compound (of note heads and note bars), primitives are identified, compounds are segmented and the segmented compound parts are identified. Finally the list is sorted and given to the post-processing part as input.

Post-processing is to retrieve information from the sorted list of symbols and interpret them. It has three steps: note identification, header (information) extraction and interpretation. First two steps consist of passes over the list. They continuously update the note information and shrink the list if necessary. The aim is to leave the list consisting of symbols that are sufficient to begin interpretation.

Below, the algorithms are presented (with pseudo codes if necessary) and the difficulties are mentioned.
3.1 Pre-Processing

3.1.1 Deskewing

Some images may have unwanted properties which make them difficult to process. One of them is skewing in the optical musical scores. That is, the stave lines may not be horizontally aligned. The most important reasons for skewing in images are user errors when scanning a note page with a scanner. Therefore deskewing is crucial part in optical music recognition and will be described briefly.

Deskewing has two major parts. These are skew angle detection and calculation the new (deskewed) positions of the pixels according to this skew angle. This section describes the algorithms of these two parts respectively.

\[
\text{CalculateSkew(Picture) = \text{LeftReferencePoint}(x_{\text{left}}, y_{\text{left}}) - \text{RightReferencePoint}(x_{\text{right}}, y_{\text{right}}) / (x_{\text{right}} - x_{\text{left}})}
\]

\text{Left Reference Point} \quad \text{Right Reference Point}

\text{Figure 2: A Skewed note sample}

Now, let’s describe how these reference points can be found:

\[
\text{LeftReferencePoint}(x_{\text{left}}, y_{\text{left}})
\]

\[
\quad \text{Goto first stave line in the image}
\]

\[
\quad \text{While (on the same same stave line)}
\]

\[
\quad \quad \text{Goto 1 pixels left}
\]

\[
\quad x_{\text{left}} = \text{CurrentPixel.x}
\]

\[
\quad y_{\text{left}} = \text{CurrentPixel.y}
\]

Obviously detailed calculations are required to go to the first stave line in the image and to detect if we are on the same stave line or not. But the algorithm mainly tries to acquire the leftmost point on the first stave line. The same calculations are done for the RightReferencePoint but this time instead of going toward left, we go towards right to detect the leftmost point in the stave line.
Once the skew angle is calculated next thing to do is actually deskewing the image by that angle:

\[
\text{Deskew}(\text{Picture}, \text{angle}) \\
\begin{align*}
x &= x' \cos(\text{angle}) - y' \sin(\text{angle}) \\
y &= x' \sin(\text{angle}) + y' \cos(\text{angle})
\end{align*}
\]

forall \([x', y']\)

\[\text{Picture}[x', y'] = \text{Bilinear}(x, y)\] //Bilinear interpolation

The reason for choosing bilinear interpolation in pixel transformation is that, its efficiency/reliability ratio is better than any other algorithm such as (bicubic interpolation or simple averaging).

In deskewing part one more operation is applied to the image. That is trimming the unnecessary white spaces at the left, right, bottom and top of the image. This operation shrinks the size of the image so in the following steps image processing becomes faster.

3.1.2 Threshold

Thresholding is a process in which a grayscale image is binarized. There are several thresholding algorithms. We preferred to combine two of them:
- Iterative Thresholding
- Average Thresholding

We saw that with iterative thresholding we lost too much of the symbols’ pixels, so we decided to combine this threshold with the result of average thresholding, which leaves more pixels black. The weights of the two thresholds can be arranged by the user in the advanced mode of the program.

\[\text{Algorithm:}\]
\begin{align*}
\text{Calculate Average Threshold} \\
\text{Calculate Iterative Threshold} \\
\text{Threshold} &< \text{- average threshold} \times w_1 + \text{iterative threshold} \times w_2 \text{ (where } w_1 + w_2 = 1) \\
\text{End}
\end{align*}

3.1.3 Noise Removal

After deskewing and threshold, there occurs noise in the image. This noise is from the lack of accuracy in deskewing and threshold. Without noise removal all of this noise would be sent to segmentation which would probably lead to a huge list of objects and time inefficiency of the segmentation algorithms. So, we have to remove the noise pixels.

Our noise removing algorithm is trivial. A 5*5 matrix traverses the image and removes the pixels which have only two black pixels when they are the center of the mask.
Algorithm:

For all the pixels of the image
   Traverse with a 5*5 mask
      If there are less than 3 black pixel in the mask whiten the center pixel.
   End

There are no problems related to this algorithm because there is no need for 100% success here. This algorithm is applied only to shrink the list of symbols by deleting the noise pixels.

3.2 Segmentation

3.2.1 Stave Line Removal

Segmentation process begins with removal of the staff lines which makes the segmentation of the musical primitives possible. The key point here is storing the coordinates, relative positions and the thicknesses of the stave lines. This issue is important because later both in recognition and interpretation phases, the stored data about the stave lines take a vital role. Another important thing when removing stave lines is not to cut the primitives that stave lines is passing on. Because an action of that kind will make recognition of the cut primitive almost impossible. In that aspect, while removing a pixel that is found to be a member of any stave line, a connectivity analysis must be done in order to understand whether that pixel is also owned by any primitive. If that is the case, the pixel in consideration must not be cleared.

Thus taking all those into account, algorithm used in the stave line removal is as the following:

1. Take the horizontal projection h(i) of the image f(i,j)
2. For each h(i)>0.7* ImageWidth mark i as horizontal peak
3. Assign directions to each peak using the following rule
   if one independent peak, give 2
   else if two successive peaks, upper=0, lower=1
   else if more than 2 successive peaks,
      uppermost=0,lowermost=1,others=-1
4. For each peak, trace the corresponding columns, 
   if direction=-1 remove the pixel
   else if direction=0 then if no pixel in upper 3 neighborhood
      clear the pixel, else keep.
   else if direction=1 then if no pixel in lower 3 neighborhood
      clear the pixel, else keep.
   else if direction=0 then if no pixel in upper and lower 3 
      neighborhoods clear the pixel, else keep.
5. Store the stave line positions
6. Calculate stave line width and inter stave line gap by using stave line positions

3.2.2 Object Location

Object location is one recursive algorithm that draws imaginary boxes around the objects. The limits of the boxes are extracted with 4-neighborhood. Algorithm searches for a black pixel in the image and when it finds it recursively enlarges the box and updates the limits if necessary.

Algorithm:
For all black pixels in the image
   Create a box with all limits equal to its coordinates.
   Paint the pixel with an auxiliary color.
   Append the 4-neighbors recursively to the box if they are black.
   Update the limits if over limit valued pixels are added.
End
For all auxiliary colored pixels in the image
   Paint pixels into black
End

No known problems in this algorithm.

3.2.2 Primitive Identification

The main purpose of Recognize.cpp is to identify the type of a symbol by calculating its characteristics that may distinguish it from other symbols. The data used to calculate further characteristics contain the minimum and maximum coordinates of the symbol in the image. Before entering this part, the inspected object is painted into a specific color so that we are sure that we do not consider any pixels from another object that may have some part in the box of the currently inspected symbol. Some basic characteristics that are used frequently for almost all symbols during the comparison are calculated in advance and placed into the symbols array for each object, while others are calculated on need. The final aim is to specify the type element in the structure. The content of a symbol structure is as follows:

symbol:

min_i, max_i: vertical border coordinates of the symbol in the image
min_j, max_j: horizontal border coordinates of the symbol in the image
h, w: height and width of the image in pixels
weight: sum of pixels of the object (the black pixels in the object box)
h_over_w: the h/w ratio
w_over_h: the w/h ratio
density: the ratio of the black
hp, vp: the dynamic arrays containing horizontal and vertical projections
hCenter the horizontal center of mass w.r.t. the with of the object
\textit{vCenter:} the vertical center of mass w.r.t. the height of the object \\
\textit{gapRatio:} the height of the object w.r.t. the gap size between the staff lines \\
\textit{type:} the symbol type assigned to the object \\
\textit{key:} the index of the staff-line group to which the symbol belongs to, which is used later to sort the symbols

\textbf{Most Important Functions for Object Identification:}

\textit{CalcVBlackPasses / CalcHBlackPasses} \\
Calculates the number of black passes when we pass vertically/horizontally through the passRatio part of the symbol \\
e.g. if symbol is a valid sharp, CalcVBlackPasses(symbol,0.5) should give the result 2.

\textit{CalcSymbolProp} \\
Calculates the most frequently used properties of the symbol which are all elements of the symbol structure except key and type.

\textit{LeftStraight / RightStraight} \\
Determines whether the left/right side of the symbol is straight or not. \\
e.g. LeftStraight of a flat symbol should return true.

\textit{VerticalPeak} \\
Determines whether there is a peak in the vertical projection of the symbol showing it is a compound note.

\textit{HorizontalHills} \\
Computes the number of hills of the horizontal projection of the symbol, that will determine the type of the rest \\
e.g. 1/8 rest has 1 hill, 1/16 has 2 hills.

\textit{ThinBottom} \\
Determines whether the symbol has a thin bottom or not, which shows that it may be a rest such as 1/8, 1/16...

\textit{AllVerticalPasses / AllHorizontalPasses} \\
Determines whether the whole part between the 'from' and 'to' of the symbol has vertically/horizontally at least 'min', at most 'max' black passes.

\textit{RecognizeMeasureNumber} \\
Recognizes the upper or lower part of the measure symbol

\textit{ProcessMeasure} \\
Divides the measure into an upper and lower part, creates two symbols and calculates their properties.
**CompoundSymbol**
Determines whether the symbol is a compound note mainly whether it contains vertical peaks or not.

**RecognizeSymbol**
This is the main function that assigns a symbol type to the symbol that is inspected. The function uses the properties that are in the symbol structure, together with the defined functions that calculate some specific properties that are important to distinguish that symbol from others. The function operates as a large decision tree. If an object satisfies all criteria assigned to a symbol type, it is assigned to that type, otherwise it is labeled as *Unidentified*.

### 3.2.4 Sub-Segmentation

After the first pass of the recognition all primitives like sharps, measure bars etc. are classified except notes. This is because of the nature of notes in musical representation. Since there are several kinds of notes like inverted notes, compound notes, etc. and compound notes can have infinite combinations among them, a further sub-segmentation of those primitives is needed. Therefore, at this point primitive subsegmentation starts. The unrecognized primitives (notes) that have been send by the Recognizer module is the input for Subsegmentation module. The algorithm of the Subsegmentation is the following: (Bounding boxes of the compound notes are taken in as input)

1. Find the width and height of the bounding box.
2. Find the vertical projection $v(i)$ of the bounding box.
3. Mark all peaks in the $v(i)$.
4. Calculate $\text{factor} = (2 \times \text{Stave Line Width}) + \text{Inter Stave Line Gap}$
5. If $\text{Width} \geq 2 \times \text{factor}$ and $\text{Height} \geq 3 \times \text{factor}$ and $\text{peakcount} > 1$
   then a compound note is founded, subsegment by boundary tracing.
   else if $\text{Width} < 2 \times \text{factor}$ and $\text{Height} > 2 \times \text{factor}$
   then an inverted note subsegment by using vertical projection tracing.
   else if $\text{Width} < 2 \times \text{factor}$ and $\text{Height} < 2 \times \text{factor}$
   then an undefined object, remove.
   else a normal note, subsegment by projection intersections.

After subsegmentation, the subsegmented primitives will then be fed to another Recognizer module which only recognizes the primitives of the compound notes which are full or half note heads, note bars and tails. Assuming proper subsegmentation, the work done by this part is not hard and gives exact solutions. However if there some defects that have been arised during subsegmentation or from other sources (frequent noise, binarization or line removal defects etc.), results may not be as accurate as the clean case.
The algorithm is as follows:

1. **Do noise removal.**
2. **For each primitive calculate**
   a. **Width**
   b. **Height**
   c. **Pixel Count (Only the object pixels)**
   d. **Box Size**
   e. **Pixel Density = Pixel Count / Box Size**
   f. **Unit Length = (2*Stave Line Width) + Inter Stave Line Gap**
3. If \((h\geq(0.5*\text{unit\_length}) \&\& h\leq(1.5*\text{unit\_length}) \&\& w\leq(1.5*\text{unit\_length}) \&\& w\geq(0.5*\text{unit\_length}) \&\& \text{pixDensity}\geq0.65)\) then FULL NOTE HEAD
4. If \((h\geq(0.5*\text{unit\_length}) \&\& h<\(1.5*\text{unit\_length}) \&\& w\leq(1.5*\text{unit\_length}) \&\& w>\geq(0.5*\text{unit\_length})\) then HALF NOTE HEAD
5. If \((w>h \&\& w>(0.5*\text{unit\_length})\) then NOTE BAR
6. If \((h>w \&\& \text{pixDensity}<0.35)\) then NOTE TAIL
7. If \((h>w \&\& \text{pixDensity}\geq0.35)\) then UNSEGMENTED CHORD

   **DO**
   a. **Find number of notes in the chord**
   b. **Subsegment the chord accordingly**
   c. **Fed back each subsegmented part to note recognizer**
8. **Else GARBAGE**

3.3 **Post-Processing**

3.3.1 Post Identification

The aim of post identification is assigning exact meanings of the primitives that are determined at the previous steps. Before starting this step, we know the type of the primitives (i.e. note head, sharp, flat). The thing that is extracted in this step is their exact meanings according the context, positions according to the stave lines and each other, they reside.

The following properties of the musical symbols are found in this step:
- Note value of note heads and chromatics (coordinates according to stave lines)
- Length of note heads (note head type, note tail, not bar and dot identification)
- Chords (coordinates according to each other)
- Length of rests (rest type identification)
- Loops in the musical scores (a colon followed by a measure bar)

These properties of the notes are stored in a newly created data structure. This data structure extends our primitive data structure with the information gathered in this process. This newly created data structure is used by the ‘ABC write’ functions to produce the final ABC code. Creation of the ABC files will be the topic of the next section; here we will briefly describe the algorithms that we produced in post identification.

new data structure ConvertAll (primitive data structure)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocateAllStaveLines</td>
<td>// all the stave line skeleton of the image</td>
</tr>
<tr>
<td>CalculateSymbolValues</td>
<td>// exact values of note heads and chromatics</td>
</tr>
<tr>
<td>ExtractLoops</td>
<td></td>
</tr>
<tr>
<td>ExtractDotAndTails</td>
<td>// symbols that affect the note durations</td>
</tr>
<tr>
<td>ApplyChromatics</td>
<td>// sharps, flats and naturals</td>
</tr>
<tr>
<td>ApplyNoteBars</td>
<td>// also affect the note durations</td>
</tr>
<tr>
<td>ExtractChords</td>
<td></td>
</tr>
</tbody>
</table>

*LocateAllStaveLines (Described in the next section)*

*CalculateSymbolValues:* The coordinate of all stave lines are known now. We also know the coordinates of all the symbols. So we can easily calculate between which two stave lines a symbol resides.

*ExtractLoops:* We look for a colon followed by a measure bar in this function. A colon followed by a measure bar means the song should return to its starting position or it should return to prior occurrence of this loop sign.

*ExtractDotAndTails:* Dots and tails following a note head lengthen or diminish the duration of the note head. Once a note head is found we check the following symbols if they are dots or tails until we reach another note head. We apply this checking for every note head in the image.

*ApplyChromatics:* Each song has a chromatic sequence. These sequences for chromatics are found at the entrance of the notes after the clef signs part and they affect all the corresponding notes in the song. These sequences are extracted at the header extraction. Also each note may have a chromatic sign next to itself which affects all the same notes in that measure. So there should be mechanism to generate these effects on note heads. We use a *mask* consisting of seven integer values for this purpose. According to the chromatic sequence at the intro part, we set the corresponding mask entries to appropriate values. We also modify this mask in measures according to chromatic information of measures. So finally, when we reach a note head, we apply the current mask to that note head to specify its exact tone.
ApplyNoteBars: Compound notes are connected together with note bars. The effect of note bars is changing the length of the notes. For example in the following compound symbol;

![Figure 3: Compound note sample](image)

Duration of the first note head is two times the length of the following note heads due to its less bar count.

In this function, when we reach a note head we count the note bars at its left and right and assign the maximum of these values as the bar count of that note. This is the correct way to assign the note bars to the note heads. Note heads always get the maximum bar count of their left and right sides.

ExtractChords: Chords are the combination of rests and note heads that should be played at the same time. The following symbol is an example of a chord.

![Figure 4: Chord sample.](image)

In this symbol, all the four note heads should be played at the same time. To achieve this result we check the neighbor note heads and rests when we reach a note head. If their ‘x’ coordinates are same or different with only a small amount according to the song, we understand that these notes form a chord and should be played together.

3.3.2 Header Extraction

Extracting header information corresponds to extracting clefs, measures and key accidentals from the list. A single pass for this purpose is sufficient. In this pass, the algorithm searches for a clef. After a clef come the accidentals. After accidentals, there is a measure. When the algorithm matches a clef, it does into a loop until it finds something different from an accidental. If this different thing is a measure it gets it to account also. Algorithm then sets measure and clef values and extracts the key information from the notes of the accidentals.
Algorithm:
For all clefs in the list
   Loop until finding a symbol different than accidental.
   If this symbol is measure, set the measure and remove it.
   Extract key information
   Remove the clef and accidentals.
End

The major problem in this algorithm is specified as follows: If the measure bar cannot be identified (non-existent in the list) and the first note of the line has an accidental on its own, then that accidental is assumed to be an accidental of the key. The key will not be identified correctly as well as that first note. A solution should be to investigate the order of the accidentals of the key for errors. Since the order of the accidentals are fixed in classical western music. Unfortunately, as mentioned now, this would be effective only in the western music.

3.3.3 Conversion

Conversion is the last part of the program, which deals with generating ABC file from the symbol list. It traverses the list and prints directly. When it sees a rest, it evaluates its length and print. When it sees a note, it evaluates its length, its final accidental to be printed (a combo of the header and the notes own accidental). It also searches for different combinations of measure bars in order to extract loop information. Another feature supported here is the chords. It merges the chord notes and prints them in singular brackets [ ].

Algorithm:

For all elements in the list
   If the element is a note head
      Calculate the length
      Calculate the accidental behavior
      Print
   If the element is a rest
      Calculate the length
      Print
   If the element is a measure bar
      Print according to the loop structures.
      Print an endline character in every 5 measure bars.
End

There are no problems of this part which effects the final MIDI file. However the characteristics of the endline character makes the PS file left aligned which is not a very important problem considering the core OMR process.
4. INTERFACES

As mentioned before, from the first day of the project we thought about MMD as an interactive environment which can be easily used in daily life. Without forgetting that it is an OMR tool, we also worked on the graphical user interfaces and related tools.

Starting the program displays the main window of MMD (Figure 5). The ingredients are trivial. From the menu, the user may open a new ABC file or open existing documents formatted in ABC, MIDI or JPEG. There are also help and about buttons present. And finally for all these links, there are shortcut buttons just below the menu.

![Figure 5: Main Window](image)

Opening an ABC file awakens the ABC editor just as the awakening of the MIDI player with MIDI files. To start the OMR process, the user should open a JPEG file (sheet music, of course) which forces the program to go into “Simple Mode”. Simple Mode has the sufficient features to immediately create the output files. On the other hand, the user should wish to see the process step by step, test the objects and change them manually. At this point user may switch to the “Advanced Mode” by clicking a single button from the Simple Mode.

Below are the screenshots and further discussions of these interfaces.

4.1 Simple Mode

![Figure 6: Simple Mode](image)

This form of our program is used to perform the basic task of MMD, without showing details of the steps of processing. One or more JPEG images can be opened and by pressing the "Convert to ABC" and "Convert to MIDI" buttons it does all the processing on the
image and directly displays the resulting ABC or MIDI file respectively. The user can switch to advanced mode if he wishes.

4.2 Advanced Mode

In addition to the features the simple mode has, in this mode, the user is able to follow the processing of the program step by step, and to do modifications as he wishes. This mode was also very vital for us during the implementation of the program. By this interface we were able to observe easily the mistakes or the unsatisfactory parts of the program and do improvements accordingly.

By pressing the "Deskewing" button, the first step of preprocessing is performed.

![Figure 7: Advanced Mode](image)

The second step the user is able to view is binarization. The user is also to specify the threshold value for binarization by a trackbar. The optimum threshold value should lie between the values obtained by Iterative Thresholding and Average Thresholding. Thus, the user is allowed to specify the threshold value as some value between these two thresholds. In the next step the staff lines are removed from the image so that only the symbols are left. After pressing the "Recognize" button all objects are located, taken into virtual boxes and the symbols are recognized. If the user checks the "Draw Boxes" button, the boxes that encapsulate each object become visible, which shows us clearly if there are symbols that touch each other and are thus identified as a single object. At this point, the color of all symbols turns into light gray, so that a nice feature of MMD gets meaning. The user is able to see all of the symbols that are assigned to one specific symbol type at once by selecting a symbol of the radio button list on the right and pressing the "Test" button. Another nice way to check whether the symbols are identified correctly is to left click on a symbol. The
name appears next to the cursor. If the symbol is not classified correctly, the user may right click on the symbol and select the correct type from the combobox above. By this way the user will be able to get the correct MIDI output despite the wrong identification of symbols, which may greatly influence the correctness of the piece. This feature is very useful to correct the assigned types of misclassified symbols, because it is probable that even if nearly all symbols are correctly identified, a symbol is assigned a wrong class, which will spoil the music and make the correct identification of all other symbols worth nothing. With this feature the user will get a perfect musical output. Finally, if all symbols are labeled correctly, the "Translate" button is pressed and the ABC file is created. Viewing the processing step by step takes longer time then the direct execution, because each time the new image is loaded which takes some seconds according to the image size. The user is also able to skip steps and only view the results of the part he is interested in.

4.3 **ABC Editor**

This form appears when the "Translate" button in the advanced mode or "Convert to ABC" button is pressed, and the ABC file is constructed. Here, the user can view the resulted ABC file and if the user has enough knowledge on ABC, he is able to make changes as he wish when he selects the "Editable" checkbox. When the "Convert to MIDI" button is pressed, a MIDI file will be constructed of the ABC file and the MIDI player will appear automatically. Beside this, the user of course is also able to directly open an ABC file from the ABC editor, or even create his own ABC file in order to convert them into MIDI, which is an extra feature of the program.

![Figure 8: ABC Editor](image)

4.4 **MIDI Player**

The MIDI player contains all buttons a basic MIDI player should have and thus can be used to play any MIDI file besides those that are created by MMD. The MIDI player appears
when the MIDI file is constructed from the ABC file, or directly from the JPEG image. It can also be directly opened from the main menu.

![MIDI Player](image)

**Figure 9:** MIDI Player

5. **TEST RESULTS**

We tested the program on 63 images. The images we chose consist of scanned sheet music and computer generated notes. We first counted the number of symbols by hand and then evaluated the success rates of the program on them. The results are shown in the below table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Success Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole Note (480)</td>
<td></td>
</tr>
<tr>
<td>Note Heads (5717)</td>
<td></td>
</tr>
<tr>
<td>Note Tail (3210)</td>
<td></td>
</tr>
<tr>
<td>Note Bar (6108)</td>
<td></td>
</tr>
<tr>
<td>2 Rest (119)</td>
<td></td>
</tr>
<tr>
<td>4 Rest (531)</td>
<td></td>
</tr>
<tr>
<td>8 Rest (322)</td>
<td></td>
</tr>
<tr>
<td>16 Rest (93)</td>
<td></td>
</tr>
<tr>
<td>32 Rest (32)</td>
<td></td>
</tr>
<tr>
<td>64 Rest (20)</td>
<td></td>
</tr>
<tr>
<td>128 Rest (13)</td>
<td></td>
</tr>
<tr>
<td>Clefs (503)</td>
<td></td>
</tr>
<tr>
<td>Flat (1912)</td>
<td></td>
</tr>
<tr>
<td>Double-Flat (14)</td>
<td></td>
</tr>
<tr>
<td>Sharp (1714)</td>
<td></td>
</tr>
<tr>
<td>Natural (720)</td>
<td></td>
</tr>
<tr>
<td>Dot (678)</td>
<td></td>
</tr>
<tr>
<td>Measure Bars (1860)</td>
<td></td>
</tr>
<tr>
<td>Measures (64)</td>
<td></td>
</tr>
<tr>
<td>Decrescendo (10)</td>
<td></td>
</tr>
<tr>
<td>Crescendo (10)</td>
<td></td>
</tr>
<tr>
<td>Up Slur (32)</td>
<td></td>
</tr>
<tr>
<td>Down Slur (30)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10:** Success Rates of Test Results of the Symbols
6. CONCLUSION

With the test results at hand we can conclude that MMD is quite successful in recognizing the symbols. The relative weaknesses in recognizing the measures do not have a fatal effect on the deliverables.

However this theoretical success may not lead to successful songs. This result takes us to a problem related to the nature of music. While playing a musical instrument or singing a song, even a single mistake crashes the entire melody. Unfortunately music is a thing that does not accept mistakes at all. This was the basic problem of OMR and hence the basic problem of us.

Despite these problems, we observed that MMD works quite well on even complex notes. At least it gives the basics of the melody. And in relatively simple notes are played exactly as they are.

To summarize, although it is not a perfect tool, MMD may considered to be used in daily life. And the algorithms presented in it should be analyzed for better OMR tools.

6.1 Future Work

We know that OMR is an open research area and with MMD we only had partial solutions to this area. MMD is not concluded by the end of this report and will not be concluded until someone develops a perfect solution. The development of MMD continues.

The next step is the improvement of the segmentation and identification modules. The improvement on the success rates is the major topic that awaits us. Line removal is another part that needs improvement. The lossy scans of sheet music yields gaps on the stave lines. Another issue is the loss occurred in the deskewing step. The stave line removal will concentrate on these problems now on.

Coinciding symbols are another problem in the primitive identification part. Because of these intersections, the compound primitives are represented in a single list item. There will be a need of a second segmentation probably using unsupervised learning methodologies.

7. REFERENCES

APPENDIX A (CONVENTIONS)

This part is for those who are interested in analyzing MMD’s source code.

Considering the source code, we accepted some conventions not to be confused in later steps. At the moment MMD has more than 6700 lines of C code which are almost clear. This points out the importance of using conventions considering the integrity of the code.

Below is the list of conventions:

- Counters: i, j, k
- Variables: variableName
- Temporary Variables: tmpVariableType<#>
- Function Names: FunctionName
- Macros: MACRO_NAME
- Global Variables: g_VariableName
- Comments: Inputs & Outputs are at the definition, other information are at the declaration
- Main matrix structure:

```c
typedef struct
{
    int w, h;
    int **array;
} matrix;
```

- Main list structure:

```c
typedef struct
{
    int x1,x2,y1,y2,key;
    ptype type;    // where ptype is integer here
} primitive;
```
APPENDIX B (SOURCE FILE ABSTRACTS)

This part is for those who are interested in analyzing MMD’s source code.

The source code of MMD is partitioned into a number of files. This has two reasons: One is to allow tractability and the other is to discriminate the different parts of the program for easy update.

Each c file has a corresponding h file including the declarations of the functions.

The main files are:

**Globals.h:** Includes all macros, global variables and type definitions.

**Deskew.c:** Includes the functions used in deskewing.

**LineRemoval.c:** Line Removal codes.

**Locate.c:** Line locating algorithms. Find the line coordinates, segments and average gap sizes.

**Recognize.c:** All primitive identification code.

**Reset.c:** Initialization routine for globals.

**Sort.c:** Sorting routine for the symbol list with the keys extracted from the segment information.

**SubSegnRec.c:** Sub-segmentation routines.

**Threshold.c:** Thresholding routines.

**Translate.c:** Post-processing routines. A number of passes in the sorted symbol list.

**WriteABC.c:** ABC construction routines.