Recognition of Handwritten Music Notation
Annón Wolman, James Choi, Shahab Asgharzadeh, and Jason Kahana.
Northwestern Computer Music (NCM), School of Music
Northwestern University
711 E. Illinois St., Evanston, IL 60208-1200
Annw@pahlo.music.nwu.edu

ABSTRACT
This program is designed to read and interpret handwritten music notation. The user is asked to scan handwritten score written on commercially available printed staff paper, the program then attempts to convert these graphics (stored in TEFF format) to other "musical" formats, so that the information can be used in a meaningful way by commercially distributed music printing or performing programs.

1. INTRODUCTION
This report will discuss a work-in-progress. When we decided on attempting to tackle this tough problem, we spent large amounts of time defining the process by which we would attempt to deal with the different aspects and manifestations of the issue of music symbols recognition.

1.1 Input
The program attempts to decipher scanned handwritten music. The only restriction posed by the program is that the user employ standard commercially available printed score paper, or music paper created with a ruler. Once the program is started, an ongoing relationship with a specific user is formed, and the user helps the program achieve its goal. It is essential for us not to try to develop a program which once started could work by itself but rather an intelligent tool which could help the user in his work.

1.2 Background
This program is a result of Annón's frustration with manuscripting programs. It seems that they do not use the musical knowledge of the user, but rather require learning a new set of tools, i.e., instead of drawing an eighth note, one has to split ones knowledge into two pitch (where), and rhythm (symbol) and then remember the correct symbols which signal each of these, when we draw on music paper we think of it as one object. We decided to try and write a program which will help those of us who will like to use paper. At first we considered attempting to decipher commercially printed music, and then go for the more difficult handwritten issue. Thanks to Miller Puckette we realized that the two problems are different and not inclusive. A program which will be able to read handwritten music, most likely will be able to read any printed music, but the opposite is not true. We also agreed with Miller that if indeed we would be able to scan handwritten music and just straighten all the "straight" lines (bar lines, stems) already the manuscript will look better.

1.3 The Problem
Although both questions are related, interpreting handwritten symbols actually means that one could not do a "regular" comparison with a pre-existing set of symbols the way most OCR (Optical Character Recognition) programs work. Rather, the problem need to develop its own set of symbols and connect them through a set of filters to a pre-existing set of symbols. For example in regular OCR program pattern matching will use one of two techniques: line tracing, (where the program follows the black bits), or histograms (where the program adds up the black bits per specified area), and then compares the result with a specific pre-stored similar description of the symbol. Handwritten symbols can be very far away from this pre-stored description. If we would like the program to "know" the users handwriting, it has to be able to adjust both the pre-stored description and the filters for the search.

2. THE PROGRAM
It became very clear that solving this problem will take us a long time. After some discussion we reached agreements about the properties of the program, and our techniques of programing.

125
2.1 The group programming notion.

Faced with the notion that this project will be developed by many people, over a long period of time, and on the other hand that parts of the project could be useful even before the whole is complete, we were forced to design a working technique which, although it will influence the resulting program. Each task in the program is developed as an independent object. One member of our group is responsible for uniformity and correctness, and another is responsible for efficiency.

2.2 The different parts of the program

- The reader reads the input.
- The interpreter aimed to understand the input.
- The compiler asks the user for help when needed.
- The learning part adjusts and remembers for next time.

Information needs to flow freely between these entities. Since we need the help of the user we could not keep him/her waiting. We decided to attempt to work with multiple tasks.

For example, the program is started and the "reader" reads the information and displays it on the screen. It then asks the user to identify all texts present on the page (with the exception of musical markings such as "mf" which it will attempt to interpret, for obvious reasons we decided not to deal with handwritten texts) the user responds by highlighting the specific areas. Meanwhile, the interpreter is calculating the possible tilt and readjusting the page. It also finds out the number of staves, their size and the distances, all of which will be useful information since it relates to the size of note-heads, bar lines and staffs. But by the time the user highlighted the texts, the image has been adjusted in the background and "erasing" the text has to take into account the angle found by the interpreter.

2.3 State of the program

At present the program can read a page, calculate the angle is the manuscript was scanned it. Adjust the tilt, find and identify clefs at the beginning of staves, and the bar lines. Here is a more detailed description of some of the methods used.

2.3.1 Finding the angle and adjusting the tilt

In general, scanned images are tilted as much as +/-3 degrees from the horizontal. Our program can adjust to a wide variety of tilts and straighten the page. Two methods were considered. The first computes a histogram based upon the number of pixels found in a single horizontal line of the page. Then, a standard deviation routine is invoked. Then, histograms are computed at various angles at a user-definable increment. We assume that the angle which gives the highest standard deviation is the angle at which the page is tilted. This assumption is based upon the expectation that the long, horizontal lines of the preprinted staves will lead to the peaks which will cause the highest standard deviation. If the histogram is calculated at an angle at which the page is not straight, there will be no large peaks. Our method "zeros in" or the correct angle by overshooting the correct angle, then going back at a smaller increment. We have achieved excellent results using this method, but it is exceedingly slow; the second is a "LINE_TRACE" method. It is a general concept that can be tweaked to a specific task. It finds an ON bit or byte and traces it up, down, left, right, based on the task at hand. It works best with straight or almost straight lines, horizontal or vertical. Thus it tends itself to finding such objects as the staff lines, bar lines, and hopefully note stems, ties, and slurs.

LINE_TRACE METHOD

Example: Trace all horizontal lines

TASKS:
1. Start at top of page
2. If pixel is on
3. trace line right, left and find line end points
4. if line is valid then save
5. Move down one row
6. Repeat steps 1-5 until bottom of page
7. Done

As can be seen from the above example, the method is straightforward and simple. Steps 3 and 4 can be made into functions that are tailored to each task. To find the angle that the page is tilted, the program at first tries to find the staff lines using the line trace method. Even with the page tilted, it usually finds most of the line. Thus the line with the
most tilt will usually be the tools accurate. Once the starting and ending points of a staff have been determined, the tilt angle can easily be calculated using the formula $\theta = \arctan \left( \frac{\text{ystart}-\text{yend}}{\text{xend}-\text{xstart}} \right)$.

**TASK: FIND ANGLE**

```c
// FIND STAFF LINES (steps 1-5)
1. for each row from top of page to bottom
2. if byte A is mostly ON
3. trace line right and left
4. if line is valid then save line
5. next row
// COMPUTE ANGLE
6. for each line found
7. compute angle
8. if angle is the largest found so far then save angle
9. next line
10. return angle
```

Once the angle is found, the transform routine is called. This is a straightforward transform routine with all the arithmetic that is allowable moved outside the loops in an effort to increase speed.

### 2.3.2 Finding the staff and barlines

Next, the program again uses the line-trace method to find the staff lines, similar to the way it found the angle. Then it groups the lines into staffs, removes the staff lines, and then checks for staffs that are grouped. Last the code finds the barlines. Again it uses the line-trace method. These are the heart of the present program. Each line tracing function and valid line check are tailored to the specific task. If is a framework that is very flexible. It can be thought of as a general method with a unique programmable filter.

**ACKNOWLEDGMENTS**

This research is supported with a grant from the American Foundation. Additional funding was provided by the Northwestern School of Music, Northwestern School of Humanities, and the Northwestern Graduate School. Additional equipment was provided by the Midwest office of NextCo. The authors wish to thank Gary Kendall and Ric Ashley for their ongoing support, Bill Pard for his programming hints, Miller Puckett for starting it all, and Roger Dannenberg for re-energizing us.