Imperial College London
Technology and Medicine
(University of London)
Department of Computing

Reading Sheet Music

Arnaud F. Desaedeleer (afd05)

Submitted in partial fulfilment
of the requirements for the MSc
Degree in Advanced Computing of the
University of London and for the
Diploma of Imperial College of
Science, Technology and Medicine.

September 2006

Abstract

Optical Music Recognition is the process of recognising a printed music score and converting it to a format that is understood by computers. This process involves detecting all musical elements present in the music score in such a way that the score can be represented digitally. For example, the score could be recognised and played back through the computer speakers. Much research has been carried out in this area and several approaches to performing OMR have been suggested. A more recent approach involves segmenting the image using a neural network to recognise the segmented symbols from which the score can be reconstructed. This project will survey the different techniques that have been used to perform OMR on printed music scores and an application by the name of OpenOMR will be developed. One of the aims is to create an open source project in which developers in the open source community will be able to contribute their ideas in order to enhance this application and progress the research in the OMR field.

Acknowledgements

I would like to thank Dr. Simon Colton for his help and support throughout this project. He made the time to meet with me on a regular basis and provided me with feedback which contributed to the success of this project.

I would also like to thank my parents and my sister for all their support. Without them, I would not be here right now.

Contents

Chapter 1

Introduction

This project will involve implementing a software toolkit that will be able to process a printed music score and transform it into a format understandable by the computer. This process is known as Optical Music Recognition (OMR) and has been an active research area since the 1970's. In contrast to Optical Character Recognition (OCR) systems, OMR systems are subject to greater complexities as music notation is represented in a two-dimensional space. The horizontal direction can be associated with the note duration (time) whilst the vertical direction can be associated with pitch.

There are several applications to OMR systems and we will now consider a few of them. Scores sometimes need to be adapted to different instruments (transposed) and having the score in a digital format greatly reduces the time and effort required to do that. Some scores are extremely old and it is convenient to archive them in a digital format. An OMR system is an ideal tool for archiving and re-printing old scores. Converting music scores in Braille code for the blind is yet another application of an OMR system. [10]

1.1 Motivation

There are several factors that led to the motivation of this project. The main motivation behind this project was driven Dr. Simon Colton. Having an extensive collection of piano scores and not having the time to learn/play each score, he wanted an application that could play those for him. Having not found any satisfactory software available, he decided to turn it into a student project.

Having a strong interest for music and an interest for computer graphics, I decided to take on this project. I was not able to find any free or open source OMR toolkits. The only available ones are expensive, and when a particular demonstration version was tested, it performed rather poorly. Having used many open source projects available on the Internet, I feel this is an opportunity for me to contribute to the open source community. Furthermore, I want to start an open source project for which passionate musicians and computer scientists can devote their time and effort in order to help improve this toolkit and advance in the research area of OMR.

1.2 Report Outline

The report for this project is organised as follows:

1. **Background** – This chapter provides a description of related works in the field of OMR. Furthermore, the basic theory to be used for the design and implementation of the project is presented.

2. **Design** – This chapter provides a detailed description of the different components of the OMR system that is implemented for this project.

3. **Implementation** – In this chapter, a technical description of the various components and how they interact with each other is provided.

4. **Testing** – This chapter discusses how the OMR application was tested along with all the parameters that were varied when the testing phase was conducted.

5. **Results** – The application was tested with several scores and the results in terms of accuracy are provided in this chapter.

6. **Future Improvements** – This chapter talks about the future development of the OMR application implemented in this project.

1.3 OpenOMR

This project is hosted on the Sourceforge website from which all the source code can be downloaded. The URL to the project home page is given below:

`http://openomr.sourceforge.net`

Chapter 2

Background

The goal of this chapter is to build the basic foundations needed in order to understand how an OMR system works along with the theory used for the implementation of the optical music recognition system. We first begin by briefly mentioning the underlying concepts of music theory after which two musical typesetting programs will be introduced. An evaluation of current commercial OMR systems is then presented, followed by a description of how OMR systems can be evaluated. We will then look at some issues and problems which make the OMR tricky. Previous research and projects are then summarised in order to gain a certain knowledge in the field of OMR. Finally, some underlying theory which is used in the design and implementation of the OMR system is presented.

2.1 Music Theory and Terminology

In order to understand how OMR systems work, it is important to have a notion of the underlying concepts in music theory and its associated terminology.

The fundamental music element in a music score is the stave (also referred to as the staff). The stave is composed of five horizontal lines and four spaces in between each line. Vertical bars (known as a bar line) are placed in order to separate different measures. There are three types of clefs and they are known as the treble, bass and alto clefs. The clef is commonly placed at the beginning of the stave although it can also appear anywhere in a measure.

Polyphonic scores are ones in which multiple independent melodies are present as opposed to monophonic scores in which only one is present. Piano scores typically use a grand stave which is composed of a bass and treble stave.

Notes are placed either on a stave line or in between a stave line. When a note is placed above or below the stave, ledger lines are used. The higher the note's placement on the stave, the higher its pitch. Depending on the type of note, it will be played for a certain duration.

Figure 2.1 shows a series of notes (top) and rests (bottom). The number below each note and rest indicates their associated temporal value as highlighted in figure 2.1. The notion of time in musical scores is represented by a meter. The meter indicates how many notes of the same type are to be played in one measure and is expressed as a fraction. The numerator indicates how many beats occur in a measure and the denominator indicates how many beats a

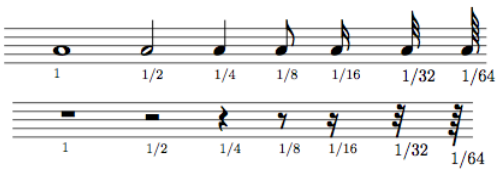


Figure 2.1: Notes and Rests (images taken from [18])

crotchet receives. A $\frac{4}{4}$ beat is also referred to as a common beat and is usually represented by a C instead of a fraction.

2.2 Music Typesetting

There exist several music typesetting packages in the open source community of which two will be briefly mentioned. The output of the OMR system that will be implemented for this project should be able to interact with a typesetting package by means of a common notation format. This will be useful for debugging purposes but will also be useful for its users to view the processed score in a user friendly format. Figures 2.1 and 2.2 were typeset using the LilyPond package.

LilyPond is a program that enables musicians to mark up music scores by using a simple ASCII notation as its input. That notation is then processed by the LilyPond program and is able to output a music score in several different formats including the *PostScript* and *EPS* formats. Furthermore, LilyPond is capable of reading different file formats including the *MusicXML* and the *MIDI* file format which are both described in section 2.4.

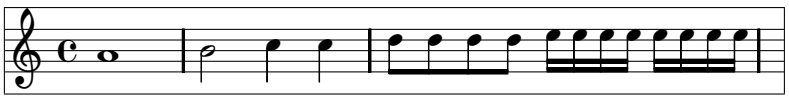


Figure 2.2: Example of a stave typeset with LilyPond

The LilyPond package is extremely well documented and many examples are provided, making it an attractive package to use. LilyPond also runs on many different platforms (Linux, Windows and Mac OS X) and does not require the installation of dependant packages. The installation of the LilyPond package is simple, making it a popular typesetting application used amongst professional musicians. [15]

For example, the code used to represent figure 2.2 looks as follows:

```
\relative c''
{
    {
        \time 4/4
        \clef treble
        a1 b2 c4 d8 d8 d8 d8 e16 e16 e16 e16 e16 e16 e16 e16 s16_" "
    }
}
```

MusiXTEX consists of a set of TEX macros enabling users to typeset music scores within TEX documents. It therefore requires the TEX package to be installed, and that the MusiXTEX package also be installed. This therefore adds complexity and is not likely to be an appropriate package for the everday user. Furthermore, the installation of the MusiXTEX package seems complicated. As stated in [14], *"If you are not familiar with TEX at all, I would recommend to find another software package to do musical typesetting"* It is however fair to say that once MusiXTEX is mastered, it is an excellent package to typeset music.

2.3 Commercial OMR Systems

This section provides a description of several commercial OMR software packages. Three of those packages were tested and a description of how they performed in comparison to one another is provided. In section 2.5, we will thoroughly discuss how OMR systems can be evaluated. Therefore this section only provides a user's perspective of how well each application performed.

1. SmartScore (Musitek)
2. SharpEye (Musicwave)
3. Notescan in Nightingale (Mac only)
4. SightReader in Finale
5. Photoscore in Sibelius (Neuratron)

Midiscan by Musitek was perhaps one of the first commercial OMR applications and first appeared in 1991, running on the Windows 3.1 platform. In 1998 it was renamed to SmartScore and in 1999 a version for the Macintosh operating system was released.

Three of the aforementioned OMR systems have evaluation versions and those were tested. All OMR systems were tested with the first page from Beethoven's "Grande Sonate Pathétique" scanned at a resolution of 300*dpi* and saved in a *jpg* format. All three systems failed to open the *jpg* file so the image had to be converted to a greyscale bitmap.

SmartScore from Musitek: The score was opened and the image was processed by Smartscore and within 15 seconds, two images were displayed: one with the original score and one with the recognised score. Additionally, SmartScore is capable of playing the recognised score in a MIDI format. However, since there were several errors present in the processed score, the playback of the original score did not at all sound like it should have. Since it failed to properly identify a rest in the second measure, the left and right hand weren't synchronised and so the MIDI file sounded incorrect. (www.sibelius.com)

SharpEye: A lack of attention given to the graphical user interface of this application made it harder to visualise the recognised partition. However, the recognition of Beethoven's Sonate was accurate and there were few mistakes present in the recognised version. On this particular tested score, SharpEye performed best in contrast to the other tested applications. Although its GUI

was poorly designed, it does off an attractive feature which enables the user to modify the recognised score. This can be useful for several reasons including: notes needing correction after the recognition process or an artist purposely wanting to make modifications to the recognised score. (www.visiv.co.uk)

PhotoScore in Sibelius: The application did not perform well at recognising Beethoven's Sonate. In particular, it missed many accidentals, chords and other features present in the original score. This was by far the worst OMR system tested. This application has a nice feature which highlights all of the stave lines when the image is initially opened. (www.sibelius.com)

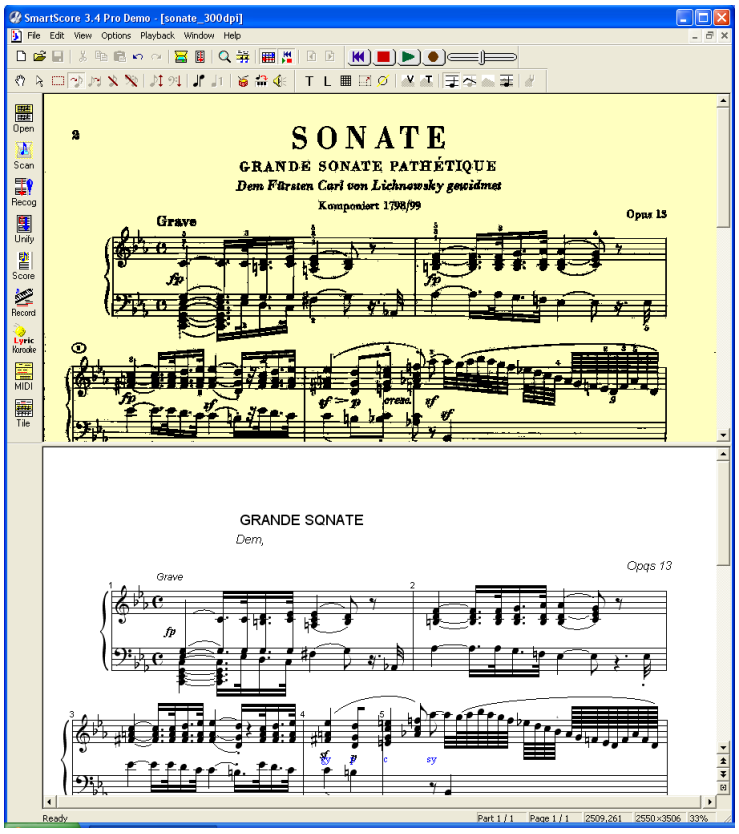


Figure 2.3: SmartScore Screenshot

2.4 Music Notation File Formats

The Musical Instrument Digital Interface (MIDI) standard was first proposed by Dave Smith in 1981 and defines a way for music to be represented in a digital format and can be played by computers. Musical notes can be represented by different electronic musical instruments such as a synthesiser, guitar and many other instruments. This standard has been widely used and is still in use today but it suffers certain drawbacks. The MIDI format does not represent stem

directions, beams, repeats and other aspects in music notation [16]. Therefore the MIDI format was not appropriate for systems in which such information was to be preserved.

As a consequence, work was carried out to create a standardised music format and the Notation Interchange File Format (NIFF) project was launched in 1994. This therefore meant that different music packages (whether typesetting, OMR or editing) would be able to share a standard format, adding a layer of compatibility between various music applications. [17]

The NIFF standardised music format was the only one to succeed but its use was limited and its format was not being maintained. Consequently, a new music format called MusicXML was developed and in 2004, its version 1.0 was released. This format is more flexible in terms of representing music and as a result this format has been adopted by music software applications (presently supported by at least 55 applications) including: Finale 2006, SharpEye Music Reader, LilyPond and many others. MusicXML is not open source but it is available under a royalty-free license from Recordare. [16]

The MIDI format is a useful format if one only wants to play a representation of a sound file on a computer but if the aim is to typeset or digitally archive a music score, then a format such as MusicXML should be considered. As the requirements for this project are to play a scanned music score in a MIDI format, the MIDI format will be adopted. The MusicXML format will also be supported due to its robustness in representing musical symbols and its compatibility with other software packages.

2.5 Evaluation of OMR Systems

As several OMR software packages have been implemented, it is important to set standards as to how such systems should be evaluated. Optical character recognition systems can be evaluated according to the total percentage of correctly identified symbols, but this is not a valid evaluation technique for OMR systems. For example, if an accidental is incorrectly identified and the output produced is a MIDI file, it will not sound correctly, when in fact there is only a small mistake [8]. Therefore other evaluation techniques must be used for OMR systems in order to benchmark OMR systems. If each author evaluates their system with different sets of music scores, then it is hard to tell how systems perform in contrast to each other. Furthermore, authors will usually use music scores on which their system performed well in order to show off the performance of their system.[5]

In order to overcome the lack of ways to evaluate OMR systems, the Interactive Music Network proposed a framework to evaluate such systems. They introduced a "Quick-Test" that would enable the judgement of the capability of an OMR system. The OMR system being benchmarked is given a music score made-up purely on the basis for testing the capability and performance of OMR systems. Version 0.1 of the OMR "Quick-Test" consists of three pages with musical features such as: time signatures, notes, beams, keys signatures, clefs, etc.

A more thorough methodology to evaluate OMR systems is described in [5]. This approach involves using a set of seven images (The first page is shown in figure 2.4) chosen from a music database that have the following features:

1. Monophonic music
2. Font variability
3. Music symbols frequently used in classical music
4. Variable density of musical symbols
5. Irregular groups of notes (triplets ...)
6. Small notes with or without accidentals (grace notes)
7. Different types of barlines
8. Clef and time signature change
9. Slurs; single and nested

The OMR system is evaluated by using different categories in which each have a corresponding weight assigned. The weights were determined from the results of a survey that was completed at a conference by a group of experts and users of OMR systems. The categories and the weights are shown in table 2.1

Note with pitch and duration (10)	Rests (10)
Notes with accidentals (7)	Groups of beamed notes (10)
Time signature and time change (10)	Key signature and key signature change (10)
Symbols below or above notes (5)	Grace notes (5)
Slurs and bends (7)	Augmentation dots (10)
Clefs (10)	Irregular notes groups (10)
Number of measures (10)	Number of staves (10)

Table 2.1: Categories and associated weights considered in OMR evaluation

The evaluation and result analysis of the OMR system is then conducted by using a set of metrics designed for this purpose. The following evaluation metrics are used for each category described above.

1. The total number of *expected* complete symbols or relationships in the original score
2. The total number of *correct (N)* symbols or relationships identified in the reconstructed score in comparison to the original score
3. The total number of *added* (n_1) symbols or relationships in the reconstructed score in comparison to the original score
4. The total number of *wrongly identified* (n_f) symbols or relationships in the reconstructed score in comparison to the original score
5. The total number of *missed* (n_m) symbols or relationships in the reconstructed score in caparison to the original score.

Then for each category, the following equation can be expressed:

$$N = n_1 + n_f + n_m$$

A table is then created based upon the results from each category and metric. This therefore enables us to compare and contrast how well different OMR systems did in relation to one another, and in which categories they performed best. [8] shows results conducted on SmartScore, SharpEye 2 and the O^3MR system (described in section 2.8.6) with this testing framework.

2.6 A Typical OMR Process

Although there is no set standard as to how an OMR system should be implemented, most authors choose to adopt the following steps.

1. Digitise music score by means of a scanner
2. Apply filters to the image
3. Identification and removal of stave lines
4. Identification and segmentation of elementary symbols
5. Reconstruction and classification of music symbols
6. Generation of symbolic representation into a symbolic format for music notation (MIDI, MusicXML, etc.)

The first step is to acquire the music score in a digital format by means of a scanner. Once the image is stored on the computer in a digital format (such as a bmp, jpg, tiff) filters are applied to the image in order to improve the quality, ensuring that the best results are obtained when that image is further processed for recognition. An example of such a filter may involve converting the image to a binary format. The next step is to identify the staves within the image. While some researchers claim this step to be unnecessary, most authors disagree and view this step as being a vital one. Depending on which method is used to further process the image, the staves may be removed at this point. The image is then segmented and elementary symbols are then recognised and classified. A representation of the music score is then created (MIDI or MusicXML) by using the information from the classification module.

2.7 Issues/Problems Associated with OMR Systems

2.7.1 Graphic Quality and Print Faults

There are many factors that can influence the quality of printed music. For instance, old printed music may have stave lines which are fading away, hence, affecting the quality of the printed music and in turn making it harder for an OMR system to perform its stave line recognition process. Skewed stave lines can make it difficult for an OMR system to perform the stave line recognition.

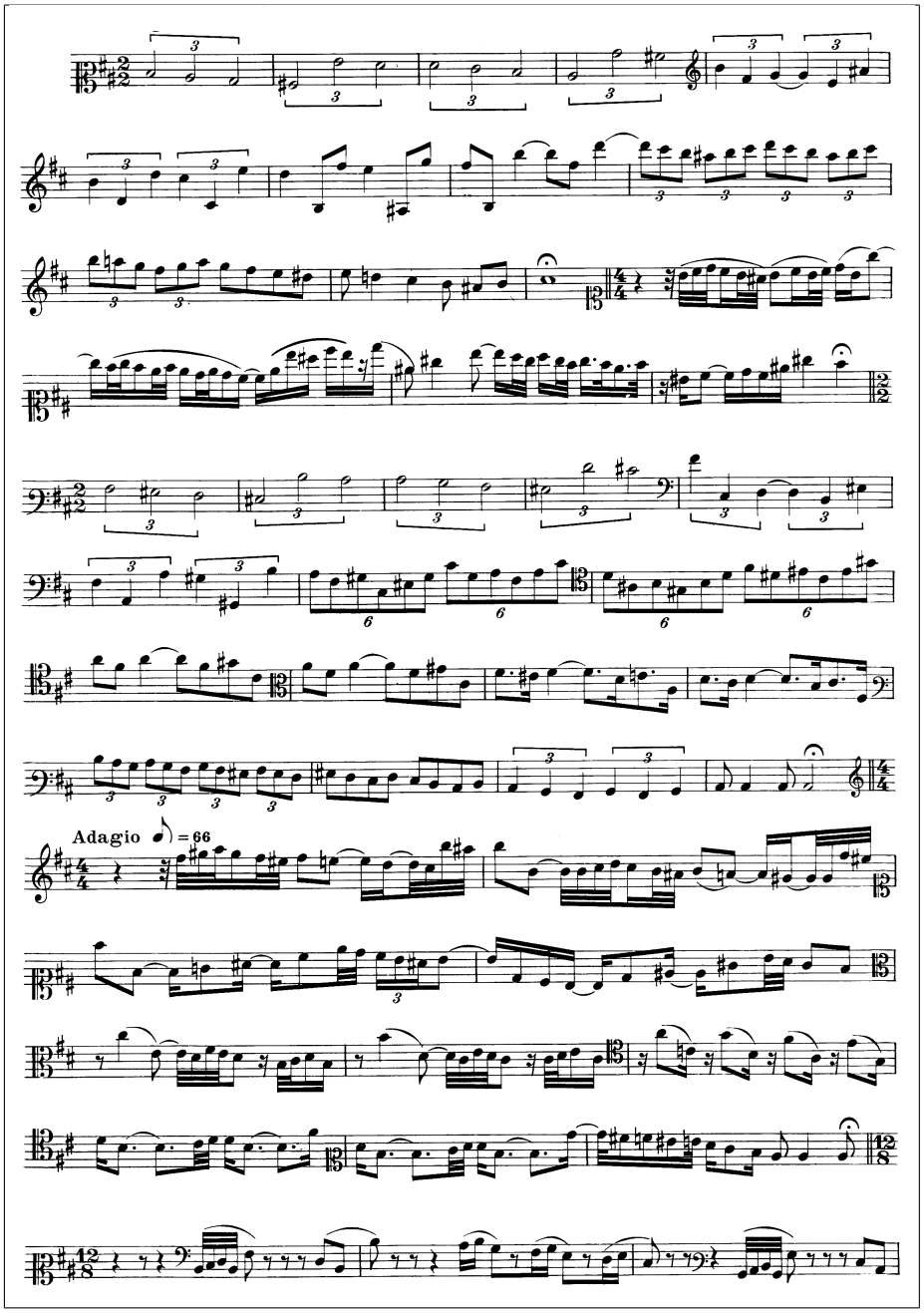


Figure 2.4: The first page of the score used to evaluate OMR systems

Therefore stave line detection algorithms must take into account skewed stave lines. Several ways have been developed to detect skewed stave lines of which one uses a Fast Fourier Transform [3]. In some older printed music, the thickness of the stave lines varied and this is therefore something that needs to be taken into account. It is possible for a note to be covering a space and a line, therefore making it hard for the OMR system to decide if the note is on a line or on a space.

2.7.2 Handwritten Partitions

Handwritten partitions are harder to process for various reasons. Each author has their own way of writing music partitions and this is analogous to everyone having their own signature. As handwritten partitions are not printed by computer systems, there can be variations in the size of stave lines and music notes, adding to the complexity of the OMR system. Due to the complexity involved with the recognition of handwritten partitions, they will not be taken account in this project.

2.8 OMR Research/Projects

2.8.1 Carter (1988)

Carter's most important contribution was the idea of segmenting the music score image by using a Line Adjacency Graph (LAG). This method involves scanning the image horizontally and vertically, searching for paths of black pixels. Graph nodes correspond to the unions of adjacent and vertically overlapping segments while arcs define the overlapping of different segments in an adjacent column (junctions) [7]. The graph obtained is then analysed to detect the stave line and symbols that overlap the stave line. By using this technique, the following features are obtained:

1. Identification of empty areas within the staff.
2. Identification of the staff (even if the image is skewed up to 10 degrees).
3. Identification of the stave lines that are slightly bent, broken or in which the thickness varies.

The phases of his OMR system can be described as follows:

1. A LAG is produced by isolating the empty parts of the stave lines. This stage also isolates all the symbols and groups of connected or overlapping symbols.
2. The objects are then classified according to the bounding box size and the organisation of their constituent sections.

2.8.2 Fujinaga (1988)

Fujinaga proposed an OMR system which heavily relied on the projections of images to identify and classify the staves and symbols. As opposed to most other systems, his system does not require the removal of stave lines. The staves

are identified by using the Y-projection of the music score and the symbols are identified by using a combination of projections. Fujinaga also provides a formal way of describing music notation by means of a context-free grammar. [8]

The phases of his OMR system can be described as follows:

1. The stave line is located by taking the Y-projection of the image (described in section 2.12) in which groups of five peaks are present.
2. Symbols are then detected by taking the X-projection of the image (also described in section 2.12). Values that are greater than the background noise suggest the presence of music symbols. The syntax proposed is then applied to support the detection of symbols.
3. The X- and Y-projections are then applied to classify the symbols. Features such as the width, height, number of peaks and number of pixels in the X-Projection are all considered as part of the classification process. He also suggests considering the first and second derivatives of the projection profiles.

2.8.3 Roth (1994)

In 1994, Roth developed an OMR system whose output was to be used with the Lipsia music notation editor (developed at ETTH Zurich) in order to digitally reproduce the original music score. The OMR architecture is built upon the seven following steps:

1. Rotation – If the scanned image is skewed, it is rotated. This step is not automated in the application and a separate program is used to rotate the image.
2. Vertical run length statistics – The distance between two staffs and the thickness of the stave lines are calculated by using elementary statistics that are described in section 2.11. From the those two parameters, the height of the staves can be determined.
3. Locate and delete stave lines – By taking the Y-projection of the image, the stave lines are located by searching for groups of five peaks. The stave lines are then removed by erasing the lines of width calculated in step 2.
4. Locate and delete vertical lines – The removal of the vertical lines (stems, bar lines ...) is accomplished by taking the X-projection (described in section 2.12) of the image and also by using Mathematical Morphology.
5. Connected component labelling – All remaining components are then identified and labelled according to where they were found in the image.
6. Symbol recognition –The components from step 5 are then classified and recognised by using a set of graphic rules.
7. Lipsia document generation – The recognised symbols in step 6 are then written in the Lipsia file format.

As mentioned in [9], the results were satisfactory although no percentage or benchmarks were given. All the rules are hard-coded and this therefore makes it hard to add new symbols to the system, Roth does suggest designing a formal language describing rules for each symbol as an extension to his system.

2.8.4 Bainbridge (1991–1997)

In [1] and [2], Bainbridge describes a system capable of recognising different music notations such as the Common Music Notation (CMN), percussion and tabulature. He first describes a way to detect the stave lines by means of taking a horizontal projection of the black pixels on each row of the image. Before proceeding to removal of the stave lines, an OCR processing phase is conducted in order to remove all text present in the score. Bainbridge also introduces a language called Primela that has the following properties:

1. Graphical description
2. Variable instantiation
3. Matching control

This language allows the specification of how to assemble primitives together. It is common to find systems in which primitives have been hard-coded, thus making it difficult to extend the system. The Primela language supports four different types of pattern recognition techniques which are categorised as follows: Projections, Hough transform, template matching and slicing techniques.

He states that the system was tested with different examples of CMN and Georgian chant music and that the results obtained were encouraging. The solution of breaking down the problem into fragments is novel and that it is more flexible than the earlier ad-hoc methods. However, the computation cost associated with those techniques increases [7].

2.8.5 Ng and Boyle (1992–2002)

The Automatic Music Score Recogniser (AMSR) was developed in 1994 at the University of Leeds. The approach used in this system is based on reverse engineering. Usually, a composer will first write the beam, then the stem and then add features such as slurs. The AMSR first looks for thick horizontal features such as slurs and then find stems and finally beams. Hence complex music symbols are decomposed into primitive ones easing the classification of the symbols [7].

The system can be described as follows:

1. The image is preprocessed and converted from a greyscale-scale image to a binary image by means of a thresholding function. If the image is skewed, the rotation of the image is corrected. As this system relies on projection methods, this step is crucial. The stave line parameters are calculated (as described in the above OMR systems).
2. In this sub-segmentation module, composite symbols are divided into primitive elements such as note heads, beams, stems. If those symbols cannot be properly classified, they are then passed into the sub-segmentation module again to be further broken down.
3. A k-Nearest Neighbour classifier is used to detect primitives. Sub-segmentation techniques are used to reconstruct the primitive symbols.
4. The output is written in a format called ExpMidi. This format is compatible with MIDI, however it has the benefits of being more expressive than the standard Midi format and is able to describe features such as accents.

2.8.6 The O^3MR

As opposed to the aforementioned OMR systems, the Object Oriented Optical Music Recognition (O^3MR) does not remove staves. The approach of the O^3MR system is based on the projection profiles to extract basic symbols that contribute to the formation of a music partition. This architecture is only concerned with the music written in the western style notation and in which only monophonic music with five stave lines are present. There are four steps involved in this OMR architecture and are as follows:

1. Segmentation
2. Basic Symbol Recognition
3. Music Notation Symbol Recognition
4. Music Notation Model Refinement

The segmentation process of the O^3MR architecture can be broken down into three further levels. Level–0 involves a tuning process in which two parameters (thickness of stave lines and distance between stave lines) are obtained. By using those parameters, the music score is then decomposed into a series of different images in which only one stave is present. Level–1 then works on the image segments produced by Level–0 and extracts vertical image segments containing music symbols. The next level then decomposes the input from Level–1 into basic symbols.

The basic symbol recognition module is responsible for recognising the basic symbols that have been output from Level–2 of the segmentation process. The output from Level–2 of the segmentation process is then normalised and input into a neural network.

The music notation symbol recognition step maps the basic symbols into elementary components of music notation symbols. For example, a stem might be mapped to to a beam.

The last part of the system consists of constructing a music model by refining the process used in the previous step. [4]

2.9 Optical Music Recognition using Kd-tree Decomposition

This is a project that was undertaken by Kerry Peake at the University of Birmingham and focuses on using a Kd-tree decomposition method and Euclidean Distance Metric to retrieve and identify symbols in a music partition. The architecture of the system can be broken down into the following 3 stages [20]:

1. Pre-processing – This stage involves correcting skewed staves, removing the stave lines and segmenting the image. Realigning the score is accomplished by using a line-chasing algorithm to identify the top stave and consequently its skew which then allows the stave line to be realigned. The staves are then removed by using a line chasing algorithm and deletes black pixels except where an object is identified. The segmentation process is done by locating the boundary of each object in the image.

2. Symbol Analysis – Most of the symbol analysis is performed by using a Kd-tree decomposition method along with Euclidean distance metrics.

3. Contextual Post-processing – Once all the symbols are identified, they can be interpreted musically from which a Midi file and a MusiXTEXfile are generated.

2.10 Music Sheet Reader

This "Music Sheet Reader" is the implementation of an OMR system by Yong Li at the University of Birmingham. The implementation of the project uses a segmentation based approach along with a neural network to classify so-called "immutable" symbols. The implementation was written in Java and the basic architecture of the system is as follows [21]:

1. Preprocessing – If the scanned image is in colour, the image is binarised by using a thresholding mechanism in order to convert the image to a binary one (black and white).

2. De-skewing and stave detection.

3. Music symbol identification – This step is decomposed into two different categories: immutable and mutable symbols. Immutable symbols such as note heads and rests are ones for which their dimension is consistent throughout the music score and mutable symbols are ones such as slurs, beams and ties for which their dimension is inconsistent throughout the score. A three-layer MLP network is used in this step to classify immutable symbols. The mutable symbols are detected at the interpretation stage.

4. Output – The output generated by this program is in both a midi and MusiXTeX format. A package called jMusic was used to generate the midi file.

2.11 Run Length Encoding

The Run Length Encoding Algorithm is briefly described in this section as it will be used to determine the thickness of the stave lines in the score and the distance between two stave lines. We will see in chapter 2 how the RLE algorithm can be applied to a music score in order to detect the parameters mentioned above.

The RLE algorithm is perhaps one of the simplest encoding algorithms which can be applied to a stream of data and works by counting each consecutive occurrences of a same value in a sequence of values. We will later refer to this as being a 'run'.

Consider the following binary values which could perhaps represent a sequence of pixels in a black and white image (where a black pixel has the value 0 and a white pixel has the value 1):

$$1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0$$

By applying the RLE algorithm, this sequence of numbers could therefore be represented as a set of tuples of the form: value, occurrence

$$(1, 3), (0, 6), (1, 8), (0, 3)$$

2.12 X- and Y- Projection Definition

It is important to understand the underlying concepts of the x and y-projections of an image as they are heavily used by several modules in the OMR design.

The X-Projection is the projection of an image onto the the X-axis. The result is a vector whose i^{th} component is the sum of all black pixels in the i^{th} column of the image.

The Y-Projection is the projection of an image onto the Y-axis. The result is a vector whose i^{th} component is the sum of all black pixels in the i^{th} row of the image [6].

Let us now look at an example of calculating the X and Y-Projections of the image in figure 2.5. The image in this figure is 6 pixels wide by 7 pixels high and represents the letter 'A'. Calculating the X-Projection of this image will yield an array consisting of 6 elements with the following values:

0, 6, 3, 3, 6, 0

Calculating the Y-Projection of this image yields an array consisting of 7 elements with the following values:

2, 4, 2, 4, 2, 2, 2

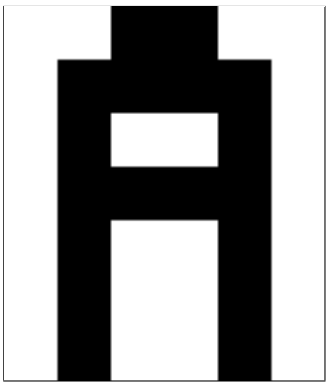


Figure 2.5: Example of a 6 pixel wide by 7 pixel high image

2.13 Discrete Fourier Transform (DFT)

An interesting approach to detect the angle by which staves are skewed in an image is discussed in [3]. This method is of particular interest to us as we want to be able to determine by how much a music score needs to be rotated in order to realign the stave(s) in such a way that they are parallel to the horizontal axis of the image. The method described in [3] uses an approach based on the 2D

Discrete Fourier transform of an image and we will now discuss the basics of the 2D FT in this section.

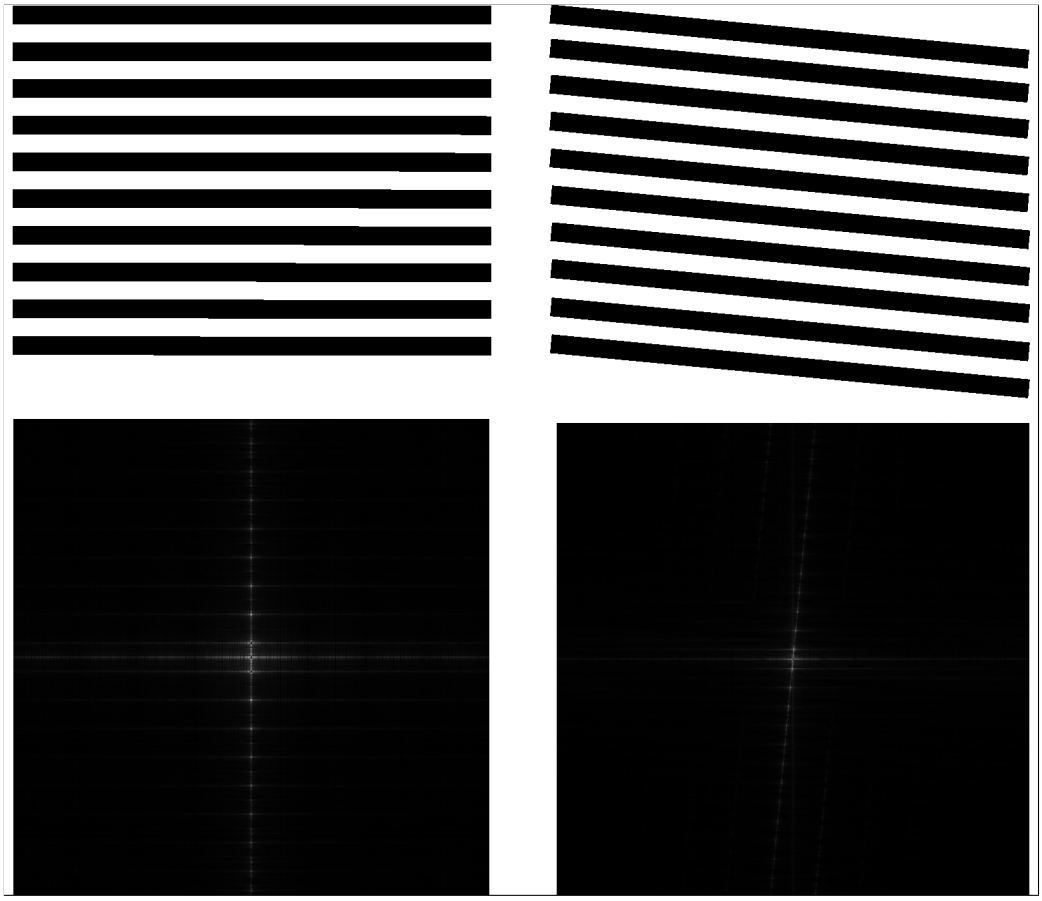


Figure 2.6: Top Left: 10 horizontal black and white lines. Top Right: 10 black and white lines rotated by 6° clockwise. Bottom Left and Right: Fourier Transform of the images directly above them

The Fourier Transform for an $N \times N$ image and its inverse are defined as follows:

$$F(u,v) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y)\, e^{(-2\pi j(ux+vy)/N)}$$

and

$$f(x,y) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} F(u,v)\, e^{(+2\pi j(ux+vy)/N)}$$

where $j = \sqrt{-1}$

$F(u,v)$ is a complex number and is represented by a magnitude and phase rather than a real and imaginary component

and

$f(x, y)$ is a function representing the pixels of the image at coordinates (x, y) and has a real value [19].

The magnitude and phases of the FT are calculated as follows:

$$Magnitude\,(F) = \sqrt{real\,(F)^2 + imaginary\,(F)^2}$$

$$Phase\,(F) = \tan^{-1}\left(\frac{imaginary\,(F)}{real\,(F)}\right)$$

The magnitude indicates how much of a certain frequency component is present whilst the phase indicates the location of the frequency components. From now on, we will refer to the Fourier Transform as being its magnitude as we will not be using the phase information of the transform.

The magnitude image of the Discrete Fourier Transform has the following properties. The u-axis represents the horizontal frequency component and runs from left to right through the centre of the image. The v-axis represents the vertical frequency component and runs from bottom to top through the centre of the image. The centre of the image corresponds to the origin of the frequency coordinate system.

The Fourier Transform attempts to represent an image as a summation of cosine-like images. The image such as the one to the top left in figure 2.6 is a pure cosine image and has a simple Fourier Transform. The image to the bottom left is the Fourier Transform of the image directly above it and we notice that it has a vertical line running through the middle of the image consisting of bright white spots. The image to the top right of figure 2.6 is identical to its neighbour except that it is rotated by an angle of 6° clockwise. The Fourier Transform of the image is shown directly beneath it and we still notice a line consisting of bright white spots, but it is no longer perfectly vertical. This time, it is at an angle to the y-axis and appears to be rotated by the same angle as the original image. This is exactly what will enable to detect if staves are rotated in an image.

Some properties of the transform as mentioned in [3] are:

1. The Fourier Transform is symmetric to the origin. That is: $F(-x, -y) = F(x, y)$
2. The most distinct feature in the Fourier Transform of a music score is the line consisting of bright white spots at an angle to the vertical axis.
3. The angle between the strong component in the Fourier transform can be determined by locating two strong components (i.e the brightest white spots) which are the furthest away from one another. The angle can thus be calculated as follows given two coordinates (x_1, y_1) and (x_2, y_2):

$$\theta = \sin^{-1}\frac{y_1 - y_2}{x_1 - x_2}$$

2.14 Artificial Neural Networks

Artificial neural networks attempt to detect patterns or trends which are too complex for other computing techniques to model. They attempt to mimic the behaviour of the human brain within certain limits. While it is estimated that the brain is made up a over a hundred billion neurons, artificial neural networks generally consist of a few hundred neurons. Neural networks have proven to be accurate for Optical Character Recognition applications. Most of the discussion to follow is based on [11] and [12].

2.14.1 Supervised and Unsupervised Training

As in most machine learning techniques, neural networks must be trained in order to provide useful outputs. A neural network is trained by collecting different data sets and feeding them into the neural network in a special "learning mode" and we will later see how the "learning mode" of the neural network affects the internal state of the network. Additionally, we will also discuss how long the training of the neural network should go on for and what parameters we can change to train the neural network. Generally, the training of neural networks falls into two categories: supervised or unsupervised training.

In supervised training, a set of inputs along with the expected corresponding outputs are given to the neural network and this is the most common type of training for neural networks.

In unsupervised learning, the neural network is only given a set of inputs and based on those it tries to find a correlation between them to classify them into different categories. The Self Organising Feature Map neural network which is briefly mentioned in section 2.14.6 is trained in such a way.

Properly training a neural network is important as they are prone to memorising the training data set and when later used with unseen data sets their performance is poor. This is a phenomenon known as overfitting and will be discussed in greater detail in section 2.11. By the same token, if a neural network is not given enough data to train on, it will also poorly perform.

2.14.2 The Artificial Neuron

The artificial neuron is the basic unit from which a neural network is constructed. It can be viewed as a many-to-one function that will either fire or not fire based on the summed value of its input(s). Figure 2.7 shows a diagram of an artificial neuron.

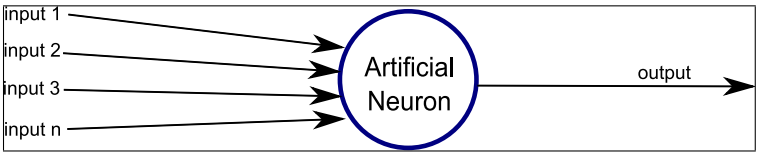


Figure 2.7: An Artificial Neuron

The Neuron is made up of the following:

1. Inputs – The inputs to the artificial neuron can come from outside the neural network (input layer) or from other neurons such as in multi-layered perceptron networks. The inputs are real-valued, i.e they take the form of a decimal number. Furthermore, each input is subject to a weight that is varied whilst the neural network undergoes its training phase. This therefore means that different inputs may have a greater impact than others.

2. A threshold value – In the case where a non-linear artificial neuron is used, a threshold value is input into the neuron. It has a weight like all other inputs, except that the input comes from a node that always outputs a 1.

3. Combination function – This is a simple linear function that calculates the strength of the incoming signals into the artificial neuron. It is defined as follows, where c is the output of the combination function with n inputs:

$$c_j = \sum_{i=0}^{n} w_{ji} x_i$$

   Where $x_0 \dots x_i$ are the real values of the input signals and $w_{j0} \dots x_{ji}$ are the respective weights for the inputs signals.

4. An activation function – The activation function determines the output of the artificial neuron (i.e the value when it fires). There exist several different activation functions of which include the step, sigmoid and hyperbolic tangent functions. The output y of the artificial neuron can then be defined as follows:

$$y_j = \phi \left(\sum_{i=0}^{n} w_{ji} x_i \right)$$

   Where ϕ is the activation function.

2.14.3 Feedforward Neural Networks

There are several types of neural networks but we shall limit our discussion to feedforward neural networks and in particular, the perceptron and multi-layer perceptron networks. Feedforward neural networks are arguably the most simple of neural networks in which information is propagated in only one direction. That is, information moves from the input nodes until reaching the output nodes. Hence in feedforward neural networks, there are no cycles or loops.

Perceptrons

The perceptron was invented in 1957 at the by Frank Rosenblatt at the Cornell Aeronautical University. It is often seen as the simplest kind of feedforward neural network and only consists of an input and output layer. This model attracted lots of attention when initially introduced until it was proven that the perceptron was only capable of solving linearly separable problems. The classic "XOR Problem" illustrated by Minsky and Papert's book was used to

demonstrate that the perceptron was incapable of learning the XOR boolean function and led to a period of inactivity in the research area of neural networks. Figure 2.8 shows how the AND and OR boolean functions are linearly separable and the the XOR boolean function is not linearly separable.

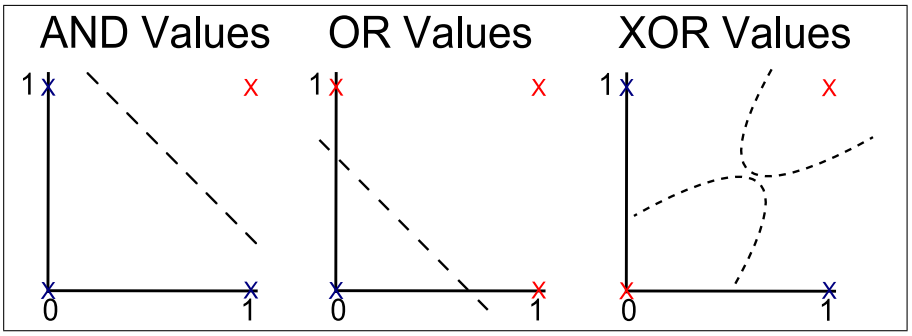


Figure 2.8: Boolean Chart: Red "X" denotes true and Blue "X" denotes false

Multi-Layer Perceptrons (MLP)

In contrast to perceptrons, multi-layer perceptrons have one or several hidden layers. The input layer propagates its outputs to the first hidden layer which in turn propagates its output on to the next node until it finally reaches the output node. Some neural networks use more than one hidden layer, although one hidden layer is sufficient for most problems. Figure 2.9 shows a simple MLP network which can be used to solve the XOR problem.

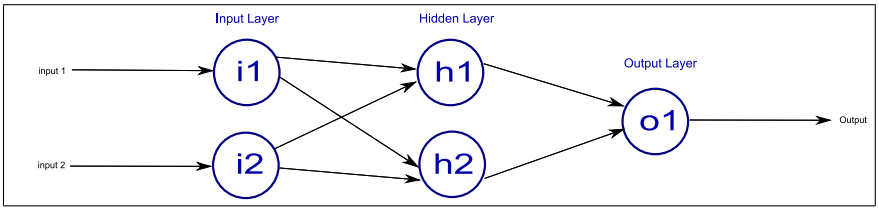


Figure 2.9: A Multi-Layer Perceptron Network

The frequently used training algorithm (backpropagation) for training the MLP network is derived mathematically by using differential calculus and we therefore require a differentiable activation function. We cannot use the step function as our activation function as it is not continuous and hence non-differentiable, so instead we will discuss the sigmoid function. When plotted, the sigmoid function has an "S" shape and therefore closely resembles the step function.

The sigmoid function is defined as follows:

$$\rho(c_j) = \frac{1}{1 + e^{-c_j}}$$

Where c_j is the combination function and e is Euler's number.

Furthermore when computing the derivative of the sigmoid function, we observe that its derivative is computationally easy to perform.

$$\frac{d\rho\left(c_j\right)}{c_j} = \rho\left(c_j\right)\left(1 - \rho\left(c_j\right)\right)$$

Backpropagation Algorithm

The MLP network is trained by altering the weights in each artificial neuron in such a way that the difference between the desired output value and the output we get is minimised. One of the most famous algorithms to train a neural network is the backpropagation algorithm and was introduced by Rumelhart, Werbos and Parker in the late 1980's. The backpropagation algorithm works as follows:

1. Initialise the network by randomly assigning weights ranging from -0.5 to $+0.5$ for all neurons in each node.
2. Test the neural network with the input data set. After running through all the input data sets, an error term is calculated and weight changes occur at each node. The error term is usually expressed as a root mean squared error and can be calculated as follows:

$$RMSE = \frac{1}{2} \sum_{j=0}^{numtrainingexamples} \left(\sum_{0=1}^{numoutputs} \left(target_o - output_o\right)^2 \right)$$

3. If a certain terminating condition is met, training stops or else step 2 is repeated.

2.14.4 Local Minima

This is a problem that can frequently occur with the backpropagation algorithm, and many other AI techniques. It occurs when the error value for the network gets stuck in a local minimum not having reached its potential global minimum. This is illustrated in figure 2.10. In such a case, trying to move away from the local minimum will result in an increase of the error value. There are several ways of getting around this problem of which two are mentioned:

1. The network can be re-initialised with new random values in hope that it won't get stuck in a local minimum.
2. Momentum is added to the weight changes. This approach involves remembering the change that was added to each weight in the previous epoch and adding a small amount of that weight to the weight in the current epoch. How much is added is controlled by the momentum parameter which ranges between 0 and 1.

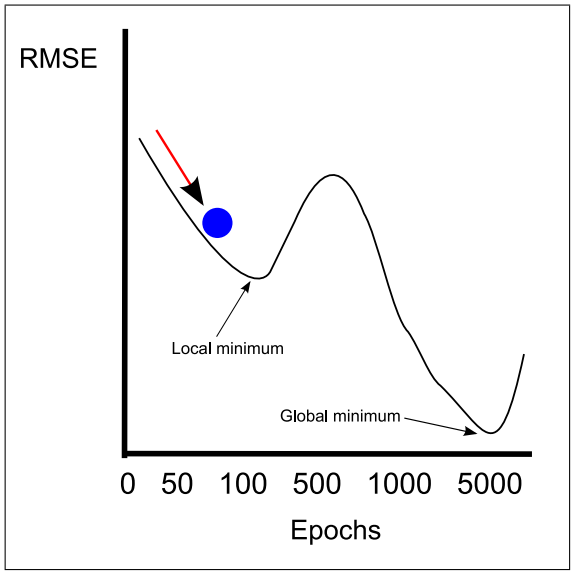


Figure 2.10: Local Minima

2.14.5 Overfitting

Overfitting can occur if a neural network is left to train for a long period of time after which it memorises the training set. In such circumstances, the network will poorly perform when given unseen data.

Figure 2.11 shows a graph of the root-mean squared error versus the number of epochs that a given neural network has been trained for. The results of a training and validation set are presented. Initially, the error for both sets decreases rapidly but after the network has been trained for a certain number of epochs, the error for the training set carries on decreasing whilst the error for the validation set increases. This is an example of when overfitting occurs. We can therefore use this information to terminate the training algorithm. After each epoch, the previous state of the neural network (i.e the values of all the weights for each neuron) is stored and if the error rate for the validation set is greater than at the previous epoch, the neural network is rolled back to its previous state.

2.14.6 Self Organising Feature Map (SOFM)

The SOFM neural network is also known as the Kohonen neural network which is named after its inventor. The SOFM differs in several ways from MLP networks. The SOFM does not have any hidden layers and is trained in an unsupervised mode. That is, it is presented with a set of data and will then classify it in a set of different classes. The output from the SOFM network does not consist of the output of several neurons but rather one output node is selected as the "winner". As with perceptrons, SOFM's can only be applied to linearly separable problems. SOFM networks are typically used for OCR applications and hence why they are mentioned.

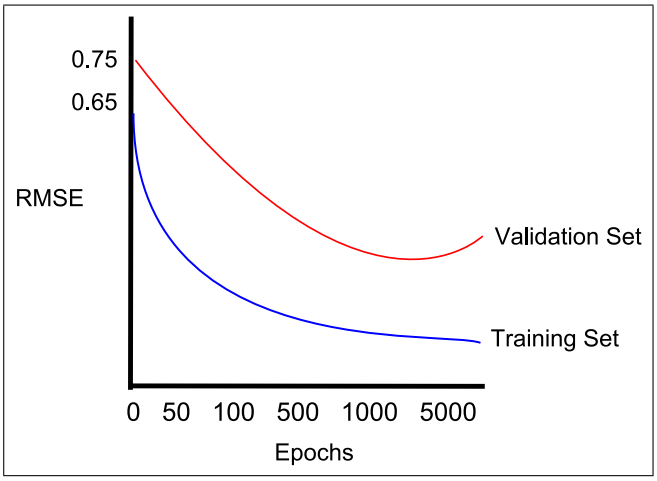


Figure 2.11: An example of overfitting

2.15 Software Implementations of Neural Networks

There exist many software implementations of neural networks, some of which are open source and other commercial applications. The following implementations were evaluated:

1. Joone – Java Object Oriented Neural Engine (open source)
2. OCHRE – Optical Character Recognition (open source)
3. Synapse – by Peltarion (commercial)
4. NeuroDimension – by NeuroSolutions (commercial)

2.16 Other Software Used

1. GUNPlot – It was used to plot all graphs in this report and is an open source project.
2. JFreeChart – It is a Java package that allows graphs to be rendered. This Java package was used to generate the graphs in the implemented application.
3. GIMP – It is an open source graphics package and was extensively used to manipulate images (rotating, resizing, etc)

2.17 Summary

This chapter provided an introduction to Optical Music Recognition and a summary of several research projects was given. In this chapter we learnt that OMR is a complicated task due to the incoherency in the typesetting of music.

Some of the theoretical concepts that will be used for the implementation of the OpenOMR application have been described and include the use of X- and Y-Projections, the Discrete Fourier Transform and Artificial Neural Networks. In the following chapter, we will see how the above concepts are used in the various components of the OMR system.

Chapter 3

Design

This chapter provides a detailed description of the design used to implement the OpenOMR application. A brief overview of the architecture will be described in the section below and an in depth explanation of each component is given in the subsequent sections.

3.1 Overview

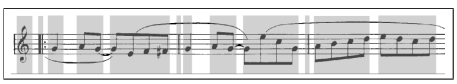
The OMR system developed in this project is based on the O^3MR discussed in section 2.8.6 architecture, although it differs in several ways. Diagram 3.1 shows an overview of the design used to develop the OMR system for this project.

1. **Scan Music Score** – The printed music score is acquired by means of a scanner at a resolution of 300 dpi (dots per inch) in black and white.

2. **Skew Detection** – The Fast Fourier Transform of the image is calculated in order to determine if the staves in the image are skewed. This method will be described following the "stave detection" section as we will first examine why the image needs to be de-skewed.

3. **Stave Detection** – The detection of the staves is probably the most critical component of the OMR architecture as all the components following the stave detection heavily rely on knowing the precise location of the staves in the image. The first step in the stave detection algorithm is to determine the thickness of a stave line and the white space between two stave lines. By using those two parameters along with the y-projection of the image, we will be able to detect the staves present in the image.

4. **Level 0 Segmentation** – This segmentation phase detects all standalone symbols and filled note heads along with groups of symbols. The segments produced by this level are then used by the next segmentation module to detect the presence of filled note heads. An example of a level 0 segment is shown below:



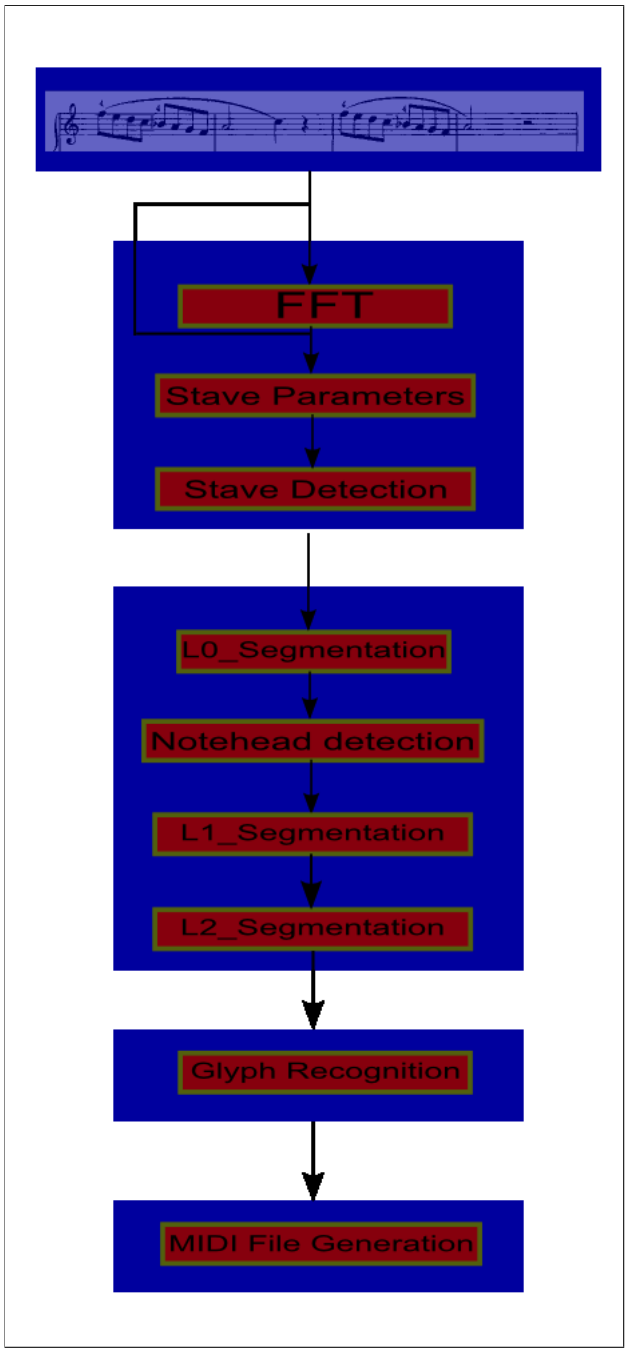
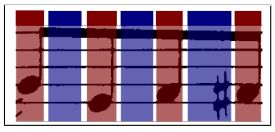


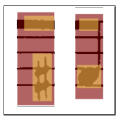
Figure 3.1: Architecture Overview

5. **Note head Detection** – The note head detection module is responsible for analysing the image segments from Level 0 in order to find all filled note heads. If any note heads are found in a Level 0 segment, that segment is labelled as having a node head.

6. **Level 1 Segmentation** – This module scans through the Level 0 segments that have note heads and splits the image into horizontal segments. An example of a level 1 segment is shown below:



7. **Level 2 Segmentation** – This module vertically segments the outputs from the Level 1 segmentation module, separating note heads from other features such as slurs, beams, etc... From now on, we will refer to the outputs from the level 2 segmentation module as glyphs. An example of a level 2 segment is shown below.



8. **Basic Symbol Recognition** – This module performs the recognition of the Level 2 glyphs by means of a MLP artificial neural network.

9. **Midi File Generation** – The pitch and duration for each note is calculated based on the output provided by the neural network and a midi file is generated.

3.2 Programming Language Choice

The C++ and Java programming languages were taking into consideration as choices to implement this application. Although I was more familiar with the C++ programming language, the Java programming language was chosen over C++ for several reasons. Java's portability feature is an attractive one, especially when working with an open source project. The Java programming language also offers an extensive range of classes which are part of the Standard Developers Kit providing the programmer with flexibility and ease. For example, reading an image into memory and manipulating it is easy to accomplish with the BufferedImage class. Java also has a good support for graphics and sound. The BufferedImage class allows different image file types (".jpg", ".bmp", ".png", ...) to be loaded from disk into memory and manipulated. The BufferedImage class provides methods to read or set individual pixel values int he image.

3.3 Stave Detection

This section provides a description of the method used to locate the stave(s) present in the image being processed. As most of the subsequent modules rely heavily on knowing the precise location of the staves, this is one of the most critical phases of the OMR architecture. If the stave detection algorithm erroneously detects a stave which isn't one, the subsequent modules will process that as being a stave and the accuracy of the OMR system will be compromised. Similarly, determining the exact coordinates of the stave lines is important as this will be used in the midi generation module to determine the exact pitch of notes.

3.3.1 Stave Line Parameter Detection

The first step in locating the stave(s) in the music score involves approximating the thickness of each stave line and the white space in between each stave line. Since the staves are the most predominant feature in a music score, a histogram of all consecutive black and white pixel will reveal the thickness of the stave lines and the distance between two stave lines. The method used to determine the two aforementioned parameters is the RLE algorithm mentioned in section 2.11.

By traversing the music score column by column, we can apply the RLE algorithm and record the number of consecutive black and white pixel runs in two arrays. After applying the RLE algorithm, the arrays contain the number of times a certain amount of consecutive black or white pixels occurred. The size of both arrays was chosen to contain 100 integers for the simple reason that a stave line and the distance between two stave lines is unlikely to exceed 100 pixels. The typical thickness of a stave line ranges anywhere from 3-6 pixels and the distance between two stave lines can range from 15 to 30 pixels. This of course depends on the typesetting of the music score [7].

We now define a range of lower and upper values for the thickness of the stave lines and the distance between two stave lines as follows:

- **d1** is $\frac{1}{3}$ the value found of the distance between two stave lines
- **d2** is $\frac{3}{2}$ the value found of the distance between two stave lines
- **n1** is $\frac{1}{3}$ the value found for the stave line thickness
- **n2** is $\frac{3}{2}$ the value found for the stave line thickness

These parameters will be referred to in the following sections.

The RLE algorithm was applied to the image shown in figure 3.3 and the results are shown in figure 3.2. We notice a peak in both graphs. The peak in the graph on the left hand side is the thickness of the stave lines whilst the peak in the graph to the right is the distance between two stave lines. In other words the sequence of 19 consecutive white pixels occurred approximately 9600 times. The sequence of 3 consecutive black pixels occurred approximately 13000 times. Hence we can estimate the stave line thickness to be 3 pixels and the distance between two stave lines to be 19 pixels. Based on this information, we will now

be able to proceed to the stave detection, knowing that the height of one stave can be approximated by:

$$Height = 5d + 4n$$

Where d is the thickness of a stave line and n is the distance between two stave lines. For the example given above, we would expect the height of one stave to be 91 pixels. This is only an approximation as the height of a stave can vary depending on the quality of the scanned music score.

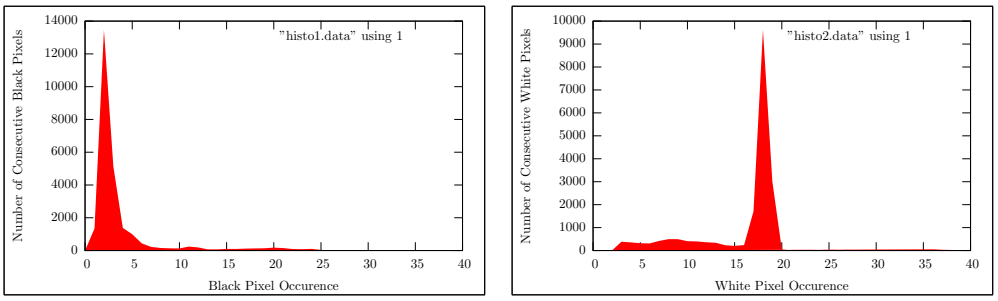


Figure 3.2: Black and White pixel Histogram for "Song for Guy" excerpt by Elton John.

3.3.2 Stave Detection

In order to detect and locate the position of the staves, we take the y-projection of the image and obtain a one-dimensional array whose contents is the summation of all black pixels along the y-axis.

The resulting array is then traversed and all local maximas are found. Having previously calculated the approximate thickness of a stave line and the distance between two stave lines, we can develop an algorithm that will scan the array, looking for five equidistant peaks in range $d1 \ldots d2$ each of approximately the same value. The peaks therefore do not have to have a value above a certain threshold but rather must be within a certain percentage of their neighbouring peaks. This therefore enables staves which may not take up the entire width of the image to be found. Figure 3.4 shows the Y-Projection of the image in figure 3.3.

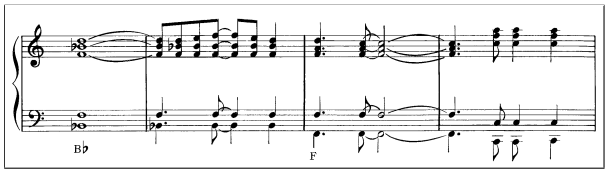


Figure 3.3: Elton John Song

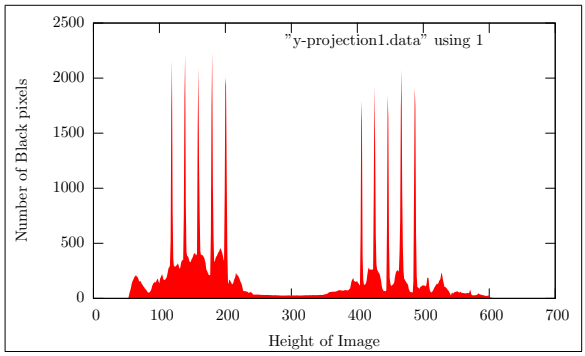


Figure 3.4: Y-Projection of figure 3.3

The algorithm described above works well when the staves are non-skewed but will however fail if the stave is skewed. Let us imagine a music score which contains exactly one stave with each stave line having a height of exactly 1 pixel. Assuming that the stave is the ONLY feature present in the image (i.e nothing but the stave lines exist in the image), the result of taking the y-projection of the image will give us an array with exactly 5 non-zero elements (the peaks) at equidistant locations in the array. If the score is now rotated at an angle and the y-projection of the rotated image is calculated, each stave line will no longer contribute to exactly one element in the array but will rather contribute to several neighbouring elements. We therefore no longer see distinct peaks when a graph of the y-projection is plotted. Figure 3.6 is the graph of the y-projection of the image in figure 3.5 and we can see that five equidistant lines as seen in figure 3.4 are no longer visible.

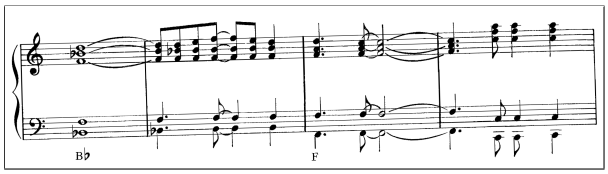


Figure 3.5: Elton John Song Skewed

When relying on the y-projection to locate the staves in a music score, it is therefore important to determine the angle by which the staves are skewed. The next section discusses the optional FFT module that is used to detect the that.

All staves found in the image are stored in a list. The following information is kept in a data structure which is used in the subsequent modules:

1. Top – The uppermost y-coordinate of the stave
2. Bottom – The bottom y-coordinate of the stave

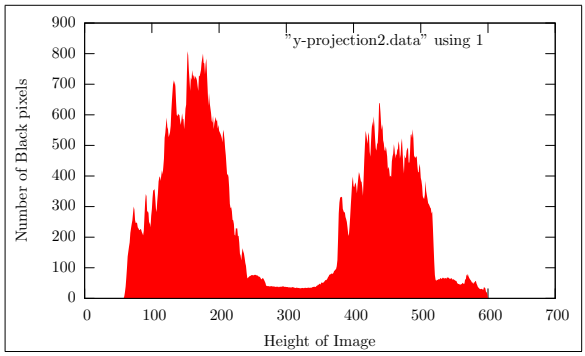


Figure 3.6: Elton John Song

3. Left – The leftmost x-coordinate of the stave
4. Right – The rightmost x-coordinate of the stave

3.4 Skew Correction

This step is optional and will not automatically be performed by the OpenOMR application. The main reason for making it optional is that the DFT is computationally expensive to perform in comparison to the time taken for the other modules to complete their execution. There exists a fast method to approximate the DFT and this is known as the Fast Fourier Transform (FFT). While the Discrete Fourier Transform is computed in $O\left(N^2\right)$ time, the FFT can be computed in $O\left(NlogN\right)$ time. Therefore an algorithm to compute the FFT was used in order to improve the efficiency of this component.

Figure 3.7 shows the graph of the magnitude components when the Fourier Transform of the image in figure 3.5 is taken. When looking at figure 3.5 we notice that there exists an almost vertical line tilted by a few degrees to the right. From this, the angle by which the staves are rotated can be calculated as mentioned in section 2.13.

This method has shown to produce accurate results (which will be discussed in the chapter "results"). There are however some circumstances under which this method will not produce accurate results. It can happen that only a few staves on a page are skewed and the rest are intact. In such circumstances, the method described above may provide an indication as to how much the image needs to be rotated by, but in reality, only a few staves may need to be rotated. Therefore by rotating the image, the staves which were originally non-skewed will now be skewed. In order to avoid this kind of situation, the Fourier Transform of smaller parts of the image can be taken. This is known a windowed Fourier Transform and is described in [3]. The current implementation does not support the windowed Fourier Transform and this is feature which is suggested for a future version of the application and is discussed in chapter 7.

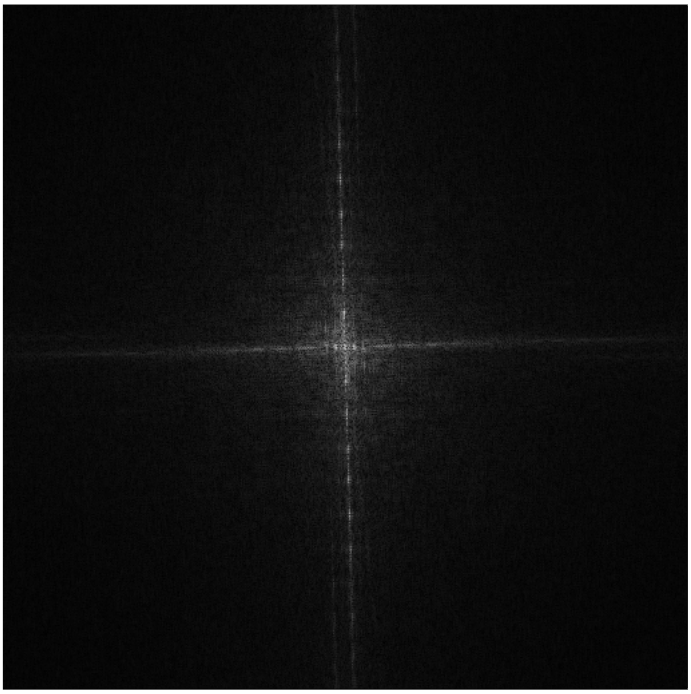


Figure 3.7: FFT of figure 3.5

3.5 Image Segmentation (Level 0)

The Level 0 segmentation module attempts to find groups of symbols, individual symbols, and individual note heads and labels them. Groups of symbols are defined as being groups of notes that are attached to one another with beams, or glyphs such as accidentals which are close to one another. Figure 3.8 shows an example of what we would expect a level 0 segment to look like. The main purpose of this module is to reduce the search space of the image in order to reduce the processing time required for the subsequent phases. It is unnecessary to process areas of the image which we are certain contain no musical symbols.

The method used to locate groups of symbols relies on the x-projection. The x-projection for each stave is calculated and stored in an array for post processing. We saw above how we could determine the height of a stave and we now introduce an equation that defines the minimum number of black pixels that are required in order for the image segment to have something other than just the stave lines.

$$MinBlackPixels = 5n_2$$

We choose $n2$ (i.e maximum stave line height) here as we want to try to set a minimum threshold that will allow us to find empty staves. The array containing the x-projection is then scanned and as soon as we find values above the minimum threshold, we start a counter. The process then continues until a value below the minimum threshold is found. The counter then defines the

width of the new segment that was found and its coordinates are stored in a data structure that is discussed in section 4.1.7.

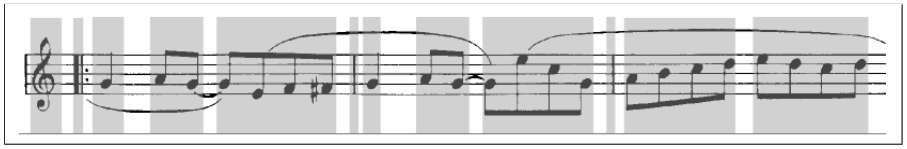


Figure 3.8: Sample Level 0 Segmentation

3.6 Note Head Detection

The purpose of this module is to detect and label any segments containing filled note heads. It does so by analysing the y-projection of the Level 0 segments.

A level 0 segment is shown in figure 3.9. In this particular example, we notice that the Level 0 segment contains a total of six note heads. This is reflected in the graph on the right of the image, where we can see six peaks. It should also be noted that the height of the peaks relates to the y-coordinate of the position of the note in the level 0 segment.

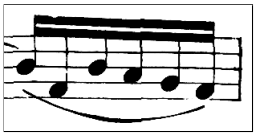
At this stage, the stems of the notes are also located. All filled note heads have a stem attached to them and the only reason a stem would not be present is if the scan of the music score is of poor quality. Stems are a distinct feature in an image as they are fairly tall in comparison to the height of the note head. By using those principals, we can scan the portion of the image in which a note head was found and determine if the stem is to the left or to the right of the note. In section 3.15, we will see that in order to determine the duration of the note, it is important to know on which side of the note the stem is positioned.

Although the current implementation of the note head detection algorithm is able to locate the stem of a note, this information is not used an extra check to make sure that the algorithm correctly detected a note head. As we will see in chapter 6, some notes are falsely detected and in order to minimise the number of false negatives, we could require the note head detection algorithm to find a stem to the left or to the right of it in order to classify it as being a note head.

All note heads which were found during this segmentation process along with the position of the stem (left or right) are stored in a data structure which is contained within the level 0 segment data structure. Additionally, a special flag "note_found" is set in the level 0 segment.

3.7 Symbol Segmentation (Level 1)

In the current implementation, the Level 1 segmentation module will only process Level 0 segments that have been labelled as having a note head. The idea



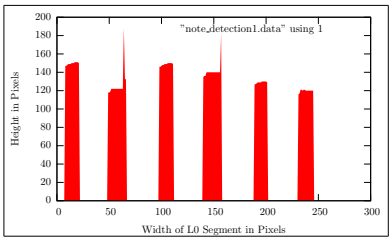


Figure 3.9: Level 0 segment and its corresponding graph after the note head detection algorithm is applied.

behind this segmentation level is to separate symbols vertically. That is, we would expect this module to produce new level 1 segments as shown in figure 3.10. The algorithm used for this segmentation module is described in section 4.1.7.

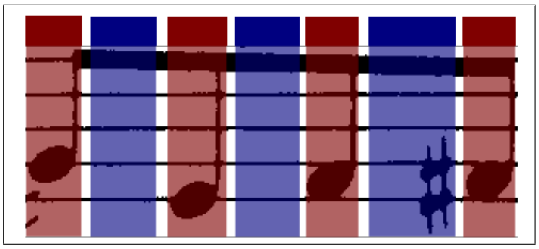


Figure 3.10: L1 Segment

3.8 Note Processing (Level 2)

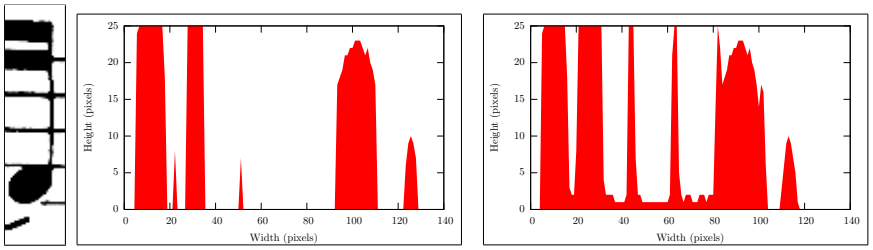


Figure 3.11: Y-Projection of Level 1 segment

The goal of this module is to separate note heads from other symbols. The level 1 segmentation module has separated the symbols vertically and this module now horizontally segments each of the level 1 segments. This is done by taking the y-projection of the image and applying two filters.

1. Stem removal – The stem is removed from the y-projection of the level 1 segment. Since we know the approximate width of the stem, we can

traverse the y-projection in search of a sequence of values within that range and reset them (i.e set them to 0).

2. Stave line removal – The stave lines are removed from the projection. We know the approximate coordinates for all five stave lines and we can make use of that knowledge to remove them from the projection.

Figure 3.11 shows the segmentation process. A level 1 segment is input into the level 2 segmentation module and the first filter (stem removal) is applied to the y-projection. The y-projection is then passed through a second filter which removes the stave lines. The graph to the left shows the result after the first filter is applied and the graph to the right shows the result after the second filter is applied.

Having applied these two filters, we can now search for glyphs in the projection. Any sequence of non-zero values in the projection which are separated by less than $D1$ consecutive 0 values in the projection are considered to be one glyph. This is an experimental value that was chosen based on the following: glyphs which are attached together such as beams do not appear to be separated by more than the distance between two stave lines. This is of course a parameter which can be varied for experimental purposes.

The relevant coordinates of each glyph found are then stored for processing by the neural network.

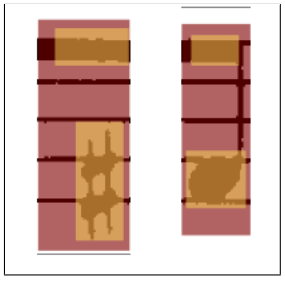


Figure 3.12: L2 Segment

3.9 Neural Network

Several design factors had to be taken into consideration when designing and choosing the type of neural network that would be used for this project. Two types of neural networks were considered and are listed below:

1. Multi-Layer Perceptrons (MLP) Backpropagation Neural Network
2. Self Organising Feature Map (SOFM) Neural Network

As we saw in section 2.14.1, the MLP network can be trained in supervised/unsupervised mode whilst the Kohonen network can only be trained in unsupervised mode. We also noted that the Kohonen network has one output node per class of data and only one of those output nodes fires when the network

is interrogated. In contrast, the MLP network will output a percent confidence for each output node when interrogated and this is one of the main reasons for choosing the MLP network for the design of our application. The percent confidence of the output nodes is used in the current implementation of the application. If the percent confidence of the classified symbol is below a certain percentage, it is rejected. A future implementation could use this information to then investigate what the supposedly recognised symbol might be.

3.9.1 Neural Network Design Considerations and Choices

As the implementation of a neural network was outside the scope of this project, an existing implementation of a neural network was used. The neural network software implementations the were surveyed in section 2.15 were tested and the commercial versions were not used as this project was to become an open source one.

In order to test each of those neural networks to determine which implementation would be best for our application, training data had to be gathered. In our case, the training data was composed of musical glyphs coming from different music scores. Those were collected by cutting out hundreds of different glyphs and saving them into separate files. In chapter 7 we will discuss how this can be avoided by having our application work semi-automatically. In chapter 6, we discuss which types of symbols and how many of each symbol was used to train the neural network.

The neural network engine from the OCHRE applet was taken and by providing several modifications our data was trained with that neural network engine. Encouraging results were obtained but this implementation suffered a flaw in the sense that it did not validate the training data. As we discussed in section 3.9, it is extremely important to train a neural network with a training data set and a validation data set in order to avoid "overfitting". Albeit working on our training set, it was impossible to detect when overfitting was occurring and the OCHRE neural network engine was dropped from the OpenOMR application. The Joone implementation was tested again and with persistence, it worked (and also provided a data validation feature when training the network).

We will now discuss the architecture of the neural network that was chosen and how the data is presented to the neural network. The number of inputs chosen for the neural network reflects the size of the normalised image and 128 inputs were chosen. This is a parameter which can of course be varied but it was chosen as the O^3MR architecture uses a 8x16 (128 values) normalised image as its input to the neural network. It was chosen to have one output node per glyph category.

3.9.2 Normalising Image Segments

The image segments produced in the Level 2 segmentation phase will have different sizes and with a fixed amount of input nodes in our neural network, we must normalise (or downsample) each image segment to reflect the amount of input nodes from our neural network. In our case, one pixel per input node is

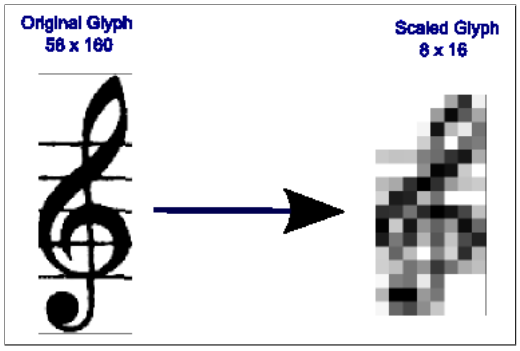


Figure 3.13: Normalising a Treble Glyph

used. The image can be normalised by selecting a scaling algorithm depending on the quality of the normalised image we require. As will be seen in section 4.1.1 a scaling algorithm that is available in the **Image** class of the Java SDK was used. The reason for normalising the level 2 segments is that the inputs to the neural network are fixed. Figure 3.13 shows a treble clef glyph segment and its corresponding normalised image (8 by 16).

Figure 3.14 shows how the image is input into the neural network. The image is represented as a two-dimensional array. In order to input the image into the neural network, we can convert the 8 by 16 image into a single array by concatenating each column of the image one after another. Images can be represented in several colour models of which include the RGB and HSV (or HSB) model. The RGB model represents each pixel in the image as a Red, Green and Blue value whilst the HSV represents each pixel as a Hue, Saturation and Value. The Value of the HSV colour model represents the brightness of the pixel and this is what we use as our input into each node of the neural network.

3.10 Midi Music Generation

The goal of this module is to generate a midi file which can be played through the computer speakers. In essence, this module takes all the level 2 segments that were produced and constructs a midi representation of them based on some rules. We are concerned with finding the pitch and duration for each note. The pitch for each note is relatively simple to find as we already know their x and y coordinates from the note head detection module. We now need to determine the duration of the note and this is done based on which side of the note the stem is located. If the stem is located on the left of the note, we need to look at the glyph directly below the note and if the stem is on the right, we need to look at the segment to the top right of the node. This follows the way that music scores are written in general.

Assuming that our neural network would output the string “quaver_line” when it recognises a quaver beam and “crotchet” when it recognises a crotchet, we would expect the following sequence of strings to be output when the score in figure 3.15 is processed.

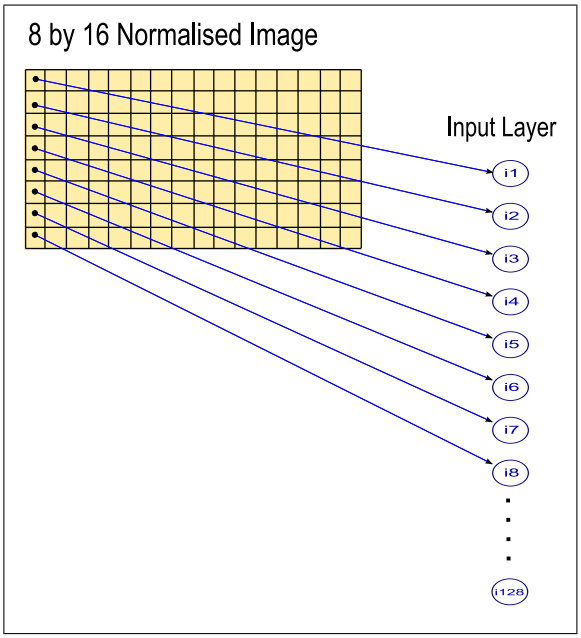


Figure 3.14: Normalised image input into NN

```
quaver_line
crotchet
quaver_line
quaver_line
crotchet
quaver_line
quaver_line
crotchet
quaver_line
sharp
quaver_line
crotchet
```

3.11 Summary

In this chapter, we discussed the different modules used for the design of the OpenOMR application. We saw how the staves are detected by taking the Y-Projection of the image and looking for five equidistant peaks in the projection. When staves are skewed, the staves are impossible to locate and we therefore introduced the FFT module that is able to detect the angle by which staves are skewed. The Level 0 segmentation module was then introduced and its main objective is to isolate groups of symbols. We then saw how note heads are detected by using a custom a combination of the RLE algorithm and y-projections. Once all note heads are detected, the level 0 segments are passed to the level 1 segmentation module which isolates symbols vertically. The last

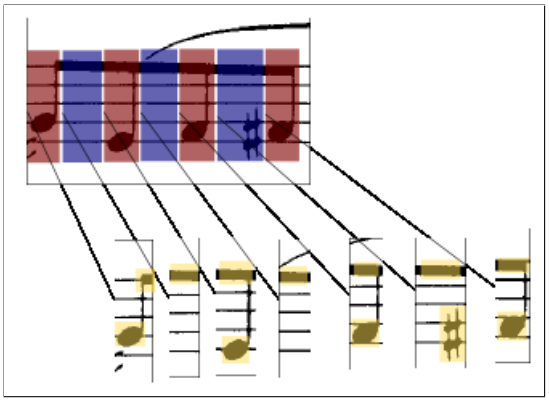


Figure 3.15: Note recognition process

phase of the segmentation process is the level 2 segmentation and it separates glyphs in the horizontal direction. The segmentation phase is complete after level 2, and the level 2 segment is input into the neural network for recognition. Finally, a midi file is constructed from all the level 2 segments.

Chapter 4

Implementation

This chapter discusses several considerations taken into account when the OpenOMR system was implemented. Notably justifications as to why the Java programming language was used are provided. A detailed description of the Java package structure of the project is given and in some cases, examples of using those classes are provided.

Throughout the implementation phase of this project, it was important to consider the future development of this application. As one of the main goals was to turn this project into an open source one, it was crucial to keep the structure organised in such a way that will be understandable by others who wish to help develop the OpenOMR application.

The package structure was designed in such a way that individual components could be altered or changed if required (neural network, FFT implementation, ...). For example, the OCHRE neural network implementation was replaced by the Joone neural network engine. This shows that with a little effort, the whole neural network engine was replaced.

4.1 Package Structure

As an open source project was created on sourceforge.net, a name for the open source project had to be chosen and the project was named **openOMR**. Therefore the package structure of the OpenOMR application takes the form: openomr.packagename. The package layout is shown in figure 4.1.

4.1.1 The openomr.ann Package

This section discusses the implementation of the artificial neural network and how it is used. As mentioned in section 3.9, the JOONE implementation was used.

Data Organisation

It was important to structure the way in which the training, validation and testing data were stored. A directory structure to store the data sets was devised and is shown in figure 4.2.

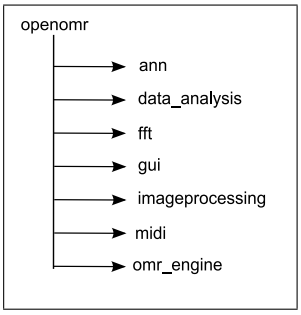


Figure 4.1: Package layout of OpenOMR

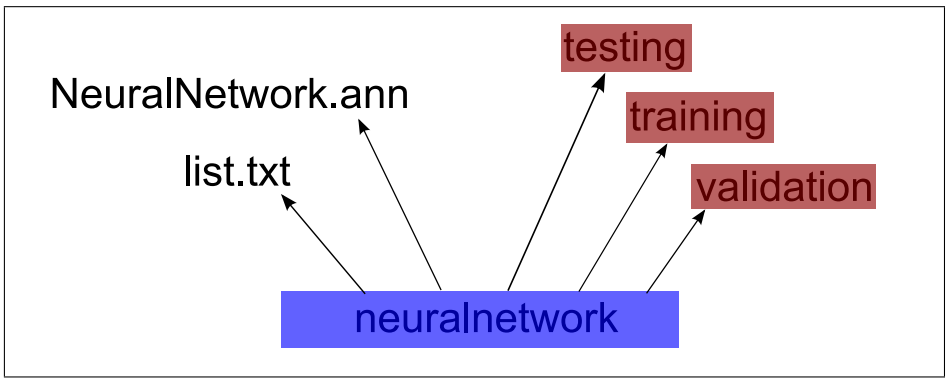


Figure 4.2: Directory structure of the ANN files

1. **NeuralNetwork.ann** – This file contains the saved state of the neural network after the training phase has been completed. Every time the OpenOMR application is loaded, it looks for this file in order to restore the saved state of the neural network.

2. **list.txt** – This file contains the list of glyphs that the neural network has been trained to recognise. This is simply a text file and each line contains the name of a glyph terminated by the end-of-line marker.

3. **training** – This is a directory which contains all the image files that the neural network should use when in training mode. It does not matter how the images are arranged in this directory (i.e sub-directories are allowed) but the file name is what matters. Each file name must begin with one of the lines from the **lists.txt** file in order to be processed when the neural network is trained. For example, if the file **lists.txt** contains a line "crotchet", then a legal name would be "crotchet1.png" but "1crotchet.png" would be illegal.

4. **validation** – This is a directory which contains all the image files of the different glyphs that have been selected for the validation set when training the neural network. The same rules as discussed for the training directory apply for the structure of this directory.

5. **testing** – This is a directory which contains all the image files of the different glyphs that have been selected as the test set. The same rules as discussed for the training directory apply for the structure of this directory.

Classes

This openomr.ann package consists of the following classes:

1. **Trainer** – This class is responsible for creating and training the neural network.
2. **DataPrepare** – This class has a set of methods that are used to prepare the data with which the neural network is to be trained. Tasks such as normalising an image are performed within this class.
3. **Interrogator** – This class is designed to interrogate the neural network. That is, it can load the state of the neural network (NeuralNetwork.ann) if its not already loaded in memory and pass a normalised image into the network that needs to be classified.
4. **BatchModeTester** – This class will normalise and feed all the images present in the **testing** directory and produce an output file. Each line of this output file contains:
   (a) The name of the glyph as found in the **testing** directory
   (b) The name of the glyph as classified by the neural network
   (c) The percent confidence of the neural network

Training the Neural Network

We will now look at how the data is input into the neural network. In order to train the neural network, we have to provide it with training and validation data sets (in our case, images). The images are read from the 'neuralNetwork' directory and the `getScaledInstance()` method of the Java **Image** class is used to scale each image to an 8 pixel wide by 16 pixel high image.

The following two arrays must be created for both the training and validation sets.

1. Input training data – This is a two-dimensional array with 128 columns (for each pixel value of in the image) and $amountSymbols$ rows. $amountSymbols$ specifies the number of images that were found in the training or validation directories. Each column is formed by reading an image into memory and converting the two-dimensional representation of the image into a one-dimensional representation. The greyscale intensity component of each pixel is used to fill the arrays.
2. Desired training data – This is a two-dimensional array with $numSymbols$ columns and $amountSymbols$ rows where $numSymbols$ is the amount of different symbol classes we are training or validating the neural network with.

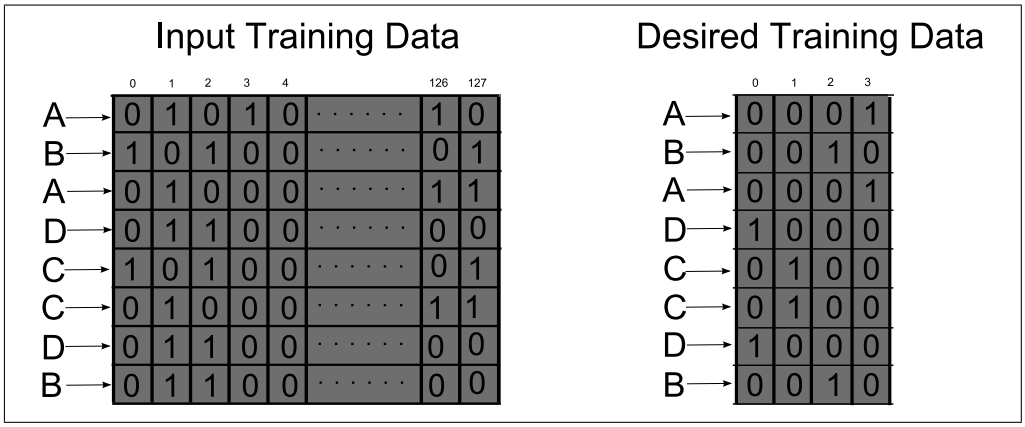


Figure 4.3: A representation of the input training data and desired training data arrays

Figure 4.3 depicts the layout of the input training data and desired training data arrays.

When the network is interrogated in order to classify a glyph, it will output a one-dimensional array of type double with *amountSymbols* elements. That array contains the percent confidence values for each symbol class. That array is then parsed, and the value with the highest percent confidence is taken to be the glyph classified by the neural network.

4.1.2 openomr.data_analysis

This package is mainly used to either generate graphs or save data in a text format. These classes are mainly intended for developers who wish to either save or view data in a graphical format since we extensively use x and y-projections and it can be useful to graphically view them. It consists of the following two classes:

1. **GNUPlotGenerator** – This class will generate a text file in a format that will understood by the GNU Plot program in order to generate charts. Its constructor and method are provided below:

   (a) `GNUPlotGenerator(String fname, int data[], int size)`

   (b) `void generateFile()`

2. **XYChart** – This class will render a chart in the form of a BufferedImage. This is convenient from a developer's point of view as charts can easily be generated and displayed in a GUI. Its constructor and method are as follows:

   (a) `XYChart(name, int data[], int size)`

   (b) `BufferedImage getChart(int width, int height)`
   – where width and height specify the size size of the image to be rendered.

4.1.3 openomr.fft

This package contains the **FFT** class which calculates the Fast Fourier Transform of a BufferedImage. This class was originally developed in C++ by Stephen Murrell at the University of Miami and we converted it to Java. Its constructor and public methods are as follows:

1. `public FFT(BufferedImage buffImage, int size)`
2. `public void doFFT()`
3. `public double getRotationAngle()`
4. `public BufferedImage getImage()`

This class can be used as follows:

```
FFT fft = new FFT(buffImage, 2048);
fft.doFFT();
double angleRad = fft.getRotationAngle();
```

4.1.4 openomr.gui

An extensive graphical user interface was developed for several reasons. From a developers point of view, a graphical user interface is invaluable in order to test new functionalities. It provides a quick way to test new features on several music scores without having to waste time reloading the whole application each time.

The graphical user interface was very useful when checking that the segmentation was properly done, that the staves were being correctly recognised and that the note heads were being properly recognised.

From a users point of view, a graphical user interface provides a certain level of comfort especially for novice users. The implementation of the GUI was programmed with Swing and new features/functionalities can easily be added or changed.

Screen shots of the GUI are provided in the user manual in Appendix A.

4.1.5 openomr.imageprocessing

This package contains two classes, one of which is used to rotate an image and the other to create and copy a new buffered image.

1. **RotateImage** – This class is used by the openomr.FFT package to rotate an image when correcting skewed staves.
2. **CopyImage** – This class is used to copy a BufferedImage to a new one.

4.1.6 openomr.midi

This package contains the class that is responsible for generating the midi file. At present, we are limited to representing monophonic scores and after describing how this class works, we will show how it can be extended to provide support for polyphonic scores.

1. `public MidiGenerator(int key, int tempo, int resolution)`

2. `public void start()`

   – This method will play the midi representation of the file that was created through the computer speakers.

3. `public void add(int note)`

   – Add a note with pitch `note`with a duration of 1 beat to the current track

4. `public void add(int note, int length)`

   – Add a note with pitch `note`and with a duration of `length`beats to the current track

5. `public void addSilence(int length)`

   – Add a rest of duration `length`to the current track

A midi representation of a scale could then be created and played through the computer speakers as follows:

```
MidiGenerator midi = new MidiGenerator(60, 30, 8);
midi.add(62);
midi.addSilence(1);
midi.add(64);
midi.addSilence(1);
midi.add(65);
...
midi.addSilence(73);
midi.play();
```

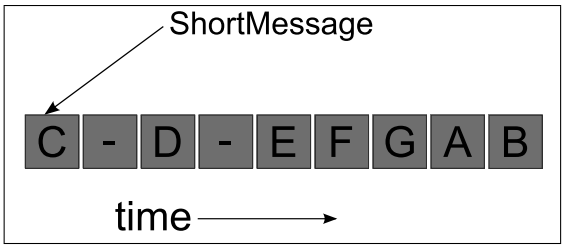


Figure 4.4: Midi representation of 'C' scale with two rests represented by '-'

We will now briefly look at how we could extend the void add(int note) method in order to add 2 notes to the midi event. The void add(int note) method is defined as follows:

```
private void addStartEvent(int note) throws InvalidMidiDataException
{
   ShortMessage message = new ShortMessage();
   message.setMessage(ShortMessage.NOTE_ON, 0, note, volume());
   track.add(new MidiEvent(message, pos));
}
```

The midi representation of the song is stored in a track. A ShortMessage object allows us to create notes and silences in a midi track. If we wanted to add two notes for the same beat, we could modify the above method as follows:

```
private void addStartEvent(int note1, int note2) throws InvalidMidiDataException
{
   ShortMessage message = new ShortMessage();
   message.setMessage(ShortMessage.NOTE_ON, 0, note1, volume());
   track.add(new MidiEvent(message, pos));
   message.setMessage(ShortMessage.NOTE_ON, 0, note2, volume());
   track.add(new MidiEvent(message, pos));
}
```

We are now able to play two notes at the same time. This shows that the midi file generator can be extended to provide a richer set of features. At present time, there is no support to save the midi representation of the song to a file but this is a suggested feature for a future version of this application and is discussed in chapter 7.

4.1.7 openomr.omr_engine

This package contains the core OMR engine that is used to recognise a music score. The most important classes that are part of this package are described below.

1. **XProjection** – This class calculates the x-projection of a BufferedImage. It is currently being used for the development of a more reliable note head detection algorithm which will be present in a future release of this application and is discussed in chapter 7.

2. **YProjection** – This class calculates the y-projection of a BufferedImage. It is currently used by the stave detection, note head detection and level 2 segmentation algorithms. The public methods available are:

   (a) `void calcYProjection(int startH, int endH, int startW, int endW)`
   -startH is the starting y-coordinate for the y-projection
   -endH is the end y-coordinate for the y-projection
   -startW is the starting x-coordinate for the y-projection
   -endW is the ending x-coordinate for the y-projection

   (b) `int[ ] getYProjection()`
   This method simply returns the y-projection calculated

   (c) `void print YProjection()`
   This method is intended for debugging purposes and will print the y-projection calculated to standard output.

3. **StaveParameters** – This class is responsible for determining the stave parameters which were discussed in section 3.3.1. Its constructor takes a BufferedImage as a parameter. The following public methods are provided and return the parameters described in section 3.3.1.

(a) `int getD1()`

(b) `int getD2()`

(c) `int getN1()`

(d) `int getN2()`

4. **StaveDetection**

   This class is responsible for detecting all the stave lines present in an image. Its constructor is given the y-projection of the image and the stave line parameters. It then looks for five equidistant peaks in the projection and stores the coordinates in an object which has the following fields:

   ```
   private LinkedList<L0_Segment> symbolPos;
   private int top;
   private int bottom;
   private int staveNumber;
   private int noteDistance;
   private int start;
   private int end;
   ```

   The top and bottom field store the y-coordinates at which the stave found start and ends. The start and end fields store the starting and ending x-coordinates. When all level 0 segments are found at a later stage, they are stored in a linked list within this object (symbolPos field).

   The public methods are:

   (a) `void locateStaves()`

   (b) `void calcNoteDistance()`

5. **L0_Segment**

   This class stores the starting and ending coordinates of the level 0 segments and also indicates whether or not a note is present in that segment. This class also analyses the level 0 segments to find any level 1 segments and for each level 1 segment found, a new level 1 object is created. That new level 1 object is stored in the **L0_Segment** class.

   ```
   public int start;
   public int stop;
   public boolean hasNote;
   \ldots
   ```

6. **L1_Segment** The algorithm to vertically separate note heads from other symbols works as follows:

   (a) If a Level 0 segment begins with a note head, create a Level 1 segment which has the height of the level 0 segment and the width of the note head. If no note heads are present at the start of the level 0 segment, create a level 1 segment which has the height of the level 0 segment. The width of this segment will be defined to be the distance from the start of the level 0 segment until the start of the first note head in the level 0 segment.

(b) If a level 1 segment containing a note head was created in the previous step, take the ending coordinate of that note head and find the beginning coordinate of the next note head and create a new level 1 segment of that width. If the previous level 1 segment did not contain a note, find the next note and create a new level 1 segment with a width of the note head. Repeat this process until the whole width of the level 0 segment has been processed.

Of course we can set a threshold so that the segmentation module does not create level 1 segments which would be extremely small (in the region of say 5 pixels). Such small segments would probably not be of much use and we can therefore omit them. A small segment could appear if notes are tightly packed together or if we are processing the end of a segment.

This class records the starting and ending coordinates of the level 1 segments. This class also analyses all the level 1 segments in order to produce level 2 segments.

```
private int xStart;
private int xStop;
\ldots
```

7. **L2_Segment**

This class stores information about all Level 2 segments as described in section 3.8. When the Level 2 segment is passed through the neural network, the output from the neural network in terms of symbol name and percent confidence is stored in this object.

```
private int yPosition;
private String symbolName;
private double accuracy;
```

8. **NoteHead**

We can approximate a note head to have a height of $2N2 + D2$ by considering the case when a note head is placed between two stave lines. The process of detecting the note heads is as follows:

(a) Process each column of the image one by one.

(b) Apply the RLE algorithm to a column and where a run of black pixels exceeds $D2$, set all the values for that run to 0.

(c) Iterate through the column again and remember the longest run of black pixels and record this in a new array if the value of the run exceeds $2 * N$. This ensures that runs which are similar to the height of a note are recorded.

(d) Repeat steps 2 and 3 until all columns have been processed.

(e) We then apply a simple filter which eliminates any values which are less than $\frac{D}{2}$ wide. We are now left with an array which contains sets of peaks. Those peaks correspond to the note heads found by the above steps and we locate those notes by finding the peaks in the array.

This class stores information about all the note heads. It stores the x and y coordinates of each note along with the stem position (left=0, right=1).

```
public int x;
public int y;
public int stemInfo;
```

9. **PitchCalculation**

This class calculates the pitch of a note based on the y-coordinate of the note. We know the distance between two notes, the y-coordinate of the last stave line in a stave and the y-coordinate of the centre of the note and given that information, we can calculate the pitch of the note as follows:

$$Note = \left\lfloor \frac{refPos - yPos}{\frac{1}{2}noteDistance} \right\rfloor$$

Figure 4.5 shows the different parameters required to calculate the pitch of a note.

Assuming: $refPos = 465$, $yPos = 500$ and $noteDistance = 35$ we can calculate $Note = -2$ from which we can determine the pitch of the note as follows:

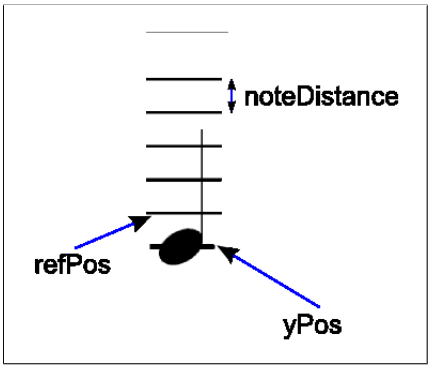


Figure 4.5: Calculating the pitch of a note

We know that there are 7 notes ('A'-'G') and that $refPos$ is located at note 'E'. Therefore given the value of $Note$, we can determine the letter corresponding to the note.

```
if (Note \% 7 == 0)
  System.out.print("E ");
else if (Note % 7 == 1)
  System.out.print("F ");
else if (Note % 7 == 2)
  System.out.print("G ");
else if (Note % 7 == 3)
  System.out.print("A ");
```

```
else if (Note % 7 == 4)
  System.out.print("B ");
else if (Note % 7 == 5)
  System.out.print("C ");
else if (Note % 7 == 6)
  System.out.print("D ");
```

From our example above, $-2 \bmod 7 = 5$ which correctly tells us that the note is a 'C'.

4.2 Using the OpenOMR Engine

We discussed the individual packages and their functionality within the OpenOMR application. We will now discuss how the core engine of the OpenOMR package could be used to integrate it with another application or as a stand-alone application. One example of a stand-alone application not requiring a GUI is a music recognition web service provided through a website. A music score is uploaded through a website and the OpenOMR engine processes that score at the server side and uploads a Midi file to the user representing the recognised score.

The code below shows how an image (assumed to be loaded and stored in a variable buffImage) could be recognised and played through the computer speakers.

```
//Take the y-projection of image
YProjection yproj = new YProjection(buffImage);
yproj.calcYProjection(0, buffImage.getHeight(), 0, buffImage.getWidth());

//Calculate the stave line parameters
StaveParameters params = gui.getStaveLineParameters();
params.calcParameters();

//Find all the staves
StaveDetection staveDetection = new StaveDetection(yproj, params);
staveDetection.locateStaves();
staveDetection.calcNoteDistance();

//Segment image (Level0, 1 and 2) and recognise all symbols
OMREngine engine = new OMREngine(buffImage, staveDetection, neuralNet);
engine.processAll();
```

At this stage, the OMREngine has now processed the entire score and has generated an internal representation of the recognised score. We can now generate and play a midi file as follows:

```
ScoreGenerator midiFile = new ScoreGenerator(staveDetection.getStaveList());
```

```
midiFile.makeSong(64);
midiFile.start();
```

4.3 summary

In this chapter we discussed the implementation details of the different Java Packages used for the OpenOMR application. In particular, we payed close attention to the openomr.ann and openomr.omr_engine packages as they are the core of the application. We also identified how the openomr.midi package can be extended to provide polyphonic support in future releases of this application.

Chapter 5

Testing

This chapter will discuss how the overall system was tested and the results will be presented in the next chapter. It is critical to thoroughly test an application in order to identify which components should be improved in future releases.

5.1 Testing Environment

The application was tested under the WindowsXP, Linux and Mac OS X operating systems. Even though Java programming language was selected due to its cross-platform compatibility, some of the GUI components did not always display as expected on the Mas OS X platform. It was therefore important to test any new GUI features on the three operating systems in order to preserve the cross-compatibility feature of the application.

5.2 False Positives and False Negatives

"We want to decide if a person will fail as a police officer. So a **false positive** is if we incorrectly say that a person will fail. A **false negative** on the other hand is if we incorrectly predict that person won't fail. " [22]

In terms of evaluating our system:

1. **A false positive** – is for example if the application falsely identifies a note head which isn't one.
2. **A false negative** – is for example when the application fails to detect a note present in the image.

5.2.1 FFT Module

The FFT module will be tested by taking one score and rotating it at different angles. The score used to test this module will have all its staves perfectly aligned with the horizontal axis. That score will then be rotated in an image editor (such as Gimp) and tested with the FFT module. The case for which only a few staves in the score are rotated will also be simulated.

The test cases are as follows:

1. Rotate the original image by 0°
2. Rotate the original image by 0.2°
3. Rotate the original image by 0.5°
4. Rotate the original image by 1°
5. Rotate the original image by 2°
6. Only rotate the bottom half of the original image by 0.5°

All of the images above will be input into the FFT module which will calculate the angle of rotation. That will then be compared to the angle at which the original score was rotated by, from which a percentage error can be determined.

5.2.2 Stave Detection

The simplest way to check that the staves were all correctly identified is to paint them over the recognised score. This module will therefore be visually assessed and if the staves as painted by the OpenOMR application do not appear to completely cover the staves in the original image, then we will reject it as being correctly identified.

5.2.3 Note Heads Detected

The note heads are painted on top of the original notes as blue square boxes and this will enable us to visually assess how many note heads were correctly and incorrectly identified. We will count the total number of note heads in the original score, the number of false positives and the number of false negatives. This will then allow us to calculate the accuracy of the note head detection algorithm in terms of notes correctly recognised.

5.2.4 Pitch Calculation

For each note head found, the program outputs a letter 'A' through 'G' based on the calculated pitch of the note head. The pitch for each note head on the original score needs to be manually determined and this can then be compared with the pitch found by the program.

5.2.5 Note Duration

For each note head found, the program outputs the duration of the note as an integer having a value from '1' to '3'. A value of '1' represents a semi-quaver, a value of '2' represents a quaver and a value of '3' represents a crotchet. The output value is then compared with the actual duration of each note head in the original score.

5.2.6 Neural Network

In order to test and train the neural network, several hundred music glyphs were extracted from a selection of music scores. The glyphs in the 'testing' directory described in section 4.1.1 were used to determine the percent accuracy of the neural network and we want to know how many symbols are correctly and incorrectly classified.

Ideally, the glyphs produced from the level 2 segmentation module should be used to determine the overall accuracy of the neural network. However, the level 2 segmentation module is not producing accurate results as is and if the level 2 segmentation module is improved in a future release, this method for testing the neural network should be used.

5.3 Graphical User Interface Testing

The graphical user interface was tested to in order to determine its stability and reliability. Two common software engineering testing methods were used to test this software and they are the 'monkey testing' and 'stress testing'. Monkey testing is a technique for which a user is asked to use the application and find bugs (such as entering a letter when a digit is required). Stress testing is when a program is tested beyond its known operating capabilities.

5.3.1 Monkey Testing

The monkey testing for this application is to be performed by two students from the Department of Computing at Imperial College. They will thoroughly test the application and make a note of any cases for which the application crashed.

5.3.2 Stress Testing

We want to determine how well the program will cope if:

1. It is given a huge image (greater than 5Mb)
2. If no music is present in the image. (For example, the portrait of someone)

Chapter 6

Results

This chapter provides results from the tests that were described in chapter 5.

6.1 FFT Module

The test cases discussed in section 5.2.1 were performed on the score in figure A.1 and the results are shown below:

Original	FFT Module	% error
0°	0°	0
0.2°	0.253°	26.5
0.5°	0.506°	1.2
1°	1.0127°	1.27
2°	2.099°	4.95

Table 6.1: FFT Module Results

The total overall error is 6.78% and this error does not include the test case which saw the bottom half of the score rotated by 0.5°. That was ommited from the overall error as the FFT module does not work when only a few staves are rotated. A windowed FFT could overcome this problem and is discussed chapter 7.

6.2 Stave Detection

The stave detection module was tested on the scores in figures A.1 and A.2. Additionally, the stave detection module was also tested when the score in A.1 was rotated by 0.5°. The results are shown in figures 6.1, 6.2 and 6.6.

Although only two staves are shown in figure 6.1, all staves were correctly recognised for this score. The stave in figure 6.2 was also correctly recognised. However when the the score in figure A.1 was rotated by 0.5°, the stave detection algorithm did not work properly and that is seen in figure 6.6.

We can therefore conclude from these test cases that when the stave is slightly skewed, the stave accuracy of the stave detection algorithm deteriorates and this shows the important role that the FFT module has in order to accurately identify staves.

6.3 Note Head Detected

The note head detection module was tested with the scores in figures A.1, A.2 and A.3. Table 6.2 below shows the results obtained after running the note head detection module on those scores.

	# Note Heads	# Note Heads Found by Module	False Positives	False Negatives
Score 1	371	404	38	5
Score 2	15	0	0	15
Score 3	165	167	18	16

Table 6.2: Note Head Detection Results

The results shown in table 6.2 provide encouraging results for the note head detection module.

6.4 Pitch Calculation

The pitch calculation for the score in figures A.1, A.2 and A.3 are shown in tables 6.3, 6.4 and 6.5 respectively. The top row shows the pitch found by manually going through the score whilst the bottom row shows the pitch as calculated by the application.

C	E	G	C	G	E	D	F	G	B	G	F	C	E	G	C
C	E	G	C	G	E	D	F	A	C	G	F	C	E	G	C
G	E	C	G	B	D	G	B	D	G	B	D	B	G	F	D
G	E	C	G	B	D	G	B	D	G	B	E	B	G	F	D

Table 6.3: Actual pitch versus Calculated Pitch for figure A.1

C	D	E	F	G	A	B	C	B	A	G	F	E	D	C
C	D	E	F	G	A	B	C	B	A	G	F	E	D	C

Table 6.4: Actual pitch versus Calculated Pitch for figure A.2

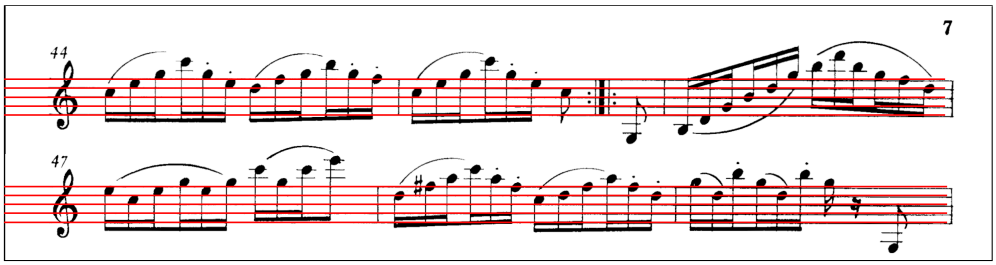


Figure 6.1: Stave Detection Results for Figure A.1

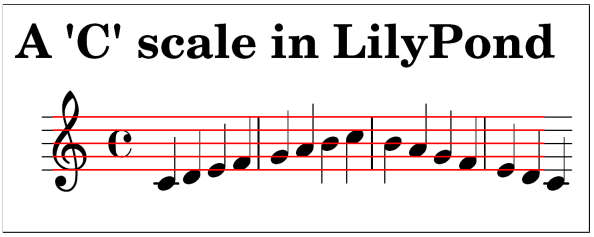


Figure 6.2: Stave Detection Results for Figure A.2

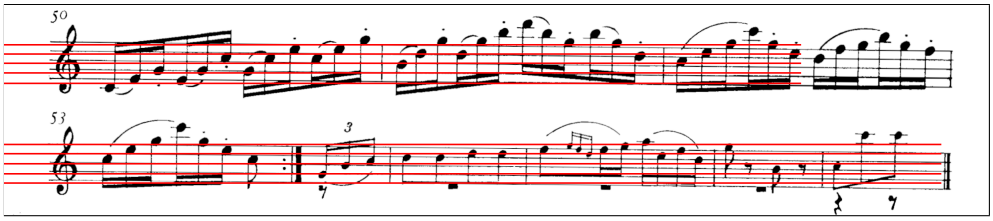


Figure 6.3: Stave Detection Results for Figure A.1 (rotated)

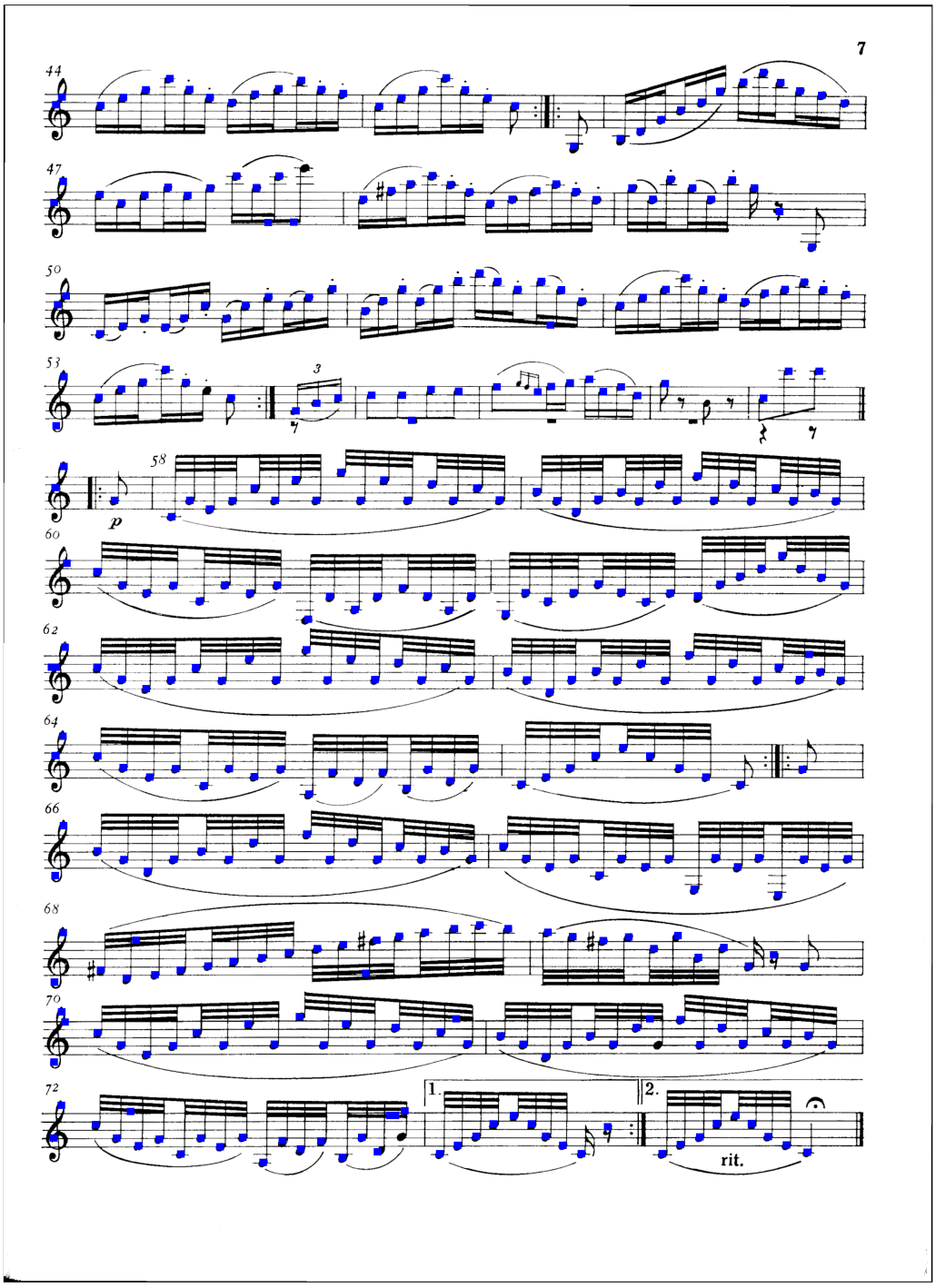


Figure 6.4: Note Head Detection Results for Figure A.1

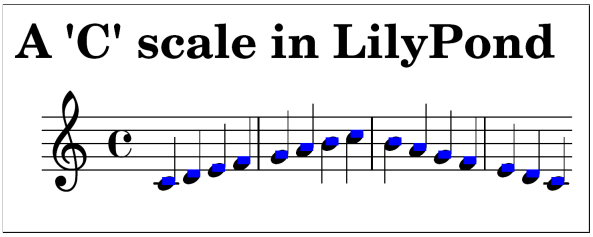


Figure 6.5: Note Head Detection Results for Figure A.2

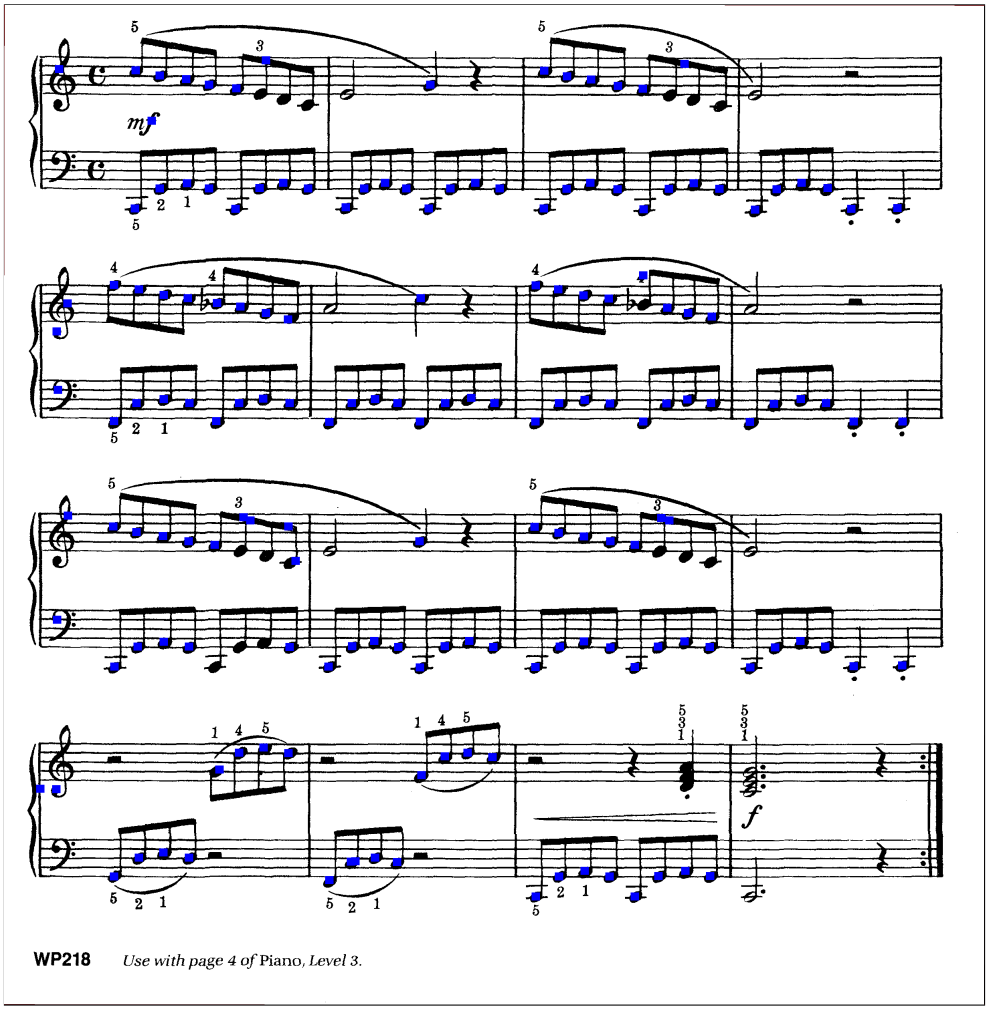


Figure 6.6: Note Head Detection Results for Figure A.3

C	B	A	G	F	E	D	C	G	C	B	A	G	F	E	D	C
C	B	A	G	F	F	_	_	G	C	B	A	G	F	_	_	_

Table 6.5: Actual pitch versus Calculated Pitch for figure A.3

1. In figure 6.3, 90.6% accuracy was obtained
2. In figure 6.4, 100% accuracy was obtained
3. In figure 6.5, 64.7% accuracy was obtained

6.5 Note Duration

The note duration for the score in figure A.2 obtained 100% accuracy but the scores in figures A.1 and A.3 did not perform well at all and this is due to the output from the level 2 segmentation module. We can therefore conclude that the level 2 segmentation module does not provide accurate results at present and should be improved in future releases.

6.6 Neural Network

The results of running the 'testing' set as described in section 5.2.6 through the neural network are shown in table 6.6. An image of a sample glyph for each category is provided in appendix B.

Glyph Name	Total Glyphs	Average Accuracy	Average Confidence
bass	5	100%	61.58%
crotchet	42	100%	84.19%
demisemiquaver_line	24	100%	99.55%
flat	10	100%	96.89%
minim	38	100%	93.89%
natural	17	100%	93.02%
quaver_br	5	100%	99.51%
quaver_line	3	100%	99.51%
quaver_tr	3	100%	91.82%
semibreve	3	100%	75.47%
semiquaver_br	1	100%	19.80%
semiquaver_line	55	100%	94.87%
semiquaver_tr	1	100%	71.13%
sharp	76	92.11%	84.95%
treble	10	100%	99.60%

Table 6.6: Neural Network Results

Out of the 15 different classes of glyphs trained by the neural network, the only class of data that did not obtain 100% accuracy was the 'Sharp'. Neural

networks are trained to remember certain patterns and in the case of the sharp its shape is similar to the natural glpyh which therefore explains why it is sometimes classyfing it as a natural.

6.7 Monkey Testing

The feedback provided by the two students was positive in the sense that the only way they managed to crash the application was to input string values in various input boxes. This is currently being fixed and each input box will be verified before accepting the input.

6.8 Stress Testing

When large images (greater than 5Mb) are used, the displaying of the score becomes extremely slow. However the system is still able to recognise the score.

Chapter 7

Future Work

As one of the major goals of this project was to turn the OMR application developed into an open source project, it was important to identify aspects of the application that could be improved. Below is a set of suggested changes or enhancements that could be made to the current implementation and each suggestion is listed as having a low, medium, or critical priority.

1. **Improved Graphical User Interface (medium)** – Although much time was spent developing the current GUI, much improvement can be brought to it. The GUI was programmed in Swing and it may be worth investigating if it would be suitable to use SWT instead. Providing a zooming feature for the different images displayed is one example of a GUI enhancement.

2. **Interactive Features (low)** – A nice feature would be to allow users to click on the recognised score in order to modify incorrectly identified notes. For example, if a note was omitted, the user could simply click on that note and the application would add it. Another interactive feature would be to illuminate the notes in a different colour as they are played.

3. **Level 2 Segmentation (high)** – The level 2 segmentation needs to be improved as this is currently affecting the overall performance of the application.

4. **Multi-Threading (medium)** – The current version of the GUI is not multi-threaded. This is only a minor issue which is unlikely to affect the execution time when recognising a score but would enhance the user's experience of the application.

5. **Detect Minims and Semibreves(critical)** – It is possible to detect filled note heads by relying on the y-projection of the music score as described in SECTION. It is however very difficult to identify semibreves and minims by using the current method used to locate note heads. A method technique to detect minims and semibreves needs to be implemented.

6. **Support for Grand Staves (medium)** – Grand stave are typically used in piano scores where one stave is used for the right hand and the one below is used for the left hand. They are usually attached with vertical

bars which is a feature that could be used to detect them. This feature could simply be implemented by having the user specify whether the score being recognised has a grand stave and if it is the case, two staves are processed at once when generating the midi file.

7. **Detect chords (critical)** – Chords are presently not implemented and in order for this application to support polyphonic scores, this feature needs to be implemented.

8. **Use Bass, Treble and Alto Clefs (critical)** – Although those clefs may be recognised and correctly classified at present, they do not affect the way in which the application generates the midi score and the default is the Treble clef.

9. **Outputting to LilyPond and MusicXML Formats (low)** – This would involve taking the internal representation of the recognised score and representing that the LilyPond or MusicXML formats. This is where perhaps a plug-in manager could be developed so that other output formats can be added at a later stage if necessary. Creating a LilyPond or MusicXML format only would a parser which knows the LilyPond and MusicXML formats.

10. **Saving Generated Midi Files (low)** – This extended feature could be provided as an extension to the openomr.midi package described in section 4.1.6 and would enable users to save generated midi files to disk.

11. **Verifying correct length of measure (medium)** – If we are able to correctly identify the start and end of each measure in the music partition, we could add a module that checks the duration of the notes found in a measure add up correctly. For example, if we are in common time and our system detected five crotchets in a measure, we would know that an error occurred somewhere. We could then perhaps re-analysing that measure by varying certain parameters.

12. **Windowed FFT (medium)** – We saw in section 6.1 that when only a portion of the image has staves rotated, the FFT module performs poorly. The FFT module could be changed in such a way that smaller portions of the image are analysed.

13. **Semi-Automatic Neural Network Trainer (low)** – This feature would enhance the training process and overall accuracy of the neural network. We could have the program save all level 2 segments to disk and then sort the files according to the glyph types. This would avoid having to cut out glyphs in a an image editor which takes a lot of time.

Chapter 8

Conclusion

Optical music recognition (OMR) is the process of converting printed music scores into a format understandable by computers. Although several commercial applications exist, we have seen that they don't always perform accurately and this was the motivation for this project. We investigated the research that has been conducted in the past in order to gain a better understanding in this field. A segmentation based approach was used for the design of this project and was based on the O^3MR project that was discussed in section 2.8.6.

The first step in a OMR application is to detect the thickness of stave lines and the spacing between the stave lines. We achieved this by using the RLE algorithm in order to produce a histogram for consecutive black and white pixel runs. The next phase involved detecting the stave lines and we saw that by taking the y-projection of the image, we were able locate the stave by looking for five equidistant peaks in the projection. When the staves were skewed, the stave detection algorithm did not perform well and a method to detect the angle by which the staves were rotated was used. This method involves taking the FFT of the image. We then proceeded to the level 0 segmentation process and found symbols that were close to one another. The note heads were then detected by means of the RLE algorithm and the y-projection. The level 1 segmentation phase was then performed and this horizontally segmented level 0 segments which contained note heads. The last segmentation phase was level 2 and this involved vertically segmenting the image to produce glyphs. The glyphs were then input into the neural network and the outputs where used to reconstruct the score.

The results showed that the note head detection module of the OpenOMR application performed extremely well but the overall accuracy of the system was degraded by the level 2 segmentation module. The level 2 segmentation is a critical component of this system and is listed as having a critical priority for any future developments of this application. We also identified some areas which need improvement in chapter 7.

Appendix A

Scores

This appendix provides several scores that were used for the testing phase of the OpenOMR application.

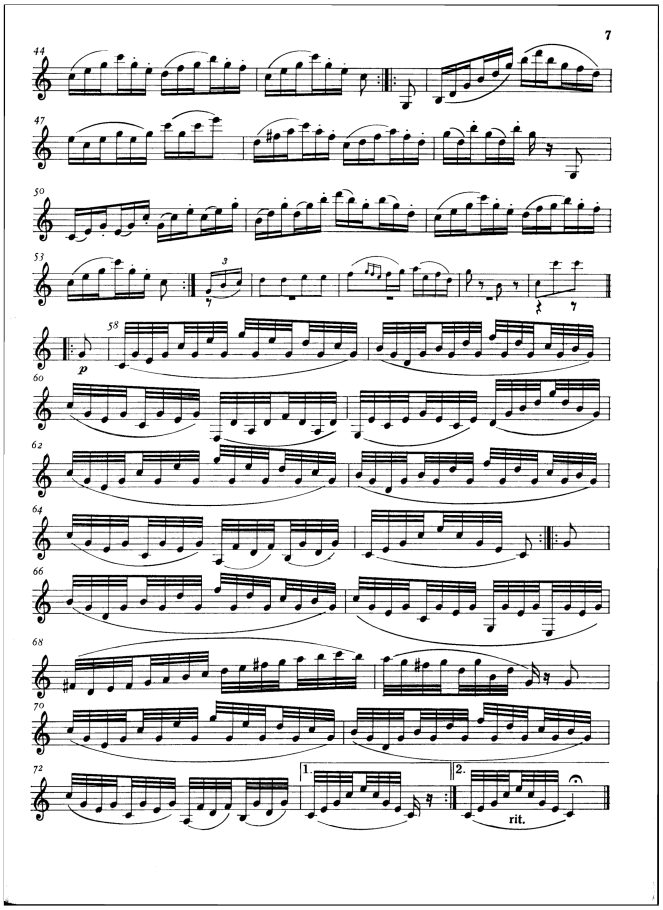


Figure A.1: Testing Score Number 1

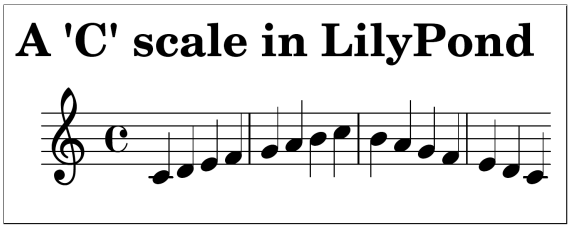


Figure A.2: Testing Score Number 2

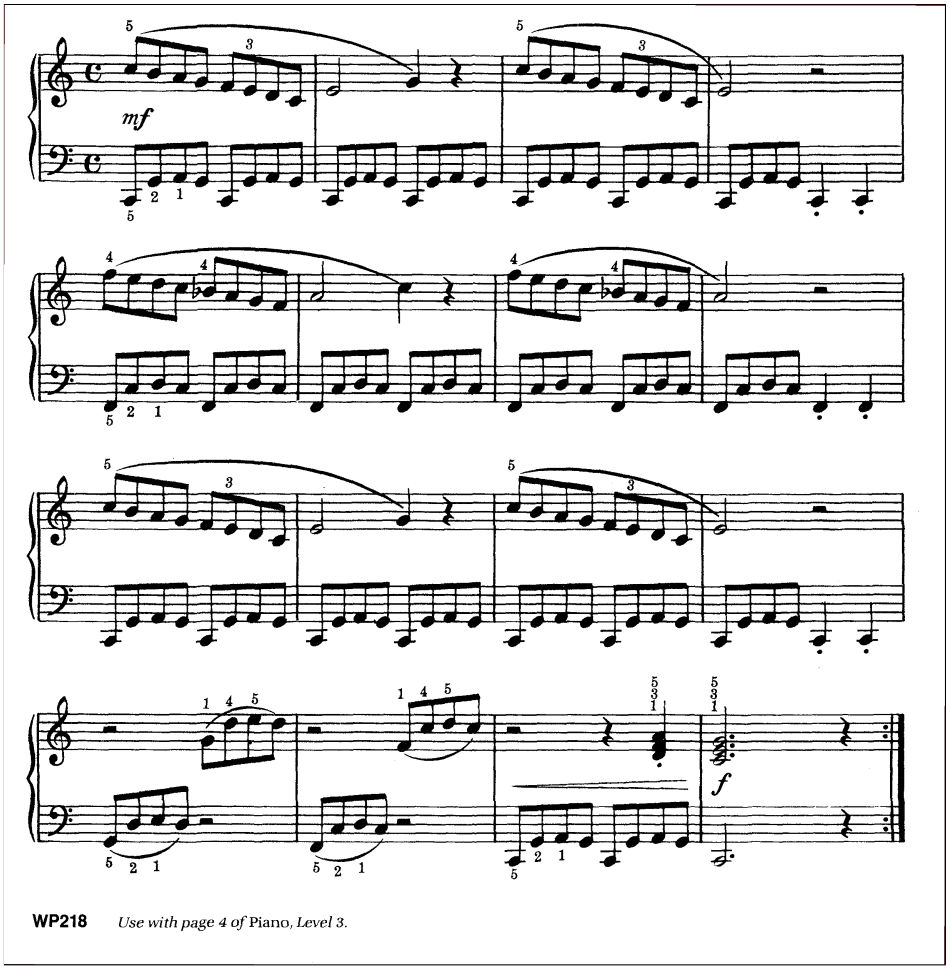


Figure A.3: Testing Score Number 3

Appendix B

Musical Glyphs Used

This appendix provides all the glyphs that were used to train the neural network.

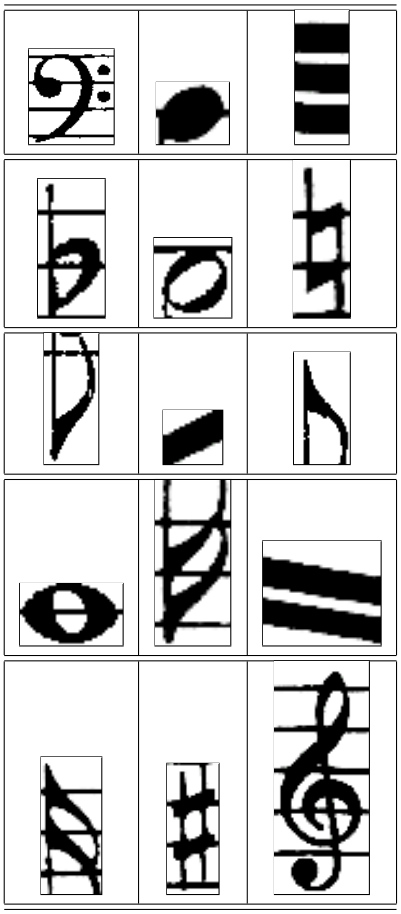


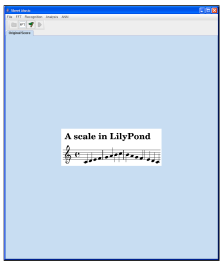
Table B.1: FFT Module Results

Appendix C

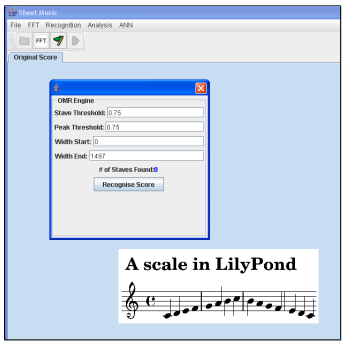
OpemOMR User Guide

This appendix provides a user guide along with screen shots for the OpenOMR application.

1. **Opening an image:** The first step to recognise a scanned music score. You can open a music score by either clicking on the 'open folder' in the toolbar or by going to the "File – Open" menu. You will then be prompted to select a file. Upon selecting a file, the image of the score opened will be displayed in the main window. You should now have a screen as shown below:

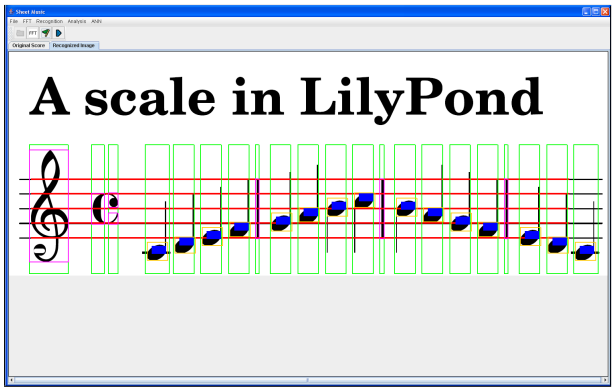


2. **Recognising a score:** The next step is to recognise the score and this can be done by clicking on the "green flag" button in the toolbar or going to the "Recognition – Recognise" menu. You will then be prompted with a box as shown below:



You should first try recognising the score with the default parameters. If once the recognition is over and you find that the wrong number of staves were found, you may adjust the parameters in the input boxes and try the recognition process.

To view the recognised score, go to the "Recognise – View Recognised Score" menu. The recognised score will then be displayed as shown below:

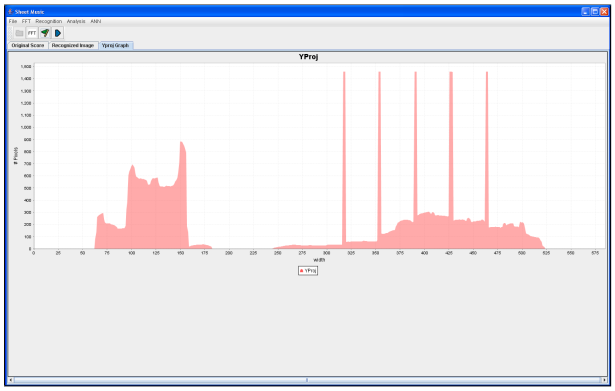


3. **Skewed staves:** If you are under the impression that the staves are skewed, you can de-skewing it by pressing the "FFT" icon on the toolbar or by going to the "FFT – FFT" menu. A window will be displayed as shown below:

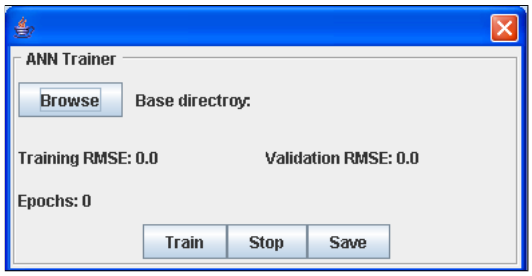
The window will prompt you for the window size of the FFT and the input must be a power of 2. That is, is can be 512, 1024, 2048, etc. Now click on the "Do FFT" button. The application will try to determine the angle by which the staves are rotated. When the window disappears, you can then proceed to the recognition process of the score.

4. **Playing the score:** You can play the recognised score by pressing the blue triangle in the toolbar menu.

5. **Viewing graphs:** Various graphs of x and y-projections can be viewed through the "Analysis" menu. When selecting a graph to view, a new tab in the main window will be created as shown below:



6. **Training and testing the neural network:** To train the neural network, go to the "ANN – Trainer" menu. A new window will be displayed as shown below:



You must now choose the directory which contains the "training" and "validation" sub-directories in which contain the sample glyphs. Now click on the "Train" button. The neural network will now train and the training will stop when the validation RMSE starts to increase. If after a large amount of epochs (i.e 3,000) the neural network has not stopped training, you can manually stop the training by pressing the "Stop" button. You will now want to save the state of the trained neural network and this can be done by clicking the "Save" button.

Bibliography

[1] Bainbridge, D. (1996). Optical Music Recognition: A Generalised Approach *Department of Computer Science, University of Canterbury.* Christchurch, New Zealand

[2] Bainbridge, D. (1997). An extensible optical music recognition system *Nineteenth Australasian Computer Science Conference.* Melbourne, Australia

[3] Bainbridge, D. (2005). Fast Capture of Sheet Music for an Agile Digital Music Library *Department of Computer Science, University of Canterbury.* Christchurch, New Zealand

[4] Bellini, P. (2001). Optical Music Sheet Segmentation *Proceedings of the First International Conference on WEB Delivering Music*

[5] Bellini, P; Bruno I; Nessi, P. Assessing Optical Music Recognition Tools

[6] Bolstein, D. A Critical Survey of Music and Image Analysis *Department of Computing and Information Science, Queen's University.* Ontario, Canada

[7] George, S. (2005). Visual Perception of Music Notation: On-Line and Off-Line Recognition *IRM Press. ISBN: 1-931777-94-2*

[8] Ng, Kia; Barthelemy, J; Ong B; Bruno I; Nesi P. CIMS: Coding Images of Music Sheets *The Interactive-Music Network, 2005*

[9] Roth, M. (1994). An Approach to Recognition of Printed Music *Swiss Federal Institute of Technology.* Zurich, Switzerland

[10] Sau Dan, L. Automatic Optical Music Recognition *Department of Computer Science, University of Hong Kong*

[11] Colton, Simon: Introduction to Artificial Intelligence *http://www.doc.ic.ac.uk/∼sgc/teaching/v231/index.html Department of Computing, Imperial College* London, UK

[12] Johnston, Alex: Classifying Persian Characters with Artificial Neural Networks and Inverted Complex Zernike Moments *Department of Computing, Imperial College* London, UK

[13] Joone - Java Object Oriented Neural Engine http://www.jooneworld.com *Joone reference manual*

[14] http://icking-music-archive.org/software/indexmt6.html *MusiXTEXreference manual*

[15] http://www.lilypond.org

[16] http://www.recordare.com/xml/faq.html

[17] http://www.wikipedia.org/wiki/NIFF

[18] http://en.wikipedia.org/wiki/Musical_notation

[19] http://www.cs.unm.edu/∼brayer/vision/fourier.html *Introduction to the Fourier Transform*

[20] Kerry Peake: Optical Music Recognition using Kd-tree Decomposition *University of Birmingham* UK

[21] Yong Li: Music Sheet Reader - An Implementation of Optical Music Recognition System *University of Birmingham* UK

[22] Peltarion: Tutorial II - Good Cop/Bad Cop http://www.peltarion.com/WebDoc/Tutorials/tutorial2.html