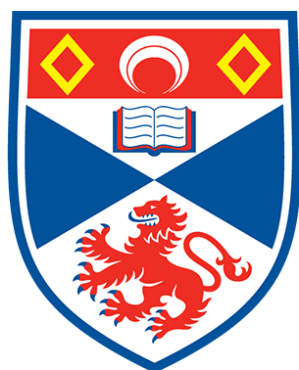


Automatic Reduction of Orchestral to Piano Scores

Joshua Wood



University of
St Andrews

Supervisor: Ishbel Duncan

Date of Submission: 30 April 2020

Abstract

Piano reductions of orchestral works are invaluable for analysis and rehearsal performances, but producing them is a time consuming task that is usually done manually. This report describes different techniques for automating this process and describes an implementation of these techniques.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 12,930 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. I retain the copyright in this work, and ownership of any resulting intellectual property.

Contents

Abstract	2
Declaration	3
Contents	4
1. Introduction	6
2. Context Survey	7
2.1. Elements of a Musical Score	7
2.2 Similar Works	13
3. Requirements Specification	15
3.1. Primary Objectives	15
3.2. Secondary Objectives	15
3.3. Tertiary Objectives	16
4. Software Engineering Process	16
4.1. Tools Used	16
5. Design	17
6. Implementation	18
6.1. Keeping Time	18
6.2. MusicXML Parser	19
6.3. Internal Representation of the Score	22
6.4. MusicXml Serialisation	25
7. Transformations	29
7.1. Merge By Average	29
7.2. Distribute Staves	30
7.3. Adjust Octaves	30
7.4. Merge	32
8. Evaluation and Critical Appraisal	32
8.1 Evaluation Against Requirements Specification	32
8.1.1. Primary Objectives	32
8.1.2 Secondary Objectives	33
8.1.3. Tertiary Objectives	33
8.2. Evaluation Against Musical Works	34
8.2.1 Bruckner - Aequale No. 1	34
8.2.2. Sullivan - Overture to Patience	36

9. Conclusions and Futures	40
Appendices	41
A. Testing Summary	41
B. User Manual	44
References	44

1. Introduction

An orchestral score, also known as a full score or conductor's score, is a type of sheet music which shows the music of all instruments in the ensemble, allowing the conductor to accurately see what each instrument is playing at all times. For rehearsals with the ensemble, this representation is necessary in order for the conductor to direct individual players and cue their entrances. However, in many situations, this level of detail is not appropriate. For works in which the orchestral music is not the only focus, such as opera, ballet, and musical theatre, rehearsals with the other performers tend to be accompanied by a single pianist, who plays a piano version of the music. This has numerous advantages; having a single pianist is cheaper, takes up less space, and is easier to direct. However, in order for this to be achieved, the pianist must have a version of the music arranged from the full orchestral score, known as a piano reduction.

Creating a piano reduction is typically a time consuming manual process. A good reduction contains the cues necessary for the performers to follow along, while also being physically playable by the pianist. Within a typical orchestral score, not every part is completely unique. Parts are often doubled in unison, or in octaves, and often these can be omitted in the reduction. While the orchestra can play simultaneously across the full range of the piano, the pianist normally plays at most five notes across approximately an octave in each hand. So inevitably, for most scores some notes must be transposed, or removed.

In section 2 I discuss the tools available currently to aid in the making of orchestral reductions, and the research that has already been done in the area. I also provide explanations of some of the music theory concepts necessary to understand the paper. In section 3 I lay out the requirements that the software was built against that I chose at the start of the project. In section 4 I explain the processes that I used to build the software, as well as the tools I used to help me. In section 5 I discuss the design of the project, and explain some of the difficulties that the design allows me to overcome. In section 6, I explain how each section of the software is implemented, and in section 7 I explain how each transformation of the score is implemented. In section 8 I evaluate the finished project against the original requirements, as well as running it on some musical works and evaluating how it performs in contrast to a handmade reduction. In section 9 I sum up the work done, and present possible extensions and improvements that could be made to the software in the future.


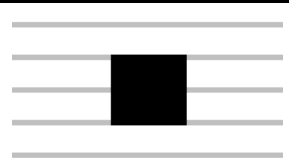

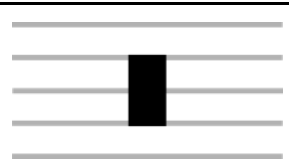
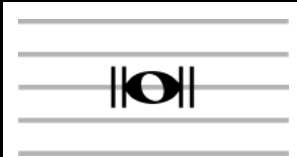
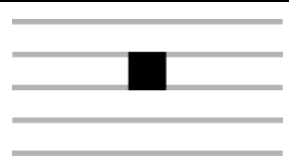
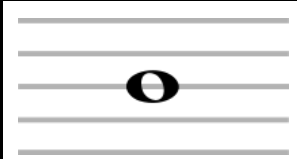
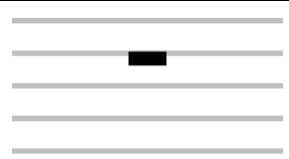

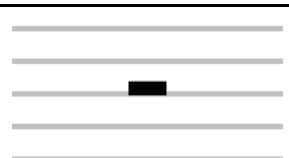
2. Context Survey

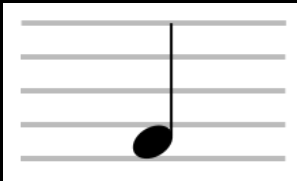
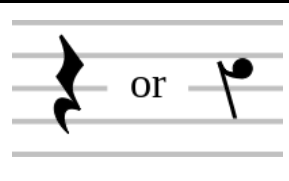


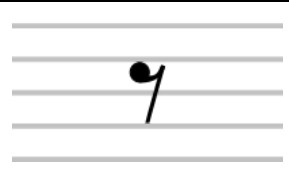







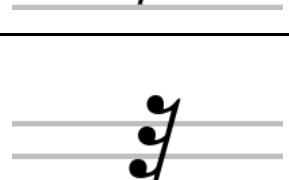


2.1. Elements of a Musical Score

A musical score represents a timeline of notes, indicating the location, pitch, and duration of each note, along with which instrument they are played by. A score is made up of a vertical list of 5 line staves, with each staff being played by an instrument, or section of instruments.

Note durations are specified by the type of note, as shown in table 1. Confusingly, there are different naming systems for note types in the UK and US. MusicXML uses the US standard, whereas I am more familiar with the UK standard. Each note type is half the duration of its predecessor, so a long is half the length of a maxima.

Table 1: Note types. Images adapted from https://en.wikipedia.org/wiki/List_of_musical_symbols#Notes_and_rests

Note symbol	Rest Symbol	UK Note Name	US Note Name
		Maxima	Maxima
		Long	Long
		Breve	Breve
		Semibreve	Whole note
		Minim	Half note

	 or 	Crotchet	Quarter note
		Quaver	Eighth note
		Semiquaver	16th note
		Demisemiquaver	32nd note
		Hemidemisemi- quaver	64th note
		Semihemidemi- semiquaver	128th note
		Demisemihemi- demisemiquaver	256th note

The curves on the top of notes from a quaver or smaller are known as flags. If multiple notes with flags are placed consecutively, they can be beamed together by connecting the common flags with a line. Notes and rests can be followed with a dot, which increases their length by half. Multiple dots may be added, where each in turn increases the

duration by half, although this is less common. Other note lengths are defined by tying notes together, which means the note sounds once, but lasts for the duration of both notes. An example is in Figure 1.



Figure 1: Tied notes. The first group has a duration on 7 semiquavers (4 from the crotchet + 3 from the dotted quaver), and the second has a duration of 9 semiquavers (1 from the semiquaver + 8 from the minim)





A sequence of note types and rests defines a rhythm. A rhythm is generally split up into beats, which are defined by the time signature. A beat is the unit that musicians use to count in, and offset notes in a rhythm against. The time signature of the piece defines what the beats are, and how they are counted. A time signature is represented similar to a fraction, with a numerator (the number of beats) and a denominator (the beat type). The denominator is a power of two¹, where 2 (the smallest commonly used number) represents a minim, 4 represents a crotchet, etc. The upper number represents the number of these beats in a bar. So 4/4 means four crotchet beats in a bar, 3/4 means three crotchet beats in a bar, and 6/8 represents six quaver beats in a bar. You may notice that 3/4 and 6/8 add up to the same duration. The difference between these two time signatures is in the way the beats are grouped. In 3/4 the beat is counted as 3 groups of one crotchet, as in a waltz, but 6/8 is counted as 2 groups of three quavers, with a strong beat and a weak beat within this. A good example of this is *We Are The Champions* by Queen, in which throughout you can count a steady *1 and a 2 and a*, with the 1 and 2 representing the strong and weak beats respectively. Bars are delimited by a vertical line known as a barline, and bars are counted using bar numbers, which are stated at the beginning of each line (except the first), and counting starts at 1. Time signatures are stated at the first bar of the piece, and only again if they change within the piece.

The pitch of a note is specified by its position on the five line stave. A symbol called a clef indicates a baseline that defines where the cycle of notes begins. There are three types of Clef symbol, G, C, and F, and four commonly used clefs using those three symbols, Treble, Alto, Tenor, and Bass. The letter of the clef indicates the note name the clef defines, and convention dictates the octave it sounds in. For a piano score, a treble clef is typically used for the right hand and a bass clef for the left hand, although this can change if the register of the piece is particularly high or low. Alto clef is used by viola and alto trombone, and tenor clef is sometimes used by tenor trombone, and by cello and bassoon when playing

¹ Not technically always true, but true for all conventional music. For more information research irrational time signatures.

in their upper registers. Table 2 shows the commonly used clefs. Clefs are stated at the start of every stave.

Table 2: Common clefs. Images taken from <https://en.wikipedia.org/wiki/Clef>

Symbol and Note	Clef Name
	Treble Clef
	Bass Clef
	Alto Clef
	Tenor Clef

Now that the base note has been defined, we can work out what every note on the stave represents. Notes always cycle from A - G, in ascending order, alternating between a space and a line on the stave. Figure 2 shows this for treble, alto, and bass clef.

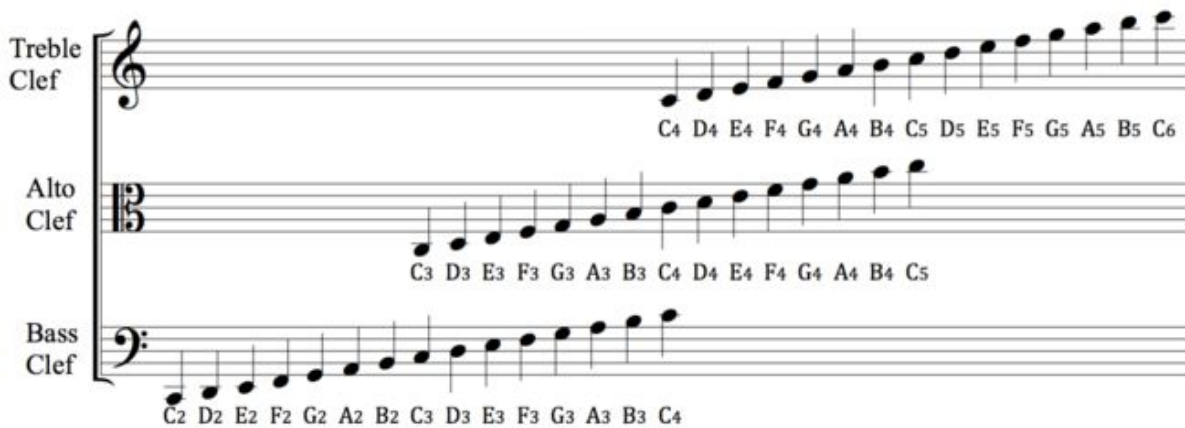






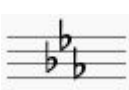

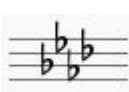

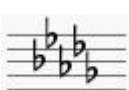
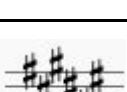
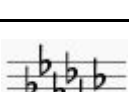
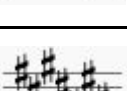
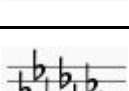


Figure 2: The note names in different clefs. Image taken from <https://en.wikipedia.org/wiki/Clef>

The distance between the two nearest notes with the same name, for example from C₂ to C₃, is called an octave. An octave is split into 12 semitones. Most unaltered note names have two semitones between them, although E-F and B-C are only one semitone. Notes between are defined with an alteration. There are two main types of alteration, a sharp (#), which raises the written note by a semitone, and flat (♭), which lowers the note by a semitone. Thus, the note between a C and a D, can be written as either a C# or a D♭, and the two spellings of this note are said to be *enharmonically equivalent*. As the gap between an E and an F is only one semitone, an F♭ is enharmonically equivalent to an E. Other types of alterations exist, such as a double sharp (x), which raises a note by two semitones (equal to a tone), and double flat (♭♭), which lowers a note by a tone. When these alterations are applied to a note on the stave, they are called an accidental. They apply to the note, and all subsequent notes of the same pitch in the same register for the remainder of the bar. Alterations can be cancelled with the natural (♮) accidental.

A piece of music is often said to be in a key, which is defined by the key signature. This is represented as a collection of either sharps or flats at the beginning of each stave. Each sharp or flat shows that every note with the pitch of the symbol is played as a sharp or a flat. For example, a key signature of a sharp on the top line of a treble clef stave means that all Fs are played as sharps. Sharps and flats are always added to a key signature in a specific order. For sharps this is F#, C#, G#, D#, A#, E#, B#, and for flats this is B♭, E♭, A♭, D♭, G♭, C♭, F♭. This means that a key signature with three sharps will always have an F#, a C#, and a G#. As the name suggests, different key signatures represent a different key that the work is based around. This key could either be major or minor, depending on the piece. The minor version is called the relative minor, and the pitch of the minor version is always a tone + a semitone below the major. The names of different keys are shown in Table 3.

Table 3: The different possible musical keys and their key signatures. Key signature images from MuseScore

Key Signature	Key	Key Signature	Key
	C major A minor	-	-
	G major E minor		F major D minor
	D major B minor		Bb major G minor
	A major F# minor		Eb major C minor
	E major C# minor		Ab major F minor
	B major G# minor		Db major Bb minor
	F# major D# minor		Gb major Eb minor
	C# major A# minor		Cb major Ab minor

Key signatures appear at the beginning of each stave, or wherever they are changed within the piece. They are used to reduce the number of accidentals within a piece.

To make things more complicated, some orchestral instruments are known as *transposing instruments*. This means that the notes they play are written the same way as for other instruments, but they sound at a different pitch, determined by a constant offset from the written pitch defined by the key of the instrument. An instrument with no transposition is said to be in “C”. For example, a clarinet in Bb sounds one tone lower than the notes that are written, because a Bb is one tone lower than a C. This instrument transposition also means that the key signature is transposed by the same amount. So if the piece is in the key of G major, then the clarinet in Bb part is written with a key signature of A major, so that the written A sounds as a G.

2.2 Similar Works

There has been a limited amount of research into automating the reduction of orchestral scores, and a number of existing solutions are available, of varying quality.

The simplest, although perhaps most convenient solutions are the ones built into music notation software. A common function is a *reduce* or *implode* function that takes music on several staves and reduces them to a fewer number of staves. This tends to be a fairly literal reduction, often simply placing the notes of each staff into a new voice.

The version of this in MuseScore is the implode tool[1]. This tool is very basic, as it simply copies the contents of each staff that is being imploded into a new voice on the target staff. For homophonic rhythms, there is no attempt to merge notes into chords, and even unison notes are shown twice. This results in quite a convoluted score when there are more than two staves being reduced. An example is shown in Figure 3, where a homophonic passage, and then a polyphonic passage is imploded to a reduction staff. The ColorVoices plugin is used to show the colours of the voices in the reduction.



Figure 3: MuseScore implode tool. Tested using MuseScore version 3.4.2.9788

It is easy to see that for a complex orchestral score with tens of staves this would very quickly become unmanageable, and if you are using MuseScore for this purpose it may be easier to do the entire reduction manually.

Sibelius offers a similar function, called reduce[2]. This is similar to MuseScore in that polyphonic passages are simply copied into different voices. However, homophonic passages are condensed into a single voice with chords, which is an improvement over

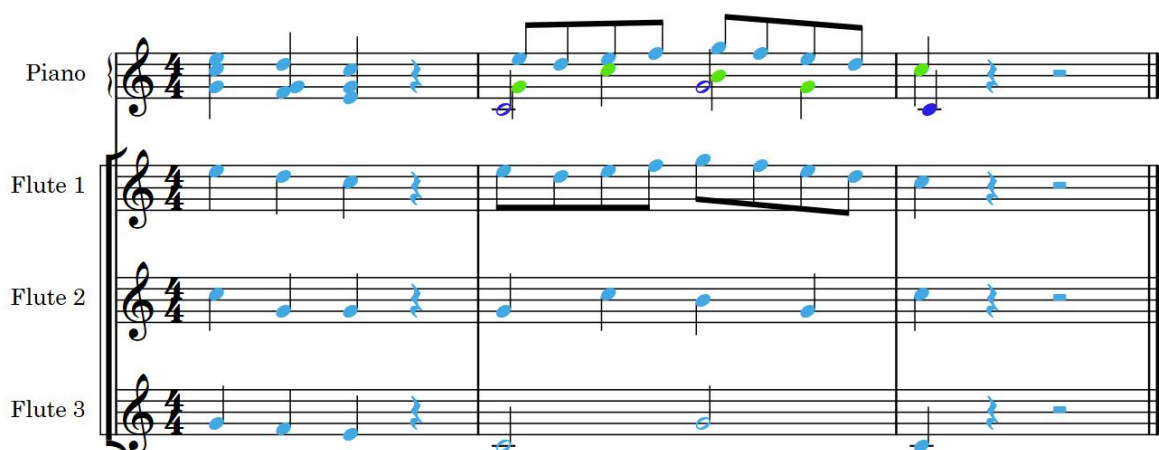
MuseScore. The results of the same experiment can be seen in Figure 4.



The image shows a musical score for Piano and three Flutes. The Piano part is in the top staff, and the three Flutes are in the bottom three staves. The score is in 4/4 time. The Piano part has a complex melody with many notes, while the Flutes have simpler, more homophonic parts. The score is a single system with four staves.

Figure 4: Sibelius reduce tool. Tested using Sibelius 7.5.

Dorico also offers a reduce function[3], and the result is similar to Sibelius, in that homophonic passages are reduced into a single voice, but polyphonic passages are reduced into multiple voices. The output can be seen in Figure 5.



The image shows a musical score for Piano and three Flutes, labeled Flute 1, Flute 2, and Flute 3. The Piano part is in the top staff, and the three Flutes are in the bottom three staves. The score is in 4/4 time. The Piano part has a complex melody with many notes, while the Flutes have simpler, more homophonic parts. The score is a single system with four staves.

Figure 5: Dorico reduce tool. Tested using Dorico 3.1.10.1032.

While these tools might be the most convenient to use, they are very limited in scope, and for orchestral scores which will often have many different rhythms at the same time, it is impractical to use these tools for anything other than the simplest of passages.

There have been attempts at producing a tool, similar to the one I have produced, which have been described in academic papers.

Chiu et al.[4] describe a system in which passages within a musical work are categorised into different arrangement elements, based on their role within the piece. These elements are lead, foundation, rhythm, pad, and fill. The musical work is split up into segmented tracks, or phrases, and these phrases are classified using a support vector machine. The

phrases are then assigned a utility value according to the type of arrangement element, and finally a phrase selection algorithm is employed to select as many phrases as possible to keep the output playable and with the highest total utility value.

Nakamura et al.[5] presents the problem as one of optimisation, in which the difficulty of the reduction is balanced against the fidelity to the original score. The difficulty of the reduction is calculated using a piano fingering model based on a Hidden Markov Model. The fidelity to the original score is based on the probability of different editing techniques. The reduction can then be produced by specifying a range on the difficulty, and maximising the fidelity within that range.

Li et al.[6] works in a similar way to Chiu et al., by splitting the music into phrases and keeping as many phrases as possible, while maintaining playability. However, instead of using heuristics to classify staves, this solution uses musical entropy to find the most important elements of the score. Musical entropy is used to describe the complexity of a passage.

3. Requirements Specification

At the start of the project I settled on this list of requirements.

3.1. Primary Objectives

- Create reductions of small ensemble pieces with simple but non-homophonic rhythms.
- Allow input scores to have a variety of instruments with different clefs and transpositions.
- Allow simple annotations such as dynamics and tempo markings to be maintained.
- The software should do some work to create an idiomatic piano reduction, by making sure intervals in each hand are within a handspan, if possible.

3.2. Secondary Objectives

- Support for more complicated rhythms and notations (e.g. triplets, chords in instrument parts).
- Provide a configuration interface for configuring the input to the program (though not the output).
- Support for larger ensembles by allowing additional staves over the default 2 staves for the piano.
- The software should do more work on creating an idiomatic piano part by working to eliminate awkward leaps by having a floating split point between the right and left hand.

3.3. Tertiary Objectives

- Provide an interface to view the output of the file automatically.
- Further analysis of the musical content of the score and perform further reductions such as removing notes and adjusting octaves.

4. Software Engineering Process

The process I took to developing the software was a goal based one, in which every week I would aim to have a tangible goal completed, as discussed in the weekly meetings with my supervisor. These goals would often be small, for example allowing the program to parse chords instead of just single notes, but it would mean that every week I had something to work on. Once I had reached a point where I had a minimum viable product, I attempted to ensure I always had a working solution that took a valid input and produced a valid output, even if the product wasn't much more useful beyond that.

This agile approach ensured I could easily keep track of my progress, and the short sprint cycles enabled me to easily change track if something turned out to be more difficult than I had originally anticipated.

4.1. Tools Used

The program is written using the Rust programming language[7]. I chose this language due to its powerful type system, including components such as sum types, which allow me to attach different types of data to different enum variants, and its functional-style iterator system, which allows me to use transformations such as maps and filters on iterators in order to transform data. It also has a very easy to use dependency management and deployment system in the form of Cargo[8], which allows me to easily develop on different operating systems without having to spend time setting up an environment. I used git version control to store my code and track progress.

My project has several dependencies:

roxmltree[9] - This efficiently parses an XML document to a read only tree, that allows it to be queried and iterated over.

quick-xml[10] - While this also includes an XML parser, I use this as an XML writer to write the output MusicXML file.

clap[11] - This is a command line parser, and allows me to easily build the fairly complex command line interface that my program uses.

itertools[12] - This includes extra iterator adaptors and functions that I use in the transformations.

5. Design

The input and output format of the program is MusicXML[13]. This is widely used as a format for moving scores between different music notation software, and so is ideal to be used as the file format for input and output files. A downside of using this format is that it requires the original score to already be present in music notation software, so it can be exported as MusicXML. If the composer of the work is the user of this program, this is not a problem, but for existing works it means that the score must be entered manually, as the notation files are generally not distributed, and may not even exist. There has been an effort to digitise scores so that they can be used in these kinds of scenarios[14], but the range is fairly limited, and is only really possible for public domain works. With that being said, this is the only real option for a project like this, as trying to convert from a rendered file such as an image or PDF is still a young field, and current attempts generally provide very inaccurate results[15].

While MusicXML is useful for input and output, the file structure is not ideal for the actual processing of the score. While there are two variants of MusicXML defined by the specification[16], partwise and timewise, only partwise is commonly used. In this format, a musical score is made up of a list of parts. Within each part is a list of numbered bars, each containing the notes belonging to that bar, as well as any musical structure elements such as key and time signatures. These structural elements are repeated for that bar in every part.

To work around this, this project developed a program that parses the MusicXML into an internal representation that is easier to work on. A number of transformations are applied to this representation, that does the work of reducing the score, before it is re-exported as MusicXML. Similar to the approach in Chiu [4], the internal representation is based on the idea of phrases. A phrase represents a linear timeline of non-overlapping, but not necessarily contiguous elements, and generally represents the notes of a musical idea that an instrument plays, such as a line of a melody. Each element is either an individual note, or multiple notes (a chord), and has an associated position and length. Concepts such as bars are discarded, and structural items such as key and time signatures are stored separately, so that bars can be reconstructed at the end.

Parsing the score breaks down the musical work into phrases, which can be considered the basic building blocks that the program operates on. The program then builds up the work again to a piano score by operating on the phrases in the transformations, by assigning staves, adjusting octaves or by removing phrases.

A key part of the program is that certain transformations can be turned on or off, allowing the user to experiment with different settings to produce different outputs, which they can then compare themselves to get the best reduction for a particular musical work.

The way the data is represented in the program, as well as the transformations that can be applied to it is shown in Figure 6.

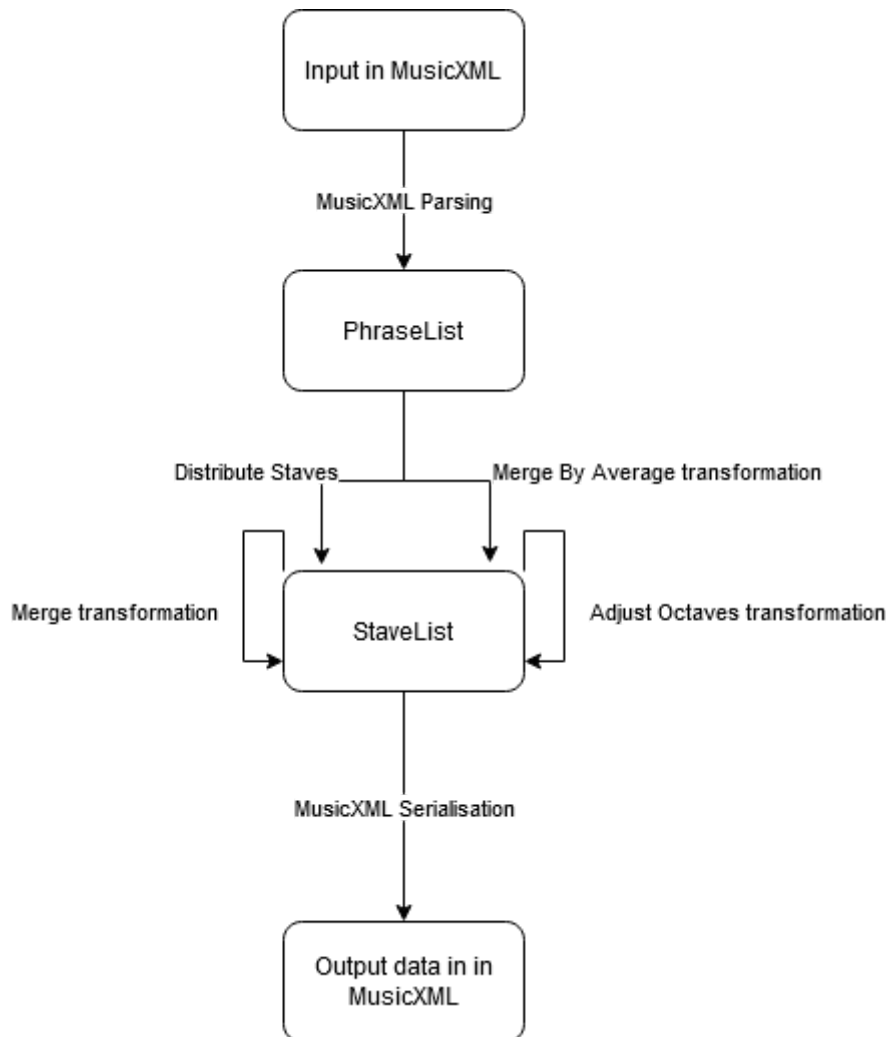


Figure 6. The data flow of the program.

6. Implementation

6.1. Keeping Time

One of the essential dimensions to music is time. Notes in a musical work have a position and a duration, and we need a way to represent this in the program. As concepts such as bars are separated out at the start of the program, it is not possible to work in terms of bars. For simplicity's sake it would be easier for two identical note values to be represented the same way regardless of what the beat type is as defined in the time

signature. To do this I created a Fraction data type, which functions like an ordinary mathematical fraction, in that it is composed of a numerator and denominator, and mathematical functions such as addition, subtract, multiplication and division can be applied, as well as comparison operations. This allows note lengths to be adjusted as required.

I chose a value of 1 to represent a crotchet, as it's a very common beat type. Another obvious choice would have been a semibreve, which means that when time signatures are represented as fractions they represent the length of a bar, e.g. a time signature of 4/4 indicates the bar is a semibreve long, with a crotchet as the beat unit. Further, this would be consistent with the American standard for specifying note values, where a semibreve is a whole note. However, as a crotchet is a very common beat type, e.g. in 3/4 or 4/4, I found it more familiar to use that, although ultimately the decision is fairly arbitrary, as long as it is consistent throughout the program.

Fraction is implemented as a struct containing two signed integers representing the numerator and denominator. Operations are defined using the `std::ops::*` traits, which means that `+`, `-` etc. operators can be used. Fractions are automatically simplified through use of a greatest common divisor function whenever they are modified or created, which makes debugging easier as identical fractions will have an identical representation. This technique allows notes and therefore combinations within phrases to be easily manipulated within the program.

6.2. MusicXML Parser

Due to the complexity of MusicXML, creating the parser proved to be an unexpectedly difficult task that I spent a lot of time on early on in the project. The XML is parsed using the `roxmltree` [9] library, which parses the document into a read only tree structure that can be queried and iterated. As MusicXML files can contain a lot of information unrelated to my program, such as formatting, the program generally queries the tree to get information, rather than iterating through the whole tree and processing everything in sequence. For example, to get the parts in the score, the program filters the children of the root element for all elements that have the tag name `part`, and then processes each part.

To process each part, a similar filter is applied to get the bars. Each bar has an `attributes` element defining properties about the bar and indicating changes of elements such as key and time signature changes, and instrument transposition. These values are stored as variables and are updated for each bar that is processed.

The elements in the bar are defined by note elements in the XML, so the program iterates through all the notes to process them. As MusicXML does not define note positions, just their durations, it is necessary to keep a counter object that is incremented with the

duration of each note. Confusingly, both notes and rests have the `note` tag, and while rests have a `rest` element, notes have a `pitch`. They both have elements that define their duration.

Durations in MusicXML are defined in two different ways. There is the intended duration, and the notated duration. The intended duration is defined as a whole number multiplier of divisions, which are defined in the bar attributes. Divisions in a bar is typically the division of the smallest note value in the bar from a crotchet. So, a bar of all quavers, each half the size of a crotchet, could have a bar division of 2, with each note having a duration of 1. Similarly, a bar of all minims (twice the length of a crotchet) could have a bar division of 1, with each note having a duration of 2. The notated duration is defined in terms of note values (whole, half, etc.). Typically, these durations are equivalent, but there are exceptions. A whole bar rest will always have a notated duration of a whole note, but depending on the time signature the intended duration might be different. For this reason, the program looks at the intended duration first, and falls back on the notated duration if the intended duration is not present. For every note processed, the counter is increased by the duration, so that it represents the start position of the next note. Notes are added to a list of notes that is later put into a phrase, whereas rests are discarded, although the counter is still updated. The counter then shows progress throughout the musical work in terms of beats.

Key and time signatures are also defined in the attributes section where they first appear. Key signatures are defined by the number of sharps or flats they have in the fifths element. A positive number represents sharps and a negative number indicates flats, with 0 representing C major. For example, the key of A major has 3 sharps, so the fifths element contains the value 3. Time signatures are defined by the beats and beat-type elements, with the beats representing the numerator and the beat-type representing the denominator.

Notes in chords are specified using the chord element in all but the first note. The chord element indicates that the counter should be rewound to the previous note and that the note being defined should be part of a chord with the previous note. This is fairly simple to do, by rewinding by the length of the note being processed, getting the note at that position, and converting it to a chord with the two notes. However, I had problems with this when writing the Adjust Octaves transformation, as if a chord was larger than an octave, the program would not know how to reduce it and would loop infinitely. To work around this, notes in a chord are each added to an individual phrase, which will then get condensed together in the Merge reduction.

Note pitches are defined by three attributes: step, octave, and alter. Step is a letter A-G that defines the note name. Octave is a positive integer defining which octave a note is in, with the split between octaves being between notes B and C, and with octave 4 defining

the octave starting with middle C. Alter describes the chromatic alteration as a signed value. The value could be decimal to represent microtonal notation, but that is beyond the scope of this application, so it is safe to treat it as a signed integer. The value represents an adjustment from the step in semitones. So a C \sharp would be defined with step C and alter 1. These values can be parsed fairly easily, although before they are stored they must have the instrument transposition applied if necessary.

Ties on notes are defined in the notations element, under the tied element. A note can either have no tie, start a tie, end a tie, or both start and end a tie in the case of being in the middle of a tie chain. The MusicXML specification has two relevant values for the type attribute: start and stop, with no element being the default for no tie. The Note type has an attribute of a Tie, which is an enum with variants: None, Start, Stop, StartStop. To allow ties to be added to notes without affecting existing ties I added start and stop functions to the Tie type that modify the current tie to allow it to be started and stopped. If the start function is called on a None or Start variant, the Start variant is used, otherwise the StartStop variant is used. The equivalent is true for the Stop variant. This means that the program can iterate through all of the Tied elements, and if the element is Start it calls the start function, if it's Stop it calls the stop function. This ensures that a note with both start and stop ties is represented with a StartStop tie variant.

Instrument transposition is defined in the attributes section. It contains three elements, diatonic, chromatic, and octave. The diatonic transposition defines the number of note name steps the note must take to get from its written value to its concert pitch value. So, an instrument with a diatonic transposition of -1 playing a written A would produce a concert pitch value of a G. The chromatic transposition defines the number of semitones between the written and concert pitches. So an instrument with a chromatic transposition of -2 playing an A would produce a G, but playing a C would produce a B \flat . However, it would also be true to say that the note produced is an A \sharp , or even a C \flat . Therefore, it is necessary to use both the chromatic and diatonic transposition elements when calculating the concert pitch. To do this there must be a way of calculating the value of a pitch as a number of semitones, so that they can be easily compared. The way I implemented this was to have a base value for each note, to which the chromatic alteration value is added, and finally the octave * 12 is added (there are 12 semitones in an octave). The base values are shown in table 4:

Table 4: the chromatic alteration values of each base note

Note	Value
C	0
D	2
E	4
F	5
G	7
A	9
B	11

With this in place, it is now possible to calculate the transposed note as follows.

1. The program parses the note as it is written in the MusicXML, along with the transposition element.
2. The value of the note is calculated and recorded.
3. The note step is shifted by the diatonic value, ensuring that the octave is changed if necessary.
4. The new value is calculated and subtracted from the original.
5. The chromatic value is subtracted from that difference, to get the final offset that the note must be altered.
6. The offset is subtracted from the note's alter value to get the transposed note.

As the document is being parsed a variable containing the current phrase is maintained. As notes are parsed, they are added to the current phrase. When a rest, or the end of a part is encountered, the current phrase is added to the list of phrases, and a new phrase is started. When testing this, I found that there were situations where instruments would play continuously for long periods of time, creating very long phrases. Because of this, the program accepts a parameter specifying the maximum number of bars a phrase should be.

Key signatures must also be transposed to concert pitch for transposing instruments for the piano score. This is a little different as key signatures are defined as a number of fifths, with positive numbers indicating a sharp key signature, and negative numbers indicating a flat key signature. However, this is an easier calculation, as the transposed key signature is equal to the original key signature + the chromatic transpose * 7, remainder 12.

As mentioned previously, key and time signatures are stored separately to the musical contents. These are stored in a BTreeMap with their location as the key, which allows them to be easily queried again when reassembling the bars into MusicXML.

6.3. Internal Representation of the Score

Notes are defined by their pitch and their ties, but do not include duration information. This is done to simplify modifying durations in phrases, and ensuring that all notes in a chord are the same length. Chords are represented as a Vec of notes. To allow phrases to have either a chord or a note at a given point, the enum PhraseElement has Note and Chord variants.

```
pub enum PhraseElement {  
    Note(Note),  
    Chord(Vec<Note>),  
}
```

As explained in the design section, a phrase contains a list of non-overlapping notes or chords. The simplest way to do this would be to have a Vec of PhraseElements, which would be associated with their position and length. However, in order to make querying easier the Vec would have to always be sorted to allow a binary search to be used. This adds a maintenance overhead whenever the Phrase is modified, and is prone to errors. Instead, I used a BTreeMap again, with the key being the element's position, and the value being a tuple of the PhraseElement and the duration. This allows iterating over the phrase in order, and also allows range based queries which is useful for getting the note before or after a certain point.

There are several functions implemented for the Phrase type that allow phrases to be queried and manipulated. Most of these are straightforward, allowing the user to get the start, end, and length of a phrase, or getting values such as the highest, lowest, or average pitch at a particular point.

The most complex function is the function to add a note to a phrase, as there are many things to consider based on collisions with other notes. First, the program checks if the note being added overlaps with the next element. If it does, it checks whether the overlapping element contains a note with the same pitch as the note being added. If it does, the note being added is shortened so that it stops at the start of the next element. If it does not contain an equivalent note, the added note is shortened to stop at the start of the next element, and it is set to have a start tie. A new note is also added at the point of overlap, with length equal to the amount the added note was shortened by, and the tie set to stop. This is illustrated in Figure 7. The first bar shows the note in phrase 2 being added

to phrase one, and the note being split and a tie being added. Bar two shows the notes having the same pitch, and so the added note is shortened.

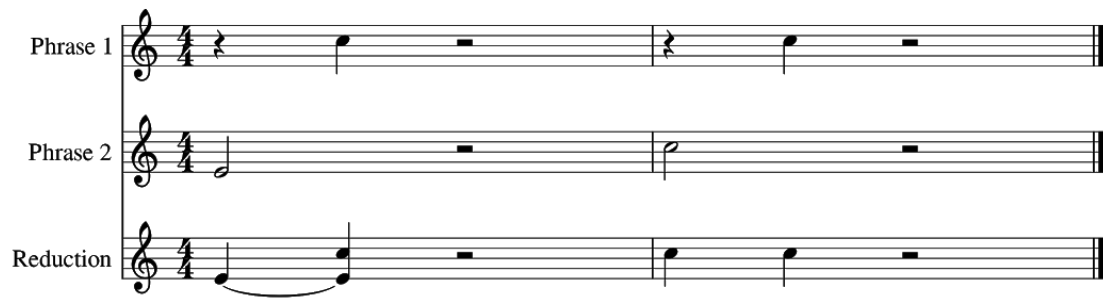


Figure 7: Adding notes to a phrase that overlap with existing notes.

Second, the program checks if the added note overlaps with the element before it. The way it deals with this is slightly inefficient but is simpler to reason about. The previous element is removed, and the new note is added. The previous element is then added back into the phrase recursively, meaning that it's handled by the next overlap section of the function. Finally, the program checks if there is an element at the position of the new note. If there isn't, the note can safely be inserted into the BTreeMap. If there is, the program compares the lengths of the existing element and the added note. If the length of the existing note is less than the length of the added note then the added note is shortened to the length of the existing note, is set to have a start tie, and is merged with the existing element. Then a new note is added to the phrase after the existing element, with the same pitch as the added note, a stop tie, and length equal to the amount the added note was shortened by. If the durations are the same, the note is merged with the existing element. If the existing element is longer than the added note, the element is replaced with the added note, and the existing element is recursively added back to the phrase. The final thing that is checked is if the existing element has a stop tie on a note with the same pitch as the added note. If it does, this should be deleted as the note should be rearticulated. This also means that the previous note, the one that starts the tie, should have the start tie tag removed. Figure 8 shows a note being added to a phrase that already has a note at that position. In the first bar, the E is split into two notes, and they are added separately, with a tie between them. In the second bar, the note is added to a phrase with a note with the same pitch, and a stop tie. The tie is removed.



Figure 8: Notes being added to a phrase with existing notes already at that location.

Functionality such as merging phrases or adding a `PhraseElement` to a phrase is built on top of the `add_note` function. Adding a `PhraseElement` is implemented by matching on the element, and if it is a note just adding the note, or if it's a chord adding each note individually. Merging one phrase into another is implemented by iterating through each element in the phrase to be merged and adding it to the existing phrase.

There are two different container classes that represent the score that transformations can be applied to. The first is `PhraseList`. This simply stores a list of phrases, as well as the key and time signatures, and is what is produced by parsing the MusicXML. The second is `StaveList`. This stores a `Vec` of a `Vec` of phrases. The first dimension represents the different staves, and the second represents the list of phrases in each stave. This is used to serialise the reduced score back into MusicXML.

6.4. MusicXml Serialisation

In order to create an output MusicXML document it's necessary to transform the data into a form that more represents the structure of a MusicXML document. This means reconstructing the piece into bars and parts. One simplification we can make here is that since the output is a piano score, this counts as only a single part, despite it having multiple staves. This means that the information about the part, such as the part list, can be hardcoded, and the piece can just be split into bars.

In order to split the piece into bars it's necessary to split at bar boundaries. This is done by looping through every phrase in each stave and calculating which bar it starts in. The `BarNumbers` type handles this by using the list of time signatures. When the type is constructed it takes the list of time signatures and calculates the bar number from these, by taking the gaps between time signatures and dividing the length by the time signatures to get the number of bars. The bar number is then stored with the time signature in a `BTreeMap` as before. It would have been possible to read this data at the start when the time signatures are read, but there is no requirement in the MusicXML specification that bar numbers have to be consecutive or unique. The bar number can then be calculated by getting the bar number and position of the previous time signature, then getting the number of bars between the time signature and the start of the phrase and adding that on to the bar number of the time signature.

For phrases that are shorter than a bar long, this works well, but many phrases will be longer, and therefore it is necessary to test if the phrase crosses a barline, and to then split at that point if it does. This is also handled by the `BarNumbers` type, with the `crosses_bar` function that takes the start and length of the phrase as parameters. The function works by getting the position and value of the previous time signature. It then normalises the position of the start of the phrase by subtracting the position of the start of the phrase and

the position of the time signature. It also calculates the length of a bar by multiplying the value of the time signature as a fraction by 4. Using this it calculates the normalised position of the next bar by subtracting the normalised value modulo the length of a bar from the normalised value to get the position of the start of the bar, then adding the length of a bar to get the next one. It then checks if the position of the end of the phrase is greater than the position of the next bar, and if it is, returns the position of the next bar. The phrase can then be split at that point, and the first half of the split can be stored in the appropriate bar, and the second half can be processed along with the remaining phrases.

Splitting a phrase is generally straightforward. The way this is handled is by creating two empty phrases and looping through each element in the original phrase. If the element is before the split point, it's added to the first phrase, and if it's after the split point it's added to the second. The exception is if the element crosses the split point, in which case the element must be duplicated, with the first set to start ties and the second set to stop them. The first element can then be added to the first phrase with length up to the split point, and the second can be added to the second phrase with the remaining length.

The phrases are organised into a two dimensional Vec, with the first dimension representing each bar, and the second dimension representing the phrases in each bar, along with their stave, in a tuple. The loop also records the minimum duration of each bar, which will be used to specify the duration of notes in the MusicXML.

Once the piece is in this format it can easily be converted to MusicXML. The MusicXML struct facilitates writing to a buffer, and contains functions for writing elements to the buffer, such as bars, notes, and rests, as well as other important elements such as backups. This code is mostly boilerplate involved with writing the correct xml elements, and so contains very little algorithmic interest. The most complex thing is the mechanism for converting from note names to divisions. When a bar is started the smallest division is converted to divisions of a crotchet. This is done in a match statement, where note values of less than a quarter note (crotchet) are valued as increasing powers of two, and note values from a quarter note up are valued as 1, as defined by the MusicXML specification and shown in Figure 9.

```
pub fn to_divisions(&self) -> u32 {
    match self {
        NoteType::N1024th => 256,
        NoteType::N512th => 128,
        NoteType::N256th => 64,
        NoteType::N128th => 32,
        NoteType::N64th => 16,
        NoteType::N32nd => 8,
        NoteType::N16th => 4,
        NoteType::Eighth => 2,
        NoteType::Quarter => 1,
        NoteType::Half => 1,
        NoteType::Whole => 1,
        NoteType::Breve => 1,
        NoteType::Long => 1,
        NoteType::Maxima => 1,
    }
}
```

Figure 9: Getting the bar divisions based on the shortest note.

To convert each note to a duration in divisions, another match statement is used, this time converting each note length to values of increasing powers of 2, starting from the longest possible note, which is a Maxima. This results in a quarter note having a value of 32, and so divisions for the bar are multiplied by 32, then divided by the calculated value, giving the length of the note in terms of divisions in the bar, as shown in Figure 10.

```
pub fn divisions(&self, current: u32) -> u32 {
    let division = match self {
        NoteType::N1024th => 8192,
        NoteType::N512th => 4096,
        NoteType::N256th => 2048,
        NoteType::N128th => 1024,
        NoteType::N64th => 512,
        NoteType::N32nd => 256,
        NoteType::N16th => 128,
        NoteType::Eighth => 64,
        NoteType::Quarter => 32,
        NoteType::Half => 16,
        NoteType::Whole => 8,
        NoteType::Breve => 4,
        NoteType::Long => 2,
        NoteType::Maxima => 1,
    };
    (current * 32) / division
}
```

Figure 10: getting the length of a note in divisions based on the bar division.

To generate the MusicXML document, the program loops through the list of bars and starts a new bar in the XML document. The time and key signatures are queried from the lists, and if there is one present at that location it is put in the attributes section of the bar. In the first bar, a treble clef is inserted in all but the last stave, where instead a bass clef is placed.

The program then loops through every phrase in the bar. If the phrase is empty, it adds a full bar rest to document, otherwise it loops through every element in the phrase. Similar to parsing the document, the program maintains a counter to keep track of the current position in the score to ensure that the document is written correctly. If the element starts after the current position then rests must be added before the element. The `from_fraction()` function on a `NoteType` converts a length as a fraction to a list of `NoteTypes`. These are then added to the document as rests and the counter is updated.

If the element starts before the current counter a backup element must be used. Similarly to rests, the amount needed to backup is converted to a list of `NoteTypes` and backup elements are created with those lengths.

Once the counter is in the correct position the element can be added to the document. The element is converted to a list of `NoteTypes` which must be duplicated and tied together. The first element is set to start a tie, the last element is set to stop a tie, and all other elements are set to start and stop a tie. These can then be added to the document. If

the element is just a note it is added as normal. If the element is a chord, the first note is added as normal, and then the remaining notes are added with the chord attribute enabled.

After all of the phrases have been added, the program checks that the counter is at the end of the bar. If it isn't, rests are added to fill the remainder of the bar, in the same way as they are at the start.

A fraction is converted to a list of NoteTypes by splitting the fraction into a list of powers of two. The program starts with a chunk of value 32/1 (the largest possible note) and loops until the fraction is zero. If the chunk is less than or equal to the fraction, the chunk is subtracted from the fraction and converted to a note type which is then added to the list. The chunk is then halved. This will continue until the fraction is zero, which means that it has been completely split into chunks. This process is shown in Figure 11.

```
pub fn from_fraction(mut fraction: Fraction) -> Vec<NoteType> {
    let mut notes = Vec::new();
    let mut chunk = Fraction::new(32, 1);
    while !fraction.is_zero() {
        if chunk <= fraction {
            fraction -= chunk;
            notes.push(
                Self::from_duration(chunk.numerator() as u32, chunk.denominator() as u32)
                    .unwrap(),
            );
            chunk /= Fraction::new(2, 1);
        }
    }
    notes
}
```

Figure 11: Converting a length as a fraction to a list of note types.

7. Transformations

Transformations are applied to the different representations of the score, either as a function on the object, or as a way of converting between object types.

7.1. Merge By Average

The first transformation I wrote was one that allocated each phrase into a stave based on the starting pitch of the phrase compared to the average pitch of the last phrase added to the stave. It takes a PhraseList and outputs a StaveList. The program sets up a Vec of staves, with each stave having one empty phrase, and a Vec of averages with length equal to the number of staves. The Vec of averages is populated by iterating through all the phrases and finding the maximum and minimum values at position 0. This range is then split evenly into staves + 1 parts, the split points of which are used as the starting average.

For example, with a 2 stave configuration, a maximum value of 60 and a minimum of 30, the interval is split into 3 parts with split points 50 and 40, which are used as the starting average.

The program then sorts the phrases by their start positions and loops through every one. It calculates the average pitch at the start of the phrase and then finds the index in the averages Vec with the closest value to the calculated mean. For example, if the averages Vec had elements [50, 40] and the average pitch at the start of the phrase was 55, the index of the closest value would be 0. The phrase is then merged with the phrase in the stave with the calculated index, and the average at that index is updated to equal the average pitch of the phrase.

This transformation is not generally used as it is quite inflexible. It results in just a single phrase per stave meaning that the octave adjustment transformation cannot be used effectively on the result. It also allows the possibility that hands can swap over as the mean of a phrase can be different to the mean at the start. However, it serves as a good baseline that other transformations can be compared to, and for some smaller scores it performs quite well.

7.2. Distribute Staves

The second transformation I wrote was one to distribute phrases onto staves, while leaving them as separate phrases so that other transformations could easily be applied to them. This takes inspiration from the Merge By Average transformation, but makes some changes that fix some of the problems with that technique. This transformation works on the same principle of moving averages, but it does not use a separate list of averages. Instead, it takes the idea of splitting the range between the highest and lowest note at a particular point and uses that for every phrase instead. It does this by looping through every phrase, and by taking the position of the start of the phrase, it finds the highest and lowest pitches at that point. It then splits that range into intervals for each stave, and finds the index of the value closest to the average pitch at the start of the stave. It then clones the phrase and adds it to the stave. It's necessary to clone the phrase so that while iterating through the rest of the staves it can still get the highest and lowest pitches of all the staves, while still being able to allocate them to a stave. This is a slight inefficiency, but reduces the complexity of trying to collect many phrases from many different locations when finding the maximum and minimum pitches.

This technique generally works better than Merge By Average. By always taking the range between the maximum and minimum elements, it ensures that two phrases starting at the same point will always be allocated to the correct stave to eliminate hand crossing, although there is still the possibility of phrases crossing after that point. It also doesn't take into account similarity of phrases, so in a situation where a single line melody is

played alongside a chordal accompaniment, there is no guarantee that the chordal accompaniment will be played in one hand and the melody in the other, even though that might be more idiomatic.

7.3. Adjust Octaves

The next transformation that can be applied is Adjust Octaves, which transposes phrases in a StaveList by octaves so that each stave fits within a certain interval at any given point. It can also move phrases onto adjacent staves if necessary.

The function starts by iterating through every stave. Within each stave, it makes a list of every unique start position of PhraseElements for all phrases. It then iterates through every position. For every position, it finds the highest, lowest, and average pitches at that point. It also calculates the midpoint between the highest and lowest pitches. Then, while that interval is greater than the specified maximum it adjusts the phrases. The way in which the function adjusts the phrases is fairly complex. The first thing checked is whether there is a stave above the current one. If there is, it checks whether the phrase with the highest pitch will fit into the handsan of the upper stave. This is done through the `can_have_phrase` function. If the stave above can fit the phrase, the phrase is moved to that stave.

If there is no stave above, or if the stave above cannot fit the phrase, then the program calculates the maximum and minimum pitches, excluding the ones from the maximum and minimum. It then checks if the highest phrase can be transposed down an octave. This is possible if:

1. The mean value is less than the midpoint. This means that the highest note in the phrase is an outlier and therefore is transposed towards the midpoint.
2. If this is the top stave, then the highest pitch shifted down an octave must be greater than or equal to the second highest pitch. The highest pitch is generally the melody, so this ensures that the melody stays as the highest pitch and doesn't get lost inside the accompaniment.
3. The lowest pitch in the phrase with the highest note has a value greater than or equal to 21. This is generally true, but if it wasn't then transposing the note down an octave would mean the phrase would go off the bottom of the piano.
4. If this is the lowest stave, then the highest pitch transposed down an octave is greater than or equal to the lowest pitch. This ensures that the phrase doesn't cross over to become the bass line.

If all of these are true, then the phrase with the highest note is transposed down an octave.

If these are not true, then other tactics must be tried. The program next checks if there is a stave below the current stave. If there is, then the phrase with the lowest note is moved to

that stave. This is done to prioritise pitches being kept equivalent when possible, although it's very possible that they might be transposed when the next stave is processed. If there is not a stave below, the program checks if the phrase with the lowest note can be transposed up an octave. The conditions for this are similar to before:

1. If this is the bottom stave, then the lowest note shifted up an octave must be less than or equal to the second lowest pitch. This ensures that the bass notes stay as bass notes, which is important for the quality and stability of the chord.
2. The highest pitch in the phrase with the lowest note has a value less than or equal to 84. Again, it is uncommon that this is not the case, but if it wasn't, then transposing the phrase up an octave would mean the phrase went off the top of the piano.
3. If this is the highest stave, then the lowest pitch transposed up an octave is less than or equal to the highest pitch. This ensures that the phrase doesn't cross over to become the melody line.

If both of these are true, then the phrase with the lowest note is transposed up an octave. Finally, if none of the conditions so far have been met, the program once again checks if this is the top stave, and if so, removes the phrase with the lowest note. If it's not the top stave, it removes the phrase with the highest note. This whole process repeats until the range at that position is less than the specified maximum.

While this technique makes its best effort to ensure the piece is kept within a handspan, there are some drawbacks. As the program only checks the handspan at the current point, it's possible that when notes later in a phrase are processed, the phrase could be transposed in a way that makes intervals earlier in the stave greater than the handspan, or the melody or bass notes may be lost in the accompaniment due to crossing of phrases. It also means that very similar sections of music in a piece may produce outputs in different octaves, or with different notes removed, due to differences in voicings, or overlaps with other phrases.

7.4. Merge

The final transformation that can be applied is very simple and is used to reduce the number of phrases on a stave. After a large orchestral score has had Distribute Staves applied, there are likely to be many phrases on each stave simultaneously, which can be very difficult to leave. This transformation simply iterates through each stave and merges every phrase in that stave into a single phrase. This does prevent the possibility of two voices in a stave being present in the output, but it provides a simple solution to the problem of having many concurrent phrases within a stave.

8. Evaluation and Critical Appraisal

8.1 Evaluation Against Requirements Specification

8.1.1. Primary Objectives

- Create reductions of small ensemble pieces with simple but non-homophonic rhythms.
- Allow input scores to have a variety of instruments with different clefs and transpositions.
- Allow simple annotations such as dynamics and tempo markings to be maintained.
- The software should do some work to create an idiomatic piano reduction, by making sure intervals in each hand are within a handspan, if possible.

Most of the primary objectives have been completed. A good example of a reduction of a small ensemble piece is the Bruckner described in the next section. The program also allows instruments to have instruments in a variety of clefs and transpositions, as shown in the Sullivan in the next section. However, annotations such as dynamics and tempo markings are not maintained. I decided not to complete this as I wanted to concentrate on the transformations that are applied to phrases, rather than simply structural elements. The way that key and time signatures are stored shows the principle of how other structural elements could be processed, and if the product was to be expanded in the future it should be fairly easy to add that on. The software also does work to ensure intervals are kept within a handspan, through the Adjust Octaves transformation.

8.1.2 Secondary Objectives

- Support for more complicated rhythms and notations (e.g. triplets, chords in instrument parts).
- Provide a configuration interface for configuring the input to the program (though not the output).
- Support for larger ensembles by allowing additional staves over the default 2 staves for the piano.
- The software should do more work on creating an idiomatic piano part by working to eliminate awkward leaps by having a floating split point between the right and left hand.

Again, many of the secondary objectives have been completed. Chords in instrumental parts are supported, however, other complex features such as triplets and grace notes are not. When writing the helper functions such as merging phrases, I realised that these would require special cases in most instances, and would have made these methods significantly more complex than they already were, and as before, I wanted to concentrate on the transformations that used these functions instead. A command line interface is

used to operate the program, and this interface has a number of different options that can be enabled, as described in the operating instructions. One of these options allows the user to specify how many staves the output score should have, which fulfils the third secondary objective. Finally, the Merge By Average transformation ensures that phrases are merged into the staff with the closest current average, meaning that large leaps are eliminated as much as possible. However, this does not account for instances in which the phrase already contains a large leap, or when pitch of the music shifts register so a large jump is unavoidable.

8.1.3. Tertiary Objectives

- Provide an interface to view the output of the file automatically.
- Further analysis of the musical content of the score and perform further reductions such as removing notes and adjusting octaves.

My idea with providing an interface to view the output of the file automatically was to show a webpage containing a JavaScript rendering of the MusicXML score. However, I found that this would have been a lot of work to set up, and would not significantly improve the user experience, due to the existing widespread usage of MusicXML. The Adjust Octaves transformation allows phrases to be removed or transposed by octaves in order to obtain an idiomatic reduction. However, the extent of the analysis that this is done with is fairly basic, as the only condition it works with is to maintain the top and bottom lines, which generally represent the melody and the bass line, and doesn't analyse the work to maintain harmonic consistency, for example.

8.2. Evaluation Against Musical Works

To evaluate the efficacy of the software I used two musical works. I ran the program with the arguments specified, and opened the output file in MuseScore, where I immediately exported the output as a PNG.

8.2.1 Bruckner - Aequale No. 1

The first work I used was Aequale No. 1 for three trombones, by Anton Bruckner[17]. This work is fairly straightforward, with only three parts, and the polyphony is generally simple. However, it provided a good baseline to test the program with.

Running the program with the merge-by-average option produces the output in Figure 12.



Figure 12: The generated reduction of Bruckner's Aequale No. 1 with the -a option.

Generally, this reduction is very playable. In the first four bars the distribution is very idiomatic, with each chord fitting easily within a handspan. The shift from the left hand to the right hand of the middle voice between bars 2 and 3 is perhaps not ideal, but it is still not difficult to play the transition smoothly. The fifth and sixth bars have the biggest problems. The interval on the third beat of bar 5 in the left hand is too large for many hands, and the left hand stops playing in the sixth bar unnecessarily. However, the rest of the reduction is very idiomatic, and most experienced pianists would have no problem sight reading it.

I then ran the program with the default arguments, which specifies that it should use the Distribute Staves, Adjust Octaves, and Merge transformations, with the handspan set as 12 semitones, which is equal to an octave. This produced the output in Figure 13.



Figure 13: The generated reduction of Bruckner's Aequale No. 1 with default options

There are a few differences in this reduction to the previous one. The large interval in bar 5 has been resolved by the Adjust Octaves transformation by moving the middle voice to the right hand, and Distribute Staves transformation has ensured that there is something on each hand in bar 6. However, the split in bar 16 is slightly more awkward due to the imbalance of notes between each hand. This occurs because the middle E \flat is equal to the midpoint between the two staves, and in that case it is rounded to the staff above, rather than below. The ideal solution for this would probably be for the middle voice to be played by the right hand from bars 14 to 16, which would eliminate the large leap that the left hand has to make between bars 15 and 16. However, the reduction is still playable, and much easier to read than trying to read off the original 3 staves.

8.2.2. Sullivan - Overture to Patience

The second work I tested is the Overture to the operetta *Patience*, with music by Arthur Sullivan. This work provides a much greater challenge, as it is scored for a larger orchestra, and over the course of the piece contains a much wider variety of textures. Due to the length of the work I have chosen a few passages that represent different textures to discuss, rather than looking at the whole output. The entire output can be found in the `Patience Default.musicxml` file in the tests directory. For this test I used the default settings. I compared the output to the G&S archive edition of the *Patience* Vocal Score[18], which includes a manually written piano reduction of the score. The orchestral score is my own edition.

The first passage I examined is the first 12 bars of the piece, which starts with a full orchestral tutti for the first 3 bars, then quickly scales back to just a few instruments at a time with a much thinner texture. The generated output can be seen in Figure 14, which has been exported from MuseScore with the musical contents untouched, but with a stave break added so that the passage lines up with the vocal score. Figure 15 shows the vocal score version.



Figure 14: The first 12 bars of the generated reduction of the Patience Overture by Sullivan



Figure 15: The first 12 bars of the Patience Overture by Sullivan from the vocal score

The generated output and the vocal score took a fairly different approach to the tutti at the start. While the generated output produced densely voiced chords in the left hand higher up the keyboard, the vocal score version opted for octaves in the left hand, with heavier chords in the right hand instead. The choice between these two approaches is largely down to personal preference. While denser chords in the left hand can produce a bigger sound, the octaves in the left hand accentuate the bass line more and can produce a clearer sound. My program tends to favour denser chords to avoid affecting the harmony by removing important notes.

Another important thing to notice in the second bar is that the bass line is different in the two versions. The vocal score version is correct here, and the error in the generated part is due to a limitation of the Adjust Octaves transformation. This happens because the conditions for when an octave transposition occurs only apply to the notes that are playing at the same time as the current position. So, all of the transformations would have been applied when the counter was at the first beat of the bar. In this case, this has happened because there is a phrase containing the E \flat to B \flat leap, which, when transposed down an octave overlaps with the bassline in octaves. However, as the bassline is still intact on the first note of the bar, the program does not see this, and so the bass line later on in the bar is adjusted. This is illustrated in Figure 16, which shows the octave adjustments applied in turn to that bar, while the counter is at the first beat. The notes highlighted green shows the offending phrase that gets transposed to become the bass line.



Figure 16: The octave adjustments applied individually to bar 2

In this case, the result of this is acceptable. The B \flat happens to be part of the chord on the second beat, meaning that the quality is not changed significantly, although it is in a different inversion. However, this does have the potential to significantly affect the quality of the reduction and is something I would look to fix in the future.

While the approach taken to the tutti at the start is fairly different, the approach taken to the lighter texture afterwards is fairly similar. The notes in these two sections are almost identical, although in bars 5 and 7, the middle voice is assigned to the left hand in the generated reduction, as opposed to the right hand in the vocal score. Again, the approach taken in the vocal score is more idiomatic due to the identical rhythms of the top two phrases, but the program made the choice because the middle line is physically closer to the bottom line, and so was placed with it. The generated version is still very playable though.

The other significant difference, excepting notational differences due to the use of multiple voices in the vocal score, is the voicing of the last two bars. The notes in the two versions are identical, except the generated score has the bottom line in octaves, rather than just the top octave in the vocal score. This is more faithful to the original score, although it does introduce a larger leap in the left hand between bars 10 and 11. Again, this choice is generally subjective, and is related to the discussion of the tradeoff between difficulty and fidelity to the original score in Nakamura et al.[5].

The second passage I chose is the 4 bars following the first passage. This again features a completely different texture, with a medium trumpet solo in the middle of the range accompanied by high/mid range woodwinds. The generated output can be seen in Figure 17, while the vocal score version can be seen in Figure 18.



Figure 17: The generated output of bars 13-16 of the Patience Overture

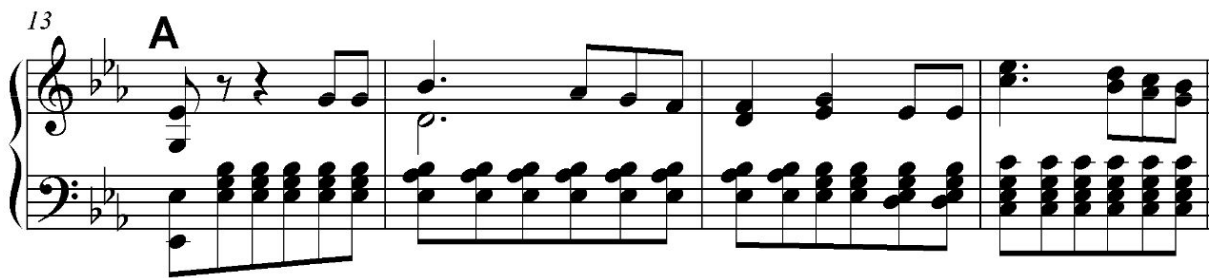


Figure 18: The vocal score version of the same passage

Unfortunately, this kind of texture is one that my program fails to deal with adequately. In the original orchestral score, the contrast in character between the trumpet playing the melody and the woodwinds playing the accompaniment is such that the melody can clearly be heard, despite playing in the same register as the accompaniment. The vocal score deals with this by transposing the accompaniment down to below the melody, which is more common on the piano, in order for the contrast to come across.

Unfortunately, my program is not able to make similar provisions, as instrumental information is stripped away when the score is parsed, and as such it cannot tell the difference between the melody and the accompaniment. I've highlighted the melody in green to show how it sits within the accompaniment, but when playing the reduction it is very difficult to pick out.

The third passage I used is later on in the piece, from bars 41 - 46, when the tempo changes to a fast 4/4, rather than a slow 3/4. The generated output can be seen in Figure 19, while the equivalent in the vocal score is shown in Figure 20.



Figure 19: The generated output of bars 41 - 46 of the Patience Overture

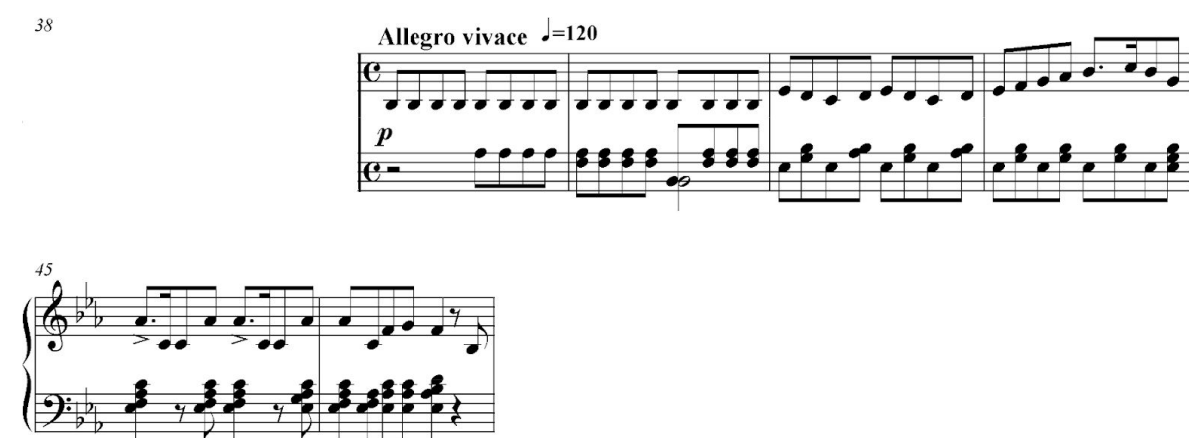


Figure 20: The vocal score version of the same passage

Here, the two versions take fairly similar approaches, but there are some key differences. The vocal score keeps the um-pah accompaniment all in the left hand, by transposing the bass notes up an octave, and revoicing the chords so that they easily fit. The generated reduction instead keeps the original voicings, which gives the accompaniment a stronger sound, although is harder to play. The chords in bar 45 and 46 are also revoiced in the vocal score, allowing the right hand to just play the melody, although it misses out on the strength of the octaves in the left hand that the generated reduction has. This revoicing does have the advantage of keeping the melody on top on the second quaver of bar 46, when in the orchestral score and generated reduction, it dips below the accompaniment. The moving chords in the right hand in bar 46 are also a little difficult to play, although it is still very possible. Again, this functions as a tradeoff between difficulty and fidelity to the original score, although the handmade reduction also has the advantage of knowing when to account for differences in orchestral colours which would otherwise be lost in the reduction.

9. Conclusions and Futures

Creating an orchestral reduction has always been a difficult task, and this project highlights many of the problems and tradeoffs that can arise during the writing of one. While for many types of music this kind of automation can work very well, and indeed my project provides good solutions for, there are other cases where other context about the music is needed, such as how different orchestral instruments sound together, the identification of a melody, or what makes a piano part idiomatic. With that being said, my project usually generates a reduction that provides a very good base for writing the final reduction, and could potentially save a lot of time for someone tasked with creating a reduction.

This project has given me a lot of experience in writing larger software products. I've gained invaluable experience writing in Rust, as well as writing algorithms for transforming and manipulating complex data. While there were difficulties I didn't anticipate, such as the parsing of MusicXML, overall I am very happy with how the program turned out, and I believe it could be useful in helping people develop orchestral reductions.

There are many ways the project could be extended or improved. One way would be to improve the friendliness of the program by providing a web interface that renders the output, allowing the user to see the results of changing different parameters in real time. There are also improvements that could be made to the transformations, such as allowing two voices per stave in the output, rather than one, or by changing the Adjust Octaves transformation to take into account the entire phrase, rather than just the notes at each individual point. There could also be more transformations added which could allow the program to account for some more of the many different textures and writing styles that occur in orchestral music.

Appendices

A. Testing Summary

To test the program works correctly, I used a combination of unit tests and test files. I used unit tests to verify some of the smaller components of the program, and the test files to generate outputs that I could then verify are correct.

The unit tests I created are described in Table 5.















Table 5: unit tests

Test	Description
fraction_balance: fraction.rs	Tests that when a fraction is created it is put in its simplest form.
fraction_add	Tests the different kinds of add functionality, by adding fractions with the same denominator that don't need rebalancing, adding fractions with the same denominator that do need rebalancing, and adding fractions with different denominators.
fraction_subtract	Tests the different kinds of subtract functionality, by subtracting fractions with the same denominator that don't need rebalancing, subtracting fractions with the same denominator that do need rebalancing, and subtracting fractions with different denominators.
fraction_multiply	Tests a variety of different fraction multiplications.
fraction_divide	Tests a variety of different fraction divisions.
phrase_split_start: phrase.rs	Tests that a phrase split at the start of the phrase results in an empty phrase and the original phrase.
phrase_split_end	Tests that a phrase split at the end of the phrase results in the original phrase and an empty phrase.
phrase_split_middle	Tests that a phrase split in the middle of the note results in two phrases, one with the note up to the split point with a start tie, and the other with the note after the split point with an end tie.
phrase_start_end	Tests that the start and end functions on a phrase produce the correct results.
note_values: phrase_element.rs	Tests that note values are correctly produced, and correctly handle accidentals and octave shifts.
note_types_from_fraction	Tests that converting a fraction to a list of note types works correctly.

transpose_ordinary: score_representation.rs	Tests note transposition within an octave with no alterations
transpose_alteration	Tests note transposition that results in a note with an alteration
transpose_octave	Tests note transposition where the resulting note is in a different octave
bar_numbers: output_score.rs	Tests that BarNumbers type can correctly calculate the bar numbers
crosses_bar	Tests that BarNumbers correctly detects when a passage crosses a bar, and where the split point is

To test the phrase merge function, I created some test files. They are run with the number of staves set to 1 in order to force the staves to be merged. This also tests the MusicXML import and export. The files I used are described in Table 6.

Table 6: Test files

Input	Output	Description
test1.musicxml 	test1.out.musicxml 	Tests merging two notes with different pitches and the same rhythm
test2.musicxml 	test2.out.musicxml 	Test merging two notes with different pitches and different rhythms. This shows that the merged notes maintain their original rhythm
test3.musicxml 	test3.out.musicxml 	Test merging two notes with the same pitch and different rhythms. This shows that the longer note is kept
test4.musicxml 	test4.out.musicxml 	Test merging two overlapping notes of the same pitch. This shows that the first note is cut short
test5.musicxml 	test5.out.musicxml 	Test merging three notes where the longer note overlaps a different note. This shows that the longer notes maintain their lengths
test6.musicxml 	test6.out.musicxml 	Test merging three notes where one note overlaps with the another with the same pitch. This shows that the first note is cut short
test7.musicxml 	test7.out.musicxml 	Test merging two notes with different pitches, where one note encompasses the other. This shows that both notes maintain their rhythm

test8.musicxml	test8.out.musicxml	Test merging two notes with the same pitch, where one encompasses the rhythm of the other. This shows that the longer note is cut short by the overlapping note.
----------------	--------------------	--

B. User Manual

To build the project requires the Rust compiler and the Cargo package manager. These can easily be installed using the Rustup tool, which will automatically download the necessary components. This can be installed by following the instructions at <https://rustup.rs/>.

Once Rust has been installed, the program can be run with the `cargo run --` command in the root directory. The two dashes are important to ensure the command line arguments are passed correctly.

There are a number of command line arguments that can be used. These are:

- i (--input) <file> - The input file [required]
- o (--output) <file> - The output file [default: output.musicxml]
- s (--staves) <n> - The number of staves the output should have [default: 2]
- h (--handspan) <n> - The maximum interval per hand in semitones [default: 12]
- l (--max-phrase-length) <n> - The maximum phrase length when parsing the MusicXML. Use 0 to indicate no maximum [default: 1]
- a (--merge-by-average) - Use the Merge By Average transformation instead of Distribute Staves
- n (--no-adjust-octaves) - Don't run the Adjust Octaves transformation
- m (--no-merge) - Don't merge phrases in a staff together before exporting
- help - Show the help page

References

- (2020) "Tools | Muscore" Muscore Handbook [online] Available at <https://musescore.org/en/handbook/tools#implode> [Accessed 30 Apr. 2020]
- (2018) Sibelius Reference Guide [online]. Available at: http://resources.avid.com/SupportFiles/Sibelius/2018.1/Sibelius_2018.1_Reference_Guide.pdf pp. 297 [Accessed 30 Apr. 2020]

3. (2020) "Reducing music onto fewer staves" Dorico Help [online] Available at https://steinberg.help/dorico/v3/en/dorico/topics/write_mode/write_mode_arranging_reducing_t.html [Accessed 30 Apr. 2020]
4. S. Chiu, M. Shan and J. Huang, (2009) "Automatic System for the Arrangement of Piano Reductions," 11th IEEE International Symposium on Multimedia, San Diego, CA, 2009, pp. 459-464.
5. Nakamura, Eita and Shigeki Sagayama. (2015) "Automatic Piano Reduction from Ensemble Scores Based on Merged-Output Hidden Markov Model." ICMC.
6. Li, You et al. (2019) "Automatic Piano Reduction of Orchestral Music Based on Musical Entropy." 2019 53rd Annual Conference on Information Sciences and Systems (CISS) : 1-5.
7. (2020) "Rust Programming Language" [online]. Available at: <https://www.rust-lang.org/> [Accessed 30 Apr. 2020]
8. (2020) "Introduction - The Cargo Book" [online]. Available at: <https://doc.rust-lang.org/cargo/> [Accessed 30 Apr. 2020]
9. (2020) "roxmltree - crates.io: Rust Package Registry" [online]. Available at: <https://crates.io/crates/roxmltree> [Accessed 30 Apr. 2020]
10. (2020) "quick-xml - crates.io: Rust Package Registry" [online]. Available at: <https://crates.io/crates/quick-xml> [Accessed 30 Apr. 2020]
11. (2020) "clap - crates.io: Rust Package Registry" [online]. Available at: <https://crates.io/crates/clap> [Accessed 30 Apr. 2020]
12. (2020) "itertools - crates.io: Rust Package Registry" [online]. Available at: <https://crates.io/crates/itertools> [Accessed 30 Apr. 2020]
13. (2017) "MusicXML" Final Community Group Report 07 December 2017 [online]. Available at: <https://w3c.github.io/musicxml/> [Accessed 30 Apr. 2020]
14. (2017) "Introducing Openscore | MuseScore" Musescore Blog [online]. Available at: <https://musescore.org/en/user/57401/blog/2017/01/11/introducing-openscore> [Accessed 30 Apr. 2020]
15. Rebelo, A., Fujinaga, I., Paszkiewicz, F. et al. [2012] Optical music recognition: state-of-the-art and open issues. [online] Int J Multimed Info Retr 1, 173–190. Available at: <https://doi.org/10.1007/s13735-012-0004-6> [Accessed 30 Apr. 2020]
16. (2020) "Adapting to a Hierarchy - MusicXML" MusicXML tutorial [online]. Available at: <https://www.musicxml.com/tutorial/file-structure/adapting-to-a-hierarchy/> [Accessed 30 Apr. 2020]
17. (2017) "Bruckner - Aequale No. 1 (WAB 114)" Musescore.com [online]. Available at: <https://musescore.com/openscore/scores/4074271> [Accessed 30 Apr. 2020]
18. (2015) "Patience" The Gilbert and Sullivan Archive [online]. Available at: <https://gsarchive.net/patience/html/index.html> [Accessed 30 Apr. 2020]