Jack Lawrence-Jones

Optical Music Recognition

Part II Computer Science Dissertation

Downing College

May 18, 2012

Proforma

Name:	**Jack Lawrence-Jones**
College:	**Downing College**
Project Title:	**Optical Music Recognition**
Examination:	**Computer Science Tripos Part II, July 2012**
Word Count:	**11068**
Project Originator:	**Jack Lawrence-Jones**
Supervisor:	**Dr. Chris Town**

Original Aims of the Project

To create a functional Optical Music Recognition system for use by musicians. That is, a program which, when given an image of a music score in Common Music Notation, of an appropriate level of complexity, can extract the notational information depicted and export it in a suitable format. This should be achieved within a reasonable time frame to a reasonable level of accuracy.

Work Completed

I implemented a functioning Optical Music Recognition system. Given an image of a printed score, my system can produce three outputs; an audio file containing a performance of the piece, a computer typeset score and a representation of the score in a music score description format. In order to achieve these tasks, I implemented submodules performing image preprocessing, segmentation, symbol classification, music notation reconstruction and final notation construction. I also implemented a lightweight Java 2D image processing library to support my application. As extensions I increased the permitted complexity of input scores and evaluated my system on handwritten scores.

Special Difficulties

Optical Music Recognition is a relatively small field, with few (one) reference books, therefore most research was from papers. Much research has been done in Portugal and China, and the translated papers were often less than lucid. Being a relatively young field, there are few standard techniques available for Optical Music Recognition so certain areas of the implementation involved elements of trial and error. The lack of a suitable 2D image processing library in Java meant I had to implement my own which added significantly to the implementation effort required before I reached my success criteria.

Declaration

I, Jack Lawrence-Jones of Downing College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed .....

Date .....

Contents

List of Figures

Acknowledgements

I would like to thank Chris Town for his expertise, encouragement and unwavering joviality.

Chapter 1

Introduction

1.1 Definition of OMR and uses

Optical Music Recognition is the automatic extraction of information from music manuscripts. Images of music scores, typically captured using a scanner, are examined and an ideally equivalent model of the music presented is constructed in a format amenable to subsequent computer based manipulation.

Optical Music Recognition (hereon known as OMR) is one type of Document Analysis and Recognition (DAR) - the general set of problems relating to extracting information presented on paper. DAR is itself one small subfield of Computer Vision, specifically restricted to documents (structured information on paper). It can be framed as a transformation from an image of a document to a symbolic representation that can then be processed by computers.

Uses of implementations of this transform include:

- *Storage*: The important information in a document is its semantics (meaning), rather than a total description of the image. For example, when storing a book in an online library, the text itself is important, not the specks of dust present on each page when the book was scanned or its typographic quirks. Storing each page's text rather than an image of the page is much more spatially efficient - a form of lossy compression.

- *Editing*: A computer based representation of a document can be much more easily changed and revised than a physical one. Automatic tools such as spell checkers require computer based representations.

- *Analysis*: Metadata such as likely composer or author can be automatically derived using computers' ability to examine large amounts of data. Automatic fraud and plagiarism detection systems analyse documents.

- *Indexing*: The collecting and parsing of data to facilitate fast and reliable information retrieval. Once documents have been converted to a computer based format, they can be searched.

Of the typical DAR uses, most relate to the digital cataloguing of music scores. Many musical works written in the past are still available only as original manuscripts or as photocopies. Their transformation into computer-based formats ensures their preservation and makes their dissemination and management more efficient. Organisations that curate online libraries of music such as IMSLP [6], Répertoire International des Sources Musicales [20] and the Choral Public Domain Library [13] use OMR for this very purpose. Scores can be found, searched, pitch transformed and downloaded, making such libraries useful to musicians and academics alike. They also reduce the cost of music publication and exposure, meaning works can be featured that are not otherwise commercially viable [12].

Interesting applications enabled by to OMR include:

1. The ability to automatically transpose scores (certain instruments read music in non standard keys, e.g. clarinets are in Bb, therefore their music must be transposed down a tone to be heard at correct concert pitch - much more easily achieved using a computer).

2. Part extraction from full scores. A composer will produce a full score containing all instruments simultaneously - each individual instrument's part must be extracted to be given to the relevant musicians. Or visa versa.

3. Automatic page turning - the musician reads the score off a device like an iPad, and the pages are automatically advanced when the end of the page has been reached [22])

4. Converting the score into alternative music formats and notations such as Braille[25], for different software packages or re-engraving (music typesetting) an old score.

5. Playing back of music through loudspeakers. An invaluable tool for composers, this allows them to hear their composition in its nascent stages, before handing the score over to a conductor and his orchestra would be remotely appropriate.

1.2 Music notation

As previously mentioned, OMR systems take as input images of music scores. Music scores come in a myriad of different formats and styles depending on the genre of music, the instruments written for and the cultural heritage of the composer. The main syntactic style used for the last 400 years in the West is known as "Modern Common-Practice Western Music Notation", which I will abbreviate to CMN (Common Music Notation) as is done in the literature (it is also sometimes known as WMN - Western Music Notation). I will discuss the syntax and semantics of CMN in the Preparation section, as considering it when designing my system will be of utmost importance.

1.3 Aims

This project's aim is to implement an Optical Music Recognition system. Given an image of a printed music score in CMN, of appropriate complexity, it will automatically create a computer based representation (in the form of an open source music score description format), a graphical representation of the score or a music file containing a performance of the music depicted.

1.4 History and State of the art

OMR is a relatively young field and as such there exists few universally adopted techniques - the literature is diverse and conflicting.

The field of OMR began with two MIT doctoral dissertations by Pruslin in 1966 and Perau in 1970. Early work was limited by technology, and the acquisition of image data itself was a significant endeavour. Both systems could also only address music of limited complexity, using a small subset of Common Music Notation.

The next significant development occurred in the early 80's when a team at Waseda University, Japan, constructed a robotic organist that could 'sight read' (play in real time previously unseen) sheet music placed in front of its electronic eye. It played the organ using its mechanical hands and feet, and could even accompany someone singing a tune it already knew, keeping in time with the singer if they varied the tempo (speed) and automatically transpose the key if the singer went out of tune.

The OMR component of WABOT-2 was the first such system capable of recognising a comprehensive set of CMN, including chords[25].

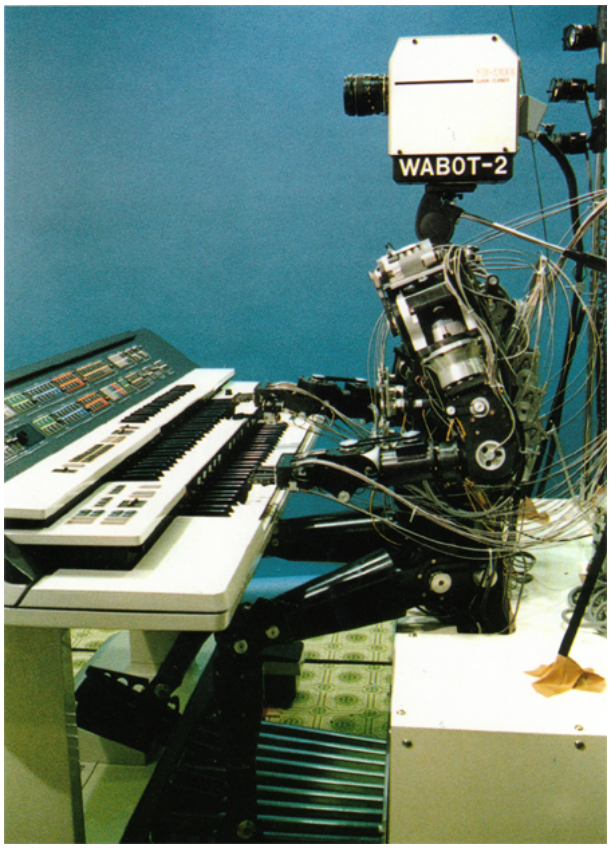


Figure 1.1: WABOT-2, the robotic organist. Copyright by Humanoid Robotics Institute, Waseda University

Inexpensive optical scanners became widely accessible only in the late 80's, facilitating growth in the field [24, 21]. Since then over a hundred[23] papers have been published. The first commercial solutions appeared in the early 1990s, the most recent (2011) and supposedly most advanced[19] being *Neuratron Photoscore 7*.

Despite advances in the field and growth encouraged by commercial success, there are still unsolved problems, and no existing solutions are close to having satisfactory performance in terms of precision and robustness[23], especially with handwritten scores. Typical accuracy rates lie at around 90% [23] for printed music and much lower for handwritten (difficult to quantify due to the massive variation in inputs). The problem of how to evaluate OMR systems will be addressed later.

Chapter 2

Preparation

2.1 General OMR system structure

In the OMR literature a 4 part modular structure is almost universally used. Control flow is linear - each module acts as one stage in a feed forward pipeline. The structure is as follows:

1. Preprocessing
2. Symbol recognition
3. Notation reconstruction
4. Final representation construction

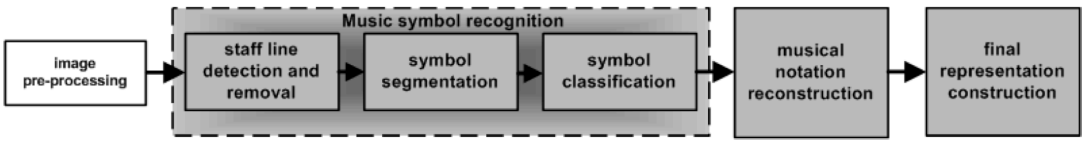


Figure 2.1: General OMR framework [41]

Preprocessing

The input for this module is the raw image of the page of music obtained from the optical scanner. Typically the image is deskewed, noise is removed and the image is thresholded to obtain a binary image. These procedures produce an image more amenable to subsequent analysis.

Symbol Recognition

This module contains the majority of the OMR system. We take the binary image produced by the preprocessing step and extract all the musical features it depicts.

In principal OMR is an extension of Optical Character Recognition, the process of recognising text in images. The same common symbol recognition problems like symbol degradation due to noise and skew apply. However for music notation, segmentation presents a significantly more difficult challenge, for a number of reasons:

- The notation is *two dimensional*, as opposed to one dimensional for printed text (ignoring the fact that the text must be split up into lines to fit on a page). The same symbol in the same horizontal position can mean different things depending on its position on the vertical axis.

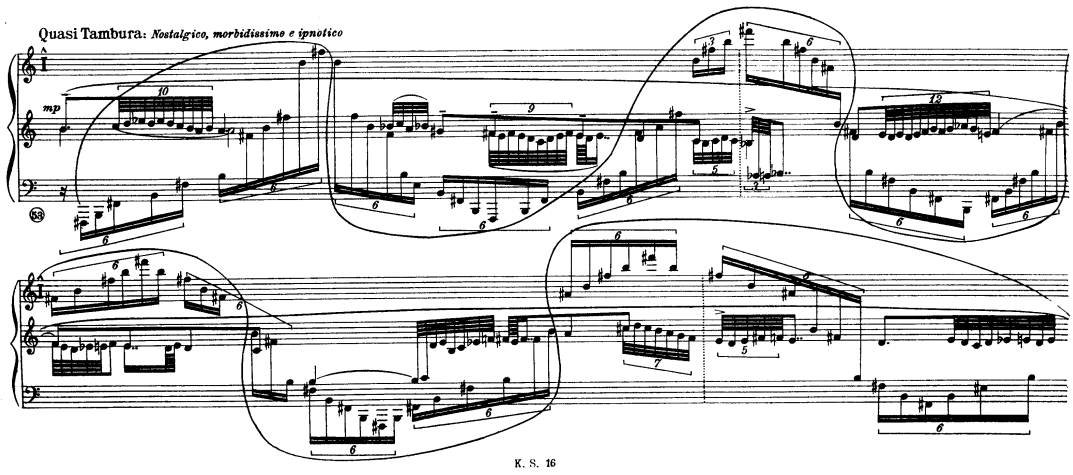
- In Latin languages there is a well defined set of common symbols coding for their characters. In music notation there is an *almost unlimited symbol set* as composite symbols can be constructed by combining primitives.

- The vertical position of symbols, which denotes their pitch, is performed in reference to the stave lines, a set of 5 horizontal, parallel reference lines on which the music symbols are placed. However this makes segmentation much more difficult as it connects all the symbols - how do we know where stave line ends and music symbol begins?

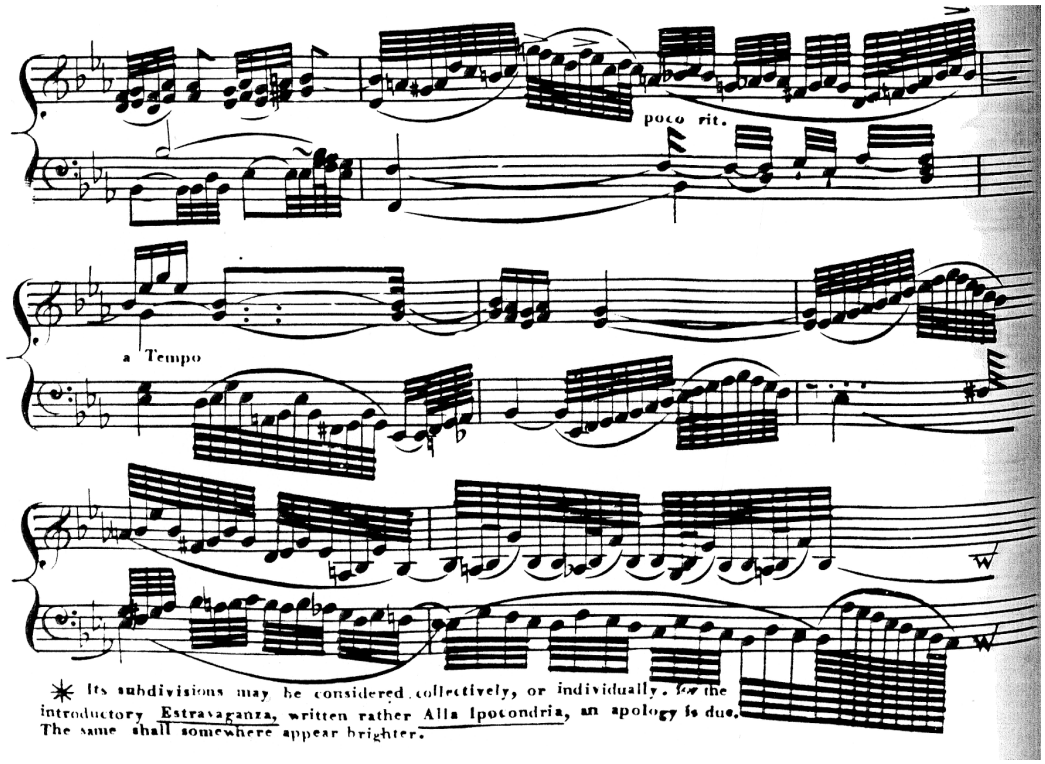
- Due to the freedom the syntactic rules allow (which they must, as the notation must be able to represent as closely as possible the full palette of sounds music allows), the notation can become extremely complex and dense. Music engravers (engraving is typesetting for music scores) employ techniques to ensure the score is as clear and easy to read as possible, which itself introduces huge amounts of variation even between identical scores produced by two different publishers. Music engraving literature stresses the importance of a clear layout with no touching symbols, however due to sloppy craftsmanship or particularly complex music, touching symbols are often encountered - this is problematic for segmentation.

Figure 2.2 illustrates some of these issues. We will explore the properties of music notation in the subsequent section so we can take these difficulties into account.

Therefore the main steps to accomplish in this module are:



(a) An extraordinary slur. From Sorabji's *Opus Clavicembalisticum* (1930), *IX Interludium B* (Curwen ed., p. 175-176)



(b) Two demisemihemidemisemihemidemisemiquavers. From Anthony Phillip Heinrich's *Toccata Grande Cromatica* from *The Sylviad*, Set 2, m. 16 (ca. 1825)

Figure 2.2: Examples of complex music notation from Donald Byrd's *Gallery of Interesting Music Notation*[16]

1. Stave line removal
2. Symbol segmentation
3. Symbol classification

Notation Reconstruction

Using the classification and position information calculated by the previous module, here an internal logical representation of the music information is constructed by applying music notation rules. Again we must explore the syntax of CMN before we can design this module.

Final representation reconstruction

The internal logical representation is transformed into the desired output format. This may be a music score description format such as MusicXML, a sound file format such as MIDI or a graphical representation such as a PDF.

Now we have a better understanding of the OMR modular structure, we can construct a more detailed framework, illustrated in figure 2.3.

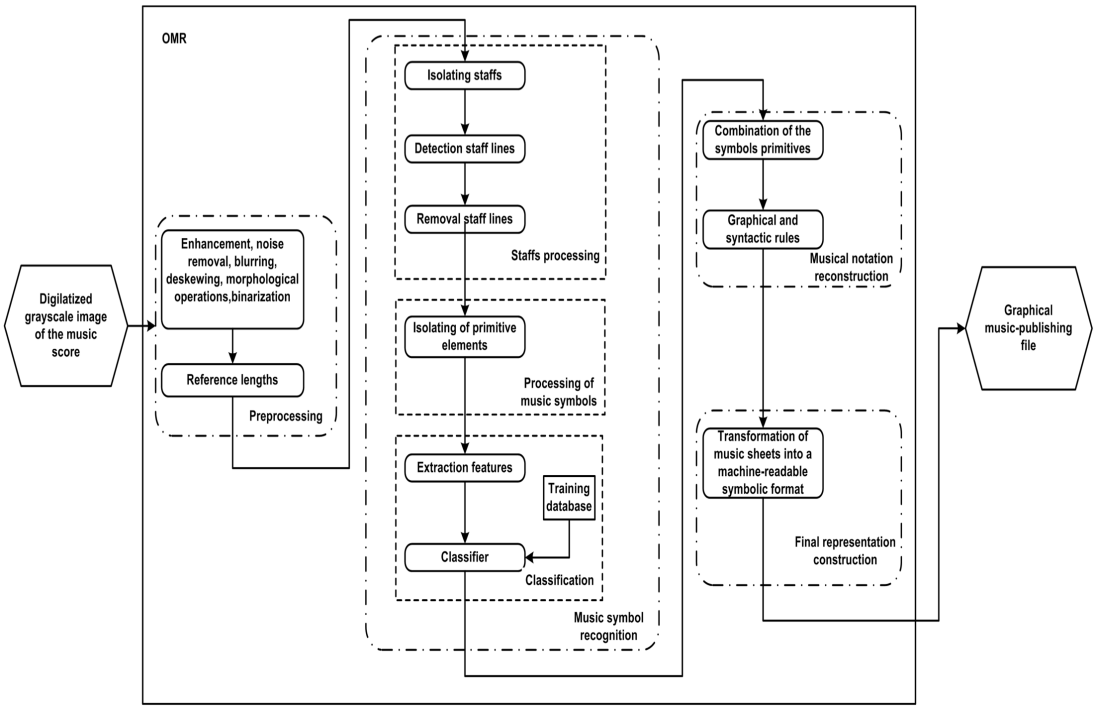


Figure 2.3: More detailed OMR framework [24]

2.2 Common Music Notation Syntax/Semantics

It is of utmost importance when designing an OMR system or indeed any Computer Vision application to consider the form the input data will take. Information by definition has structure, some inherent properties that can be formalised. By using previous knowledge about the inputs we greatly reduce the problem space and can design more specific algorithms to extract information from them. See appendix A for more.

2.3 Computer Vision Techniques

In this section we visit some common Computer Vision algorithms which will prove useful when discussing the implementation of my system.

2.3.1 Binarization (thresholding)

Most OMR systems process binary images. This is because conceptually scores only have two classes of pixels - foreground and background. However, scanned scores are often in the greyscale colour space. In order to convert these greyscale image to binary, we iterate over each pixel in the image, setting it to black if the greyscale value is below a predetermined threshold, white if it's above. For example, an 8 bit greyscale image has 256 possible levels, ranging from 0 (black) to 255 (white). If we choose a threshold of 230, all pixels of value less than 230 will be set to black and all higher will be set to white.

2.3.2 Projections

Projections are useful for calculating object boundaries (bounding boxes) and examining changes in object features. They can be calculated using the generalised projection transform, the Radon transform. This integral operator allows for continuous projections at any angle. In the discrete case, for projections of image I(x,y) of size $m * n$ at $\theta = \frac{\pi}{2}$ and 0 (projections onto the x and y axes, respectively), we get:

$$X(x) = \sum_{y=0}^{n-1} P(x, y) \ , \qquad 0 \leq x < m$$

and

$$Y(y) = \sum_{x=0}^{m-1} P(x, y) \ , \qquad 0 \leq y < n$$

For a binary image, performing a projection onto the x axis gives a vector whose ith component is the sum of the black pixels in column i of the image. The Y projection correspondingly results in a vector whose jth component is the number of black pixels appearing in row j of the image.

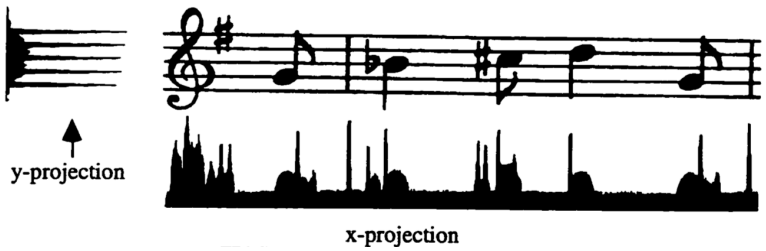


Figure 2.4: X and Y projections of a section of a stave. From [37]

2.3.3 Run-length encoding

Run-length encoding is a simple data compression technique where a sequence of consecutive identical values is represented by the value and the length of the sequence. For example, the sequence

$$\{7, 7, 7, 3, 3, 3, 3, 9, 9, 3, 5, 5\}$$

can be coded as the following list of *(value, run length)* pairs:

$$\{(7, 3), (3, 4), (9, 2), (3, 1), (5, 2)\}$$

For a binary image, where each pixel can have one of two values (0 or 1), the run-length encoding can be made even more compact. As there are only two possible values, and consecutive runs must by definition have different values, alternating runs will have alternating values. If we arbitrarily assume that the first run of the encoding always consists of 1's (if the sequence starts with a 0, the length 0 is used), we can now uniquely encode all possible binary sequences. For example, the sequence

$$\{1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1\}$$

can be compressed to

$$\{6, 3, 10, 1, 1, 4, 2\}$$

By encoding each column or row of a binary image we can greatly reduce the memory required to store it, depending on the image contents (typically to around one tenth of the original image size[37]) . Furthermore, performing image processing operations directly on this representation can often lead to more efficient algorithms[37].

2.3.4 Connected component labeling

In graph theory, a connected component in an undirected graph is a subgraph whose vertices can all be reached from any other vertex in the subgraph, and which contains no other additional vertices. Vertex B can be reached from, or is connected to, vertex A iff there is a sequence of one or more edges linking B to A:

$$A = v_0, \ldots, v_n = B \ . \ \ v_i \in V$$

where V is the set of all vertices in the graph and

$$\text{exists}(\text{edge}(v_{i-1}, v_i)) \text{ for } 0 < i \leq n$$

If we treat a binary image as a graph, with pixels as nodes and adjacent pixels having edges between them, we can define a connected component for the purposes of image processing. For a binary image $I(x, y)$, where each pixel can take a value v of 1 or 0, two pixels x', y' and x'', y'' are connected with respect to value v iff there is a sequence of pixels

$$(x', y') = (x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n) = (x'', y'')$$

in which

$$I(x_i, y_i) = v \text{ for } i = 0, \ldots, n$$

and

$$(x_i, y_i) \text{ neighbours } (x_{i-1}, y_{i-1}) \text{ for } i = 1, \ldots, n$$

Neighbouring pixels are adjacent in the image. Two common types of neighbourhoods or connectedness are 4 and 8 connectedness - only 8 connectedness includes the diagonally adjacent pixels:

	2	
1	*	3
	4	

(a) 4-connected

2	3	4
1	*	5
8	7	6

(b) 8-connected

Figure 2.5: Different types of connected-ness

The sequence of pixels $(x_0, y_0), \ldots, (x_n, y_n)$ forms a connected path from (x', y') to (x'', y''). A connected component of value v is a set of pixels P, each having value v, such that every possible pair of pixels in the set are connected with respect to v.

A connected components labelling of a binary image I is a labelled image L in which the value of each pixel is the label of its connected component[44]. The labels uniquely identify each connected component - usually positive integers are used.

There are a number of different algorithms which implement the connected component labelling procedure with varying efficiencies, including recursive, iterative and parallelised methods. I implemented an iterative algorithm that uses a union-find data structure to keep track of components as it is efficient for large images.

2.4 Classifier Theory

A classification problem is that of determining which category a new data point lies in, based on previously observed training data consisting of a set of points of known category memberships. In Computer Vision, a classifier usually refers to an algorithm used to determine what kind of object the image contains, based on a set of features calculated from the object's properties. Common applications include character recognition in OCR systems - e.g. inferring that an image is in fact of the letter 'd'.

2.4.1 Basic Classifier model

We start with a set of n known classes of objects. Each class is defined by either a description of the class or by having a set of examples of objects from each class.

An ideal class is a set of objects all sharing common properties. These objects are denoted as members of this class by assigning them class labels. Classification is the process of, given a new object, deciding which label to assign to it based on its properties. A classifier therefore takes a representations of an object as input, and outputs a class label.

The question of which properties to use to classify images remains. These properties are called features, and we use feature extraction to calculate them from the image. Ideal features are those relevant to classification in that they most distinguish objects of different classes. However calculating features can be computationally expensive. The simplest feature set is that of all pixel intensities - inputting a whole, untransformed description of the image to the classifier, one feature per pixel. Other commonly used features include:

- Area

- Perimeter
- Bounding box size/dimensions
- Location of its centre (centroid)
- Circularity
- Spatial moments
- Number of holes
- Number of strokes

2.4.2 Support Vector Machines

Support Vector Machines (or SVMs) were first proposed by Vladimir Vapnik in their early form in 1963. The current standard implementation was devised in 1995 [32]. SVMs are a form of non-probabilistic binary linear classifier. They work by constructing a hyperplane or set of hyperplanes in a high dimensional feature space in order to separate and therefore classify the data points. A good classification model is achieved by finding a hyperplane that is furthest away from the nearest data points from each class, minimising the classification error.

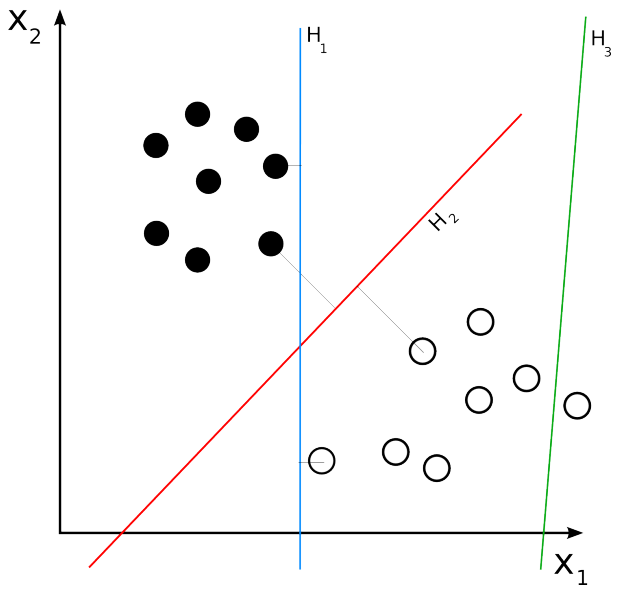


Figure 2.6: H_1 - H_3 are possible hyperplanes dividing this two dimensional feature space. H_3 doesn't separate the two classes, H_1 does but with a small margin and H_2 does so with a maximum margin. From [3]

The locating of this maximum-margin hyperplane can be seen as an optimisation problem. Given a training set of l instance-label pairs, each with n features

$$(\mathbf{x}_i, y_i).\ i = 1, \ldots, l.\ \mathbf{x}_i \in R^n.\ y_i \in \{1, -1\}$$

the SVM calculates the solution of the following optimisation problem (soft margin, [32]):

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0 \end{aligned}$$

Where $\mathbf{w}$ represents the normal vector to the hyperplane, b the offset of the hyperplane and $\boldsymbol{\xi}$ the slack variables for soft margin classification. The training vectors $\mathbf{x}_i$ are mapped to a higher (possibly infinite) dimensional space by the function ϕ in order to find a linear separating hyperplane. $C > 0$ is a penalty parameter of the error term.

$K(\mathbf{x}_i, \mathbf{x}_j) \equiv \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$ is known as the *kernel function*. Possibilities include linear, polynomial, sigmoid and radial basis function (RBF). RBF is the most generally applicable kernel as it has many advantages over the others such as handling non-linear data and fewer hyperparameters [30]. The RBF kernel is as follows, with kernel parameter γ:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2),\ \gamma > 0$$

SVMs are binary classifiers - they separate testing data into two classes. In order to classify data into 3 or more classes, necessary for the myriad of symbols in music notation, multiclass SVMs must be used. This is achieved by "breaking the single multiclass problem into multiple binary classification problems" [3]. A competitive method is the "one-against-one" approach. If k is the number of classes ($k > 2$), then $k(k-1)/2$ binary classifiers are constructed and each one is trained using data from only two classes. Every sub-classifier is queried during classification; a voting strategy is used to decide the final class.

Artificial Neural Networks

Artificial neural networks are computational structures inspired by the structure of animal brains. They consist of a network of interconnected processing elements. Another machine learning technique, artificial neural networks (or ANNs) must

be trained in order to provide useful outputs. The basic unit of an ANN is the artificial neuron, illustrated in figure 2.7.

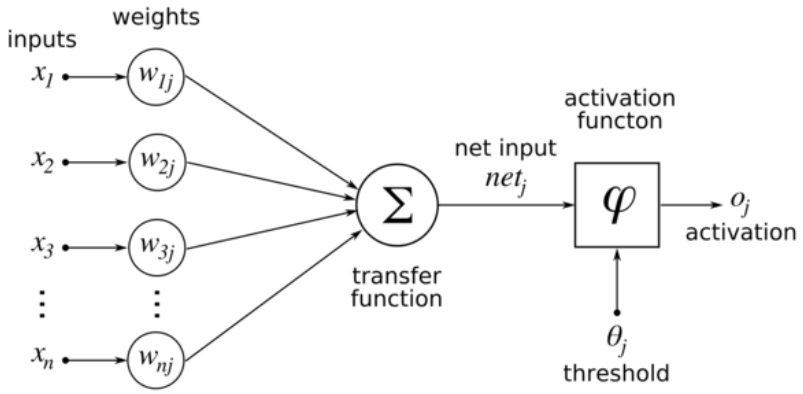


Figure 2.7: An artificial neuron. From [2]

The inputs are multiplied by their respective weights and are summed by the transfer or combination function $c_j = \sum_{i=1}^{n} w_{ij}x_i$. The activation function decides the output o_j of the neuron; whether, for those inputs, the neuron will fire based on the threshold parameter θ.

Neurons are combined to make networks of various topologies. A common type is the *feedforward network*, where each neuron in the network is located at a particular level, receives input from all neurons in the previous level and feeds its output into all neurons in the next level. There is no feedback - outputs are never transmitted to previous layers. Figure 2.8 illustrate this.

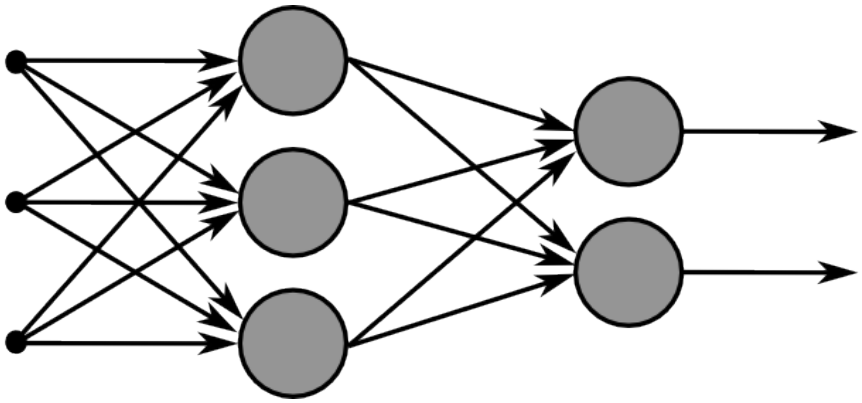


Figure 2.8: A multilayer feedforward artificial neural network. From [2]

ANNs are trained by adjusting individual neuron input weights. A variety of different algorithms exist for both supervised and unsupervised learning, includ-

ing *Back Propagation*, *Resilient Propagation*, *LevenBerg Marquadt* and *Simluated Annealing Training*.

2.5 Proposal refinement

Having researched more thoroughly techniques and problems in the field of OMR, I decided to refine my proposal to ensure I had a clear and realistic specification. I originally intended to use existing library functions from OCRopus, OpenOMR and Audiveris to allow me to ignore the low level OMR functionality and focus on the classification and notation reconstruction modules. However this proved very difficult in practice - the internal complexities of each system made integrating their functionality hard. Also music notation reconstruction is a particularly hard problem - few detailed results are available in the literature and most involve the use of complex two dimensional graph grammars, the implementation of which falls outside the scope of this project.

Therefore I decided to refocus on building a complete OMR system. Due to the huge variation in possible inputs, it is important to carefully restrict the forms the inputs can take in the specification. Due to the complexity of the system, I proposed two iterations of development; the first addressing basic OMR functionality and the second attempting to analyse scores of higher complexities.

2.6 Software Engineering

2.6.1 Specification & Constraints on the problem

I planned two main development iterations - these I will call BasicOMR and ComplexOMR.

BasicOMR implements basic OMR functionality. Its inputs are restricted to single lines of monophonic (single part, single stave, one note at a time) music, with no composite symbols. The input images may not display high noise levels or be skewed by more than 5°. The symbol set was carefully designed to allow the recognition of a large range of scores whilst maintaining simplicity and accuracy. The system should be able achieve results in the same order of magnitude as those in the literature. We will discuss how to quantitatively evaluate OMR systems below.

The second major iteration, called ComplexOMR, will hopefully successfully analyse inputs that match closer real world OMR system use cases. I will still restrict the music to a single part, however the system will take an entire page

of music, and will hopefully analyse composite symbols. This involves a massive increase in complexity, and therefore I don't expect to complete this iteration within the timeframe allowed by a Part II project - I foresee this iteration being more of an extension, with BasicOMR as the core of the project.

2.6.2 Requirements Analysis

Both iterations of my OMR system should be able to extract the notational information from the input scores and produce three types of output - a MIDI sound file containing an automatically generated performance of the music, a computer typeset document (most likely a PDF file using the excellent open source typesetting program LilyPond [7]) and a logical description of the score in a music score description format. These three formats are useful in real world OMR applications, and facilitate different forms of evaluation.

The system must take as input an image file in the PNG (Portable Network Graphics) format, chosen for its compatibility with common I/O routines and wide adoption amongst the online music document community. The standard resolution adopted for OMR input images is 300dpi [42].

2.6.3 Spiral methodology

The complexity of OMR systems means their development isn't suited to one top-down iteration (the waterfall model). Instead I used a spiral development methodology, with a fixed number of major development iterations. Due to the loosely coupled modularity of the design, the system is easily expanded and refactored.

2.6.4 Programming Languages

I decided to code the main bulk of my system in Java for its platform independence, the availability of powerful developing tools and its familiarity, having studied it for the last two years at Cambridge.

I plan to use MATLAB to process large amounts of image data when constructing training and testing sets. MATLAB is not only fast but has excellent 2D image processing functionality. I didn't know any MATLAB before starting this project, therefore in preparation I worked through some basic tutorials and exercises.

Part way through my project I discovered GAMERA - a little known document analysis application framework. It features plugins, one of which has

artificial degradation algorithms specially aimed at OMR. To get these running I had to learn a small amount of Python which I'd never used before.

I used LaTeXto typeset dissertation - having never used it before I looked at some online tutorials and learned a lot along the way.

The typesetting and MIDI file generation are performed by LilyPond, an open source music engraving package. LilyPond functions similarly to LaTeX- plain text files are compiled to produce the outputs. I had to learn the LilyPond score description language in order to implement the final notation construction module. It was also useful for generating ideal test input scores.

I also used Dot - a diagram specification language, to draw up a few of the figures for this dissertation.

2.6.5 Evaluating OMR Systems

OMR systems are notoriously hard to evaluate. Droettboom and Fujinaga [34] point out that "a true evaluation of an OMR system requires a high-level analysis, the automation of which is a largely unsolved problem". In the evaluation section we will explore different OMR evaluation methodologies.

Chapter 3

Implementation

3.1 Introduction

3.1.1 Module Overview

As with any large software engineering project, a clear structure with well-defined interfaces between modules will help enormously during implementation. I chose to adhere to the common basic modular structure appearing in the literature. So my project consists of the following Java packages:

1. **Preprocessing** omr.preprocessing
2. **Music symbol recognition** omr.symbol_recognition
   (a) **Score metric calculation** omr.symbol_recognition.score_metrics
   (b) **Stave identification & removal** omr.symbol_recognition.stave_detection
   (c) **Symbol segmentation** omr.symbol_recognition.segmentation
   (d) **Symbol classification** omr.symbol_recognition.classifier
3. **Musical Notation Reconstruction** omr.notation_reconstruction
4. **Final Representation Construction** omr.final_representation_construction

I also developed a utility package containing a lightweight 2D image processing library, as well as common functionality such as file I/O - **Utilities** (omr.util)

I will subsequently describe the implementation of each module.

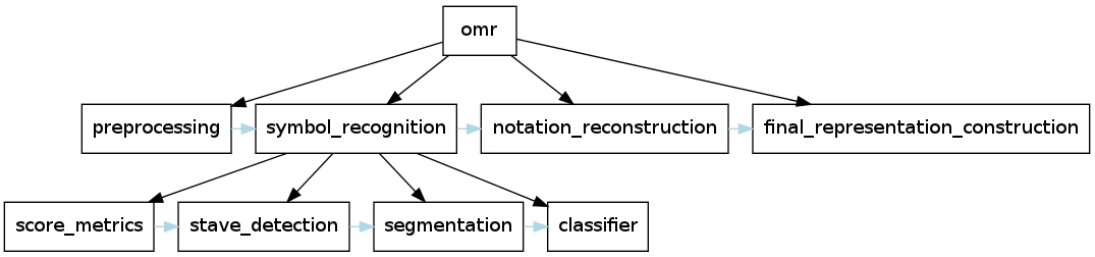


Figure 3.1: Project structure

3.1.2 Libraries Used

I used LibSVM [14] for a Support Vector Machine implementation and Encog [15] for an Artificial Neural Net implementation.

GAMERA [4] and the MusicStaves plug-in [5] provided functionality for my stave line removal evaluation code and some artificial noise generation.

I used the open source music engraving program LilyPond [7] to generate PDFs of typeset scores and MIDI files from LilyPond source code.

I used the JFreeChart Java graph library [18] to visualise some intermediate image processing steps.

I adapted some code by Alex Wong taken from javapractices.com [17] to design a class to recursively walk through a directory structure and return all the images contained.

I used some code from stackoverflow.com [10] to generate a list of unique colours of arbitrary length to populate an image of labelled connected components in my ConnectedComponentAnalysis class.

I made extensive use of both Java's and MATLAB's internal I/O and image processing libraries.

All other code is my own.

3.2 Preprocessing - *omr.preprocessing*

In order to keep the scope of this Part II project reasonable, I decided not to include this module in the success criteria specified in my Proposal. However in order to make a complete functioning OMR system, I implemented some basic routines in both MATLAB and Java to convert images from RGB to greyscale and to threshold greyscale images. I also implemented a simple noise removal algorithm based on run length smoothing - a very efficient way of removing salt and pepper noise. This involves performing both vertical and horizontal run length encodings of the image. When a single pixel run is found representing a

single flipped pixel in an area of constant colour, it is switched, smoothing the area.

3.3 Music Symbol Recognition - *omr.symbol_recognition*

This module contains the main OMR functionality.

In order to classify the music symbols, we must first segment them by extracting relevant foregound pixel regions from the input image. In conventional OCR systems that recognise printed text, segmentation is relatively simple to achieve. One line of text contains symbols whose positions vary horizontally. Their positions along the x axis encode meaning - knowing the ordering of the letters is necessary to decipher words, phrases and sentences. However, music notation is inherently more complex due to its 2D nature - not only do symbol positions vary horizontally in a single line of music, but also vertically. Variation along the y-axis, against the grid of the stave lines, encodes pitch (i.e. what note to play). This 2D layout makes segmentation much harder.

Another complexity in music notation is that certain symbols can be joined up to make composite symbols. A naïve solution would be to include such composite symbols as individual symbol in the classification stage. However the notation rules allow for almost unlimited variation, therefore defining a set of classes is near impossible. Even restricting the criteria to only commonly appearing composite symbols leads to an impractically large symbol set.

Therefore we must establish a way of breaking down such symbols into primitive symbols that can then be sent to the classifier, and then reconstructed in the latter OMR stages. This ensures the classification symbol set is kept to a manageable size. The problem of defining what the primitive symbols are is an important and non trivial one which will directly affect both segmentation and classification complexities and success rates - this will have to be carefully considered.

3.3.1 Score Metrics calculation - omr.symbol_recognition.score_metrics

This class calculates critical metrics that are then used throughout the segmentation process.

Why do we need it?

Just as text fonts come in a myriad of different styles and sizes, so do music engraving (typesetting) styles. However, there are certain conventions to which all these styles adhere to due both to the graphical rules of CMN and stylistic conventions. For example, because a note head has to fit snugly between two stave lines, we can deduce that a good estimate of the note head symbol's height will be this distance. Measurements such as this prove extremely useful during the segmentation phase.

All such measurements can be estimated using two critical metrics common to all CMN scores - the average stave line height and the average space between two stave lines (stave space height). From these we can estimate the 'size' of the music font.

How does it work?

The problem remains of how to calculate the two metrics. The method almost universally used in the literature [38], is finding the most frequent (modal average) white and black runs in the vertical run length encoding of the image of the score. This works because the stave lines are the most common, regular, repeated features on a page of music. This estimate is also immune to severe rotation of the image [38].

However, this method can sometimes fail due to the presence of image artefacts commonly generated when scanning in a page of music such as single pixel granular noise and large dark areas due to the page lifting off the scanner document table during scanning.

In 2010 Cardoso and Rebelo proposed a modified version of the common algorithm that produces much more reliable stave line/ space height estimates [29]. This was based on two key observations:

"first, the length of the run of white pixels before and after the isolated black pixels will vary a lot, 'randomly'. Second, a local fluctuation in the thickness of the [stave line] due to noise is often compensated by a variation with an opposite sign of the local distance between lines"

These imply that rather than considering isolated runs, some way of also considering their neighbouring runs could result in a more robust estimate. This is achieved by "finding the most common sum of two consecutive vertical runs" (either a black run followed by a white run or the reverse).

I chose to implement both the standard and new algorithms as I thought it would be interesting to explore the new method (not yet widely adopted) and evaluate it against the old way. It should result in a more robust system.

score_metrics.ScoreMetrics

Calls ScoreMetricsCalculator to calculate the two metrics using the algorithm specified by the input parameter, then calculates other dimensional metrics such as note head height and note head width and stores them in an object. This can then be passed to each of the subsequent stages of the segmentation process.

score_metrics.ScoreMetricsCalculator

Takes an image of a score and calculates the two essential metrics using one of two algorithms:

Algorithm 1: standard method

1. Calculate vertical run length encoding of the score image.
2. Count the number of black and white runs of each and every length (i.e. create two histograms - one for black runs and one for white runs, counting the frequency of each run length).
3. Get the global maximum frequency of each histogram - the most frequent black run length is the average stave line height, and the most frequent white run length is the average stave space height.

Algorithm 2: Cardoso, Rebelo 2010 [29]

1. Calculate the vertical run length encoding of the score image.
2. Iterate through this, summing pairs of runs, creating a new list of run pair sums.
3. Find the most frequent run pair sum - the global maximum of the 2D histogram.
4. Extract the black and white run portions from this most frequent run pair sum; they are the average stave line height and the average stave space height respectively.

3.3.2 Stave removal & identification - *omr.symbol_recognition.stave_detection*

In CMN the symbols are placed onto a two-dimensional grid known as the stave. Each stave line represents a particular pitch, relative to the other stave lines (and

stave spaces). A clef (a special symbol) located at the beginning of every line of music defines the absolute pitch each line and space represents.

Most OMR solutions contain a step that removes the stave from the image. This allows us to segment the remaining symbols much more successfully as otherwise the stave lines overlap the other music symbols and the stave appears as one large connected component. Stave removal results in white space between the symbols, leaving an image more amenable to segmenting.

Graphically, ideal stave lines are a set of five identical, straight, horizontal, evenly spaced parallel lines. However, scanned images of stave lines may contain irregularities such as curvatures, local skew and discontinuities. Stave line identification algorithms should be able to cope with such errors, and removal algorithms should be able to remove the stave lines without deforming the music symbols around them.

After reviewing the literature, most notably a recent comparative study [31], I decided to implement a new stave line removal algorithm described in [35]. This performed favourably against existing methods and hasn't yet been thoroughly evaluated which I thought would be interesting.

A stave line removal algorithm: Anjan, Dutta, 2010 [35]

This paper considers a stave line as a roughly "horizontal linkage of vertical runs of uniform height". It uses this property, as well as the fact that individual stave lines always have another roughly parallel line above or below them, if not both. It should output two images - one of just the music symbols, the other of just the staves. Therefore the algorithm must look at every foreground (black) pixel in the input image and decide image or set to put it in - a classification problem. I have included images of the GUI outputs of each stage during a typical run; the original image is shown in figure 3.2.

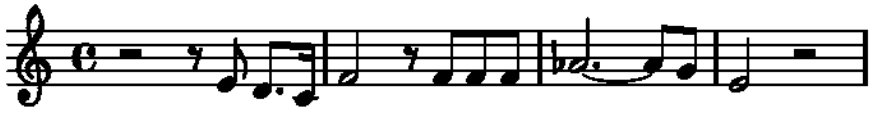


Figure 3.2: An extract from Schumann's *Ich Grolle Nicht* from *Dichterliebe*, typeset using LilyPond

The algorithm can be broken up into four main stages:

1. Horizontal run-length smoothing with parameter t_1. This should join up broken portions of stave line due to noise:

- All horizontal white runs shorter than t_1 ($staveLineHeight$, experimentally set) are converted to black.

2. Core algorithm:

   - Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of all vertical runs of black pixels in a document.
   - Let V^s be the set of vertical black runs in stave lines - the set we want to find.
   - Let V^m be the set of vertical black runs in music symbols - i.e. all those not in stave lines.
   - Then:

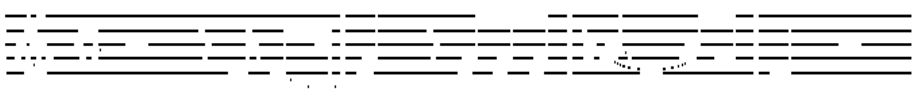
   $$V^s + V^m = V$$

   - A particular vertical black run $v_n \in V$ may be a part of the stave line (and therefore a member of V^s) iff:

   $$|staveLineHeight - \delta| \leq runLength(v_n) \leq |staveLineHeight + \delta|$$

   ($runLength(v_n)$ gives the length of the run v_n, i.e. how many pixels it contains)
   - This gives us a set of vertical black runs V^c (c for candidate). $V^s \subseteq V^c$, and is a good first approximation.
   - In the paper they experimentally set δ to 2 pixels, but having tried a few values I obtained better results with $\delta = 3$ pixels.

(a) Just symbols



(b) Just staves

Figure 3.3: Results after step 2 of stave line removal

3. Removal of non stave line segments from V^c:

   - First eliminate (remove from V^c , add back to V^m) all candidate stave line components having width less than t_2 ($2*staveLineHeight$, experimentally set), as most of the valid stave line components are expected to be wider than that, so these small ones are likely to be slices cut out from other symbols.
   - Next perform a 'neighbours check' on all remaining candidate stave line components (based on the second intrinsic property of stave lines discussed above):
     - Let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of all components from the image encoded by what's left of V^c (those wider than t_2 pixels).
     - For each component $c_n \in C$, we define its four neighbouring components as follows:
       (a) A component $c_n^l \in C$ is called the left neighbour of c_n if c_n^l is connected (4 connectivity) to the left of c_n in the original document.
       (b) A component $c_n^r \in C$ is called the right neighbour of c_n if c_n^r is connected (4 connectivity) to the right of c_n in the original document.
       (c) A component $c_n^t \in C$ is called the top neighbour of c_n if c_n^t is above c_n and

       $$verticalDistance(c_n, c_n^t) < (staveLineHeight + staveSpaceHeight)$$

       (d) A component $c_n^b \in C$ is called the bottom neighbour of c_n if c_n^b is below and

       $$verticalDistance(c_n, c_n^b) < (staveLineHeight + staveSpaceHeight)$$

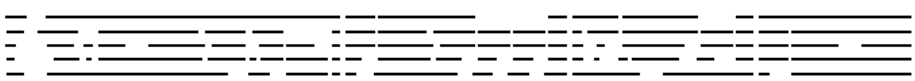
       ($verticalDistance(a, b)$ denotes the minimum vertical distance between components a and b)
     - Now, any arbitrary component $c_n \in C$, with the potential list of neighbours $c_n^l, c_n^r, c_n^t, c_n^b \in C$, is a part of a stave line if it has at least one of left or right neighbours and at least one of top or bottom neighbours:

$$\big(exists(c_n^l) \vee exists(c_n^r)\big) \wedge \big(exists(c_n^t) \vee exists(c_n^b)\big)$$

Therefore remove any components not fulfilling this criterion from C and add them to V^m. This eliminates most of the non stave line components from V^c.

Note how in Figure 3.4 the slur (resembling a right bracket rotated clockwise 90°) in the third bar has been reconstructed (compare to Figure 3.3).

(a) Just symbols



(b) Just staves

Figure 3.4: Results after step 3 of stave line removal

4. Re-addition of wrongly removed stave line segments

   This final step corrects over the over zealous removal of candidate stave line components in step 3. Due to noise and removal of small components, some small portions of stave line might have been eliminated (put back into the symbols image). Therefore we re-add some wrongly eliminated portions of stave line which satisfy the following criteria:

   (a) Let $S = \{s_1, s_2, \ldots, s_n\}$ be the set of all stave line components assembled thus far (all component in our current working approximation of V^s)

   (b) Let $E = \{e_1, e_2, \ldots, e_n\}$ be the set of all components we have eliminated (put back into V^m) from C in step 3.

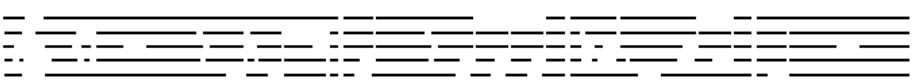
   (c) Return an eliminated component $e_n \in E$, with potential list of neighbours in S $s_n^l, s_n^r, s_n^t, s_n^b \in S$, to S if it has at least one of left or right neighbours (in S) and at least one of top or bottom neighbours (in S):

$$\big(exists(s_n^l) \vee exists(s_n^r)\big) \wedge \big(exists(s_n^t) \vee exists(s_n^b)\big)$$

This gives us a new set S':

$$S' = S \cup \{x | x \in E \text{ that satisfys the above condition}\}$$

(a) Just symbols



(b) Just staves

Figure 3.5: End results of algorithm

This algorithm gives us two images - one of the pixels identified as being part of a stave line, and the other of all the other pixels (making up all other music symbols). Figure 3.5 shows the end results from the input in Figure 3.2. Little change can be observed between stages 3 and 4 as the input is near ideal, and the post-processing step (4) is only useful when encountering complex or noisy scores See the evaluation section for more illustrative examples and edge cases. Therefore stave removal has been accomplished. However we must still detect the staves, so I developed my own method to complement this algorithm.

Stave Identification -
omr.symbol_recognition.stave_detection.StaveIdentification

Knowing the location and shape of each stave line is essential for the notation reconstruction phase - it allows us to compare the location of each symbol to the reference y values so that we can calculate its pitch. Certain symbols are y-axis invariant, however most aren't - all notes, accidentals and some rests change meaning depending on their location relative to the stave. Specific details of this stage are surprisingly hard to come by in the literature, so I decided to design my own solution.

I began by creating a simple data structure to contain a skeleton of the stave lines - for each x position along the image of one line of music, it contains the y location of the midpoint of each of the five stave lines. This is represented by the **stave_detection.StaveSkeleton** class. The structure is stored internally as a

2D integer array with each cell of the first array (of length equal to the image width) storing a pointer to a further 5 element array.

I then devised an algorithms to populate this data structure, explained below.

stave_detection.StaveIdentification

This class uses a type of template matching procedure based on a stack to locate and track stave lines, meanwhile populating a stave skeleton data structure. It uses the image of just stave lines from the Stave Removal process, first performing a vertical run length encoding. It then looks at every column of the image individually, attempting to match the run length encoded data to an ideal stave template (calculated using $staveLineHeight$ and $staveSpaceHeight$, allowing for small error margins). Starting at the top of the image, a black run of approximate length $staveLineHeight$ is located. Then another black run is looked for approximately $staveSpaceHeight$ away. If no matching run is found, the previous one is discarded and the process repeats. This ensures noise above and below the staves doesn't break the algorithm. If a match is found, the algorithm repeats, until all 5 stave lines have been found. In order to deal with gaps in stave lines due to removed crossing symbols, the algorithm includes a post-processing step which draws in the missing skeleton using the average of the two surrounding stave line ends. Figure 3.6 illustrates the output of this procedure in a simple GUI. Note how the algorithm has successfully tracked the stave lines even with unwanted components above and below the stave, and where there are gaps in the stave it had successfully inferred the stave skeleton values.

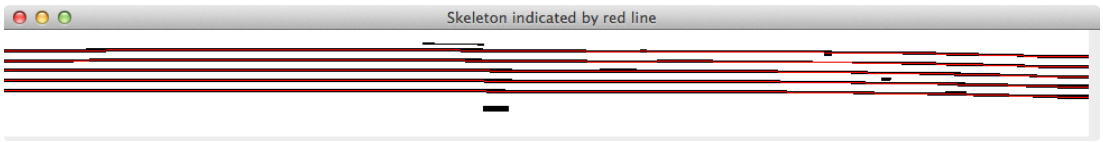


Figure 3.6: A GUI displaying the stave skeleton (in red) calculated on an image of just stave lines generated by StaveLineRemoval

3.3.3 Symbol Segmentation - *omr.symbol_recognition.segmentation*

Segmentation is a classic Computer Vision problem. It aims to partition the image into multiple sections in order to make it easier to analyse. Segmentation is typically used to locate objects and boundaries in images, both of which are regions of pixels with similar visual characteristics. In OMR systems segmentation

is used to extract symbols from the image to send to the classifier. A hierarchical decomposition of the image is performed in an attempt to isolate symbols accurately.

Crowded and broken symbols both present significant challenges. The segmenting of composite symbols into primitives is a non-trivial problem - how do we rigorously define what constitutes a symbol? A common approach in OCR is to define a symbol as a connected component. Therefore a connected components labelling can be performed on the input image, each component extracted, and the results sent to the classifier. However this is not possible with CMN, even if we momentarily discard issues of symbol crowding, as composite symbols such as as chords and beamed note groups introduce almost unlimited variation into the symbol set. Commonly such symbols are further segmented into primitive symbols such as note heads, stems and beams.

The literature in this area is diverse - no current solutions are 100% reliable. After experimenting at length with a number of techniques, I found the best solution was a combination of methods taken from a number of papers.

Heirarchical Decomposition Structure

Stave segmentation - *omr.symbol_recognition.segmentation.Stave_Segmentation*

This first segmentation section was added as an extension to my project. It uses the connected component analysis method from [37]. An analysis is performed, and all components below a certain size are discarded. Neighbour rules are use to ensure notes above or below the stave line aren't also removed

Level 0 segmentation - *omr.symbol_recognition.segmentation.L0_Segmentation*

Level 0 segmentation aims at reducing the search space for subsequent segmentation levels by detecting the main blocks of music in the stave. For simple scores this may suffice to extract the symbols, whoever composite symbols require further segmentation, as illustrated in 3.8b.

A projection of the stave onto the x axis is performed. This is then iterated over, from left to right. If the level of the projection in the current column exceeds a threshold t_3, a counter is initialised, and an L0 segment is defined as the subimage from this current column onwards to the right, until the projection values return to below threshold t_3.

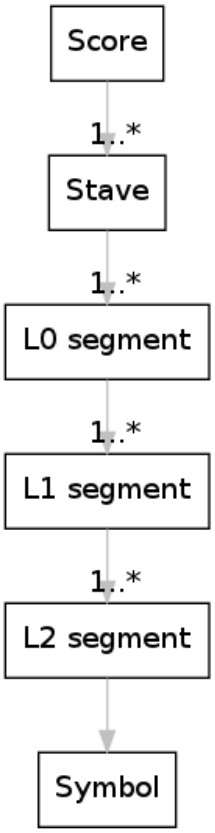
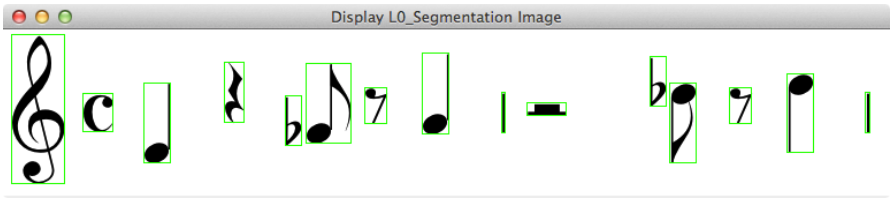
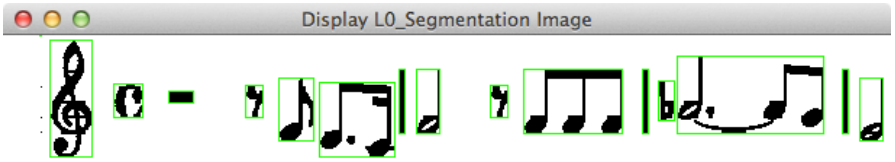


Figure 3.7: Hierarchical graphical decomposition structure

Each L0 segment is subsequently restricted along the y axis by calculating its bounding box. Figure 3.8 illustrates the Level 0 segmentation output for simple and complex scores.



(a) Level 0 Segmentation of a simple score



(b) Level 0 Segmentation of a complex score

Figure 3.8: Level 0 Segmentation results for simple and complex scores, displayed in GUI

Level 1 segmentation - *omr.symbol_recognition.segmentation.L1_Segmentation*

For SimpleOMR, Level 0 segmentation is usually sufficient. However when we encounter composite symbols and symbols that overlap and touch one another such as slurs, the segmentation necessarily becomes more complex - they must be decomposed to primitive symbols. We can visualise Level 1 segmentation as extracting segments that represent units of time in the music - each will contain one rest or note or chord. Therefore we need a way of segmenting beamed note groups appropriately.

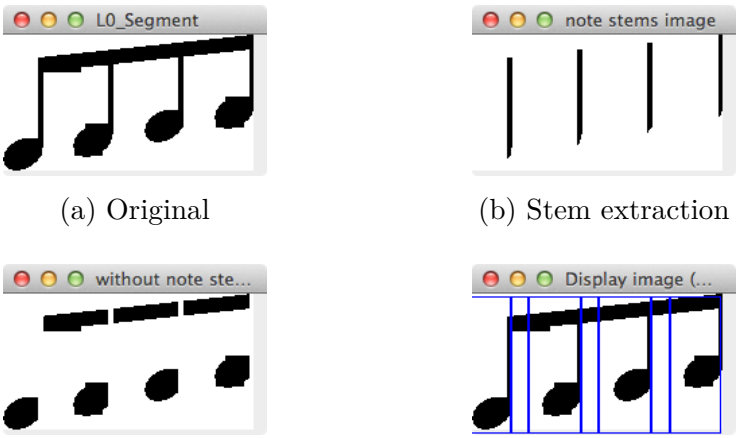
L1 Segmentation examines the L0 segments previously calculated. An estimate of the maximum possible primitive symbol width is made (experimentally set to $4 * staveSpaveHeight$. If segments have width less than this they are likely to be primitive symbols already, and no further segmentation is required. For those wider than this we invoke the L1 segmentation algorithm.

Most techniques in the field yield mixed results. I decided to implement a stem detection based algorithm inspired by [26]

We will first examine the latter as it is computationally simpler, being based on RLEs and projections. Note stems offer a unique graphical feature whose properties we can exploit to locate notes in a beamed group. The only other

long vertical lines commonly appearing in CMN are the bar lines, which don't usually feature in beamed note groups as they rarely span bars. If we locate the stems, we have not only located the boundaries of the L1 segments but also have a means of segmenting the composite symbols into primitives. In Belinni 200?, run length encoding is used to locate the note heads. I used a similar method to locate note stems to see if this yielded better results.

The algorithm's main transformations are illustrated in Figure 3.9. Vertical runs of heights within a certain range (estimates of the minimum and maximum possible note stem heights, experimentally set) are removed from the image and added to a new one. To find their positions, an x projection is performed (see Figure 3.10) and searched for peaks. Once located, the L1 segment edges can be calculated using estimates of the maximum note head width based on the 2 score metrics (specifically, $2 * staveSpaceHeight$). The side of the stems the note heads appear is also significant when defining L1 segment boundaries. This is calculated by examining the x position of the furthest right note stem - if it is near the right edge of the L0 segment, the note head is necessarily on the left; if they're about a note's width apart, the note heads are on the right of the stems.



(a) Original (b) Stem extraction

(c) Without stems (d) Done - L1 segments

Figure 3.9: Level 1 segmentation of a Level 0 segment containing a beamed note group (extracted from a real score using stave removal and L0 segmentation)

The method proposed by Fuijinaga in [37] should provide a more accurate segmentation due to its use of connected component labelling, a more advanced segmentation technique.

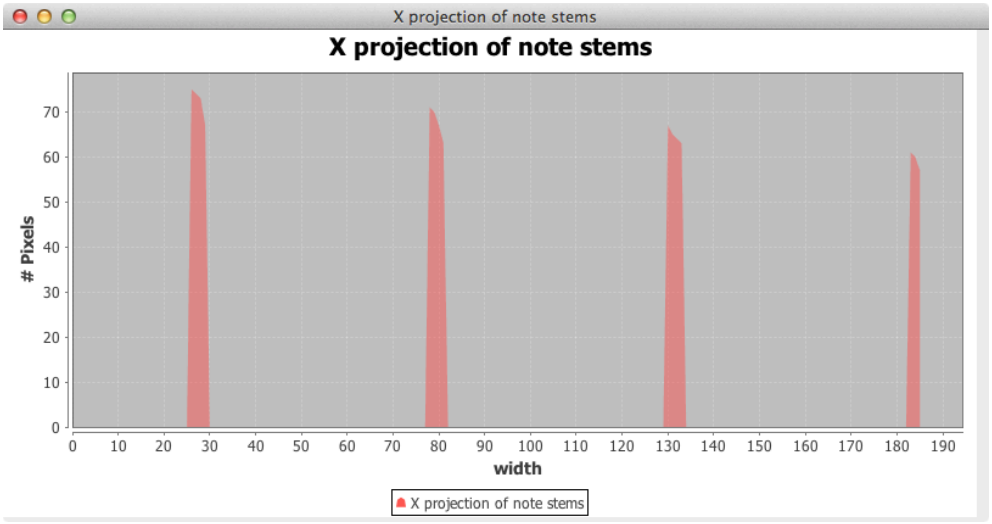


Figure 3.10: The x projection of the extracted note stems from Figure 3.9, generated using JFreeChart

Level 2 segmentation - *omr.symbol_recognition.segmentation.L2_Segmentation*

Level 2 segmentation finally extracts the individual primitive symbols from the image. Its functionality and necessity depends on the method of L1 segmentation used above as each leads to slightly different results. In the case of the stem detection method, L1 segments (equivalent to individual moments of time/rhythmic niches) must be further segmented into primitive symbols. Using the image with stems removed, horizontal cuts can be made by examining the y projection of each L1 segment, to get the bounding box of each primitive symbol. This is illustrated in Figure 3.11

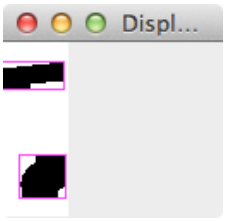


Figure 3.11: Level 2 Segmentation of the second L1 segment in Figure 3.9

Thus we have performed a hierarchical decomposition of the image, successfully decomposing composite symbols into their primitive symbol components.

3.3.4 Symbol Classification - *omr.symbol_recognition.classifier*

I decided to use Support Vector Machines and Artificial Neural Networks to perform symbol classification. Both methods perform well in the OMR literature [23] and can cope with the large variation in inputs due to different music fonts, sizes and noise.

As features I decided to use 20*20 pixel binaries images. Using individual pixel intensities as feature values is the least computationally expensive form of feature extraction, and performs well in the literature [23].

I will subsequently describe the training, testing and system usage of each classifier.

Training and testing data

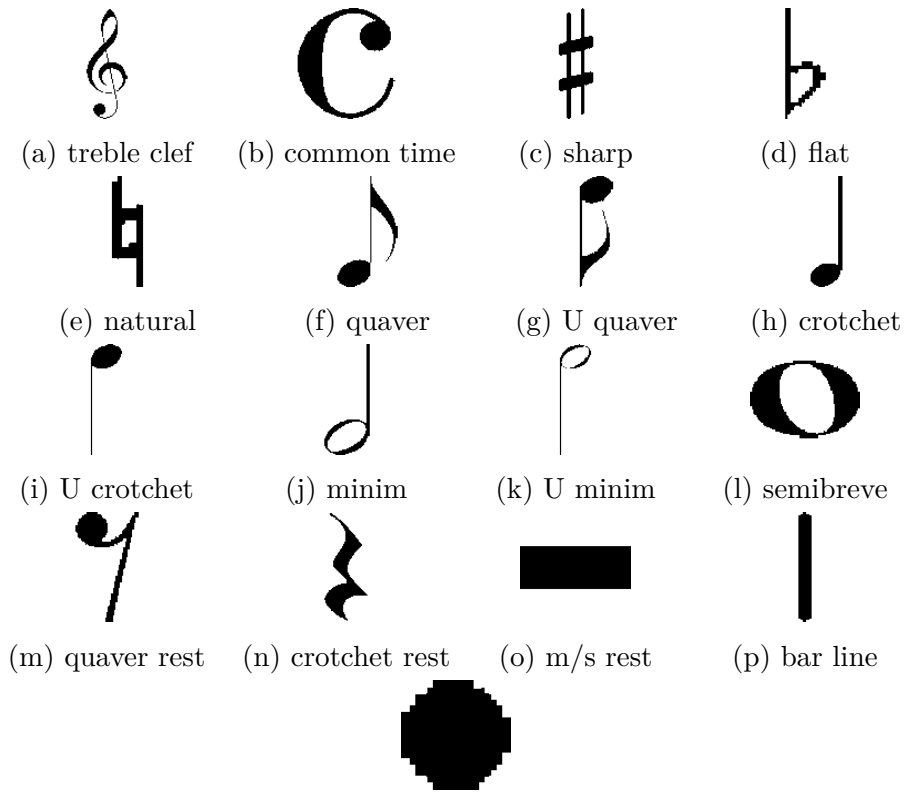
I obtained training and testing data from a large number of sources. Firstly I searched for the term "music fonts" (and variations of this) using Google, Yahoo and Microsoft Bing. Taking the first 20 hits for each search engine, from each site I downloaded all the open source music fonts I could find, compiling a list of 56 different fonts. Using the Mac OSX Font Book application, I generated 'repertoire' documents for each font - these contain every single symbol in the particular font, displayed in grids spanning multiple pages. I converted these to PNG images and wrote a Java class to automatically extract all the symbols.

Another symbol image source was those included with the open source OMR system OpenOMR [33]. I also used various music engraving programs including LilyPond, Sibelius and Finale to generate simple scores in a range of fonts which I ran my segmentation algorithms on to extract the symbols.

Once I had aggregated this image set, I used MatLab to standardise the size, colour space and padding of each symbol. I then manually picked and classified all the symbols of interest based on the symbol class set I defined for BasicOMR, shown in Figure 3.12.

Support Vector Machines - *omr.symbol_recognition.classifier.svm*

After reviewing the state of open source SVM implementations, I decided to use LibSVM. Although originally written in C++, LibSVM also offers a robust Java implementation. Having both implementations allows training using the C++ version, which greatly speeds up the process, and the flexibility of allowing me to incorporate LibSVM into my Java OMR system. This was very easy - LibSVM has a simple, well defined interfaces.



(a) treble clef (b) common time (c) sharp (d) flat

(e) natural (f) quaver (g) U quaver (h) crotchet

(i) U crotchet (j) minim (k) U minim (l) semibreve

(m) quaver rest (n) crotchet rest (o) m/s rest (p) bar line

(q) augmentation
or sticatto dot

Figure 3.12: Set of symbol classes considered in BasicOMR. 'U' codes for 'upside-down', and 'm/s' for 'minim/semibreve' .

I used MatLab to extract the pixel by pixel features from the training and testing images, constructing a text file in a format suitable for input to LibSVM. I randomly divided the dataset into training and testing sets (ratio 60:40). I trained the SVM using the radial basis kernel function, as it was recommended both by the developers and in recent OMR literature [23].

Artificial Neural Network - *omr.symbol_recognition.classifier.ann*

Having looked at various ANN implementations, notably Joone [9], Neuroph [8] and Encog [15], I decided to go with the latter. Encog is "an advanced neural network and machine learning framework", developed by Heaton Research. It again provides implementations in Java and C++ (and .Net), and contains advanced and highly optimised training algorithms.

Training data was parsed from the text file prepared for LibSVM. I used the Resilient Propagation (RPROP) training method as it has no parameters that must be set and is one of the best general purpose algorithms in Encog. Training was performed directly in Java. Incorporating the classifier into my system was more complicated than for LibSVM, but still relatively straight forward.

3.4 Music Notation Reconstruction - *omr.notation_reconstruction*

The analysis performed so far should almost entirely describe the graphical properties of the input score. Now we use this information to reconstruct the music notation, extracting the music semantics from the positional and classification data. We build up a logical internal representation of the score using the notation rules. In OMR this is a difficult problem - the complex two dimensional layout makes it much harder than in conventional OCR systems.

A common approach in the literature is the formalisation of music knowledge using a grammar. One of Fujinaga's first papers used *context-free* and $LL(K)$grammars to characterise music notation. These methods, however, we're outside the scope of this project. I decided to implement a solution based on a "fusion of musical rules and heuristics", more similar to approaches pursued in [43].

Symbol recognition in SimpleOMR produces a Score object containing one or more staves, containing a list of L0 segments, each of which have been labeled with a symbol class. This module first looks for the stave's starting clef, key signature and time signature. These symbols all act as global modifiers - they

set the pitch and rhythm contexts for all subsequent symbols (until another clef, time signature or key signature is found). As the time signature is often not repeated after the first stave, this is remembered between staves as the score's current default time signature.

Once the global stave context has been set, the rest of the symbols are iterated though from left to right along the stave. For those symbols which are y-axis variant (in other words change their meaning depending on their y position - this includes notes, accidentals and some rests), their pitch is calculated by comparing their y position to the previously calculated stave skeleton. Whereas some symbols' semantic centres (i.e. where you measure their pitch from) coincide with their physical one, this isn't always the case - flats are one example. Therefore in each symbol object we store a semantic centre variable calculated taking the symbol's class into account. We calculate the symbol's pitch using the method presented in [33]:

$$\text{Note } = \lfloor \frac{refPos - yPos}{\frac{1}{2}staveSpaceHeight} \rfloor$$

Where $refPos$ is the y value of the lowest stave line in the stave skeleton for this s position, and $yPos$ the semantic centre of the symbol.

Individual bars must add up to the rhythmic value specified in the time signature.

For the first iteration of my system, BasicOMR, I implemented a lightweight notation reconstruction module that almost directly translates the score's graphical representation into LilyPond code, as each of the three potential final outputs are derived from LilyPond source files. As part of the second implementation iteration, I developed a completely output-agnostic, detailed object-oriented logical representation of music scores, a simplified but parallel approach to that undertaken in the WEDELMUSIC Object-Oriented Music Notation Model [1]. Unfortunately due to project time constraints I did not have time to fully and successfully incorporate this back into my system - further work would be to complete this process.

3.5 Final Notation Construction - *omr.final_representation_construction*

This final module generates the desired outputs from the assembled LilyPond source file by adding relevant parameters and invoking the LilyPond compiler through a system call. I chose a MIDI file, a typeset PDF score and the raw Lily-

Pond source code as outputs as they are not only useful in real world applications but allow evaluation from various perspectives.

3.6 Utilities - *omr.util*

3.6.1 Image processing - *omr.util.imageProcessing*

I implemented all the algorithms discussed in the Computer Vision section in Preparation. This included:

- Image I/O, including MatLab-esque quick image previews
- Run length encoding
- Projections
- Thresholding and RGB to Greyscale conversion
- Connected Component Analysis using a Union find data structure. This included the construction of a set of unit tests illustrating edge case inputs. I created algorithms to convert between image and matrix (2D array) representations so images could be stored directly in the code.

3.6.2 Other

Other smaller algorithms I developed included:

- training image extractors
- debugging class
- MatLab scripts to create degraded/noisy images, extract features from images for classification, rotate and resize images and convert images between colour spaces.
- Python scripts invoking GAMERA evaluation classes

Chapter 4

Evaluation

OMR systems are difficult to evaluate. One approach is to evaluate each module separately - I will evaluate my system like this first. I designed it with well defined, loosely coupled modules, amenable towards testing each one separately as such. I will then look at techniques for evaluating the whole system (so called *black box* evaluation) - this will enable comparison with other OMR solutions.

4.1 Testing Data

4.1.1 Sources

Another problem in the field of OMR evaluation is the lack of public databases of music scores suitable for use by the research community. Although many online music libraries now exist [11, 6, 13, 20], these are unsuitable due to inconsistent formats, a lack of ground truth data, only individual score access and licensing issues.

Two recent papers have addressed this issue, both aimed towards evaluating stave removal algorithms. The first, a comparative study of stave removal algorithms, [31] includes a small test set of typeset scores which the authors later made freely available. The other paper [36] describes the construction of a large dataset of handwritten scores called *CVC-MUSCIMA ground truth*, consisting of "1000 music sheets written by 50 different musicians", specially designed for the evaluation of stave removal and composer identification algorithms. Both datasets contain images of scores and the corresponding ground truth images containing just the stave lines and just

the symbols, illustrated in Figures 4.1 and **??**. I have used the handwritten dataset in my evaluation as I proposed to evaluate my system's performance on handwritten scores, and as it is the largest and best quality dataset available.

I wrote to the authors of both papers, who pointed me towards a location to download both sets from, which proved invaluable for the evaluation of the Score Metrics and Score Removal modules. Secondly, as a keen musician I have plenty of my own scores, and my neighbour who studies music provided ample manuscripts on request.

4.1.2 Testing Data Generation

A useful technique for testing Computer Vision applications' robustness when encountering noisy inputs is the artificial distortion of inputs. It allows the levels of noise and distortion to be quantified to stress test the system until failure.

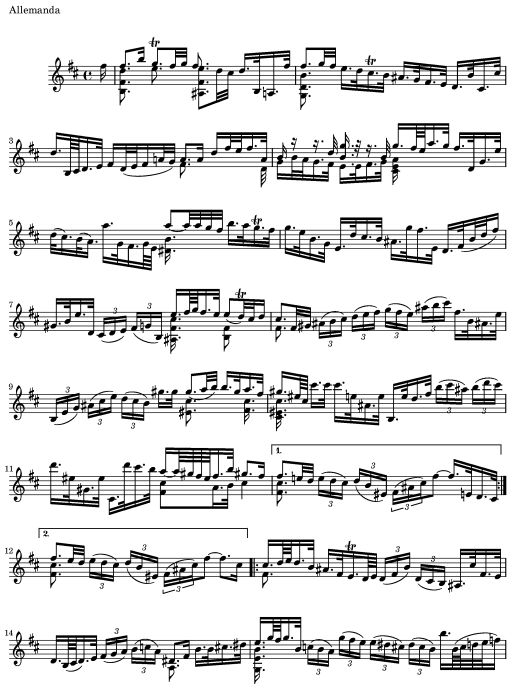
Considering likely types of noise and distortion that will affect the inputs will lead to an evaluation that better predicts real world performance. In [31] Dalitz et al. define a set of image distortions to apply to the ground truth images that hope to emulate the types of distortions that commonly appear when using OMR systems. These can be split into two categories, *deterministic deformations* and *random defects*. The distortions are summarised in figure 4.3.

Some of these deformation methods require further explanation. The *curvature* is calculated using half a sine wave for one stave width. The amount of curve is specified by the ratio between the amplitude and the width of the wave.

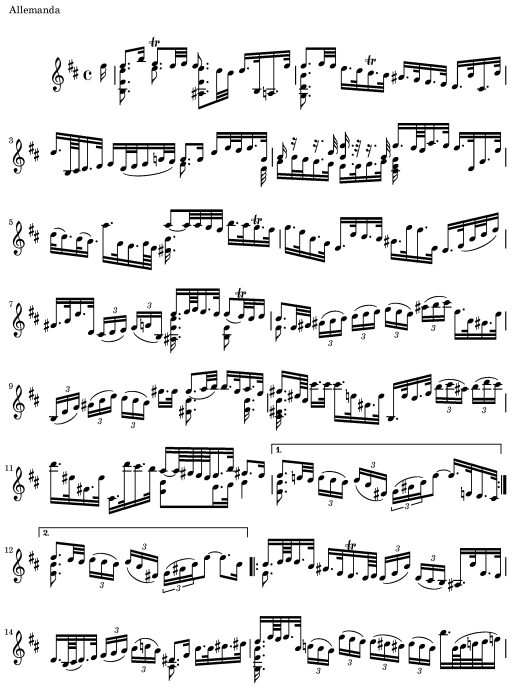
Typeset emulation attempts to recreate the kind of noise typically seen in early music scores set with lead types, giving stave line interruptions between symbols and a random vertical shift of each vertical slice of the stave containing one symbol.

Stave line *thickness variation* and *y-variation* are modelled using a Markov chain as the thickness at a particular position along the x axis depends on the thickness at the previous one. The parameter is the transition matrix P, where the $(i, j)^{th}$ element of P is:

$$p_{i,j} = P(X_{n+1} = j \mid X_n = i)$$

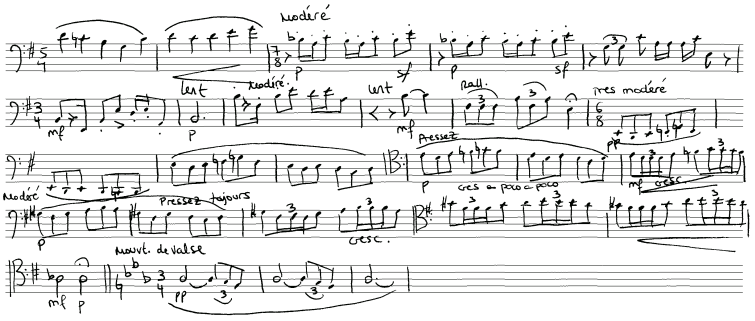


(a) Original score

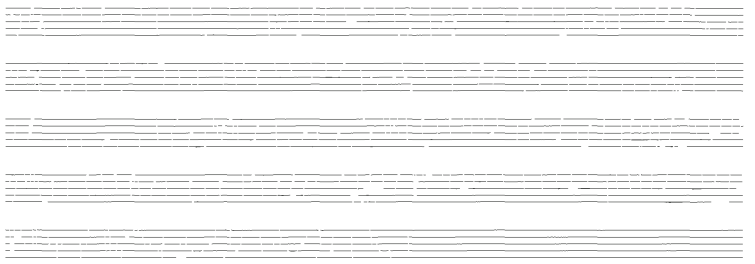


(b) Symbols ground truth

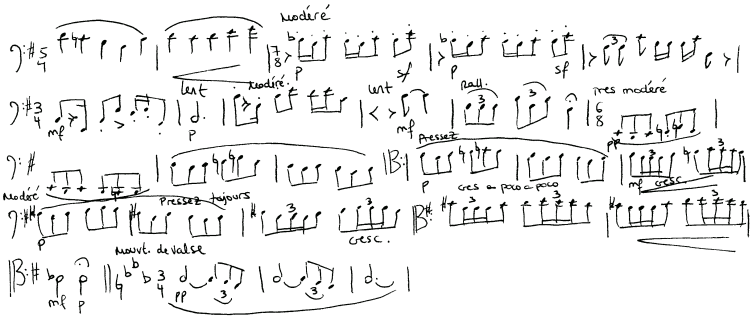
Figure 4.1: An example score from Dalitz et al's set, with corresponding ground truth stave removal symbol image



(a) Original score



(b) Staves ground truth



(c) Symbols ground truth

Figure 4.2: An example score from the CVC-MUSCIMA ground truth dataset, with corresponding ground truth stave removal images

Deformation	**Type**	**Parameters**
Resolution	Deterministic	Dots per inch
Rotation	Deterministic	Rotation angle
Curvature	Deterministic	Height:width ratio of sine wave
Typeset emulation	Both	Gap width, max height and variance of vertical shift
Line interruptions	Random	Interruption frequency, max width and variance of gap width
Stave line thickness variation	Random	Markov chain stationary distribution and inertia factor
Stave line y - variation	Random	Markov chain stationary distribution and inertia factor
Kanungo degradation	Random	$(\eta, \alpha_0, \alpha, \beta_0, \beta, k)$, see below
White speckles	Random	Speckle frequency, random walk length and smoothing factor

Figure 4.3: Types of distortions presented in [31]

In words, the probability of the stave line at a particular x-position being of thickness j given that at the immediately previous x-position (to the left) it had thickness i. Or i and j represent the amount of y-deviation at that particular x-position. The states are modelled as a stationary symmetric binomial distribution with mean value of the stave line height of the undeformed image. The inertia factor specifies the smoothness of the transitions - the closer it is to 1, the slower the state variation.

Kanungo et al. [40] proposed a degradation model aimed at replicating common artefacts generated during document photocopying, scanning and printing, which are very common in music scores. Their model, based on the mechanics of these three processes, the uses 6 parameters $(\eta, \alpha_0, \alpha, \beta_0, \beta, k)$ as follows:

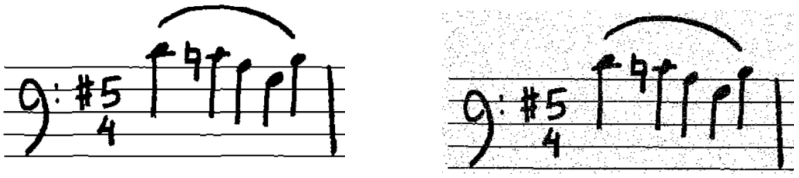
- flip a foreground pixel with probability $P_f(flip) = \alpha_0 e^{-\alpha d^2} + \eta$, where d is the distance of the closest background pixel.
- flip a background pixel with probability $P_b(flip) = \beta_0 e^{-\beta d^2} + \eta$, where d is the distance of the closest foreground pixel.
- lastly, perform a morphological closing operation with a disk of diameter k

The white speckles degradation, with parameters (p, n, k) for speckle frequency, random walk length and smoothing factor, works as follows:

- select a foreground pixel with probability p as the starting point for a random walk, length b.
- the subimage containing the random walk is smoothed using a morphological closing operation with a rectangular structure of size k.
- after all the random walks have been performed, the resultant image is subtracted from the original, resulting in flipped foreground pixels at the random walk positions.

Figure 4.5 illustrates the effects of these distortions on a simple music image.

I also used MATLAB to generate artificial salt and pepper noise in my test images. This involves randomly switching a certain number of pixels in a binary image, depending on the noise density parameter specified.



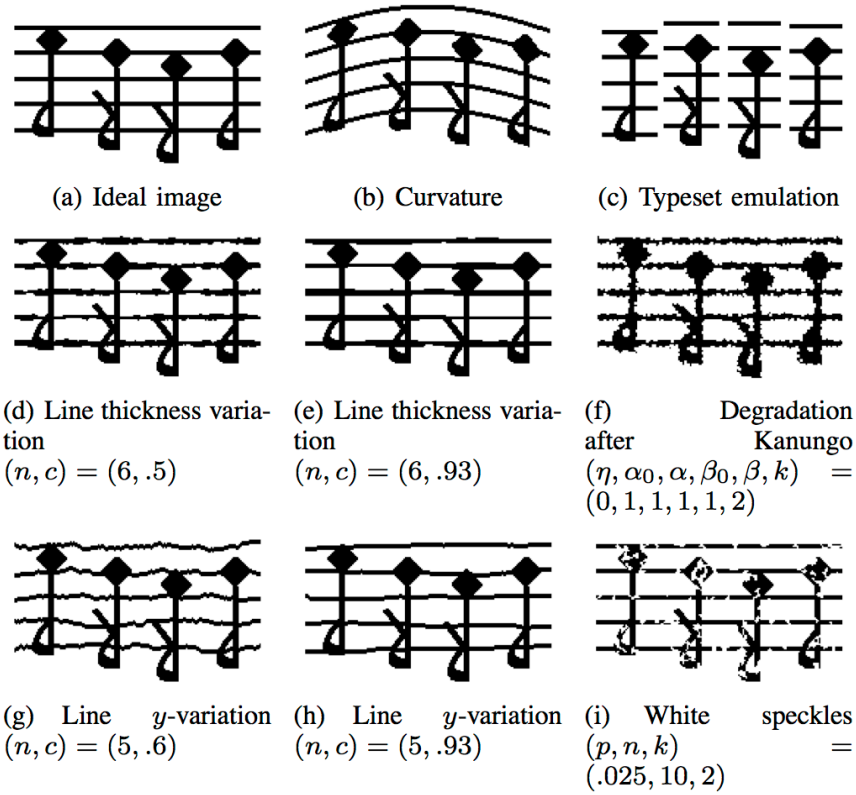
(a) before (b) after

Figure 4.4: A small section of an ideal score from *CVC-MUSCIMA* before and after the application of 'Salt and Pepper' noise with density paramater 0.035 using MATLAB

Although actual noise will typically consist of a mixture of these types, it is useful to evaluate the system against pure forms so conclusions can be drawn about why they fail [31].

4.2 Evaluating StaveMetrics module

To evaluate the two stave metric calculation algorithms, I put together a test image set comprising of ideal scores from *CVC-MUSICIMA*, distorted versions of these scores using the techniques in [31] and scores distorted by varying levels of artificial salt and pepper noise generated using MATLAB, giving a total of 5172 test images. As all the scores have standardised stave



(a) Ideal image (b) Curvature (c) Typeset emulation

(d) Line thickness variation $(n, c) = (6, .5)$ (e) Line thickness variation $(n, c) = (6, .93)$ (f) Degradation after Kanungo $(\eta, \alpha_0, \alpha, \beta_0, \beta, k) = (0, 1, 1, 1, 1, 2)$

(g) Line y-variation $(n, c) = (5, .6)$ (h) Line y-variation $(n, c) = (5, .93)$ (i) White speckles $(p, n, k) = (.025, 10, 2)$

Figure 4.5: A simple 'ideal' music image and its deformations. Figure taken from [31]

line and stave space heights, we can compare the results of the algorithms with ground truth.

The ideal and Dalitz-distorted scores (2772 images) were processed with near 100% accuracy by both algorithms - the run length method is very robust against most types of noise and distortions such as rotation. This matches results in the literature [38]. Figure 4.6 illustrates the performances of the two algorithms on the salt and pepper noise data set (3400 images). The noise density parameter can vary between 0 (no noise) and 1 (all noise, original image totally obscured). Having run my algorithms on images with various levels of noise I chose an illustrative range of noise densities from 0.015 to 0.07. The y axis is percentage of correct approximations (all the images of scores in the dataset had a stave line height of 2 pixels and a stave space height of 27 pixels) made, the x axis representing the varying levels of noise as defined by the noise density parameter.

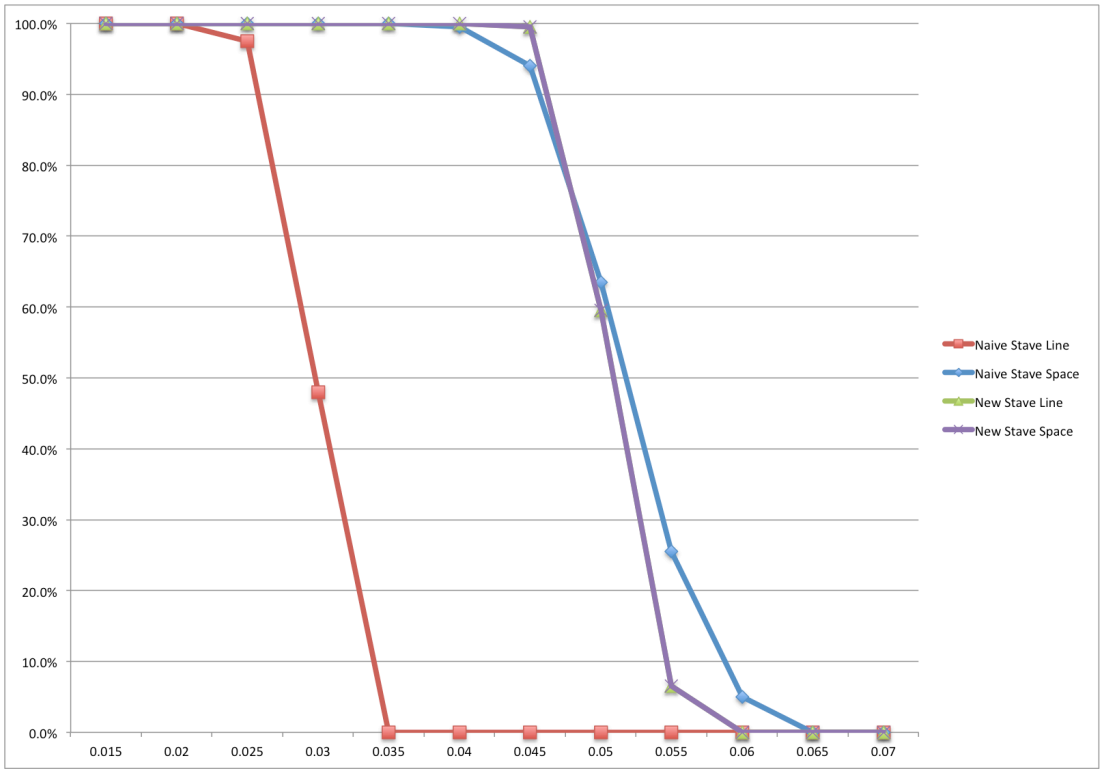


Figure 4.6: A graph showing each algorithm's stave line height and stave space height estimations for varying densities of artificial salt and pepper noise.

The new algorithm's stave line and stave space data points coincide as expected, due to the fact that the new algorithm calculates the stave line

height and stave space height as a pair by choosing the most frequently occurring sum of run pairs. The graph shows that the new algorithm consistently outperforms the naive stave line height calculation. Although the naive stave space calculation slightly outperforms the new algorithm for high noise levels, as the naive stave line height has already failed for all images at this noise level, the algorithm will fail and the encompassing OMR system is unlikely to function as desired.

4.3 Evaluating the Stave Line Detection module

The easiest way to evaluate stave line removal algorithms is to treat them as two-class classification problems at the pixel level. The algorithm should class every foreground pixel in the original image into either a stave line pixel or a symbol pixel. Then we define the error rate as:

$$\text{errorRate} = \frac{\#\text{ misclassified stave pixels } + \#\text{ misclassified non stave pixels}}{\#\text{ all stave pixels in original}}$$

This however is a rather coarse metric - although it gives an idea of how badly the symbols have been distorted compared to the ground truth symbols-only image, it gives little information as to how the algorithm cuts symbols crossing stave lines (i.e. that overlap with stave lines) [31].

We therefore use two further techniques originally developed for evaluating text document segmentation algorithms, proposed for use in stave removal algorithm evaluation in [31]. These evaluate the algorithm's performance at the segmentation region level and considering stave line interruptions.

The first, the segmentation region level, frames stave removal as a segmentation problem attempting to segment sections of stave line - all other pixels are background. We have two segmentations - one defined by the ground truth stave line image, the other the actual result produced by our system. In the set of all stave line segments from both ground truth and actual results, we build equivalence classes of overlapping segments. For each resulting equivalence class we count the number of contained segments from ground and the actual results, and can therefore detect segmentation errors. We then define the error rate among all the equivalence classes as:

$$\text{errorRate} = \frac{\#\text{ all classes } r - \#\text{ classes representing a correct region}}{\#\text{ all classes } r}$$

As it is possible to calculate the ideal stave line locations from the ground truth data, we can reduce the matching problem to that of one dimensional intervals. In both the ground truth stave lines image and the one created by our algorithm, we follow each stave line from left to right, counting stave line interruptions (that represent music symbols crossing them). For each image this yields a set of interrupting intervals which can be compared. To establish an error metric we construct a bipartite graph by adding links between intervals from each image that overlap. Figure 4.7 illustrates this process. This gives two types of errors: unlinked intervals and intervals with more than one link. Counting the latter by calculating the maximum cardinality matching of the graph, we establish the error metric as:

$$\text{errorRate} = \frac{min(\#\text{ GTI}, \#\text{ IWL} + \#\text{ RL})}{\#\text{ all classes } r}$$

where GTI are ground-truth interruptions, IWL are interruptions without link and RL are removed links.

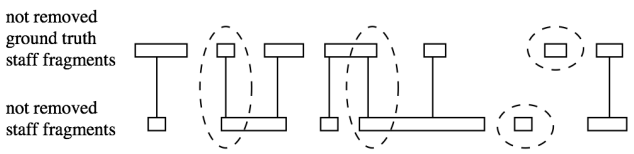


Figure 4.7: Matching stave fragments from ground-truth data to our stave removal algorithm's result. Errors are circled.

Having run my algorithm on subsets of the typeset dataset and *CVC-MUSICIMA*, I obtained the following results: 4.8 4.9

I have also included some output images for quantitative evaluation - see figures 4.10, 4.11 and 4.12.

4.4 Evalutating classification module

By dividing the symbol datasets into training and testing sets, we obtain an easy way to evaluate the two classifiers based on the percentage accuracy obtained when classifying the testing set.

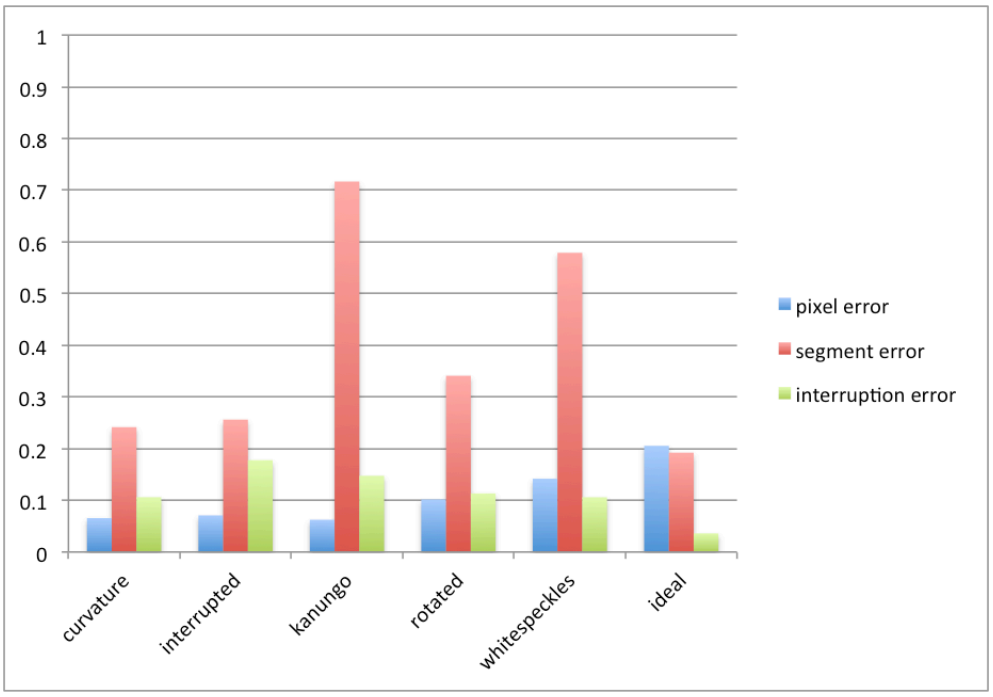


Figure 4.8: A graph showing the three error metrics calculated from results of my algorithm on the typeset dataset

4.5 Whole System Evalutation

As previously mentioned, an automated OMR whole system evaluation method has not yet been devised. How to quantified the accuracy of systems which differer in allowed input sets and output formats is made harder by the fact that certain errors carry more weight than others. Decisions must be made as to whether the errors produced compromise the score's meaning. Errors in global context setting symbols such as clefs and time signatures will obviously have a large detrimental effect on the final output, whereas some smaller forms of errors may even be dynamically fixed by considering prior knowledge.

A few interesting methods have recently been proposed. In [27], Bellini et al. propose a series of complex set of metrics evaluating the OMR system output at different logical layers.

Another solution, used in takes its inspiration from sequence matching, a problem in the field of BioInformatics. We define the evaluation metric as the number of point changes it would take, using an idealised, general Music Notation editing program, to transform the system's output into the

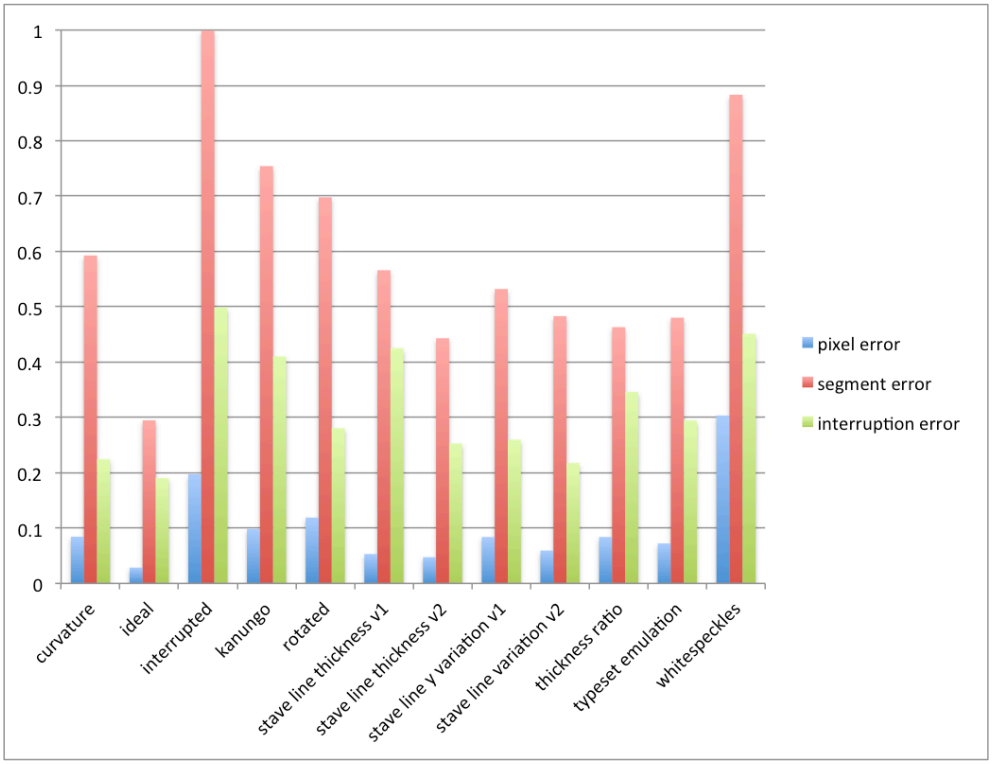


Figure 4.9: A graph showing the three error metrics calculated from results of my algorithm on the *CVC-MUSICIMA* dataset

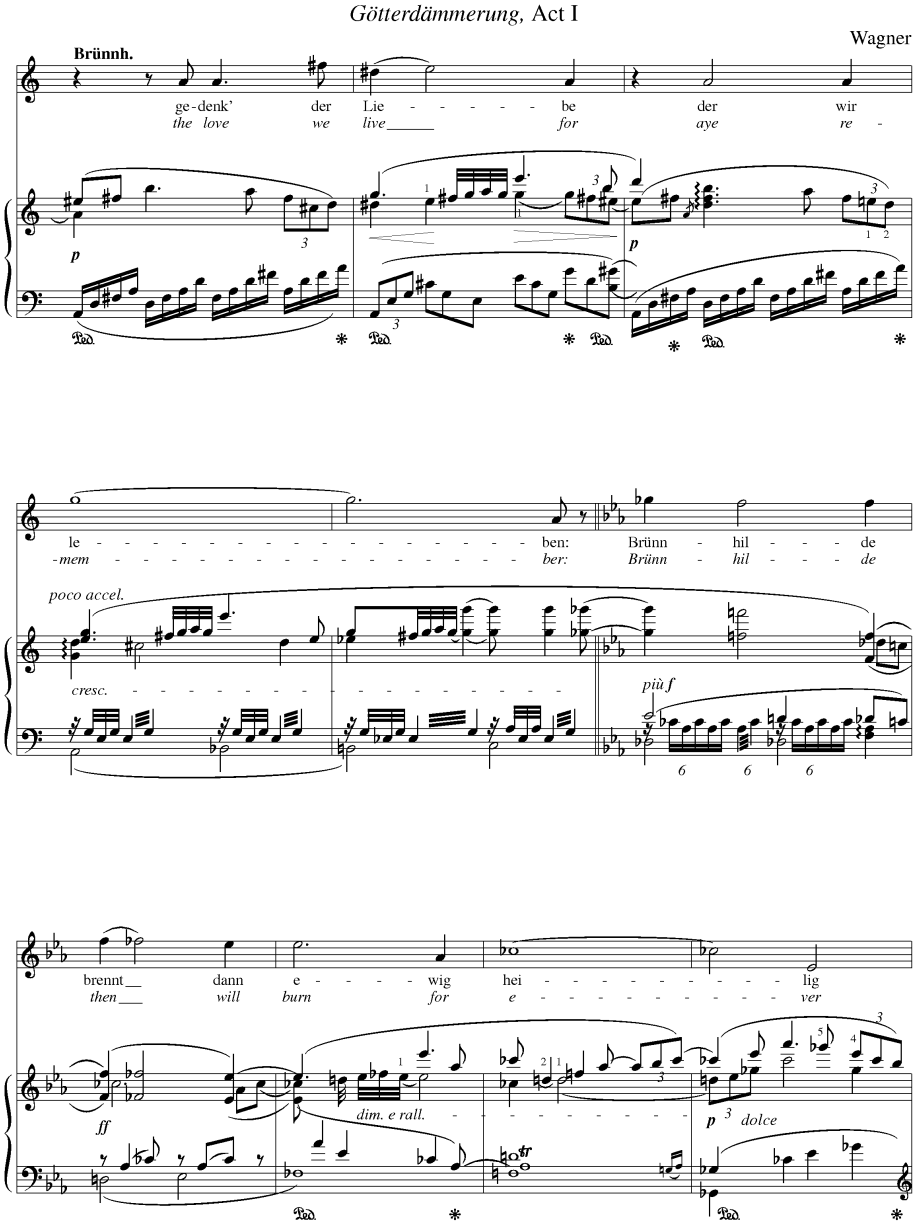


Figure 4.10: Original image

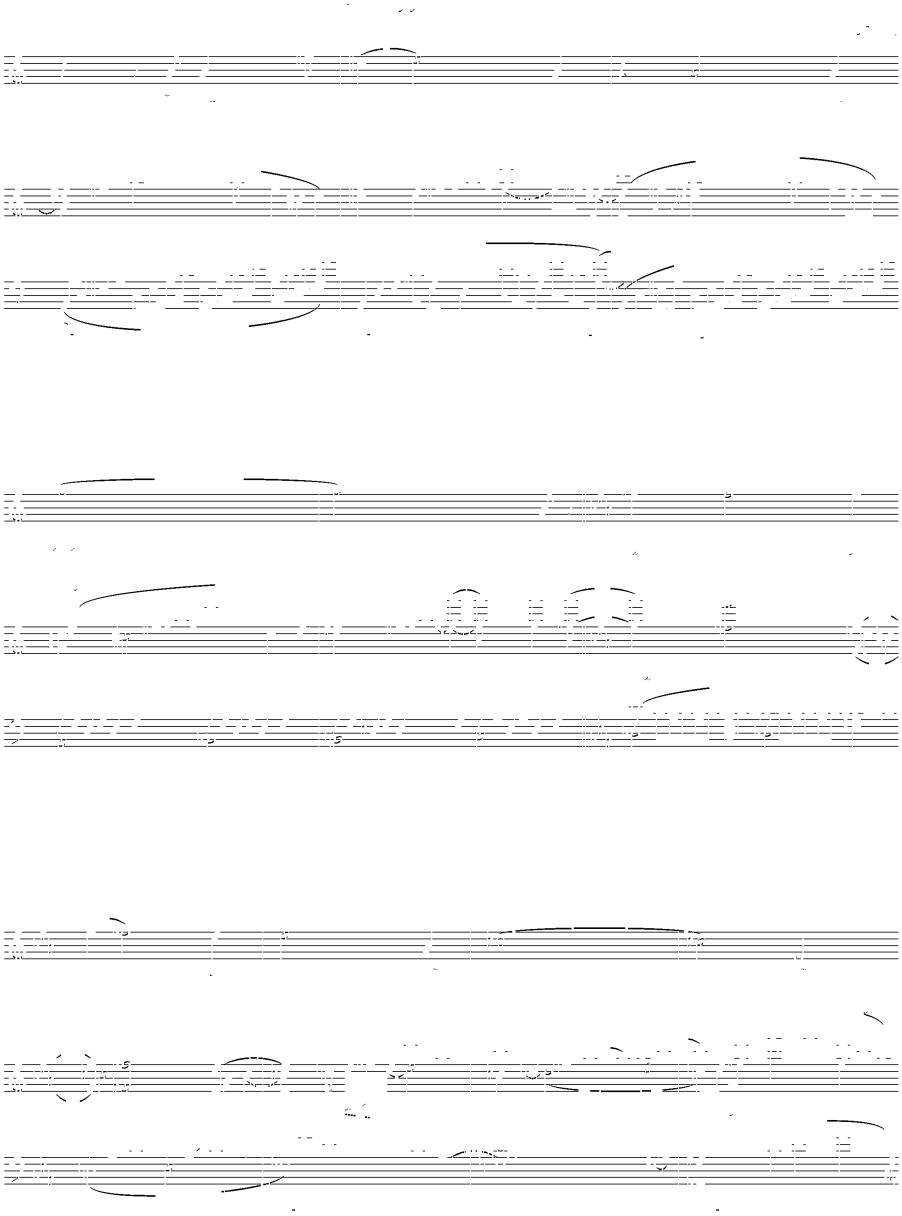


Figure 4.11: Symbols removed by my system

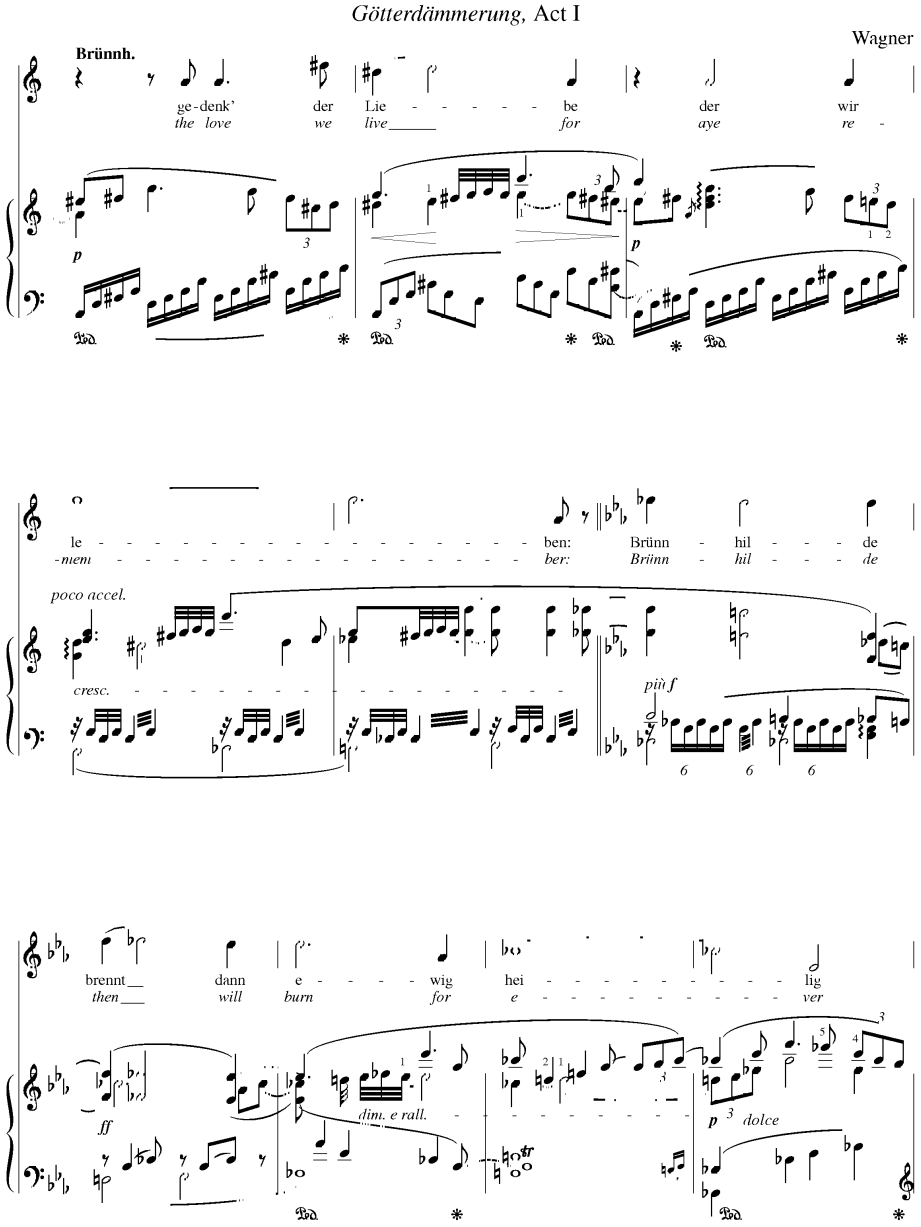


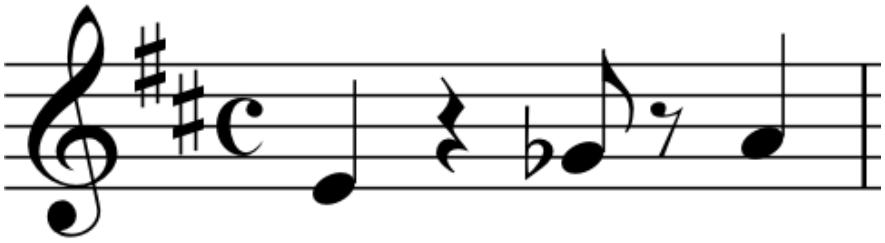
Figure 4.12: Staves removed by my system

	ANN (%)	**SVM (%)**
treble clef	86	96
common time	92	93
sharp	91	93
flat	84	94
natural	89	88
quaver	81	97
quaver upside down	85	93
crotchet	89	94
crotchet upside down	91	96
minim	86	94
minim upside down	78	86
semibreve	83	99
rest quaver rest crotchet	91 84	92 95
rest minim/semibreve	90	92
bar line	89	94

Figure 4.13: Accuracy obtained by the two classifiers on the testing symbol set

ground truth version. Unfortunately given the time constraints of a Part II project, I was unable to perform a thorough evaluation in this manner. Instead I will include some simple examples of outputs generated by my program. I also originally intended to perform a comparison between my system and other existing open source and commercial solutions. However given the complexities in implementing an OMR system, I was not able to get my system to accept inputs of a high enough complexity to make this interesting - the other systems performed extremely well on inputs my system wold struggle with. Therefore I decided to examine results qualitatively, as below.

It is easily discerned that approaching the problem of why errors occur is very difficult without having an appreciation of the system at a modular level .The error introduced by my system in 4.14 was in fact due to improper stave removal due to the proximity of the flat to the quaver.

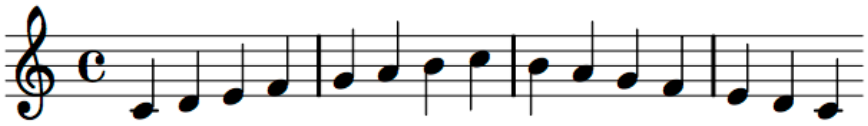


(a) input

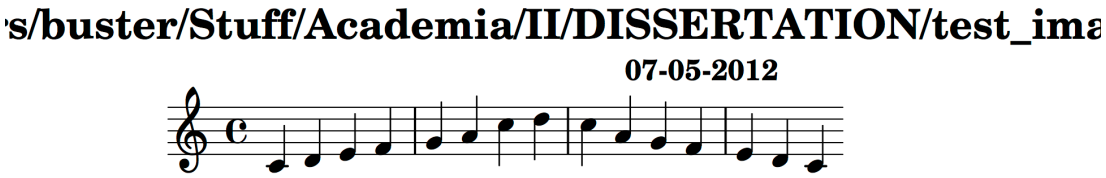
(b) output

Figure 4.14: SimpleOMR input and output

See B for an example of output LilyPond source code.



(a) input



(b) output

Figure 4.15: SimpleOMR input and output 2

Chapter 5

Conclusion

In the time available for this Part II project I have successfully implemented a working OMR system. The BasicOMR system works to a decent level of accuracy, as seen in the whole system evaluation chapter when presented with simple scores of low complexity. I was unable to complete the implementation iteration for ComplexOMR; although I devised a successful segmentation module which could decompose composite music symbols, the added complexity which would be required in the notation reconstruction module proved out of the scope of this project. Even the implementation of BasicOMR required considerable effort, especially given the lack of good lightweight 2D imaging libraries in Java.

The evaluation shows that the new score metric calculation algorithm is more robust against higher levels of salt and pepper noise. An interesting extension would be to extend this idea to looking at sums of three or more runs to see if the robustness to salt and pepper noise increases even further.

The stave line removal results were competitive with older methods in the literature. I was able to particularly thoroughly evaluate this section due to the excellent ground truth datasets available and complex evaluation algorithms specifically tailored for stave removal algorithm evaluation.

Whilst evaluating my classifiers, and in practice, the SVM implementation consistently outperformed ANNs, consistent with results in the literature. An interesting idea proposed recently in the literature is the combining of multiple classifiers to increase accuracy [28]; this is one I like to explore.

If I were to continue to work on this project I would complete the implementation of ComplexOMR, enabling a much broader range of inputs.

An interactive graphical user interface featuring a live display of the OMR system's progress would be a useful and interesting tool.

If I were to start the project again from scratch, the main difference would be a different choice of implementation language. I chose Java mainly due to my familiarity with it, however due to its image processing limitations I ended up having to implement quite a lot from scratch. I began to really appreciate MatLabs strong image processing functionality based on matrix manipulations - although less portable, MatLab would have been an ideal developing environment for this project.

Bibliography

[1]

[2] http://en.wikibooks.org/wiki/artificial_neurał$_n$*etworks/activation_functions*.

[3] http://en.wikipedia.org/.

[4] http://gamera.informatik.hsnr.de/.

[5] http://gamera.informatik.hsnr.de/addons/musicstaves/index.html.

[6] http://imslp.org/wiki/.

[7] http://lilypond.org/.

[8] http://neuroph.sourceforge.net/.

[9] http://sourceforge.net/projects/joone/.

[10] http://stackoverflow.com/questions/3403826/how-to-dynamically-compute-a-list-of-colors.

[11] http://vc.lib.harvard.edu/vc/deliver/home?_collection=scores.

[12] http://www0.cpdl.org/wiki/index.php/help:what%27s_it_all_about%3f.

[13] http://www.cpdl.org/.

[14] http://www.csie.ntu.edu.tw/ cjlin/libsvm/.

[15] http://www.heatonresearch.com/encog.

[16] http://www.informatics.indiana.edu/donbyrd/interestingmusicnotation.html.

[17] http://www.javapractices.com/home/homeaction.do.

[18] http://www.jfree.org/jfreechart/.

[19] http://www.neuratron.com/photoscore.htm.

[20] http://www.rism.info/.

[21] www.ehow.com/facts_5011392_historycomputer-scanners.html.

[22] *Identification of Music Symbols for Optical Music Recognition and On-Screen Presentation*, San Jose, California, USA, 2011.

[23] Carlos Guedes Andre R. S. Marcal Ana Rebelo, Filipe Paszkiewicz and Jamie S. Cardoso. Optical music recognition - state-of-the-art and open issues for handwritten music scores.

[24] Filipe Paszkiewicz Andre R. S. Marcal Carlos Guedes Ana Rebelo, Ichiro Fujinaga and Jaime S. Cardoso. Optical music recognition: state-of-the-art and open issues. 2012.

[25] David Bainbridge and Tim Bell. The challenge of optical music recognition. pages 35: 95 – 121, 2001.

[26] P. Bellini, I. Bruno, and P. Nesi. Optical music sheet segmentation. *Web Delivering of Music, International Conference on*, 0:0183, 2001.

[27] Pierfrancesco Bellini, Ivan Bruno, and Paolo Nesi. Assessing optical music recognition tools. *Comput. Music J.*, 31(1):68–93, March 2007.

[28] Donald Byrd and M. Schindele. Prospects for Improving OMR with Multiple Recognizers. pages 41–46, Victoria, Canada, October 2006. University of Victoria.

[29] Jaime S Cardoso and Ana Rebelo. Robust staffline thickness and distance estimation in binary and gray-level music scores. *2010 20th International Conference on Pattern Recognition*, 0:1856–1859, 2010.

[30] Chih-Chung Chang Chih-Wei Hsu and Chih-Jen Lin. A practical guide to support vector classification. 2010.

[31] Bastian Pranzas Christoph Dalitz, Michael Droettboom and Ichiro Fujinaga. A comparative study of staff removal algorithms. *IEEE TPAMI*, 2008.

[32] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995. 10.1007/BF00994018.

[33] Arnaud F. Desaedeleer. Reading sheet music. Master's thesis, Imperial College London Technology and Medicine (University of London) Department of Computing, 2006.

[34] Michael Droettboom and Ichiro Fujinaga. Symbol-level groundtruthing environment for omr. In *International Conference on Music Information Retrieval*, 2004.

[35] Fornes Llados Dutta, Pal. An efficient staff removal approach from printed musical documents. In *International Conference on Pattern Recognition*, 2010.

[36] Alicia Fornés, Anjan Dutta, Albert Gordo, and Josep Lladós. Cvc-muscima: a ground truth of handwritten music score images for writer identification and staff removal. *International Journal on Document Analysis and Recognition*, pages 1–9. 10.1007/s10032-011-0168-2.

[37] Ichiro. Fujinaga. Adaptive optical music recognition. 1996.

[38] Ichiro Fujinaga. Staff detection and removal. In Susan George, editor, *Visual Perception of Music Notation: On-Line and Off-Line Recognition*, pages 1–39. Idea Group Inc., 2004.

[39] Anssi Klapuri and Manuel Davy. *Signal Processing Methods for Music Transcription*. Springer, 2006.

[40] Song Mao and Tapas Kanungo. Empirical performance evaluation methodology and its application to page segmentation algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:242–256, 2001.

[41] A. Rebelo, G. Capela, and Jaime S. Cardoso. Optical recognition of music symbols: A comparative study. *Int. J. Doc. Anal. Recognit.*, 13(1):19–31, March 2010.

[42] Jenn Riley and Ichiro Fujinaga. Recommended best practices for digital image capture of musical scores. *OCLC Systems and Services*, 19, 2003.

[43] Florence Rossant and Isabelle Bloch. Optical music recognition based on a fuzzy modeling of symbol classes and music writing rules. In *ICIP (2)*, pages 538–541, 2005.

[44] Stockman Shapiro. *Computer Vision*. Prentice-Hall, Inc., 2001.

Appendix A

Two approaches to CMN

We can examine CMN from two perspectives:

1. the music theoretician's approach
2. the fool's approach

The music theoretician's approach - a beginner's guide to Music Theory

A music score is a document that totally describes a piece of music (whether it does totally describe a piece of music, or if this is possible only through the performance of the music and the score should be seen more as a rough guide, is a contentious debate in the field of Music as Performance, however for our purposes we assume this is true). It may be anything from a single page (perhaps a short piece of *Lieder*) to an entire book (an opera, symphony, anthology etc).

Common music notation is used to describe music written for almost all instruments used today. Variants of CMN can be used to describe drum music, synthesised music and even music to be played with a saw. This dissertation will be concerned with classic CMN, that is the kind used to describe non-experimental forms of e.g. violin, flute or voice music. The structure of a score can be hierarchically decomposed as so:

A **score** consists of a set of one or more **parts**, one for each instrument featuring in the piece it describes. Each part has between one and three **staves**, depending on the instrument (e.g. the violin has one, the harp has two, the organ has three). When a part comprises of multiple staves,

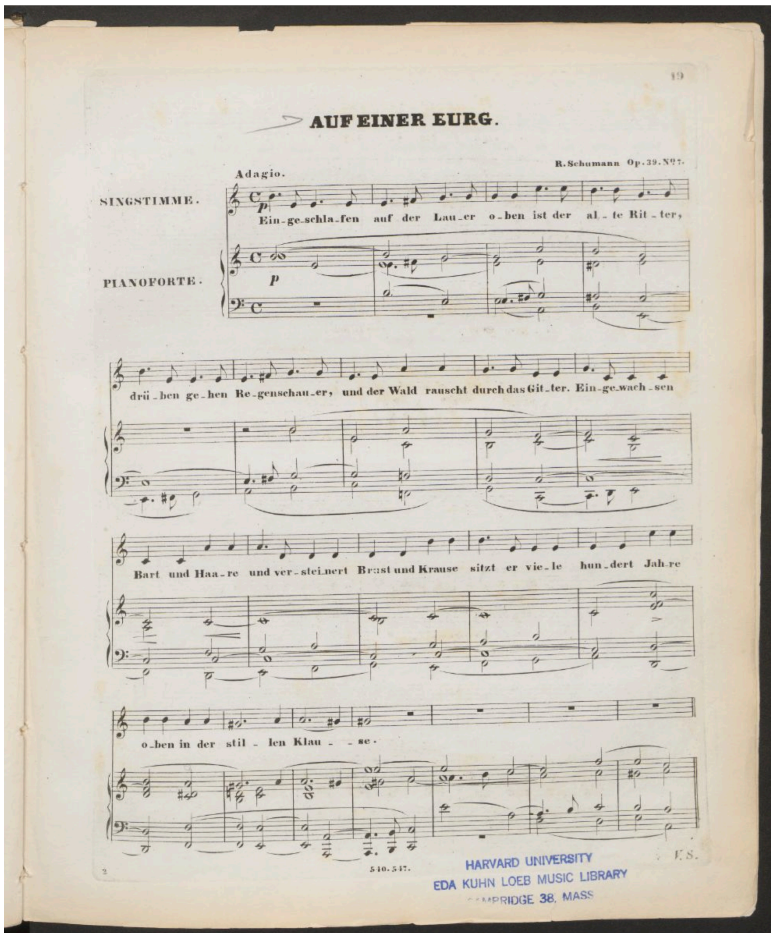


Figure A.1: A page of music - *Auf Einer Burg*, a gorgeous short song from Schumann's Op. 39, *Leiderkreis*. Taken from Harvard University's Loeb Music Library online catalogue [11]

they are grouped into a 'great stave'. The music is broken up into lines and distributed over pages. Lines are equivalent to single lines of text. Lines are read from top to bottom, left to right. Figure A.1 shows a score with two parts: the higher one for voice and the lower for piano. The piano part has two staves. The music is split into 4 lines on the page. Within a line, a particular location along the horizontal (x) axis corresponds to a moment in time. The vertical (y) axis, within each stave, corresponds to pitch.

The **stave** contains the information describing when to play which note (as in at what pitch) or combination of notes, how long to play it and how to play it. The **stave lines** present a series of reference y values on which symbols are placed. Each of the possible positions (either on a line or between two lines) represents a different relative pitch, the absolute value of which is calculated using the context of the stave's clef and key signature.

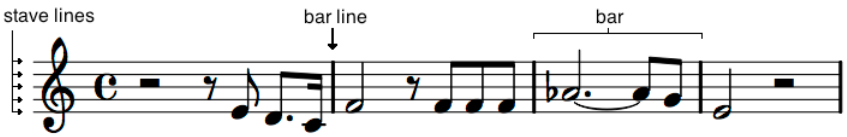


Figure A.2: A labelled stave. Extract taken from Schumann's *Ich Grolle Nicht* from *Dichterliebe*, typeset using LilyPond [7]

Due in part to the way the human ear evolved, the smallest pitch interval commonly used in Western tonal music is the semi-tone. This system of tuning is known as the twelve-tone equal temperament, as the octave is divided into 12 widths, called semitones. Although pitch may be quantified as a frequency, it's not a purely objective physical property; it's a subjective psycho-acoustical attribute of sound that allows the ordering of sounds on a frequency-related scale[39]. Our pitch sense is logarithmic with respect to frequency - the difference between each consecutive semitone sounds equal. An octave corresponds to a doubling of frequency, therefore the frequency ratio of the interval between two consecutive semitones is $\sqrt[12]{2}$. This interval is represented on the stave lines by adjacent lines and spaces.

The stave is further decomposed into **bars** (American: measures), delineated by **bar lines**. Bars are rhythmic units - each bar's duration (the sum of the rhythmic values of all the notes and rests in the bar) must equal that specified by the most recent time signature. Bars contain a list of symbols - notes and rests being the two main types, with accidentals and augmentation dots acting as local pitch and rhythm modifiers, and key signatures

and clefs as global pitch modifiers and time signatures as global rhythm modifiers.

Notes are made up of **note heads**, **stems** and **hooks** (sometimes called **flags**). The note head may be filled or hollow. The pitch of the note is defined by the position of its note head (and the context of the current clef). The presence of a stem indicates a maximum duration of a minim. Filled note heads must have stems. Each additional flag halves the note's existing value. Notes' durations may be augmented using augmentation dots (increases the duration by half). The following diagram illustrates some of these principles:

Notes may be stacked vertically to make **chords**, indicating the notes should be played at the same time. They may also be joined horizontally using **beams**. This technique is used for rhythmic clarity and to illustrate phrasing and in vocal music to illustrate which lyric matches which note(s). Beamed note groups and chords offer almost unlimited variation in symbols (see symbols in the latter example in figure 2.2) and therefore present a significant challenge to symbol classification attempts. Commonly such composite symbols are broken up into their component primitive symbols for classification.

Rests represent measures of time in which no notes are played. They do not have a pitch (although to distinguish between minim and semibreve rests, their y positions must be taken into account - a notational quirk). Quaver rests can be halved by the addition of a flag (and so on), and rests can also be augmented using dots.

Clefs set the pitch context of the stave. Every stave has a starting clef as it's first symbol. The clef can also be modified at any other point (obviously you can't have 2 consecutive clef symbols as there would be no point - clefs, like accidentals, are pitch modifiers and isolated don't encode any performance directions). The context of that clef continues until another clef is encountered.

Accidentals appear in two different contexts: in **key signatures**, and to modify the pitch of notes. Key signatures define the default 'setting' for each pitch (stave line or stave space) in the stave. Each stave has a starting key signature that appears after the clef symbol. The key signature can again be modified at any point elsewhere in a stave. An absence of an accidental in the key signature for a pitch implies a natural. Key signatures

can either consist of sharps or flats (not both), the ordering of which must follow a precise pattern:

(a) sharps (b) flats

Figure A.3: Key signatures [3]

When placed to the left of a note head, an accidental modifies that note's pitch. Accidentals are absolute, not relative - for example in F major you don't have to make a B natural to then make it sharp. We have already explored the major score hierarchy and symbols. We have so far ignored (so called in OMR research) horizontal symbols such as slurs, ties, tuplets and dynamic hairpins). Along with dynamics and performance directions, these indicate how series of notes should be played - at what volume, with what phrasing (which notes to emphasise, which notes to lengthen or shorten etc.). For the purpose of keeping the scope of this dissertation feasible, we will ignore such symbols (and explore their analysis as an extension).

A fool's (computer's) approach

Here we ask the question 'what does a score look like to someone with no knowledge of music theory?'. Although such an individual, with no notion of written music, would be rare today this is an important question when designing the lowest level image analysis in a computer vision application. Hopefully we can ascertain the kinds of basic structures that code for the notation so we can decide on what kinds of feature extraction algorithms might be useful.

The largest and most regular symbols on the page are the stave lines - long, ideally straight and horizontal, spanning most of the page. They appear in groups of five, making a stave. Other long horizontally oriented line symbols include phrase marks, slurs, ties and beams. Two important metrics when analysing a score are the average stave line height and the distance between consecutive stave lines (known as the stave space height). As a note head must fit almost exactly between two stave lines, these reference lengths

Key name	**Number of sharps**	**Sharpened notes**
C major	0	
G major	1	F♯
D major	2	F♯ , C♯
A major	3	F♯ , C♯ , G♯
E major	4	F♯ , C♯ , G♯ , D♯
B major	5	F♯ , C♯ , G♯ , D♯ , A♯
F♯ major	6	F♯ , C♯ , G♯ , D♯ , A♯ , E♯
C♯ major	7	F♯ , C♯ , G♯ , D♯ , A♯ , E♯ , B♯

(a) Sharp key signatures

Key name	**Number of flats**	**Flattened notes**
C major	0	
F major	1	B♭
B♭ major	2	B♭, E♭
E♭ major	3	B♭, E♭, A♭
A♭ major	4	B♭, E♭, A♭, D♭
D♭ major	5	B♭, E♭, A♭, D♭, G♭
G♭ major	6	B♭, E♭, A♭, D♭, G♭, C♭
C♭ major	7	B♭, E♭, A♭, D♭, G♭, C♭, F♭

(b) Flat key signatures

Figure A.4: The order in which accidentals are displayed in the key signature. Note that both patterns are the reverse of the other - due to the circle of fifths

allow the system to analyse music fonts of any size, and adapt the system for each specific input score.

Figure A.5: A section of stave, illustrating the two metrics 'stave (staff) line height' and 'stave (staff) space height' [23]

Vertical lines usually indicate note stems or bar lines. Filled note heads are distinct as the largest 'blob-like' symbols in CMN.

Appendix B

Example music score description format output: LilyPond source code

```
%{
LilyPond code generated by University of
 Cambridge Part II Project on OMR by Jack Lawrence-Jones.
%}

\version "2.15.36"

% define variable to hold the date and time:
date = #(strftime "%d-%m-%Y" (localtime (current-time)))

\header{
title = "/Users/buster/Stuff/Academia/II/DISSERTATION
/test_images/sibelius_opus_simple_stave.png"
subtitle = \date
}

\score {
{
% contains whole compound music expressions
\new Staff {

\clef treble
```

```
\key d \major

\time 4/4
r4 f'4 r8 a'4
}

}
% end of whole compound music expression

\midi {
\context {
\Score
tempoWholesPerMinute = #(ly:make-moment 72 2) }
}


}
```

Appendix C

Project Proposal

Part II Computer Science Project Proposal

Optical Music Recognition

J. Lawrence-Jones, Downing College

Originator: J. Lawrence-Jones

21 October 2011

Project Supervisor: Dr C. Town

Director of Studies: Dr R. Harle

Project Overseers: Dr M. Kuhn & Dr I. Leslie

1. Introduction

The musical score is the medium through which the composer conveys his artistic vision to the performer. Musical notation syntax has been evolving ever since the first written music, the oldest example of which is thought to be a 4000 year old cuneiform tablet found in Nippur, Iraq (*A. D. Kilmer and M. Civil, 1986*). Methods used today vary depending on culture, genre and artistic preference. The main syntactic style used in the West today is "Modern Common-Practice Western Music Notation", which I will abbreviate to WMN (Western Music Notation).

Optical Music Recognition (OMR) is the process of automatically analysing and understanding an image of a music score. The aim is to create a file representing the score that can be stored, shared, reprinted, edited (by notation software such as Sibelius) and even played automatically. A type of Optical Character Recognition (OCR) problem, it is still an active area of research as there are currently no complete satisfactory solutions.

Sheet music by its very nature isn't a particularly permanent method of preserving some of the greatest pieces of art humanity has ever conceived - the sheer size of the body of works available means its digitalisation and thereby preservation necessitates OMR. It also allows the dissemination of the music through online libraries, allowing greater exposure to lesser-known works and free access to those who wouldn't have bought the physical scores in the first place (which are often very expensive).

The aim of this project is to design and implement an OMR system - given an image of a printed music score in WMN, it will automatically create a machine-based representation, in the form of an open source music score format (so it can fulfill the criteria specified above). Although WMN is more or less standardised, there is still much variation in the symbols and syntax used in scores printed by different publishers and in different countries. Handwritten scores present even greater variation in styles. My system should hopefully be able to interpret the most popular score engraving styles currently used, and perhaps also some historic ones too. As an extension I will look into, among other things, analysing handwritten scores.

2. Starting Point

I have no knowledge of computer vision principles. The main OMR and OCR libraries are written in Java, which we have used throughout the Tripos so far. My knowledge of AI techniques is limited to the Part 1B course. I also have very limited experience with MATLAB, which I use to train my Neural Networks and Support Vector Machines. I will be typesetting my project using LaTeX, which I've never used before. I may also use a scripting language to tie various parts of my project together - I have some previous experience with Perl that could prove useful.

3. Project Details

In the literature, OMR systems are usually split up into four distinct subunits. I will use this architecture as a basic structure for my project:

1. Preprocessing

This stage involves the removal of noise, de-skewing and image enhancement. In order to limit the scope of this project to make it a viable Part II project, I will begin by assuming this part of the OMR system has already been successfully solved - I will supply noise-free, un-skewed inputs to the rest of my system, or if necessary use the functions supplied for these purposes by OpenOMR, Audiveris and OCRopus (these are the main open source OCR and OMR libraries used today). Once I have completed the core implementation of this project, I may look into implementing this module myself.

2. Music symbol recognition

This stage accomplishes the main OCR work for the system. It comprises of three subsections: stave (the five horizontal lines, spanning the width of the page of music, which form a basic framework on which the notes and other symbols are placed) detection, symbol segmentation and the symbol classifier.

3. Musical notation reconstruction

In this unit we combine symbol primitives in order to reconstruct a logical description of the score's information, in accordance with relevant graphical and syntactic rules. A knowledge base will be required to guide this process. I hope to also implement a feedback loop to the symbol recognition stage to increase error detection accuracy, as discussed in the literature.

4. Final representation reconstruction

Here we transform the information interpreted from the score into a format readable by common score editing programs (likely MusicXML, an open source format suitable for this task).

I intend to use a spiral software development model when implementing my project. To begin with I will build a system that uses existing library functions (from OCRopus, OpenOMR and Audiveris) to achieve each stage. I will then reimplement the system myself unit by unit in Java, allowing me to iteratively test each stage and the system as a whole simultaneously.

A focus will be placed on the symbol classifier and musical notation reconstruction units, whose functionalities I will further explore. I will implement, compare and evaluate two techniques commonly used in the literature for symbol classification - Neural Networks and Support Vector Machines (trained using MATLAB). This will require the data collection of music symbols in order to construct training and test sets.

I also hope to use prior musical knowledge in the form of a knowledge base to reduce symbol classification error rates (the extent to which I can formalise WMN rules will be explored). This will involve some sort of feedback loop between the symbol recognition and notation reconstruction units, the nature and efficacy of which I will explore.

The development of a full score OMR package is outside of the scope of this project, therefore the range of inputs my system will be required to handle will be carefully restricted. I will not be recognising text such as performance directions and lyrics in the scores. Also the complexity of the music itself will be limited - I will start off by analysing only single voice

parts (i.e. pieces with one stave per line of music) in WMN (ignoring the works of certain Romantic composers who often broke WMN rules for the sake of clarity and musical expression) and allowing only simple chords. The range of symbols allowed will be specified in the write-up. Although these restrictions seem to dramatically limit the range of analysable music, the system should still be able to interpret most single stave instruments' scores (such as violin, oboe, trumpet, voice), and could be extended to successfully interpret multiple voice scores without too much hassle.

4. Possible Extensions

I have considered several possible extensions to my project.

The core of my project will be a system to analyse printed scores. Broadening the range of possible inputs to handwritten scores presents a much harder challenge as it increases the quantity of errors and ambiguity the system will have to deal with. Composers often find handwritten scores a much more natural medium to work with, yet the beauty of using digital tools such as Sibelius is that you can easily make changes and also get instant feedback on how your composition sounds (even with a piece for a full Symphony Orchestra, for example, which would otherwise be particularly inconvenient to iteratively perform). Being able to analyse handwritten scores allows the best of both worlds, and it will be interesting to see how effective my system intended for printed scores is on them. However again I must be careful with the scope of this extension - I will only attempt to analyse handwritten music with printed staves (as handwritten ones introduce a whole new layer of ambiguity).

It would also be of great interest to me to be look at how and where my system would have to be adapted to successfully deal with different notation methods - for example Gregorian plainchant, the syntax of which differs significantly from that of WMN. However this could constitute a Part II Project in it's own right, so my investigations will be limited.

A technique used in the literature to formalise the prior musical knowledge base is developing a grammar (possible as music notation is a context free grammar (*Fujinaga 1988*)). I would like to investigate how one might go about doing so, and perhaps construct a very restricted one and try to incorporate it usefully into my system.

5. Success Criteria

I will consider my project a success when a working OMR system has been implemented - given an image of a page of printed sheet music (subject to the constraints discussed above), the system will be able to translate it, within reasonable time, into an appropriate format to be used by popular music notation packages, to a level of accuracy of the same order of magnitude as those commonly quoted in the literature (in practice this implies a per character recognition accuracy of about 90%). This will involve at the very least the completion of a music symbol recogniser, a musical notation reconstruction unit, and a final representation reconstruction unit.

I will evaluate my project at two levels - comparing firstly the two different types of symbol classifiers I will implement and secondly each of my own units (and my system as a whole) against the libraries' implementations (mainly OpenOMR and Audiveris). I will compare speed, complexity and success rates of the different solutions.

Much of the success of this project lies in the generation of appropriate training and test sets. I will create synthetic scores with quantifiable variations in common error sources such as broken symbols and symbol clustering.

6. Work Plan

Starting on 21st October, in 2 week time intervals:

1. **(21 Oct - 3 Nov)** Preliminary preparation
   - Review the literature available on existing OMR methods and computer vision basics
   - Acclimatise myself with LaTeXand MATLAB
   - Setup SVN
2. **(4 Nov - 17 Nov)**
   - Assemble a basic working system using the libraries
   - Draft dissertation: Introduction and Preparation
3. **(18 Nov - 1 Dec)**
   - Investigate neural network techniques and assemble testing and training symbol sets

  - Implement my own music symbol recognition module, using Neural Networks

4. **(2 Dec - 16 Dec)**
   - Implement my own music symbol recognition module, using Support Vector Machines
5. **(17 Dec - 30 Dec)**
   - Slack time for unseen delays etc.
6. **(2 Jan - 15 Jan)**
   - Investigate open digital notation formats (LilyPond, MusicXML etc.)
   - Implement my own musical notation reconstruction module
7. **(16 Jan - 29 Jan)**
   - Continue to implement my own musical notation reconstruction module
8. **(30 Jan - 12 Feb)**
   - Draft dissertation: Implementation
   - Progress Report Deadline: Fri 3 Feb
   - Progress Report Presentations: 9th - 14th Feb
9. **(13 Feb - 26 Feb)**
   - Optimisations
10. **(27 Feb - 11 Mar)**
    - Quantitative analysis of my system
    - Draft dissertation: Evaluation and Conclusion
11. **(12 Mar - 25 Mar)**
    - Extensions
12. **(26 Mar - 8 April)**
    - Draft dissertation: Appendices
    - Review, Finalise and Format Dissertation
13. **(9th April -)**
    - Slack time for extensions, unforeseen delays etc.
    - Proofread dissertation

[Dissertation Deadline (paper copies): Fri 18 May (12 noon)]

[Dissertation and Source Code Deadline (electronic copies): Fri 18 May (5pm)]