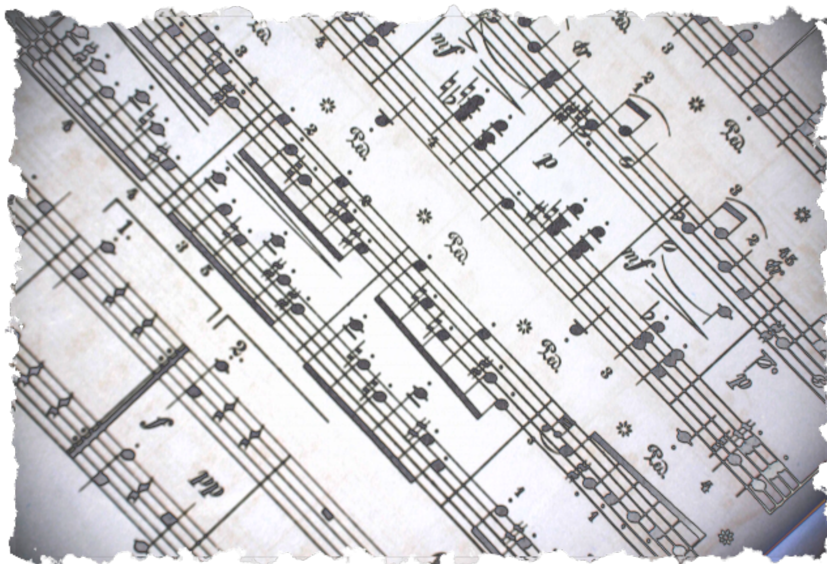
Emily Fox

Automated Canon Composition

Computer Science Tripos – Part II

Churchill College

May 10, 2016



Proforma

Name:	**Emily Fox**
College:	**Churchill College**
Project Title:	**Automated Canon Composition**
Examination:	**Computer Science Tripos – Part II, 2015-16**
Word Count:	**11577**[1]
Project Originator:	Dr Samuel Aaron
Supervisor:	Dr Samuel Aaron

Original Aims of the Project

To make an extension to Sonic Pi[4] that is able to generate musical canons[24] and play them back to the user. In line with the design aims of Sonic Pi, the extension should be ready for use by non-technical users. In addition, the canons played should be randomly generated using Sonic Pi's deterministic randomisation features in order to get different canons when different seed values have been specified. It should be possible to specify certain aspects of the resulting canons, for example the key and range of notes used.

Work Completed

An extension to Sonic Pi, implemented in Ruby, that allows canons to be generated with a single line of code. The user can specify key, note range, distribution of note lengths, canon type, length, number of voices, number of repeats, sounds used, maximum size of 'jumps' in pitch, offset between voices, voice transposition, number of repeats and time signature. The types of canon supported are: rounds[28], crab canons[26] and palindromes[27]. The extension to enable the pieces to be exported to Lilypond[1] for typesetting has also been completed.

[1]This word count was computed using TeXcount: `http://app.uio.no/ifi/texcount/index.html`

Special Difficulties

None.

Declaration

I, Emily Fox of Churchill 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

1 Introduction **15**

1.1 Project Overview . . . 15

1.1.1 Project Aims . . . 15

1.1.2 Success Criteria . . . 16

1.2 Algorithmic Composition Overview . . . 16

1.2.1 Stochastic Processes: Markov Chains . . . 17

1.2.2 Artificial Intelligence: Neural Networks . . . 17

1.2.3 Evolutionary Techniques: Genetic Algorithms . . . 17

1.2.4 Knowledge and Rule-Based Techniques . . . 17

1.2.5 The Place of this Project . . . 18

2 Preparation **19**

2.1 Logic Programming . . . 19

2.1.1 MiniKanren . . . 19

2.2 Music Theory Background . . . 21

2.2.1 Representation of Notes . . . 21

2.2.2 Keys and Scales . . . 22

2.2.3 Chords . . . 23

2.3 Canons . . . 23

2.4 Deterministic Randomisation . . . 25

2.5 Defining Note Compatibility . . . 26

3 Implementation **27**

3.1 The Logic Approach . . . 27

3.2 Stages of Development . . . 27

3.2.1 Development Methodology . . . 28

3.2.2 Initial Approach: Bottom-up . . . 28

3.2.3 The Introduction of 'Variations' . . . 30

3.2.4 Unification in a Single Pass . . . 30

3.3 The Algorithm . . . 31

3.3.1 Choosing a Chord Progression . . . 32

3.3.2 Serialising the Chord Progression . . . 32

3.3.3 Rhythmic Variation . . . 33

3.4 My Implementation . . . 33

3.4.1 Modifications to MiniKanren . . . 33
3.4.2 User Interface . . . 36
3.4.3 Choosing Parameters . . . 37
3.4.4 Generating the Scale . . . 37
3.4.5 Generating the Chord Progression . . . 37
3.4.6 Generating the Variation Random Variables . . . 38
3.4.7 Creating the Internal Melody Structure . . . 38
3.4.8 Serialising the Chord Progression . . . 39
3.4.9 Adding Rhythmic Variation . . . 40
3.4.10 Exporting to Lilypond . . . 41
3.5 An Optimisation . . . 42

4 Evaluation **43**
4.1 Success Criteria . . . 43
4.1.1 First Criterion: Generates Canons . . . 43
4.1.2 Second Criterion: Properties . . . 45
4.1.3 Third Criterion: Interface . . . 48
4.1.4 Fourth Criterion: Usability . . . 48
4.2 Analysis of Findings from the User Study . . . 49
4.2.1 User Study Structure . . . 49
4.2.2 Results . . . 50
4.3 Timings . . . 53
4.3.1 Method of Evaluation . . . 53
4.3.2 Results . . . 53
4.4 Musicality of the Generated Pieces . . . 55
4.4.1 Method of Evaluation . . . 55
4.4.2 Results . . . 55
4.4.3 Significance of the Results . . . 58
4.5 Summary . . . 58

5 Conclusion **59**
5.1 Achievements . . . 59
5.2 Further Work . . . 60
5.3 Concluding Remarks . . . 60

Bibliography **60**

A User Study **63**
A.1 Instruction Booklet . . . 64
A.2 Consent and Questionnaire . . . 69
A.3 Results Data . . . 71
A.3.1 User Experience . . . 71
A.3.2 Numerical Scores . . . 72
A.4 Canons Generated . . . 73
A.4.1 Audio Files . . . 79

B Musicality Data **81**
B.1 Numerical Scores 81
B.2 Comments 81

C Project Proposal **87**

List of Figures

2.1 The adopted naming convention. 21
2.2 A stave with its features annotated. 22
2.3 A chromatic scale over two octaves, the first octave with sharps and the second with the equivalent flats. C major and minor scales, and D major and minor scales. Note that these are a subset of the chromatic, following the defining major and minor patterns. 23
2.4 A visual demonstration of how the notes of the major and minor keys relate to each other. It shows the gap between notes in semitones, where the notes are represented using the roman numeral for their position within the scale. 24
2.5 Constructing chords from the C major scale. Note that the final chord has the note order switched. 24
2.6 An example of (the opening of) a canon. Notice how the voices are identical but delayed in time. *Editor: Gustav Nottebohm (18171882); Publisher Info.: Wolfgang Amadeus Mozarts Werke, Serie VII: Lieder und Kanons, Bd.2, No.43 (pp.4) Leipzig: Breitkopf & Hrtel, 1877. Plate W.A.M. 230. Copyright: Public Domain.* 25

3.1 The original, bottom-up algorithm. 29
3.2 A flowchart of the general algorithm used. 31
3.3 A diagram of how a chord progression ensures note compatibility. 32
3.4 How chords are ‘serialised’. 32
3.5 A melody before and after adding rhythmic variation. 33
3.6 A flowchart of my implementation of the algorithm. 34
3.7 A diagram showing the internal representation of a melody. 39

4.1 Canon Creator meets the first success criterion. 44
4.2 Canon Creator exceeds the first success criterion. The second voice is an octave lower (shown by the bass clef and repositioning of notes) and has a single bar offset. 45
4.3 Canon Creator meets the second success criterion. 46
4.4 Canon Creator meets the second success criterion. 47
4.5 The average scores given by eight users when comparing creating canons using the two methods. 51
4.6 The average run times for some canon queries. 54

4.7 Overall scores for the musicality of the two classes of canons. The second graph excludes one of the participants' melody scores for the DIY canon because it is an anomaly (as evidenced by the comment claiming that it is a canon that 'no one would write' by the student marking the piece) and causes the variance to be very large. The DIY method does marginally better in the rhythm and melody categories, but worse in the canon category. In the second graph, DIY's overall score is actually marginally higher Canon Creator's, emphasising the fact that the scores are very similar. . . 56
4.8 The scores given to each canon for their 'suitablility as a canon', and the average for each class. 57

Acknowledgements

Acknowledgement and thanks go to the following people for their welcome contributions:

- Dr Samuel Aaron for supervising me on this project
- William Byrd for his help and advice with MiniKanren, particularly the development of the 'project' function
- Stefaan Himpe for his previous work upon which I built my algorithm[15]
- Dr Alan Blackwell for his advice regarding the user studies and evaluation
- Tom Ainge and Jonathan Mash for their help with analysing the canons created during the user study
- All the people who participated in the user study.

Chapter 1

Introduction

My project is concerned with algorithmic composition of music, within the specific domain of musical canons[24]. My chosen method of implementation (dubbed, 'Canon Creator') is based on logic programming and music theory techniques, the reasons for which are outlined in section 3.1.

I have successfully implemented an extension to Sonic Pi[4] which enables the user to generate a canon using a single line of code. Users may specify the properties of the canon manually, and Sonic Pi's deterministic randomness will choose those left unspecified, as well as generating the underlying canon. Therefore changing the seed changes the output and I have met all the success criteria for this project.

In addition, I implemented an extension enabling the canons to be exported to Lilypond[1] for typeset music notation, examples of which can be found in section 4 and appendix A.4 (the latter having been manually converted to a single file for conciseness).

1.1 Project Overview

1.1.1 Project Aims

The aim of this project was to make an extension for Sonic Pi that facilitates the creation of musical canons by non-technical users for inclusion in their Sonic Pi projects. The three main stages involved in designing the system were:

1. Solidifying the aims by defining the success criteria.
2. Developing an internal representation of a canon and associated constraints.
3. Designing the user interface.

1.1.2 Success Criteria

The project proposal (appendix C) contains the success criteria, that is the minimum features the software must exhibit to be deemed successful, as well as some useful additions. These are outlined below.

Non-negociable

A concise summary is given below, for the full criteria see appendix C.

- Canons can be generated, satisfying the requirements for a 'round'[28].
- Canon properties can be specified explicitly.
- An interface exists within Sonic Pi for using the software.
- Users are able to create canons using the extension.

These define the minimal aims of the project, and give a good foundation for further extensions.

Ideal characteristics

In practice, the timings of the software affect its success. Sonic Pi is used for live coding and it should be possible to incorporate canons generated into those compositions. Moreover, Sonic Pi enforces temporal semantics[5] to ensure that threads do not get out of time with each other (creating audible discrepancies in rhythm). A long computation time would violate these semantics and so the thread running the software would be killed with a timing exception. For these reasons, the computation time should be well within the *schedule ahead time* of Sonic Pi, which by default is 0.5 seconds.

It is also preferable that the generated pieces are 'musically interesting'. I define this as having no more than four consecutive notes of the same pitch, nor all of the beats having the same rhythm. This is a minimum requirement however, and where more detailed analysis is required I use the judgments of music students. The caveat to this second approach is its subjectivity, although the nature of western music means that to some extent this is unavoidable if meaningful measures are to be constructed.

1.2 Algorithmic Composition Overview

I will now give a brief overview of some of the work that has already been done within the field of algorithmic composition, and outline where this project fits with these.

1.2.1 Stochastic Processes: Markov Chains

Markov chains have had moderate success, particularly towards the beginning of research in this field[6]. In general, n-th order chains are used to give a model for music generation with the built in assumption that a note depends on its n predecessors. The transition properties can be learned from training on test pieces, or by hard coding them using music theory based trial and error.

Where this approach has worked well has been in generating melodies due to the local statistical properties they possess. However longer pieces tend to have little consistent structure throughout the piece because of the local, rather than global, dependencies. This can be mitigated by using high values of n, but this leads to a tendency to reproduce melodies found in the training set. For this reason Markov chains have had greater success providing a source of new material to aid human composers, rather than in generating full compositions[12].

1.2.2 Artificial Intelligence: Neural Networks

Various machine learning algorithms based on neural networks have been implemented, the first of which was implemented by Todd in 1989[22], with many other variations since[12]. This is a form of supervised learning where layers consisting of *artificial neurons* within the network learn their associated probabilities (are trained) using techniques such as back propagation to create a system that models the composition process.

This approach relies on a collection of data being available for training, that consists of compositions in the musical style being modeled.

1.2.3 Evolutionary Techniques: Genetic Algorithms

Essentially, genetic algorithms work by finding efficient ways to solve a search problem with an unstructured search space. Initially random solutions are generated and then combined in order to converge on an optimal one[17].

For music composition, a set of random melodies are generated (*motives*) or provided by the user, and then combined to make longer phrases and to generate countermelodies[8]. Measures are given for 'good' and 'bad' melodies to evaluate each motive. 'Good' ones are kept to be combined further whilst the 'bad' ones are abandoned, ensuring convergence on a final piece that equates to a 'good' solution.

1.2.4 Knowledge and Rule-Based Techniques

A family of algorithms have been explored which involves representing knowledge about musical composition as a set of structured symbols. In general, knowledge can either be hard coded into the program or learning techniques used to acquire it. The earliest well

known example of a system using these techniques was developed in 1958[14] by encoding the classical rules of counterpoint.

A popular realisation of this technique has been encoding musical knowledge as logic constraints and then generating pieces using constraint satisfaction approaches. For example, E. Morales and R. Morales worked on a technique that used inductive logic programming in order to find a *hypothesis* which generates the given example pieces. This is a combination of logic programming and machine learning[18].

More recently, *fuzzy logic*[30] and *probablisitic logic*[13] have been explored, because these reflect the 'soft' classification that exists in music due to its inherent subjectivity.

1.2.5 The Place of this Project

This project fits within the domain of knowledge and rule-based systems. I used logic programming for constructing the melodies with all the knowledge about musical structure built into the source code. My work differs to the first two examples given above because I do not employ any kind of machine learning, nor is there any evolution of solutions as is found in genetic algorithms. The rules and knowledge that built in is based on music theory, the relevant parts of which are outlined in section 2.2.

In addition to logic, I built on the deterministic randomness functionality in Sonic Pi to choose pitches and durations. In this sense, my solution acts a little like a Markov process with a single state for choosing the rhythms of each beat, and the pitches undergo a similar process except with the added constraints given by the harmonies that they create. However, my approach differs greatly to the n-th order Markov model in that my model is not based on probabilistic dependencies between states but rather on random choice from within a constrained domain.

In the next section I will cover the foundations of music theory and logic programming which underpin the project.

Chapter 2

Preparation

First I will cover the logic programming concepts used in the project, including the role of MiniKanren. Then I will look at the music theory required to understand the algorithm, before discussing the use of deterministic randomness and why it is important here.

2.1 Logic Programming

Logic programs generally consist of statements expressed as formulae (called *constraints*), which can then be verified and/or substitutions found for *logic variables* to make these statements true[21]. In the case that the statements cannot all be satisfied the program is termed *inconsistent*. The computer uses rules of inference in order to determine which constraints hold and to find assignments that satisfy them. A program run will either return *SUCCESS*, with at least one substitution for the logic variables, or *FAIL* if none exist.

The only form of constraint that is relevant for this project is *unification*. A successful unification of two logic variables means that they map to the same underlying object, or have the same substitution in the current context. For example, a logic variable A can be unified with 3, *nil* or any other value in the program, but then could not be further unified with 4 if this contradicts the first unification, e.g. $3 \neq 4$.

The way in which the program searches for an answer depends on the language and implementation. Prolog[2] generally implements a depth first search with backtracking for example, whereas MiniKanren[10] chooses to operate on streams of data, sharing the time between them in some pre-specified proportion.

2.1.1 MiniKanren

MiniKanren is a domain specific logic programming language[23], and has been implemented in many different languages[10]. Its most prevalent implementation is written in Scheme where most of the research is active.

Sonic Pi is written in Ruby so a Ruby implementation was needed for this project. I used Sergey Pariev's implementation[19] since it was the most complete version available at the start of the project. However, the implementation was not adequate for my purposes so I had to modify it. I have since pushed my changes to the main repository, so they are now featured in the main Ruby implementation of MiniKanren[19]. These additions are detailed in section 3.4.1.

MiniKanren usage

Using MiniKanren in Ruby requires the following steps:

1. Creation of a MiniKanren block.
2. Creation of fresh variables.
3. Addition of constraints between the variables. `conde(arg1, arg2)` acts as logical *OR* while `all(arg1, arg2, arg3, ...)` corresponds to logical *AND*.
4. Evaluation of the query.

Below is an example program.

```ruby
require 'mini_kanren'

results = MiniKanren.exec do # Create the MiniKanren block.
  x, y, z = fresh(3)         # Create three new fresh variales.
  q = fresh                  # Create a new query variable.

  # Create an array of contraints.
  constraints = [
    conde(eq(x,y),eq(x,z)),  # x = y OR x = z.
    eq(y,5)                  # y = 5.
  ]

  # Run the query (unify q with the array containing x, y and z).
  run(q, eq(q, [x, y, z]), *constraints)
end

puts results

# Output: [[5, 5, "_.0"], ["_.0", 5, "_.0"]].
# i.e. either:
#  1) x = y = 5, z is unconstrained
#          OR
#  2) y = 5 AND x = z (unconstrained).
```

Listing 2.1: An example of a simple MiniKanren program.

Notation	Note	Octave	MIDI number
gb5	g flat	5	78
eb4	e flat	4	63
cs3	c sharp	3	49
fs5	f sharp	5	78
c3	c natural	3	48
d	d natural	4	62

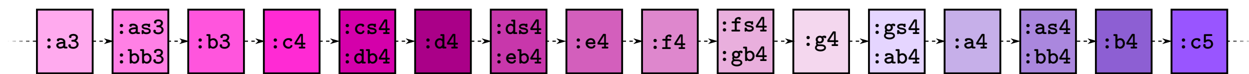


Figure 2.1: The adopted naming convention.

2.2 Music Theory Background

I will now outline the minimal amount of music theory needed for a sufficient understanding of this project.

2.2.1 Representation of Notes

Letter notation

Each *octave* contains every note, with the same notes in consecutive octaves found by doubling/halving the frequency. An octave has twelve notes, the frequencies having been split evenly to get the *equitempered*[7] scale. Each pair of consecutive notes is separated by a semitone.

Every note has a *pitch* (how high or low it sounds, related to its frequency) and a *duration* (how long it sounds for). The pitch can be represented by a letter name (from `a` to `g`), and then *natural*, *sharp* (raised a semitone, denoted `s`) or *flat* (a semitone lower, denoted `b`), where natural is assumed if not explicitly given.

To represent the octave, a number is written after the letter. The convention adopted here (as in Sonic Pi) is that octave boundaries occur at every C, with middle C being represented by `:c4`. This notation is summarised in figure 2.1.

MIDI notation

When dealing with computers, it is helpful to have a numerical representation. The standard for this is MIDI (Musical Instrument Digital Interface) which uses numbers ranging from 0 to 127, with each increment representing a semitone increase in pitch. Middle C corresponds to MIDI number 60.

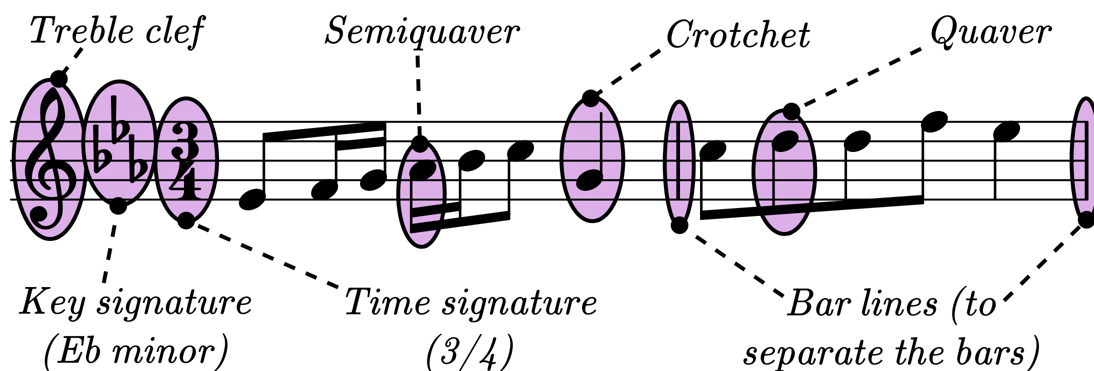


Figure 2.2: A stave with its features annotated.

Stave notation

The important concepts in stave notation are the *notes*, *staves*, *clefs*, *time signature* and *key signature* . Each is explained below.

- *Notes*: the circles sitting on/ in between the lines. The notes with no bar across are *crotchets* (quarter notes), those with a single bar are *quavers* (eighth notes) and those with two bars are *semiquavers* (sixteenth notes).
- *Stave*: the set of horizontal lines across the page. Each line represents a pitch and notes can go or in between the lines. They can also go above or below the five lines that are permanently there, and add extra (ledger) lines in order to do so.
- *Clefs*: the symbol at the beginning of each line specifying the pitch each horizontal line corresponds to. Roughly, the bass clef is lower and the treble clef is higher.
- *Time signature*: how many beats are in a bar, and how long those beats last. The ones we consider here are just 3/4 and 4/4, which mean *three crotchets*[1] *per bar* and *four crotchets per bar* respectively.
- *Key signature*: this specifies which notes are played sharp, or flat. The sharp/flat signs are in the position of the notes that are affected. See 2.2.2.

Figure 2.2 provides an annotated diagram.

2.2.2 Keys and Scales

Not every note is used in every piece; each piece of music has a *key* which determines the allowable notes. A key is specified by a note (the *tonic*) and a type (*major* or *minor*). The notes of the key make up a *scale* when played in order of pitch.

To build up the scale of a major key we start with the MIDI number of the tonic, call it x, then choose the notes $x, x+2, x+4, x+5, x+7, x+9, x+11$. For a (natural) minor key, the notes $x, x+2, x+3, x+5, x+7, x+8, x+10$ are selected. These patterns repeat

[1]Quarter notes.

Figure 2.3: A chromatic scale over two octaves, the first octave with sharps and the second with the equivalent flats. C major and minor scales, and D major and minor scales. Note that these are a subset of the chromatic, following the defining major and minor patterns.

in every octave. Figure 2.3 shows a chromatic scale (every note included) and then scales in the keys of C and D major and minor, using stave notation for comparison.

Note that the major and minors scales are the same sequence of notes with different starting positions, resulting in a cyclical pattern. Figure 2.4 demonstrates this.

2.2.3 Chords

Any group of notes is a *chord*. Technically any combination of notes will produce a chord, however music theory defines some rules to create standard ones.

A chord is defined by a number, n, and a key. The lowest note of a chord is the n-th note of the key's scale. The second and third notes are then the $(n+2)$-th and $(n+4)$-th notes of that scale. This is shown in figure 2.5.

The chords are named using the roman numeral representation for the number, n, required. This project only deals with the chords `I`, `IV`, `V` and `VI`.

2.3 Canons

A canon is a piece of music with multiple voices playing the same melody (or variations of a melody) starting at different offsets in time. Figure 2.6 has an example.

Some of the most famous canons were written by Bach in the early 1700s, constructed using the rules of counterpoint[25]. The canons in this project are different insofar as they are not concerned with counterpoint; the ones developed here contemporary in style with fewer rigid constraints.

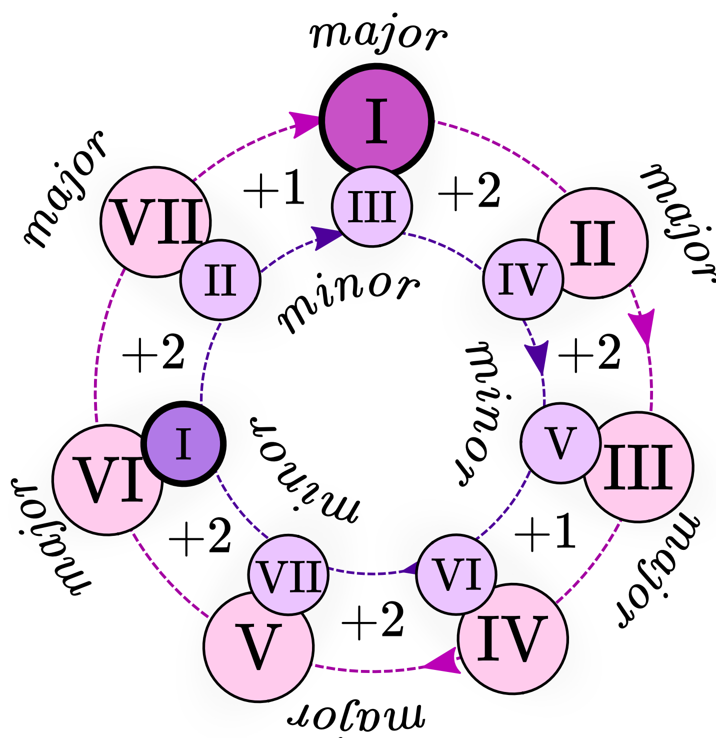


Figure 2.4: A visual demonstration of how the notes of the major and minor keys relate to each other. It shows the gap between notes in semitones, where the notes are represented using the roman numeral for their position within the scale.

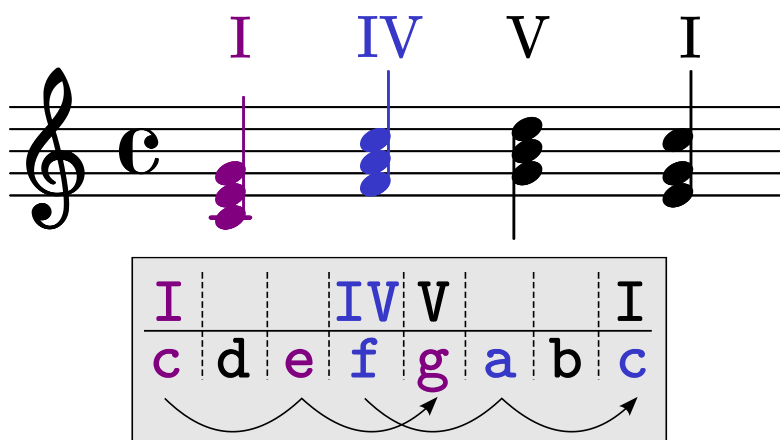


Figure 2.5: Constructing chords from the C major scale. Note that the final chord has the note order switched.

Figure 2.6: An example of (the opening of) a canon. Notice how the voices are identical but delayed in time. *Editor: Gustav Nottebohm (18171882); Publisher Info.: Wolfgang Amadeus Mozarts Werke, Serie VII: Lieder und Kanons, Bd.2, No.43 (pp.4) Leipzig: Breitkopf & Hrtel, 1877. Plate W.A.M. 230. Copyright: Public Domain.*

The types of canon this project is concerned with are:

- Rounds: Identical voices, each starting at some offset from the previous one.
- Crab: Has the additional property that the melody plays against itself in retrograde (backwards) at least once during the piece.
- Palindrome: Sounds the same whether played backwards or forwards.

2.4 Deterministic Randomisation

An important feature of Sonic Pi is its ability to exhibit deterministically random behaviour. Its utility in this context comes from the fact that it allows reproducibility of the canons if a user finds one they like. Since the output of Canon Creator is aural only (unless it's exported to Lilypond, but even then it cannot be imported back into Sonic Pi to be played again[2]), there would be no way for them to recreate any melodies they like again if it were truly random.

The seed value can be set using Sonic Pi's inbuilt function, `use_random_seed n`. This dictates the starting point for generation of pseudo-random numbers, and therefore fully constrains what all the calls to the random functions (`.choose`, `.rand` etc.) will return. Since these functions are used in a sequential order dependent on the properties that must be generated randomly at the start, there is no correlation in a musical sense between the canons generated by Canon Creator with different seeds. In other words, there is no way of fine tuning a canon based on its seed.

[2]This is a possible unimplemented extension, that would allow arbitrary pieces of music available as Lilypond source to be directly imported into Sonic Pi.

2.5 Defining Note Compatibility

For the purposes of this project, I define two notes to be compatible if they are in the same chord. This means that they will be complementary (*consonant*) rather than clashing (*dissonant*) and music theory says that that will sound good together in isolation. This convention is adopted from now on.

In the next chapter I explain the implementation of Canon Creator, and justify my approach.

Chapter 3

Implementation

This chapter firstly discusses why I chose to use logic programming, including some of successes and shortfalls of the various stages during the project's development. I then explain the modifications I made to the MiniKanren implementation to make it suitable for purpose before explaining the final algorithm and my implementation in more detail.

3.1 The Logic Approach

While it is entirely possible to write a solution to this task that does not involve logic programming (see S. Himpe's Python implementation for an example[16]), there are a few of features of the problem that lend themselves to a logic approach.

Firstly, the music to be generated has key features that can be well defined (see 2.5). This means that what it means for the *whole piece* to conform to these rules can be built up using simple constraints on the logic variables representing notes, which is a neat way of viewing the problem. Similarly, the structure of the canon is well defined, again making finding notes to constrain both possible and concisely expressible.

Moreover, the nature of logic programming means that many solutions can be generated from one set of constraints. While this has not been fully utilised in this project, it is easy to see where this might be beneficial. For example, multiple solutions could be generated before being auditioned and assessed against some heuristic that is a measure of how good that piece is to ensure that better results are returned more often.

3.2 Stages of Development

In this section I explain my development methodology and then examine the approach used in some key iterations of the project.

3.2.1 Development Methodology

When starting the project there were many unknowns because I had very little experience with any of the technologies that I was using, including having to learn both Ruby and MiniKanren from scratch. Nor did I have a precise version of the algorithm to work with as I was inventing the algorithm myself. For these reasons, I adopted an agile methodology[11] to make sure that issues were exposed early on before too much time had been invested in an algorithm that could not deliver.

I divided the work into two-week intervals and planned the work accordingly, taking into account vacations and other commitments. I included some scheduled catch up time to allow flexibility. These measures turned out to be very useful because, especially near the start of the project, there were some issues with the algorithm that led to developing a completely different approach. This extra scheduled time meant that these issues did not have a major detrimental impact on project progress.

3.2.2 Initial Approach: Bottom-up

The approach I had initially suggested is summarised by figure 3.1. The idea was to start with a list of logic variables representing notes and then to constrain them in duration and pitch to ensure that overlapping ones were compatible.

This approach did yield some canons- thousands could be generated on each run- however they consisted of mostly the same note, certainly failing the musicality test in 1.1.2. While it would have been possible to add more constraints to prevent this, it would in practice amount to over constraining the piece and infeasible run times due to its 'generate-and-test' nature on the large search space.

As it was, this approach was very tricky to code in such a way that it terminated in reasonable time ($<$ 1hr). The inefficiencies arose from dealing with the arithmetic of durations, and the unconstrained domain of the pitches. Solutions had to be generated, their variables 'projected' (their value found in the current substitution) and then discarded if overlapping notes were not compatible. Where this did terminate, it led to lots of repetition from all variables being constrained in the same way and evaluated in the same order.

Under this paradigm, all the possible pitches for a note are tried in turn in the order given by the constraint. Even with only eight possible pitches (one octave), this means a *lot* of options must be tried before an overall substitution is found satisfying the constraints.

At this point I searched for existing algorithms and discovered Stefaan Himpe's work[15]. He had overcome this problem by building in more knowledge of music theory from the start- a more top-down approach- and so I adapted this for use with logic programming. His algorithm is described in 3.3.

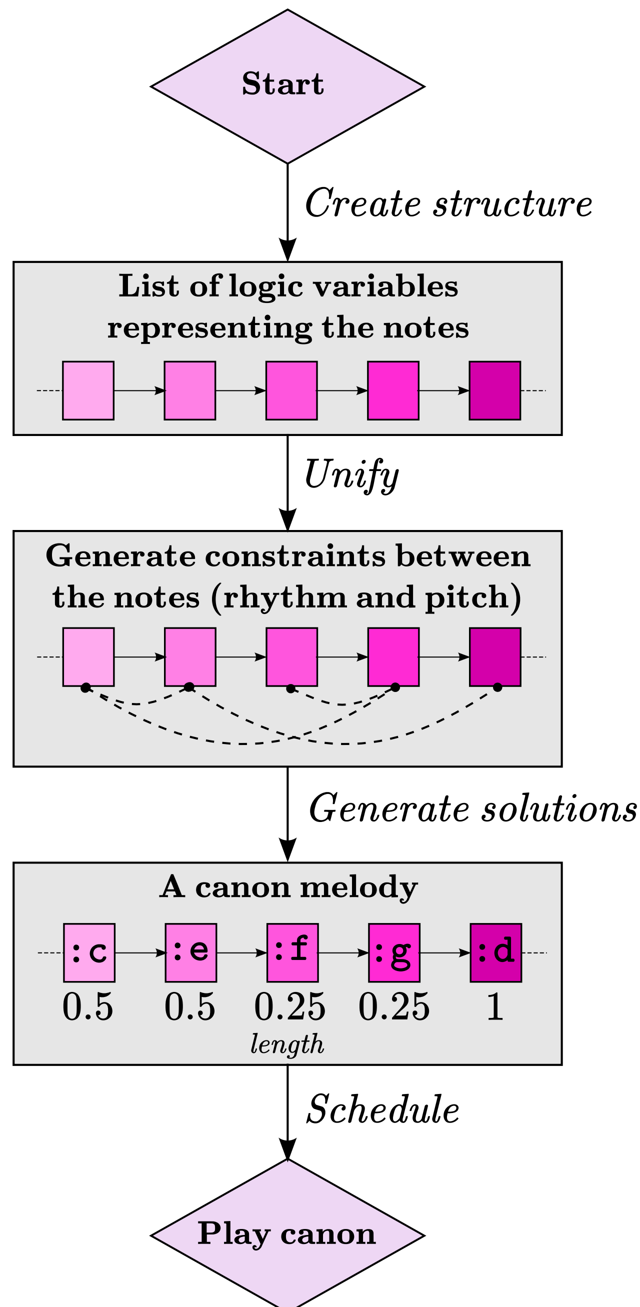


Figure 3.1: The original, bottom-up algorithm.

3.2.3 The Introduction of 'Variations'

When I first successfully generated full length canons, the results suffered from the same limitations that the Markov chain models do with too small a value of n (see 1.2.1). That is, the piece appears to wander aimlessly over time because there are no dependencies between the separate parts of the music. To fix this, I introduced the idea of *variations* which were simply melodies the length of the chord progression which could be combined to make a complete piece.

I populated an array of variations (themselves a 2D array of bars and beats) such that each pair worked together (and also in reverse for crab and palindrome canons). I then scheduled them sequentially by randomly picking ones, alternating backwards and forwards for the crab canons or mirroring each half of the piece for the palindromes. This successfully generated canons based on those variations, but only when the number of variations was low (about three) and the piece was short- no more than a few bars. In addition, Sonic Pi's schedule ahead time had to be increased dramatically in order to get any results.

The main reason for this bad scaling was the requirement that all variations had to work with every other one rather than just the neighboring bars- this is order n^2 in the number of variations. Moreover, the constraint that the final note of each must be the tonic meant that a very large range of notes was needed to end on a different tonic for every variation. This is neither feasible nor encouraged musically for the number of variations required for a piece that is not just the same melody played with itself multiple times.

To fix this I could have set aside a variation for the final bar, required to end on the tonic, while the rest need not follow this pattern. However, I decided that a better solution would be to use a variation model that only affects rhythms rather than pitch, and so I implemented the version that appears in the final version, explained in section 3.4.6.

3.2.4 Unification in a Single Pass

The final version of Canon Creator does two passes of MiniKanren; one to associate notes with beats and another to add rhythmic variation. It occurred to me as I was implementing the crab and palindrome functionality that a lot of the control flows were being duplicated and so it could all be done at once. Potentially then, more solutions could be found because more backtracking would be possible. A possible disadvantage though is that it could increase the run time since all the variables representing the root notes would have to be projected out each time they were needed. More analysis would be needed to conclude whether this trade-off is worth making.

I only considered this approach quite late on in the project and since it involved a refactor of the majority of my code there was no time to implement it, however if I were to do this project again I would try this approach first since it would conceptually make more sense than the current two-pass model.

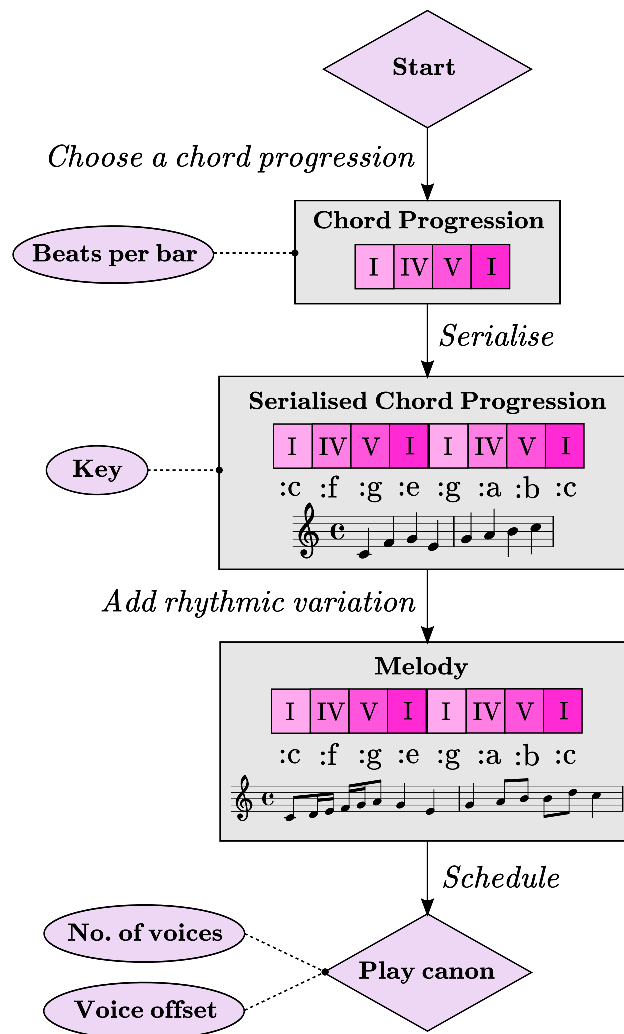


Figure 3.2: A flowchart of the general algorithm used.

3.3 The Algorithm

This algorithm was originally used and designed by Stefaan Himpe who outlines it in his blog[15] and I used his ideas to build my own solution. The main stages of the algorithm are:

1. Choose a chord progression.
2. Serialise the chord progression.
3. Add rhythmic variation.

This general procedure is shown in figure 3.2 and this section explores each stage in more detail.

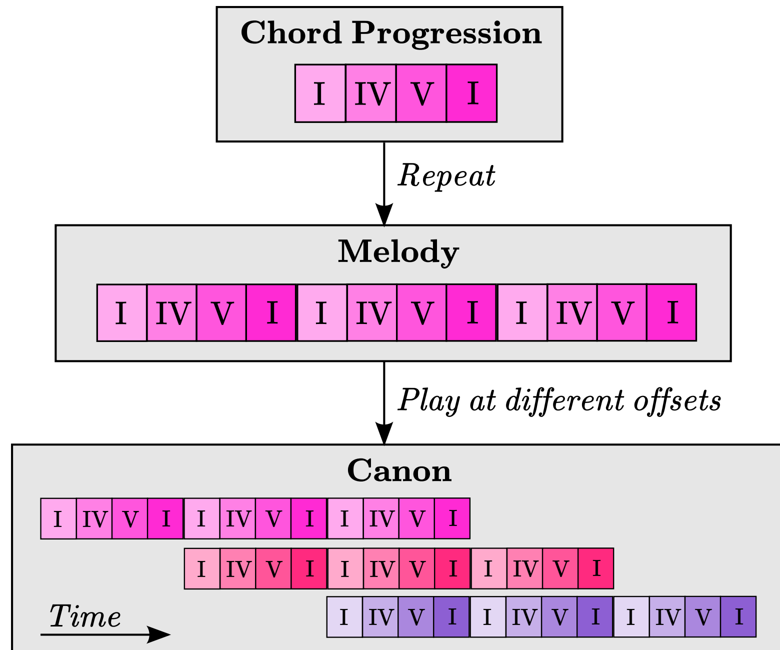


Figure 3.3: A diagram of how a chord progression ensures note compatibility.

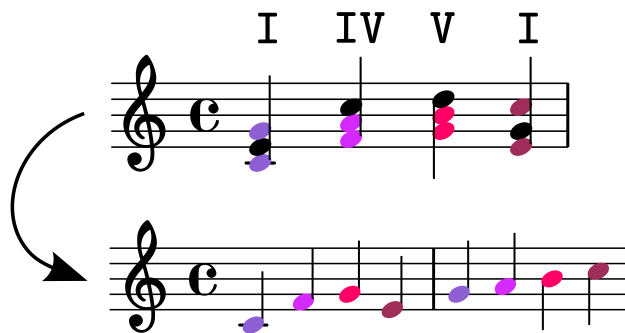


Figure 3.4: How chords are 'serialised'.

3.3.1 Choosing a Chord Progression

The chord progression associates a chord with each beat and is repeated throughout the piece. It must be some integer number of bars in length so that starting voices only after some integer number of chord progressions means that overlapping beats have the same associated chord associated, as shown in figure 3.3. The only requirement on the chord progression is that it ends with `V, I` which is a perfect cadence, meaning that the piece will sound 'finished' at the end of the piece.

3.3.2 Serialising the Chord Progression

Once every beat has an associated chord, a single note from the chord is chosen to be the *root note* of that beat. This can be described as serialising from the way that the notes in the stacked chord are spread out between bars (see figure 3.4).

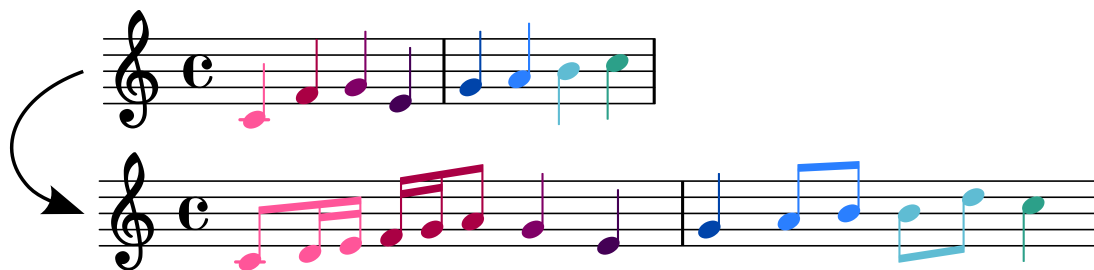


Figure 3.5: A melody before and after adding rhythmic variation.

3.3.3 Rhythmic Variation

At this point, a valid canon could be made by playing the root note of each beat for its whole duration, however this is certainly not rhythmically interesting. This final stage instead takes each beat and splits it into smaller individual notes to add some rhythmic variation (see figure 3.5). The first note of the group is the root note, and the others are chosen with some procedure for finding compatible notes.

3.4 My Implementation

The specific stages that my implementation follows are:

1. Choose parameters for the canon (done by the user and/or the software).
2. Find the scale for the piece.
3. Generate a chord progression.
4. Generate random variables to control the variation.
5. Generate the internal structure of the melody, i.e. an array of bars with arrays of beats inside each one (figure 3.7).
6. Serialise the chord progression by finding a root note for each beat.
7. Add rhythmic variation.

The first stage is done using getters and setters within a `Metadata` class that I created to represent the parameters associated with a canon, while the rest are done within the constructor of a `Canon` class which handles actual canon generation. In the rest of this section I first explain the MiniKanren modifications, and then explore each stage in more detail. A diagramatic version of my implementation is given in figure 3.6

3.4.1 Modifications to MiniKanren

The additional functionality that I required can be summarised as follows:

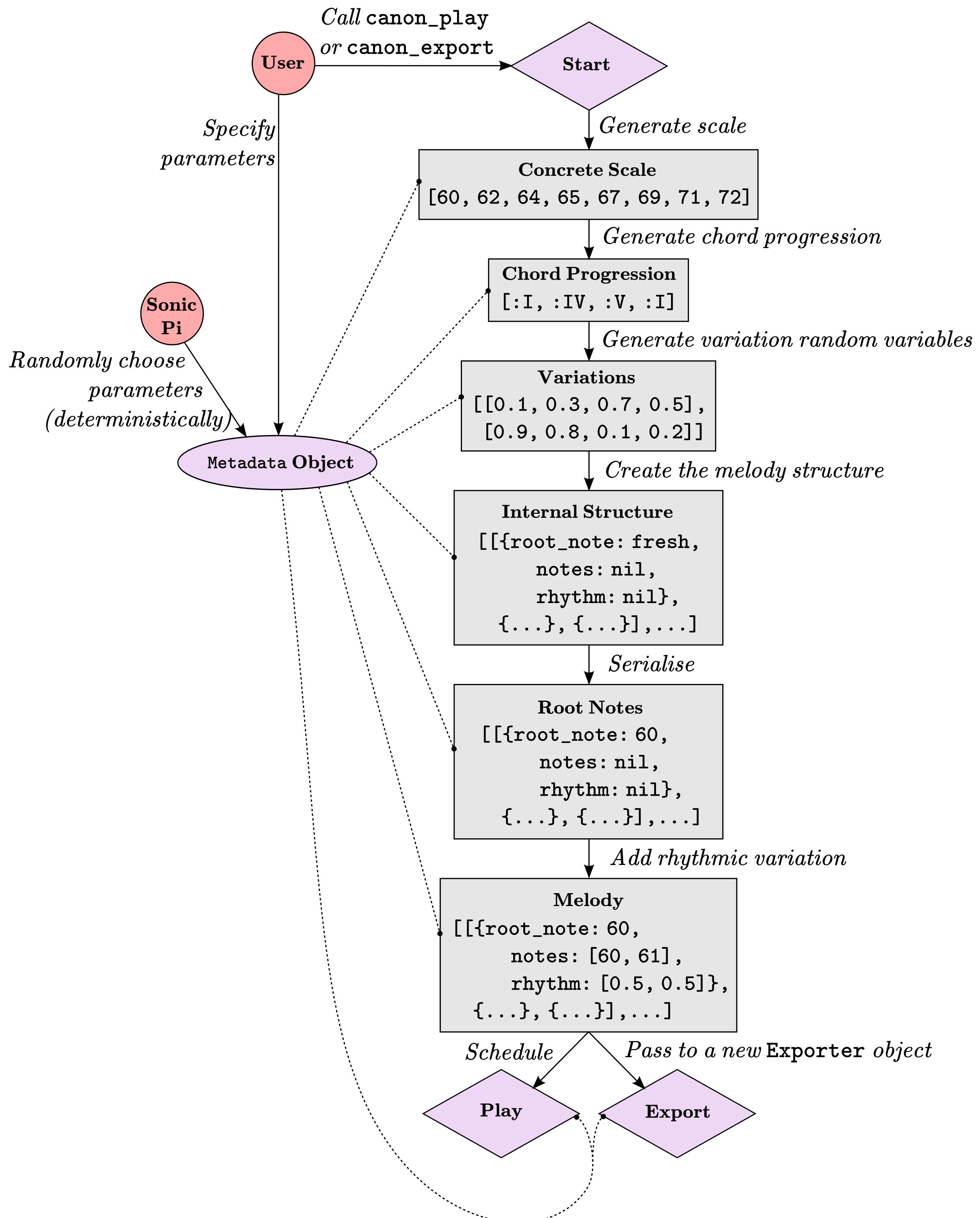


Figure 3.6: A flowchart of my implementation of the algorithm.

- A method that retrieves the value of a logic variable under the current substitution ('project').
- Full implementation of hash map unification.

The need to modify MiniKanren was unexpected and so caused a slight delay in progress until I had familiarised myself enough with the implementation to be able to make the changes. The modifications themselves are outlined below. They were later merged into the original repository[19], including some unit tests I devised.

'Project'

'Project' is a constraint function that takes a logic variable and finds its value under the current substitution, before using it in another constraint function. For example, I used it to find the actual values of overlapping notes by projecting the logic variables concerned.

William Byrd[3] (co-creator of MiniKanren) both explained the motivation for 'project' and helped to implement it in Ruby. It works by returning a lambda function (the form of a constraint in Ruby's MiniKanren) which:

1. Takes the current substitution as a parameter.
2. 'Walks' the variable that's being projected (finds the value under that substitution).
3. Calls the lambda function that it has been passed, passing in the projected variable to return a new constraint function.
4. Calls that new constraint function with the current substitution, to modify the substitution as appropriate.

```
1 # project(x, lambda { |x| eq(q, x + x) })
2 def project(u, block)
3   lambda do |s|
4     walked_u = walk_all(u, s)
5     g = block.call(walked_u)
6     g.call(s)
7   end
8 end
```

Listing 3.1: The 'project' function.

Hash map compatibility

This implementation of MiniKanren correctly worked for instances of arrays, but not hash maps. The functionality that was missing was the ability to recursively unify elements of the hash. Therefore, I had to:

- Add checks within the `unify` method to detect when both items are hash maps, and if so to call unify recursively on each of their elements.
- Modify the `occurs_check` method to detect occurrences of a circular structure within hashes.
- Modify the `reify_s` and `walk_all` methods to call themselves recursively on each element of a hash.

3.4.2 User Interface

I chose to use a *fluent interface*[20] which is an object orientated model where *method chaining* is used to try and improve code readability. This is important for Canon Creator because the users will be non-expert so they need as clear and concise an interface as possible. Method chaining is where each setter method returns 'self', so that multiple setters can be used in a row (*cascaded*). This leads to an interface that looks like:

```
use_random_seed 202
canon_play(canon.type(:crab)
                .key_type(:minor)
                .number_of_voices(2)
                .number_of_bars(5))
```

Conceptually, a canon is represented simply by the keyword `canon` and then it is played using the `canon_play` function. Properties can be specified by chaining together these method setters. In the above example, the number of bars is set to five by using `.number_of_bars(5)`. As many properties as the user wants to explicitly control can be chained in any order, arbitrarily many times.

To prevent the user having to deal with programming paradigms like constructors, I used `define` to create Sonic Pi functions that hide some of the syntax. This extra interface code is stored in `interface.rb`.

A simple example of this is allowing a `Metadata` object to be created using `canon` instead of `Metadata.new()`, which abstracts away the whole concept of metadata from the user and allows them to keep this conceptual model of operating on canon directly. The code to do this is:

```
1 define :canon do
2   return Metadata.new()
3 end
```

Listing 3.2: Abstracting away the underlying representation using Sonic Pi's 'define' function.

3.4.3 Choosing Parameters

A `Metadata` class contains all the properties of the canon in a hash map. It has setter methods for each property which the users interact with, while the getters are for internal use only. I created this as a separate class rather than having the user deal directly a hash map so that I could add validation easily, and implement the fluid interface. Moreover, any properties that are not specified by the user have to be randomly generated, and I could delegate this to the `Metadata` class to give a clean, modular design; the `Canon` class can now assume that all properties have a (valid) value.

When a getter is called but there is no value for that property in the hash, the property is assigned a default value and returned. Some are chosen at random from a list with `[option1, option2, ...].choose` while others are always the same. Properties left unspecified by the user will be generated only when its associated getter method is called. Appendix A.1 lists all the properties and their defaults, except `.repeat` which was introduced later.

3.4.4 Generating the Scale

The scale is a member variable of the `Canon` class, stored as an array. It must be stored with the notes in ascending order for the subsequent logic to work. Sonic Pi has a `Scale` class built in which can find the notes in a specific scale[1], so the only additional work was finding the part of the scale that spans the specified range and storing that as an array.

The stages for this are:

1. Find the *highest tonic* that is *lower* than (or equal to) the lowest note allowed.
2. Find the *lowest tonic* note that is *higher* than (or equal to) the highest note allowed.
3. Find the number of octaves between 1 and 2.
4. Instantiate a new `Scale` object, passing in the lower tonic and the number of octaves required. This generates the correct scale for that number of octaves.
5. Convert the scale to an array and remove notes not within the required range using `delete_if(outside_range)`. (Where `outside_range` is a lambda function returning true when the note falls outside the range.)

3.4.5 Generating the Chord Progression

The chord progression is an array of length $b * o$, where b is the number of beats per bar, and o is the offset (number of bars before the next voice starts). Every position in the

[1]This handles the logic concerned with generating scales outlined in section 2.2.2.

array is assigned a chord at random using `[I, IV, V, VI].choose`, with the exception of the final two which are always set to be `V` and `I` in order to get a perfect cadence.

Modifications for crabs/ palindromes

The user may choose to create a crab or palindrome canon using the `.type` method. In this case there is the extra requirement that the chord progression be symmetric so that it can be reversed to get the same chord sequence. Therefore the first two chords are chosen to be `I` and `V` (an inverse perfect cadence), then chords are chosen at random up to the half way point. The second half of the array is assigned to mirror the first half.

3.4.6 Generating the Variation Random Variables

In order to get some continuity through a longer piece, I introduced a concept of 'variation'. The percentage variation can be specified on a per canon basis using `.variation(percent)`. This controls how many different rhythms there are for bars in the piece[2], as a percentage of the total bars.

Variations are arrays with a length equal to the number of beats in a bar, with a random number in the range $[0, 1]$ generated for each beat using `rand`. This is used to choose how to split the beat into notes later on (see 3.4.9).

3.4.7 Creating the Internal Melody Structure

The internal representation of a melody is an array of bars with an array of beats at each index (see figure 3.7). Each beat is represented by a hash map storing the root note, notes inside the beat, and the corresponding note durations. A complete beat would look something like:

```
{ root_note: :c5,
  notes: [:c5, :e5],
  rhythm: [Rational(1/2), Rational(1/2)] }
```

This example shows a beat containing two quavers with pitches `:c5` and `:e5` respectively. The rational class was originally used for durations to allow triplets to be precisely expressed, but this functionality was subsequently removed after it resulted in rather chaotic pieces.

This structure is created within a MiniKanren block whose purpose is to assign a root note to each beat, so at this stage the notes and rhythm values are set to nil while the

[2]Not strictly true; it actually specifies only how many notes each beat is split into. In particular, two bars might be generated from the same variation, but one of them has a beat split into two semiquavers followed by a quaver while the other splits the same beat into a quaver followed by two semiquavers. As an approximation this is adequate.

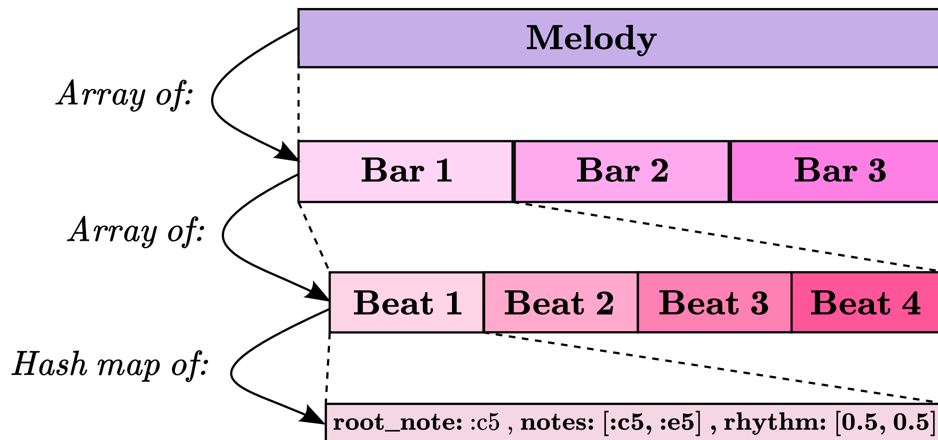


Figure 3.7: A diagram showing the internal representation of a melody.

root note is made into a fresh variable ready to be unified. All the information about the structure is provided by the `Metadata` object (number of bars, number of beats per bar, etc.).

3.4.8 Serialising the Chord Progression

Assigning a root note for each beat is done by working backwards through the piece, constraining the final root note to be a tonic note and then adding a constraint for each beat's root note that encapsulates the following:

- It is a note from the chord dictated by the chord progression to ensure compatibility (see 2.5).
- It is not the same as any overlapping beats. This adds harmony and minimises the chance of notes generated in the rhythmic variation stage clashing.

In order to implement this I wrote methods to:

- Return all the notes in a given chord.
- Find the logic variables representing notes that overlap with a given beat.
- Generate and add the constraint that the current beat's root note is:
  - In the chord
  - Not the same as an overlapping root note
  - Not too different in pitch from the next beat (this parameter is adjustable per canon using the `.max_jump` property).

  It does this by taking the variables that represent the overlapping, current and successor root notes as parameters and 'projecting' them to get their actual values. An array is then created containing every note in the chord, before discarding ones that don't meet the other two constraints using Ruby's `select` method.

Once these constraints have been added for every beat, the MiniKanren query is run and each root note is unified with an actual note. One of the results is chosen at random using `.choose` and is carried forward to the next part.

Modifications for crabs/ palindromes

For the crabs and palindromes this works slightly differently; for crab canons the final iteration of the chord progression is constrained as above, but then the penultimate one (and alternating ones before that) are instead constrained to be a mirrored version the one following it. For palindromes, constraints are added for the second half the piece as above, and then the first half is unified with the mirror image of the second half[3].

3.4.9 Adding Rhythmic Variation

In a new MiniKanren block, each beat goes through the following process to get a more musically interesting rhythm associated with it:

1. Assign fresh variables to the notes and rhythm properties of each beat.
2. Choose how many notes the beat will contain using the variations.
3. Add the constraint for the beat's rhythm.
4. Constrain the pitches of the notes.

I created four methods `transform_beat_single`, `transform_beat_double`, `transform_beat_triple` and `transform_beat_quadruple` that unify the logic variables representing the rhythm and notes with arrays containing the possible choices. A `conde` (*OR*) clause is used where there are multiple possibilities, and the arguments to these clauses are always shuffled with `.shuffle` to make sure that the solutions generated each time are random and do not always prefer a similar solution.

The logic variable for the notes is first unified with an array of n fresh variables, where n is the number of notes that this beat is splitting into. The first note note (and third, if it exists) are unified with the root note for that bar, while the other ones are each unified with a 'walking note'. I wrote a method which finds such notes by choosing one at random from between its adjacent notes[4].

An outer method `transform_beat` additionally takes in the value of a uniform random variable in order to decide which method to run on that beat, based on the probabilities of splitting into each number of notes (this is a parameter that the user can specify using `.probabilities`). The value of this random variable is provided by the variation values previously generated (see section 3.4.6) by running sequentially through them. The `transform_beat` method is run on each beat in turn from the last to the first.

[3]If the total number of chord progressions is odd, the odd one is dealt with separately and mirrored within itself.

[4]Or up to two notes outside that range to give more variation.

Finally, the MiniKanren query is run to get a completed canon.

Modifications for crabs/ palindromes

For crab and palindrome canons, the same adaptation is done as in 3.4.8.

3.4.10 Exporting to Lilypond

I created a new `Exporter` class which goes through the canon note by note converting the Sonic Pi representation into the Lilypond representation. The core methods I implemented were:

- `get_lilypond_note`: takes a MIDI number and returns the equivalent Lilypond note. The key signature is used to find the name of the note (resolving ambiguities from multiple notes mapping to the same MIDI number), then the octave number is calculated. The correct number of apostrophes or commas are appended to the note, which is how Lilypond represents octaves[5].
- `interpret_canon`: calls `add_bar` for each bar, which in turn calls `add_beat` for each beat.
- `add_beat`: appends the Lilypond representation of the note to the `$notes` array which contains all the Lilypond notes so far, in order.
- `convert_to_lilypond`: sets up the whole Lilypond environment by adding the staves and their associated information (such as the key and time signature) based on the information stored in the `Metadata` object. It then calls the above methods as appropriate, and finally writes it to file.

The interface

The interface for exporting a canon to Lilypond is provided by the `canon_export` method, defined in `interface.rb`. This provides validation and hides the underlying `Exporter` class and associated syntax.

As well as specifying the canon to play, a file path and name must be provided for the saved file. Additional parameters include the title, composer, whether to play the piece as well as export it, and the speed (in beats per minute). For example:

```
use_random_seed 202
canon_export(canon.type(:crab)
                .key_type(:minor)
```

[5]Apostrophes mean up an octave and commas mean down as octave. I used *absolute* mode for the pitch rather than *relative* so that the number required does not depend on the previous note, which would make the logic more complicated.

```
.number_of_voices(2)
.number_of_bars(5), "/home/emily/examplefile.ly",
"My Canon", "E. Fox",
true, 70)
```

3.5 An Optimisation

The original version of Canon Creator had MiniKanren generate forty results for each of the queries and then use `.choose` to pick a random one each time. However since all the `conde` constraints have their arguments shuffled before being evaluated anyway this is not necessary, and it is equivalent to generating a single solution. The difference this makes to the timings is detailed in section 4.3.

The next chapter evaluates the software based on the success criteria and the user study.

Chapter 4

Evaluation

This project meets all the success criteria outlined in the original proposal which I will verify in this chapter. Using the results from the user study I will discuss the extent to which the aims of the software have been met, and consider the additional specification outlined in section 1.1.2.

A note on the pieces used for evaluation

Since the internal representation of the canon is one that is created dynamically by calls to `play` and `sleep` within Sonic Pi, a concrete representation is needed for testing. For this I will present both the audio output recorded as a sound file, and the stave notation (provided by Lilypond).

4.1 Success Criteria

Each of the original success criteria are now given, along with a demonstration of completion and analysis of extra achievement (if applicable).

4.1.1 First Criterion: Generates Canons

Create a program which can generate canon melodies from given constraints. Specifically, the program must be able to output a melody (in some format, whether transcribed score or the internal data structure form) that satisfies the requirements of a 'round', which is a limited type of canon. Namely, it must:

1. *Consist of at least three voices* [1]

[1] During development I realised that a lot of canons only have two voices, therefore the default in my software is to specify two voices, and indeed this tends to give the best results, however it is possible to achieve the behaviour required here.

Criteria One

Canon Creator

Prophet

Pretty_bell

Beep

4

Audio: `https://goo.gl/hdscBa`

Figure 4.1: Canon Creator meets the first success criterion.

2. *Have all voices sing the same melody at the unison*
3. *Have each voice starting at different times.*

Method for verifying correctness

A piece must be exhibited that was generated by Canon Creator and has the listed properties.

Analysis

Figure 4.1 exhibits such a canon; it has three staves with a voice on each one, so property 1 has been met, all the staves have exactly the same sequence of notes, so property 2 is true and each voice starts two bars later, so property 3 has also been met.

Extensions

The final software is able to have the voices playing at the octave (i.e. some integer number of octaves apart) if the user wishes, although this is not default behaviour. This gives the user some extra flexibility and enables them to have one voice acting as a 'bass line' (i.e. lower in pitch), or a 'descant' (i.e. higher in pitch).

Criteria One, Extension

Canon Creator

Prophet

Pretty_bell

3

Audio: `https://goo.gl/AgwHIf`

Figure 4.2: Canon Creator exceeds the first success criterion. The second voice is an octave lower (shown by the bass clef and repositioning of notes) and has a single bar offset.

The time between the entrance of each voice can also be explicitly chosen (in number of bars) which allows a wider range of pieces to be generated. Both of these properties are exhibited in figure 4.2 with a piece generated using the options: `.voice_offset(1)` and `.voice_transpositions([0,-1])`.

4.1.2 Second Criterion: Properties

Be able to specify properties of the generated melody:

- *Number of parts*
- *Specified key*
- *Specified length*
- *Number of repeats*

and have the generated music conform to these.

Method for verifying correctness

Largely these criteria are correct by construction; the `play_canon` method calls the `play_melody` method the number of times specified in the `Metadata` object, for example. To demonstrate this I will choose two canons with a mix of these properties and will confirm that their outputs are consistent with what was specified.

Criteria Two, Specimen One

Canon Creator

Play 2 times

Prophet

Pretty_bell

4

7

10

Audio: `https://goo.gl/C4U8MV`

Figure 4.3: Canon Creator meets the second success criterion.

Analysis

First specimen

The first canon to be generated is:

```
canon.key_note(:g)
     .key_type(:major)
     .number_of_bars(10)
     .number_of_voices(2)
     .repeats(2)
```

By looking at and listening to the piece in figure 4.3 we can verify that the above properties hold; there are two staves representing two voices, a single sharp in the key signature as well as the piece starting and ending on a G, indicating G major, the number of filled bars in each voice is ten and there are instructions to play the melody twice. Therefore, all the desired properties hold of the output.

Criteria Two, Specimen Two

Canon Creator

Play 3 times

Saw

Prophet

Pretty_bell

4

7

Audio: `https://goo.gl/GKPXsS`

Figure 4.4: Canon Creator meets the second success criterion.

Second specimen

The second canon to be generated is:

```
canon.key_note(:eb)
     .key_type(:minor)
     .number_of_bars(5)
     .number_of_voices(3)
     .repeats(3)
```

By looking at and listening to the piece in figure 4.4 we can verify that the required properties hold; there are three staves representing three voices; there are five flats in the key signature as well as the piece starting and ending on an E flat, indicating Eb minor; the number of filled bars in each voice is five and there are instructions to play it three times as required.

Extensions

These demonstrate only a small subset of the options implemented but in the interests of space no more are included here. Appendix A.1 contains all the possible properties that can be specified by the user, with the exception of the repeat option (demonstrated here) which was only implemented after the user study.

4.1.3 Third Criterion: Interface

Interface this with Sonic Pi by introducing a well-defined way of combining available constraints in a call to the function.

Method for verifying correctness

To show this has been achieved, I will simply demonstrate the interface and show that it can be used within Sonic Pi.

Analysis

The method of interaction chosen for creating canons within Sonic Pi was one of chaining together method calls to set properties of the canon. This interface has been explained fully in 3.4.2.

The user study (see section 4.1.4) demonstrates the completion of this using the fact that users were able to generate canons using Canon Creator within Sonic Pi.

4.1.4 Fourth Criterion: Usability

Users are able to use Sonic Pi to generate canons using the new functionality.

Method for verifying correctness

I carried out a user study[2] where participants were required to create their own canons using two different methods. The first required hard coding the notes and their durations by the user, with code provided to handle playing it. The second method was Canon Creator itself.

Participants were required to fill in a questionnaire about their previous musical and computing experience and then asked to score how they found each method.

[2] In the original proposal I planned to do two user studies, however I later decided against this because the other one was superfluous in relation to the success criteria and project aims.

Clearly the only thing that needs to be shown to verify success here, is that the majority of participants managed to use Canon Creator to generate a canon. Beyond this however, I will analyse how beneficial Canon Creator is for creating canons when comparing it to the other method. This will consist of performing statistical tests on the data obtained from the user study.

Analysis

The basic criterion has been met. This is clearly true from the fact that of the eight users who took part in the user study, all were able to create a canon using Canon Creator. These are shown in appendix A.4.

Since the scope of this user study and the results obtained is so large, a more detailed assessment of this can be found in section 4.2.

4.2 Analysis of Findings from the User Study

This section contains an analysis of how far Canon Creator goes towards making canon creation a good experience for the user. It considers factors such as the ease of creation, their enjoyment, and the speed of creation. I will first explain the set up of the user study, then give a general overview of the scores for each method and what can be learned from these.

4.2.1 User Study Structure

The user study was carried out in order to gather information on how far Canon Creator succeeds in enabling users (perhaps non-skilled in music or computer science) to create their own canons within Sonic Pi. To get any meaningful results I had to compare the Canon Creator method with a base line, which I hereafter call the 'DIY method'.

DIY method

The chosen baseline is a method where the user must create their canon by specifying notes (pitch and duration) manually. All code was given to them except the melody itself, so they did not have to do any serious programming themselves except what was required to modify the template of the melody.

In order to ensure that the DIY method was measuring the same things as Canon Creator I added restrictions to the canons it could create. This included restricting the note durations and time signatures.

Study structure

The overall structure of the study was:

1. Fill in a questionnaire giving a rough idea of previous experience in music and programming.
2. Learn what a canon is, and the relevant music theory for this task.
3. Create a canon using the DIY method.
4. Create a canon using Canon Creator.
5. Fill in a questionnaire about each method by specifying level of agreement with a set of statements.

The aim of Canon Creator is to make it possible for even non-musical and non-technical people to be able to generate their own music within Sonic Pi, and to do this in a way that is enjoyable, intuitive, quick and fairly easy. Since Sonic Pi's main aim to to be educational, I also asked the users about this even though it was not a direct aim of Canon Creator.

The statements that were given to the users after they'd completed the two composition tasks were:

- *I understood quickly how to use this method to create canons.*
- *I found this approach intuitive and easy to use.*
- *I was able to create a canon(s) that I was happy with and that sounded nice to me.*
- *I found the experience frustrating.*
- *I found the experience enjoyable.*
- *I found the experience informative (i.e. I learned something new).*
- *It was quick to create new canons.*

I did one trial run of the study to refine it before carrying out an improved version on eight more users.

4.2.2 Results

A graphical summary of the results is given in figure 4.5. The overall score was taken by averaging each of the individual average scores (with the 'frustration' score being subtracted from six to give the more positive reaction a higher score whilst keeping the range from one to five).

Before commenting on the scores themselves, it is worth noting that in most cases Canon Creator's scores have a smaller variance than the DIY method, suggesting that the experience with Canon Creator was more uniform. The questionnaire revealed that the users

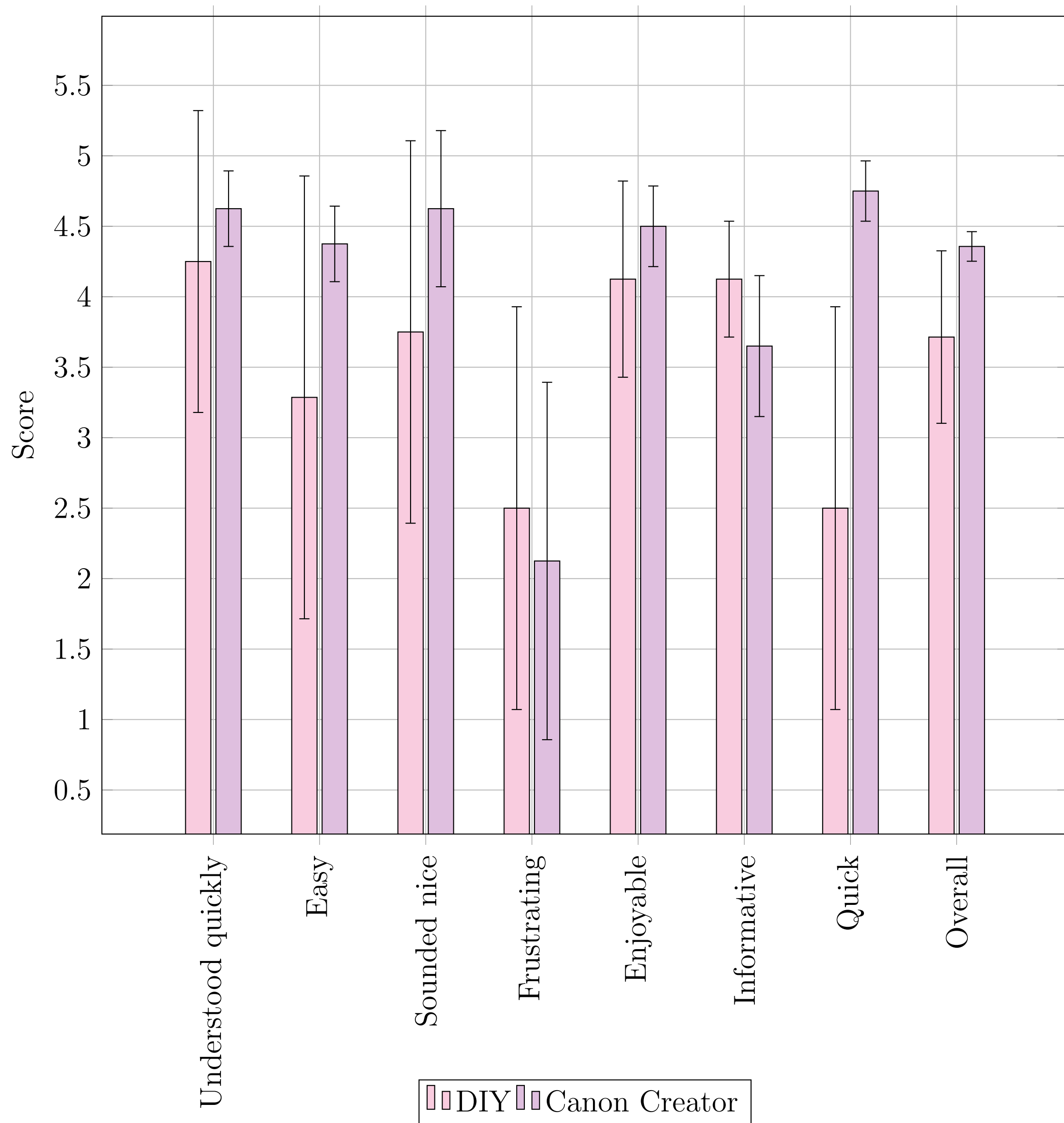


Figure 4.5: The average scores given by eight users when comparing creating canons using the two methods.

had varying levels of prior knowledge, so it is possible that this resulted in a wider range of experiences where more hard coding was required. On the other hand, Canon Creator did not expect as much input from the user and so created a more level playing field with fewer prerequisites.

It is evident from the graph that Canon Creator scores more favourably in most of the categories, including overall score, with the exception being how informative it is. In order to comment on whether the results are statistically significant, I will consider the value of the t-statistic for a two-tailed test in each category.

The following table shows the differences in means and the corresponding t-statistic. Negative values indicate that Canon Creator scored higher (numerically) than DIY, and vice versa. Note that one user filled in 'don't know' for ease of use, and therefore there is one fewer degree of freedom for two categories since their results were excluded from these.

Category	Difference in Means	t-statistic	Degrees of Freedom
Understood Quickly	-0.375	-1.426	7
Easy	-1.143	-1.922	6
Sounded Nice	-0.875	-1.594	7
Frustrating	0.375	0.574	7
Enjoyable	-0.375	-1.158	7
Quick	-2.25	-4.583	7
Informative	0.375	1.426	7
Overall	-0.694	-1.865	6

With six degrees of freedom a t-statistic of greater than 2.447 is significant, and with seven this drops to 2.365 (with $p < 0.05$[9]). Therefore, the only value significant at the five percent level is the speed of creation. This is understandable since Canon Creator abstracts away much of the complexity associated with composing the music. In fact, this difference is so pronounced that is is also significant at the $p < 0.01$ significance level. We can therefore confidently reject the null hypothesis that the methods do not affect the speed, and instead conclude that Canon Creator makes it quicker for users to create canons.

The next largest t-statistic is given for ease of use, but although this has a relatively high difference in mean of -1.143, the variance is also large which means that this difference has less meaning. This is not even significant at the ten percent level.

At twenty percent significance however, all the scores are significant except 'frustrating'. With the exception of the informative category, Canon Creator scores more highly in each of these; being understood more quickly, easier, sounding nicer, more enjoyable and getting a larger overall score[3]. It is understandable that the DIY method was deemed to be more informative; it requires more hard coding and therefore more understanding of

[3]The overall score statistic is rather arbitrary though since it assigns an equal weighting to all of the categories. Moreover, the selection of which to use in the first case was rather arbitrary- there are many others that could have also been chosen.

the material that is abstracted in Canon Creator. It was also done first so would have been where the greater learning curve was.

However a twenty percent significance with $p < 0.20$ is generally inadequate for rejection of the null hypothesis because there exists significant uncertainty. In order to get more conclusive evidence, I would have to test the program on more users to get more data and see whether there is any correlation in the scores.

4.3 Timings

As explained in 1.1.2, the timings are important to consider.

4.3.1 Method of Evaluation

I used the `Benchmark` class built in to Ruby to do some measurements of the timings. My code creates the `Metadata` object for a canon and then creates a canon from it, measuring how long this takes to complete. It does this 1000 times for each canon and then calculates the mean and variance in those results.

The measurements were made when creating the following canons:

- None: no properties specified.
- 1: C major, 3/4 with two voices.
- 2: C major, 3/4 with three voices.
- 3: C major, 3/4 with four voices.
- 4: Eb major, 4/4 with three voices.

4.3.2 Results

While conducting these tests I realised that the optimisation explained in 3.5 was possible. Figure 4.6 shows the run times with and without this optimisation.

The run times are much smaller than the upper limit of 0.5 seconds in both cases, although it is clearly still beneficial to take the option with the shorter times. Therefore, Canon Creator meets the ideal timing characteristics very comfortably.

Failures

The results from the timings also show how many times there failed to be a canon with the given properties to return at all. This is obviously not ideal behaviour because the user might not be sure why it happened and therefore not know how to fix it. In essence,

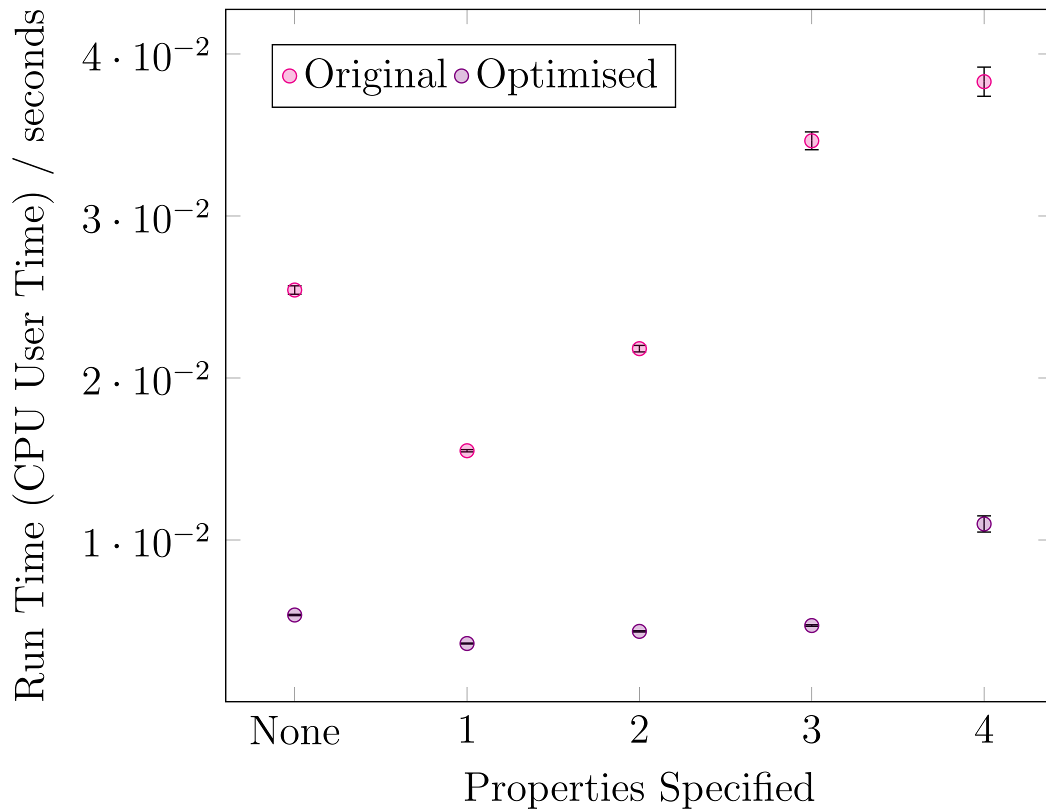


Figure 4.6: The average run times for some canon queries.

failures come from not having a big enough range of notes to be able to assign a root note to every beat that satisfies the given constraints. The user can fix this by increasing the range, increasing the maximum jump parameter, changing the range to a better one for that key, or reducing the number of voices.

Properties Specified	**Number of Failures**
None	27
1	0
2	0
3	0
4	2

An improvement here would be to improve the error reporting on this, and try to relax some constraints so that this happens less often. During the user study, a few of the participants experienced this problem which could have affected their scores.

4.4 Musicality of the Generated Pieces

4.4.1 Method of Evaluation

Firstly I evaluated the pieces created during the user study against the definition of musically interesting given in 1.1.2 and then asked some music students for scores for rhythm, melody, and suitability as a canon. I then used this data to compare the musicality of those canons created using each method. The aim is to see whether the pieces made using Canon Creator are more 'musical' than those created by hand by the same group of people.

Musicality is a subjective concept, so the results from this test are undoubtedly biased towards the opinions of the students who assessed the pieces, however should the distance between the two classes of canon be sufficiently large then conclusions can still be drawn. In order to reduce the bias from having multiple students assess the results, I ensured that they both did equal numbers of canons from each class to make sure that differences in the way they mark the pieces does not skew the results.

4.4.2 Results

Evaluating the pieces against the definition in 1.1.2 every piece passes the basic test, so this is not useful for discrimination.

The results from this study were interesting in that they suggest that there is little difference in the overall musicality of the canons created using the two different methods. A summary of the scores is given in figure 4.7.

Similar to what was observed in the user study, the variance in Canon Creator's scores is always smaller than the corresponding DIY ones. This supports the hypothesis introduced earlier that Canon Creator generates more consistent results and reduces the effect of differences in the users' personal level of expertise.

Although the DIY canons score marginally higher in terms of rhythm and melody, Canon Creator has higher scores for 'suitability as a canon' which causes it to get a total of four marks more than the DIY method (see figure 4.8). This suggests that the logic constraints were good for constraining the overlapping notes for the canon's harmony, but the very basic and random method I used to create the base melody was not conducive to well constructed pieces.

It should be noted that some of the criticisms the music students had with Canon Creator's canons are easily solvable and arose often from my ignorance of musical phrasing when building the software. For example, one comment that came up quite a lot was 'three and six bar phrases are unusual', which can be attributed to the fact that I set the default value for the number of bars to be double the number of beats in a bar, which is not necessarily a good thing for a piece with a 3/4 time signature.

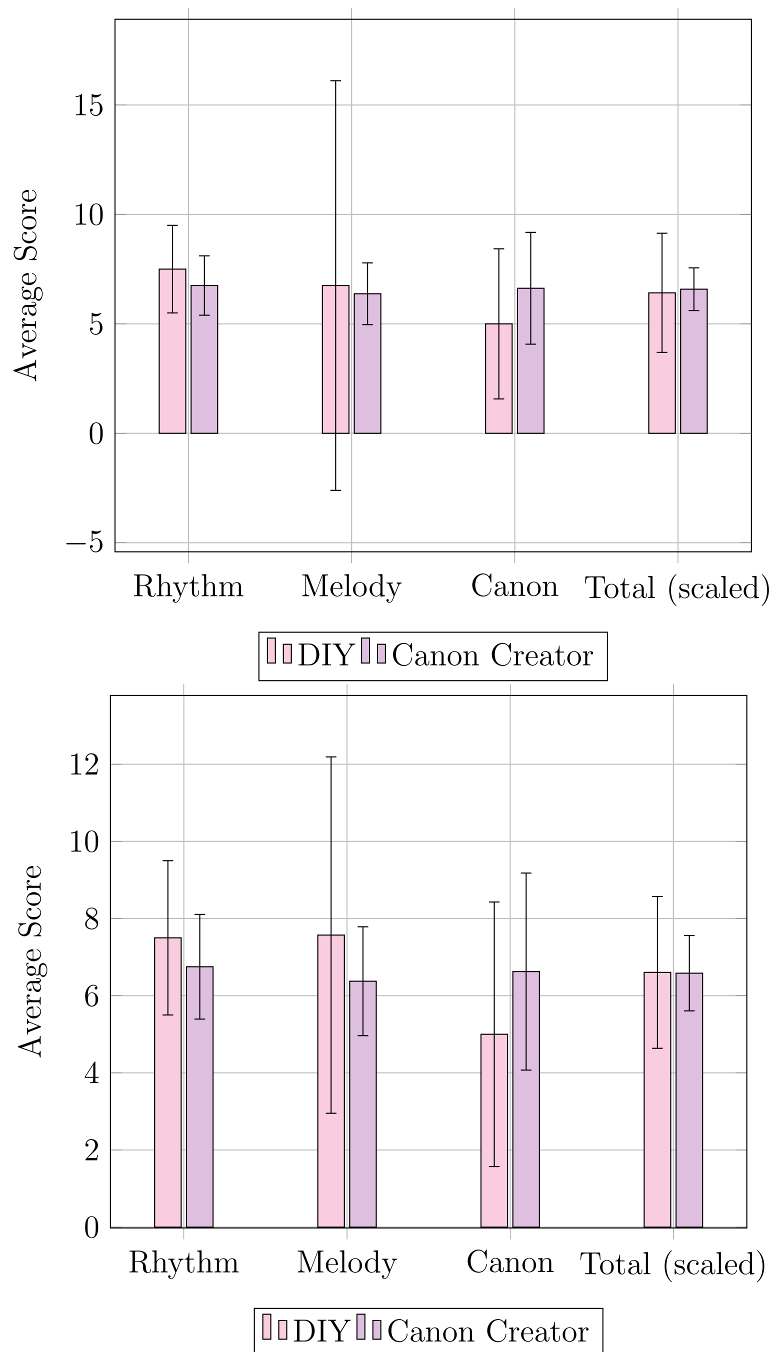


Figure 4.7: Overall scores for the musicality of the two classes of canons. The second graph excludes one of the participants' melody scores for the DIY canon because it is an anomaly (as evidenced by the comment claiming that it is a canon that 'no one would write' by the student marking the piece) and causes the variance to be very large. The DIY method does marginally better in the rhythm and melody categories, but worse in the canon category. In the second graph, DIY's overall score is actually marginally higher Canon Creator's, emphasising the fact that the scores are very similar.

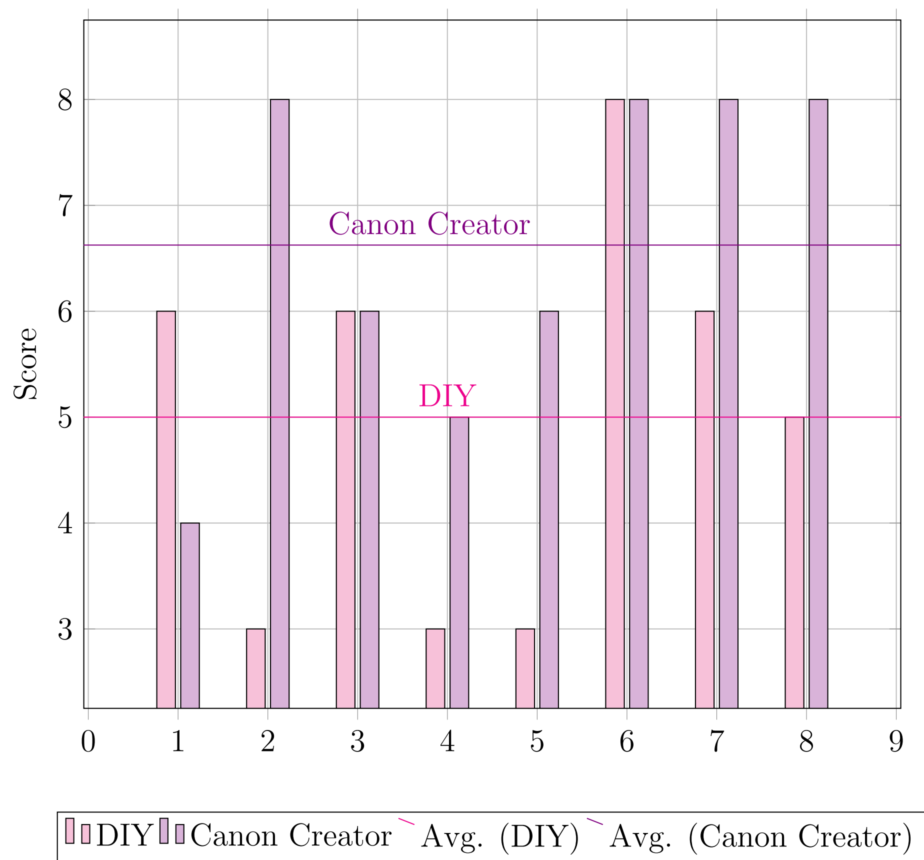


Figure 4.8: The scores given to each canon for their 'suitablility as a canon', and the average for each class.

Other common problems with the pieces made using Canon Creator was that there was no real sense of phrasing: 'Often feels quite random. Not really any sense of phrasing - feels like one long phrase that keeps going.' being the exemplar comment demonstrating this. I did realise this when developing the software, and in an attempt to improve this tried a different method using variations that I outlined in 3.2.3. The watered-down version I did implement was clearly not sufficient for achieving its aims, at least in its current form and in the pieces demonstrated.

4.4.3 Significance of the Results

The table below shows the t-statistic values for the difference between the means of the canons generated with each method (including the anomalous one since the data set is not large enough to discard it entirely).

Category	Difference in Means	t-statistic
Rhythm	0.75	1.158
Melody	0.375	0.406
Canon Suitability	-1.625	-2.089
Total	-0.5	-0.344

There are seven degrees of freedom, so the only significant result is the suitability as a canon, with $p < 0.10$. The conclusions outlined above are therefore limited in scope since the differences are small with a fairly large variance. In order to get more precision I would have to generate more data by doing the user study with more people. This was not feasible within the scope of this project because the resources of time, people and the music students to assess the results did not allow for more extensive study.

4.5 Summary

Canon Creator meets all of its success criteria, and adds extra functionality on top of what was initially required, such as specifying voices and octave offsets to use, as well as supporting exporting to Lilypond and generation of different types of canon. In general users found making canons using Canon Creator quicker than the manual method, and there is some evidence to suggest that they also found it easier too, although this is less certain. There was very little difference in the musicality of the pieces created using each method as judged by the music students, but Canon Creator generated canons with better canonic structure.

The next chapter concludes the project, summarising the achievements and suggesting further work.

Chapter 5

Conclusion

5.1 Achievements

I successfully implemented an extension to Sonic Pi that allows users to create musical canons with properties that they may specify themselves. The user study confirmed that even non-expert users were able to create a canon using this method, and generally found it quicker than doing it manually. Part of this success can be contributed to the fluid interface that I implemented which provided a simple and intuitive interface for the user.

I utilised the deterministic randomness already part of Sonic Pi in order to create a system that allowed reproducible results, whilst retaining the benefits of randomness for unique compositions. This has led to a system that can easily produce different results using random seeds. Previously, creating randomised music in Sonic Pi required the user to code it manually, but Canon Creator builds on top of provided functionality to automate the fine grained melody generation.

Two extensions were successfully developed for the application. The first one allows the user to export their pieces to Lilypond for typeset stave notation. This required a lot of code to deal with the information about the canon in order to find the correct notes and provide a mapping between the two representations. It provides exporting functionality for every key supported by Canon Creator, as well as allowing plenty of optional extra features such as including the title, composer and tempo in the exported file. In order to do this I had to familiarise myself with Lilypond and its relevant syntax.

The other extension completed was allowing crab and palindrome canons to be created. This involved adjusting the way in which constraints were generated depending on their position in the piece; some bars were unified with the bar they were mirroring instead of a new melody. These kind of constraints are more difficult for users to do on their own because the extra requirement of mirroring some/all of the melody making it harder to visualise, which makes it particularly welcome in this software.

5.2 Further Work

There are many improvements that could be made to the current system, as well as areas for further work in the general field.

A natural extension for the project would be to enable the creation of table canons[29], which have two voices playing the same melody together, but with one both reversed and inverted[1]. This is another one that it tricky for a human to compose because of the transformation to get the other half of the piece. It is conceivable that a modification to the existing palindrome code could be made which adds the additional constraint to the pitch of the notes that they work well with the inverted note, rather than the note itself.

Currently, the exporter creates a Lilypond file but the user must compile it themselves to create the PDF. This is not ideal for non-technical users and so a good addition would be some code that takes the exported code and automatically compiles it.

Thinking about the application to live coding, it would be useful to create a similar piece of software that simply creates short melodies rather than whole canons, for use in live coded compositions. Lots of the same techniques used here would apply, but could be changed so that the chord progression is provided explicitly as well as giving a bigger emphasis to the musicality of the melody in isolation.

Throughout the project it has become clear that in the absence of machine learning it is necessary to build in a lot of music theory. This was made especially apparent by the music students' comments about the pieces created during the user study (see section 4.4 and appendix B.2). If I were to build the next iteration of this project I would gather more advice from music students beforehand about the important aspects that need to be built in, like the chord inversions and use of dissonance which was largely ignored in this version. More research into the existing techniques introduced in chapter 1.2.4 would also be beneficial.

5.3 Concluding Remarks

I have enjoyed working on this project because of the way it combines both music and computer science. I have learned to write in a new language (Ruby), a new logic language (MiniKanren) and how to typeset music notation using Lilypond. Through the process I learned more about domain specific languages, and gained enough understanding of the MiniKanren implementation to be able to contribute to its development. I have also experimented with some techniques that could in the future be applied to expanding the current support for creating randomised music within Sonic Pi.

[1] i.e. played upside down.

Bibliography

[1] Lilypond... music notation for everyone. http://www.lilypond.org/.

[2] Prolog. https://en.wikipedia.org/wiki/Prolog.

[3] William E. Byrd. http://webyrd.net/.

[4] Dr Samuel Aaron. sonic-pi.net. http://sonic-pi.net/.

[5] Samuel Aaron, Dominic Orchard, and Alan F. Blackwell. Temporal semantics for a live coding language. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 37–47. ACM, 2014.

[6] C. Ames. The markov process as a compositional model: A survey and tutorial. *Leonardo*, 22(2):175–187, 1989.

[7] J. Murray Barbour. *Tuning and Temperament*. Michigan University Press, 1951.

[8] John A. Biles. Evolutionary music tutorial. http://igm.rit.edu/~jabics//EvoMusic/BilesEvoMusicSlides.pdf.

[9] Christopher S. Butler. *Statistics in linguistics / Christopher Butler*. B. Blackwell Oxford [Oxfordshire] ; New York, 1985.

[10] William E. Byrd. Minikanren. http://minikanren.org/.

[11] David Cohen, Mikael Lindvall, and Patricia Costa. Agile software development. *DACS SOAR Report*, 11, 2003.

[12] J. Fernández and F. Vico. AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, 48:513–582, 2013.

[13] R. Madrid A. Martnez Y. Padilla G. Aguilera, J. Galn and P. Rodrguez. Automated generation of contrapuntal musical compositions using probabilistic logic in derive. *Mathematics and Computers in Simulation*, 80(6):1200 – 1211, 2010.

[14] L. A. Hiller, Jr. and L. M. Isaacson. Musical composition with a high-speed digital computer. *J. Audio Eng. Soc*, 6(3):154–160, 1958.

[15] Stefaan Himpe. Algorithmic compostion. http://a-touch-of-music.blogspot.co.uk/2013/08/algorithmic-composition-generating.html.

[16] Stefaan Himpe. canon-generator. https://github.com/shimpe/canon-generator.

[17] B. L. Jacob. Composing with genetic algorithms. *Proceedings of the International Computer Music Conference*, 1995.

[18] Eduardo Morales M. and Roberto Morales M. Learning musical rules, 1995.

[19] S. Pariev. Minikanren: an implementation for ruby. `https://github.com/spariev/mini_kanren`.

[20] Shushobhickae Singh and Chitrangada Nemani. Fluent interfaces. *ACEEE Int. J. on Information Technology*, 1(2):19–22, 2011.

[21] Michael Spivey. *An Introduction to Logic Programming Through Prolog*. PrenticeHall International, 1996.

[22] P. M. Todd. A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4):27–43, 1989.

[23] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[24] Wikipedia. Canon (music). `https://en.wikipedia.org/wiki/Canon_(music)`.

[25] Wikipedia. Counterpoint. `https://en.wikipedia.org/wiki/Counterpoint`.

[26] Wikipedia. Crab canon. `https://en.wikipedia.org/wiki/Crab_canon`.

[27] Wikipedia. Palindrome. `https://en.wikipedia.org/wiki/Palindrome`.

[28] Wikipedia. Round (music). `https://en.wikipedia.org/wiki/Round_(music)`.

[29] Wikipedia. Table canon. `https://en.wikipedia.org/wiki/Table_canon`.

[30] A. E. Yilmaz and Z. Telatar. Note-against-note two-voice counterpoint by means of fuzzy logic. *Knowledge-Based Systems*, 23(3):256 – 266, 2010.

Appendix A

User Study

This chapter contains the user study materials:

- The instruction booklet
- The consent form and questionnaire
- Data gathered from the questionnaires
- The canons generated:
  - In typeset stave notation
  - As a link to the audio file

Note that the users are numbered rather than names for anonymity, and that they are numbered from 02 to 09. This is because the the test with user 01 was a trial run to refine the study before carrying it out on the rest of the subjects.

A.1 Instruction Booklet

Before we start

Please read and sign the consent form, and fill in the first section of the questionnaire.

Some background understanding is needed to complete these tasks. Please read the information that follows (up until task one) and **ask any questions that you need to**. If you have no musical knowledge you will probably need it all. If you are already familiar with music theory you might be able to skim read it fairly quickly.

What is a Canon?

Information taken from https://simple.wikipedia.org/wiki/Canon_(music)

A **canon** is a piece of music in which two or more voices (or instrumental parts) sing or play the same music starting at different times.

There are different kinds of canon. Canons can be described according to distances between the entries of the voices. If the second voice starts one bar (one measure) after the first voice, this is called a “canon at the bar”. If it starts after only half a bar, it is called a “canon at the half-bar”. It is even possible to have very close canons, e.g. “canon at the quaver (eighth note)”. Olivier Messiaen wrote a 3 part canon at the quaver in his *Thème et Variations’* for violin and piano. The pianist’s right hand (playing chords), his left hand and the violinist are the three parts.

“Strict canon” means a canon where each voice imitates the first voice exactly all the way through the piece.

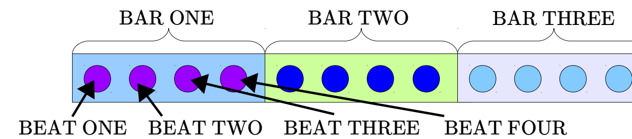
Here is an example of a five part, strict canon, with the only variation being the octave of some of the parts: https://www.youtube.com/watch?v=vMZtVmWEetQ. (Credit to Stefaan Himpe.)

What Kind of Canons Are We Considering Here?

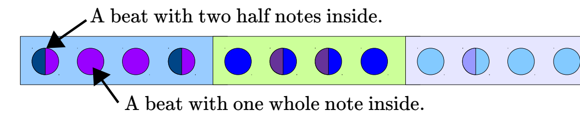
Here we are only considering strict canons with three or four beats in a bar, and with only crotchets, quavers and semi-quavers (whole, half and quarter notes respectively).

Music Theory Crash Course

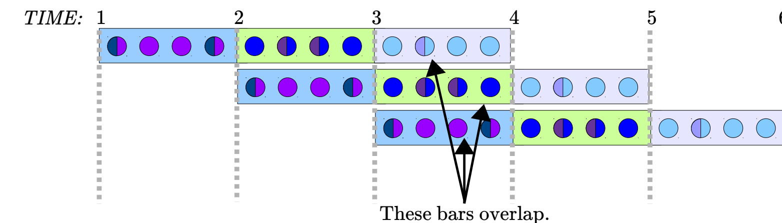
Rhythm: Every piece of music is split into **bars** which have a certain number of **beats** in them. For example, a piece of music with four beats in a bar might look like:



Each beat can have multiple notes in- we can half notes and quarter notes too. For example, we could change to make the piece look like this:



To make this into a canon, we could play the melody three times, each one starting a bar later. We’d then end up with a piece that looks like this:



Pitch: Every note also has a *pitch*, that is how high or low it is. The notes are named from A to G, and are grouped into octaves. In Sonic Pi, a note is represented by a symbol with the name and the octave number, for example C in the 5th octave will be represented as `:c5` and D in the 3rd octave will be represented as `:d3`, so that is the notation we will use here.

The sequence of notes from lowest to highest note goes:

… , `:c3, :d3, :e3, :f3, :g3, :a3, :b3, :c4, :d4, :e4, :f4, :g4, :a4, :b4, :c5,` …

You will want to keep your music in around the 4th-6th octaves to keep it in a good range for listening in general.

If no octave is specified, then Sonic Pi assumes that it is in the 4th octave, i.e. `:c` is the same as `:c4`, and `:g` is the same as `:g4`.

As a general rule, notes next to each other in the sequence above do not sound nice when played together. For example, playing `:c3` and `:d3` at the same time will sound bad (try it!).

Ignore this next part if you do not know about musical keys, sharps, flats and accidentals.

We can also represent sharps and flats by appending '`s`' or '`b`' to the note name, respectively. For example, C sharp in the 5th octave is represented by `:cs5`, and A flat in the 4th is represented by `:ab4`.

Note that B sharp is in the same octave as the B with that octave number, and the same for C, i.e. `:b4` is the same as `:cb5`, and `:bs4` is the same as `:c5`.

This allows you to write canons in different keys other than C major, and/or add accidentals.

Task One: DIY Canon

Overview: In this task you have been provided with all the template code for creating your own canon by composing it yourself. *All the code to play the canon has been written*, all you need to do is write your melody and press play.

Aim of the task: To write your own canon. When you have done one (or more) that you are happy enough with (or want to give up!) let me know and we can move on.

Instructions: For now, look at the section entitled 'Your Canon!'. You have been provided with an example canon here. To hear it run the code by pressing the **`Run`** button or **`Alt-r`** (you can stop it any time by pressing the **`Stop`** button, or **`Alt-s`**.)

```
canon = [
 {pitch: :c, length: 0.25}, {pitch: :d, length: 0.25}, {pitch: :e, length: 0.5}
]
```

The melody is represented as an *array* (or list) of notes.

Every note is represented by a *hash*, with a pitch and a length associated with it. For example, `{pitch: :c, length: 0.25}`. This is plays C for a quarter of a beat.

Now look at the section entitled 'Information About Your Canon'. Here we can specify some things about this canon. To set one just change the value that is assigned to it in the code after the equals (=) sign, for example to change the number of voices to 2 type `number_of_voices = 2`.

Variable	Description	Possible choices
number_of_beats_per_bar	This sets the number of beats in each bar.	3 or 4.
number_of_bars_before_starting_the_next_voice	This sets how many bars to wait before starting to play the melody again (so that they	1 or 2 recommended.

	overlap).	
number_of_voices	This sets how many parts you want- how many different voices play the melody.	2 or 3 recommended.
sounds	This sets what sounds you want to use for each part. It's an *array* (list) of synths. (See buffer three to test some sounds out.)	`[voice1, voice2...]` where some suggested sounds are: `:beep, :dpulse, :pretty_bell, :prophet, :pulse, :saw, :tb303.`
transpose	This says if you want any of the voices to be played up or down an octave (or multiple).	`[transpose1, transpose2...]` where transpose1 are the number of octaves to transpose by. Choose from -2, -1, 0, 1 and 2.

Now if you play the piece again, you should be able to follow the example melody as it plays.

If you have any questions at this point, please ask.

Now it's time to **create your own canon**. Delete the example canon and **unleash your creativity**!

When you have completed the task, please fill in the section concerning task one in the questionnaire.

Task Two: Canon Creator

Overview: In this task you may use Canon Creator to help you write canons. You have been provided with a template for setting some basic properties of the piece.

Aim of the task: To write your own canon. When you have done one (or more) that you are happy enough with (or want to give up!) let me know and we can move on.

Instructions: To get a canon we just type: `canon`.

However this will do nothing by itself- we must play it. To play it, type: `canon_play(canon)`.

Now **run the code** to play the canon by pressing the `Run` button, or `Alt-r`. (You can stop it any time by pressing the `Stop` button, or `Alt-s`.)

Congratulations, you're a composer!

Sonic Pi uses random numbers to create your canon. By changing the 'seed value' (the number it uses to find other random numbers) you can change how it acts. Add the line `use_random_seed 97` above your code to change the seed. Try **changing the number** to whatever you want and hear how different the canons sound. For example:

```
use_random_seed 10001
canon_play(canon)
```

We can **specify certain properties of** the canon. Let's start by setting the key type to 'major'. To do this, we write a dot (.) followed by the name of what we want to set, then the value we cant to set it to in brackets. So to get the major key we type: `canon.key_type(:major)`

and to play it we need the same structure as before, and then we press play:

```
use_random_seed 103
canon_play(canon.key_type(:major))
```

We might also want to set the lowest note to be middle C. We can add this afterwards: `canon.key_type(:major).lowest_note(:c5)` . We can chain together as many or as few properties as we like in this way.

All the possible properties are given on the next pages. Use these to **create your own works of art**.

When you have completed the task, please fill in the section concerning task two in the questionnaire.

Property	Valid arguments (inputs)	Description of what this property does/is	Default value (the property the computer will assign if not specified)
key_note	`:c`, `:d`, `:e`, `:f`, `:g`, `:a`, `:b`, `:cb`, `:cs`, `:db`, `:ds`, `:eb`, `:fs`, `:gb`, `:gs`, `:ab`, `:as` or `:bb`.	This is the **tonic (root) note** of the key.	Random.
key_type	`:minor` or `:major`.	This is the **type of the key**, very broadly, `:minor` is sad/mysterious and `:major` is more cheerful.	Random.
beats_per_bar	3 or 4.	This sets the number of **beats in a bar**. Each beat will be split into multiple shorter notes later.	Random.
lowest_note	Any valid note.	The **lowest note** that can be played in this piece.	`:c4`, or two octaves below the highest note if there is one set.
highest_note	Any valid note.	The **highest note** that is can be played in this piece.	`:c6`, or two octaves above the lowest note if there is one set.
probabilities	`[p1, p2, p3, p4]` where `p1`, `p2`, `p3`, and `p4` are numbers adding up to one, e.g. `[0.5, 0.3, 0.1, 0.1]`.	This sets how likely a beat is to **split** into multiple notes. Specifically, `p1` is the probability of splitting into a single note, `p2` of splitting into two notes, `p3` splitting into three and p4 splitting into four. So, `[1,0,0,0]` would have all beats in the canon being a single note, and `[0,0,0,1]` would have all beats splitting into four notes, while `[0, 0.5, 0, 0.5]` would have about half the beats split into two notes, and the other	`[0.5, 0.25, 0.15, 0.1]`

		half split into four notes.	
number_of_bars	Any whole number between 2 and 50.	This is the **number of bars** in the melody of the piece.	Two times the number of beats in the bar.
number_of_voices	1, 2, 3 or 4.	This is the **number of parts** in the canon- how many times the melody is played over itself.	2
type	`:round`, `:crab` or `:palindrome.`	This says what **type of canon** to create. A round has no restrictions on the melody, a crab canon plays each bit of the tune against itself backwards, and a palindrome sounds exactly the same forward as it does backwards.	`:round`
variation	Any number between 1 and 100.	This is the **percentage variation in rhythm** of the piece. A value of 1% means that every bar has the same rhythm (except perhaps where there are three notes in a beat- these might appear in a different form), while 100% means that there is not necessarily any repeats of rhythms (although this may happen by chance).	Randomly choose one of 50 and 100.
voice_transpositions	`[t1, t2, t3,...]` where `t1`, `t2`, `t3` etc. are the number of octaves to shift this voice by and are equal to -2, -1, 0, 1 or 2.	This controls whether any voices are **moved up or down** by an octave(s). For example, `[0,1,-2]` will leave the first voice alone, shift the second one up an octave and the third one down two octaves.	`[0, 0, 0,...]`
voice_offset	1, 2, 3 or 4.	This controls the number of bars to play **before the next melody** comes in.	Randomly choose between 1 and 2.

sounds	`[s1, s2, s3,...]` where `s1`, `s2`, `s3` etc. are any valid Sonic Pi synths. These include: `:beep`, `:dpulse`, `:pretty_bell`, `:prophet`, `:pulse`, `:saw`, `:tb303.`	The synth (**sounds**) that will be used for each voice. For example, `[:pretty_bell, :saw]` will use `:pretty_bell` for the first voice and `:saw` for the second.	A random choice for each voice, choosing from: `:beep`, `:pretty_bell`, `:prophet` and `:saw`.
max_jump	A whole number, greater than or equal to 5.	The **maximum permissible jump** between consecutive (root) notes. It is generally recommended to keep this at about 6 otherwise it tends to lose musicality, however experimenting with larger values can be interesting.	6

***Thank you** for taking part in this user study, your time is much appreciated!*

A.2 Consent and Questionnaire

Canon Creator

User Study Consent and Questionnaire

Please read the following and sign below if you accept and consent to take part in this evaluation.

- I agree to my answers to the following questions, and the music and code that I produce, being stored for evaluation of Emily Fox's Part II Project at the University of Cambridge.
- I understand that:
  - i. I can opt out at any point.
  - ii. I can request to find the results of the study by emailing ef337@cam.ac.uk.
  - iii. The results of this study WILL NOT reflect my own ability, but solely the quality of the project.
  - iv. There will be both written and verbal instructions, and if at any time I do not understand the study I can ask for clarification.
  - v. If I feel uncomfortable at any point, including but not limited to: uncomfortable sound level, lighting level, questions being asked, I can request to have that rectified, and/or stop the study early.
- I would like this to be stored anonymously (delete as applicable): YES / NO

Signed: ....

Name (print): ....

Date: ....

Questionnaire

BACKGROUND IN MUSIC AND COMPUTING (COMPLETE **BEFORE** DOING THE TASKS)

*1) What formal **musical** training/experience do you have? (Please tick all that apply.)*

◯ Music lessons at school
◯ GCSE music (or equivalent)
◯ A-level music (or equivalent)
◯ Music degree (complete, or in progress)
◯ One or more grades 1-4 in an instrument or voice
◯ One or more grades 5-8 in an instrument or voice
◯ Diploma in an instrument or voice
◯ Engagement in a choir, orchestra, band or similar
◯ Other (please specify) ....

*2) What formal **computing** training/experience do you have? (please tick all that apply.)*

◯ Key stage 3 ICT/computing (or equivalent)
◯ GCSE computing (or equivalent)
◯ A-Level computing (or equivalent)
◯ Undergraduate degree in Computer Science/Software Engineering/Computing or similar (complete, or in progress)
◯ Some training as part of a degree that is not directly computing related (e.g. Engineering, Mathematics)
◯ Other course, e.g. evening class
◯ Other qualification (please specify) ....

TIME SPENT ON THE TASKS (COMPLETE **AFTER** DOING **EACH TASK**)

3) Please give an indication of the time that you spent on each task.

Time spent on task one: ....

Time spent on task two: ....

EXPERIENCE OF TASK ONE (COMPLETE **AFTER** DOING **BOTH TASKS**)

4) Please give your level of agreement to the following statements, considering the first method of creating canons (task one), i.e. the DIY approach. For each, please circle the number that matches your experience, or tick 'I'm not sure'.

I **understood quickly** how to use this method to create canons.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I found this approach intuitive and **easy to use**.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I was able to create a canon(s) that I was happy with and that **sounded nice** to me.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I found the experience **frustrating**.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I found the experience **enjoyable**.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I found the experience **informative** (i.e. I learned something new).

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

It was **quick** to create new canons.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

EXPERIENCE OF TASK TWO (COMPLETE **AFTER** DOING **BOTH TASKS**)

5) Now please give your level of agreement to the following statements, considering the second method of creating canons (task two), i.e. with Canon Creator. For each, please circle the number that matches your experience, or tick 'I'm not sure'.

I **understood quickly** how to use this method to create canons.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I found this approach intuitive and **easy to use**.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I was able to create a canon(s) that I was happy with and that **sounded nice** to me.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I found the experience **frustrating**.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I found the experience **enjoyable**.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

I found the experience **informative** (i.e. I learned something new).

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

It was **quick** to create new canons.

STRONGLY DISAGREE NEUTRAL STRONGLY AGREE

1 2 3 4 5

◯ I'm not sure.

EVALUATION COMPLETE. **THANK YOU** FOR GIVING UP THE TIME TO TAKE PART IN THIS STUDY.

A.3 Results Data

A.3.1 User Experience

02

Music: Music lessons at school; Other: self taught piano. *Computing*: GCSE computing (or equivalent); A-level computing (or equivalent); Undergraduate in Computer Science/ Software Engineering/ Computing or similar (complete, or in progress); Other: internships.

03

Music: Music lessons at school; Other: one year of piano and one year flute lessons. *Computing*: Key stage 3 ICT/ computing (or equivalent); GCSE computing (or equivalent); A-level computing (or equivalent); Undergraduate in Computer Science/ Software Engineering/ Computing or similar (complete, or in progress).

04

Music: None. *Computing*: Key stage 3 ICT/ computing (or equivalent); A-level computing (or equivalent); Undergraduate in Computer Science/ Software Engineering/ Computing or similar (complete, or in progress).

05

Music: Music lessons at school; Engagement in a choir, orchestra, band or similar. *Computing*: Undergraduate in Computer Science/ Software Engineering/ Computing or similar (complete, or in progress).

06

Music: Music lessons at school; One or more grades 5-8 in an instrument or voice. *Computing*: Key stage 3 ICT/ computing (or equivalent).

07

Music: Music lessons at school; One or more grades 5-8 in an instrument or voice. *Computing*: Key stage 3 ICT/ computing (or equivalent).

08

Music: Other: piano lessons once a week for ten years; Other: musical education in a music school after regular school twice a week for around 4-5 years. *Computing*: Key stage 3 ICT/ computing (or equivalent)

09

Music: Music lessons at school (until year 8). *Computing*: Key stage 3 ICT/ computing (or equivalent).

A.3.2 Numerical Scores

DIY method (task one)

Question	02	03	04	05	06	07	08	09	Avg.	Var.
Understood Quickly	5	5	5	4	4	5	4	2	4.25	1.071
Easy	2	2	4	2	5	4	4	n	3.286	1.571
Sounded Nice	3	3	5	2	5	4	5	3	3.75	1.357
Frustrating	4	4	1	3	1	2	3	2	2.5	1.429
Enjoyable	5	3	4	3	5	5	4	4	4.125	0.696
Informative	4	3	5	4	4	4	5	4	4.125	0.411
Quick	1	1	3	2	4	3	4	2	2.5	1.429

Canon Creator method (task two)

Question	02	03	04	05	06	07	08	09	Avg.	Var.
Understood Quickly	5	5	5	5	4	5	4	4	4.625	0.268
Easy	5	4	4	5	4	5	4	4	4.375	0.268
Sounded Nice	5	5	5	5	5	5	3	4	4.625	0.554
Frustrating	1	2	1	1	3	4	3	2	2.125	1.268
Enjoyable	5	4	5	5	5	4	4	4	4.5	0.286
Informative	3	3	3	4	4	4	5	4	3.75	0.5
Quick	5	5	5	5	4	5	5	4	4.75	0.214

A.4 Canons Generated

02's DIY Canon

♩ = 100

Pretty_bell

Saw

Tb303

02's Canon Creator Canon

♩ = 80

Saw

Pretty_bell

5

8

03's DIY Canon

♩ = 60

Saw

Pulse

2

03's Canon Creator Canon

♩ = 80

Saw

Pretty_bell

Saw

4

7

10

13

16

3

04's DIY Canon

♩= 70

Prophet

Pretty_bell

Tb303

3

04's Canon Creator Canon

♩= 200

Prophet

Pretty_bell

5

4

8

05's DIY Canon

♩= 70

Pretty_bell

Saw

Hoover

05's Canon Creator Canon

♩= 70

Saw

Beep

4

6

8 5

6

4

06's DIY Canon

♩= 70

Dpulse

Saw

Tb303

Hoover

Pulse

6

7

8

8

06's Canon Creator Canon

♩ = 70

Beep

Saw

Pretty_bell

5

9

13

17

07's DIY Canon

♩ = 70

Pretty_bell

Saw

Tb303

3

07's Canon Creator Canon

♩ = 70

Pulse

Pretty_bell

Saw

5

08's DIY Canon

♩ = 70

Hoover

Beep

Dpulse

4

08's Canon Creator Canon

♩ = 70

Saw

Pulse

4

09's DIY Canon

♩ = 70

Pulse

Prophet

Dpulse

3

11

09's Canon Creator Canon

♩ = 70

Pretty_bell

Pulse

Dpulse

4

7

10

13

12

15

18

Music engraving by LilyPond 2.18.2—www.lilypond.org

A.4.1 Audio Files

02's DIY canon: https://goo.gl/lzBpIi

02's Canon Creator canon: https://goo.gl/Iwtvmw

03's DIY canon: https://goo.gl/XVBxzD

03's Canon Creator canon: https://goo.gl/8HNqPg

04's DIY canon: https://goo.gl/fJeDrR

04's Canon Creator canon: https://goo.gl/EyvQ8S

05's DIY canon: https://goo.gl/ncqrHh

05's Canon Creator canon: https://goo.gl/ONVDYn

06's DIY canon: https://goo.gl/xg7bRk

06's Canon Creator canon: https://goo.gl/1OGr11

07's DIY canon: https://goo.gl/xJJmbJ

07's Canon Creator canon: https://goo.gl/HHVZr0

08's DIY canon: https://goo.gl/H28A1R

08's Canon Creator canon: https://goo.gl/ROcr3r

09's DIY canon: https://goo.gl/I38xZV

09's Canon Creator canon: https://goo.gl/AVr8Er

Appendix B

Musicality Data

B.1 Numerical Scores

User (Number)	Method Used	Rhythm /10	Melody /10	Canon Suitability /10	Total /30
02	Canon Creator	6	6	4	16
02	DIY	8	9	6	23
03	Canon Creator	7	5	8	20
03	DIY	7	7	3	17
04	Canon Creator	7	5	6	18
04	DIY	5	3	6	14
05	Canon Creator	7	6	5	18
05	DIY	6	8	3	17
06	Canon Creator	5	6	6	17
06	DIY	8	1	3	12
07	Canon Creator	6	8	8	22
07	DIY	9	9	8	26
08	Canon Creator	7	8	8	23
08	DIY	8	9	6	23
09	Canon Creator	9	7	8	24
09	DIY	9	8	5	22

B.2 Comments

02 (Canon Creator)

Rhythm: Nice variety of note values, although there are points where is loses momentum (e.g. bar 3).

Melody: The opening shape is nice (pentatonic, for some reason), although the tonic (Db) seems to only appear arbitrarily.

Canon: Its mostly consonant, but a lot of the intervals between the parts are perfect (fourths, fifths, and especially octaves), which leads to quite a bare sound. There are also a few moments where the parts cross over and effectively cancel each other out (e.g. second half of bar 5).

02 (DIY)

Rhythm: Although its in , a lot of bars actually imply 6/8 (e.g. 1-2), which then leads to some nice hemiola in bars 3-5, when and 6/8 are effectively played at the same time.

Melody: This melody is actually completely pentatonic. It has a nice shape, with the first phrase ending on the dominant (A) and the second on the tonic (D).

Canon: A lot more dissonance in this one, although I personally think it works quite well due to the pentatonic nature of the melody. Theres some overly close writing in bar 2 though, which means the entry of the second part is somewhat hidden, particularly when both it and the first part play As in unison.

03 (Canon Creator)

Rhythm: Nice variety, and it doesnt really lose momentum, although the consistent ending crotchets at the end of every bar does become a bit boring.

Melody: This canon is quite a bit longer than the others, so staying in C minor the whole time gets a bit boring. Also, the phrasing seems quite arbitrary. Once again, quite pentatonic.

Canon: The harmony is much richer, with lots more thirds and sixths between the parts that fill out the chords.

03 (DIY)

Rhythm: Theres a lot of variety in a very short space of time, but the phrasing doesnt feel very well-defined and I think that may be due to the rhythmically abrupt end to the phrase.

Melody: I like how the phrase starts within the range of a third and then expands up to the top C. The G sharps arent really classically correct, but I like them.

Canon: Its seems almost purposely dissonant, in that the augmented fourth and sevenths dont really resolve, so dont really seem to mean anything.

04 (Canon Creator)

Rhythm: Like No.1; good variety, but it does lose momentum at times. That said, the much faster tempo means this is much less of a problem.

Melody: There are a lot of tonics and dominants (every single bar ends on either an A or an E), so the phrases have a clear enough shape but also sound rather too repetitive.

Canon: Nice variety of intervals between the parts, although there are points where they get a little too close or the harmony becomes a little too bare.

04 (DIY)

Rhythm: This one is very much down to personal taste. Obviously, there is no variety here; every beat has the same rhythm, so you could say its quite boring. On the other hand, it does have an insistent nature that I almost like.

Melody: As with rhythm, theres very little variety, with every beat being an arpeggio. The overall shape though seems quite arbitrary, and I dont really like the ending on the dominant.

Canon: Again, very much down to personal taste. The harmony is all very full and triadic, which is nice. Everything moves in parallel (mostly parallel triads), which could be heard as boring, although I quite like it.

05 (Canon Creator)

Rhythm: Nice variety, and it keeps moving.

Melody: Nice shape. The large leaps mostly work, although not always. It does start to sound a bit rambly towards the end, again because most of the bars end on either the tonic or dominant.

Canon: Nice variety of intervals, with some dissonance. However, again, the dissonance doesnt really seem to lead anywhere. Also, the parts harmonically undermine each other at times (e.g. at the end of bar 4, the upper part seems to be cadencing onto chord V, but the lower part moves towards chord I instead).

05 (DIY)

Rhythm: Hard to say much for such a short phrase, but the rhythm gives it a nice shape. It feels a bit more like 2/4 instead of 3/4, which isn't necessarily a bad thing.

Melody: Its only an opening phrase, so its perhaps a bit unfair to compare it to the others, but it has a nice arch from tonic to dominant and back.

Canon: Theres barely any counterpoint here at all because the phrase is so short and the entries are so relatively far apart. What counterpoint there is is mostly fine, although the clash between the C and D in bar 2 undermines the nice ending of the upper line.

06 (Canon Creator)

Rhythm: Often feels quite random. Not really any sense of phrasing - feels like one long phrase that keeps going.

Melody: Some quite disjunct leaps and lots of leger lines making it difficult to read/-play.

Canon: Quite unusual to imitate at the same pitch. Crossing parts are also not best practice. Sometimes a bit clashy and dissonant e.g. bar 3 and sometimes the harmony is a bit bare e.g. fifth in bar 4 or the octave in the same bar. Nice palindromic canon though.

06 (DIY)

Rhythm: Good driving rhythm. Probably want to finish on a strong beat with a strong ending.

Melody: Extremely disjunct and unreadable/unplayable. Repeated note in bar 4 Fb is equivalent to E.

Canon: E against F in bar 8 is quite a strong unprepared dissonance. Lots of crossing parts again and all centred around two entries an octave apart. Is theoretically a canon but not one that anyone would write.

07 (Canon Creator)

Rhythm: Puts quite a lot of emphasis on the second beat which is quite unusual for but otherwise nothing is particularly wrong.

Melody: Unusual in the fact that its in a six-bar phrase but other than that perfectly fine - nothing particularly outlandish.

Canon: Would normally arrange with the highest entering part at the top. Rather than having the third part enter at the same pitch it would be more normal to have it enter between the octaves at the fifth. Unisons in bar 2 make the entry of the second part basically inaudible. Some awkward harmony/unprepared dissonances.

07 (DIY)

Rhythm: Good driving rhythm. Four-bar phrases. Good.

Melody: Good melodic shape. Nothing incorrect about it.

Canon: Would be more normal to have the second part a fourth below. Second bar is almost entirely dissonance. Crossing part in bar 2. Works as a kind of dissonant counterpoint 20th Century canon.

08 (Canon Creator)

Rhythm: Again quite a lot of emphasis on the second beat in which is unusual. Might want to finish on a strong beat.

Melody: Generally fine - not too disjunct or anything. Could possibly be lengthened into two four-bar phrases: 3/6 bar phrases are unusual.

Canon: Quite a lot of second inversion chords (with the fifth in the bass) - unusual unless used in the context of a Ic-V-I cadence. Lots of crossing parts still. Should probably end with a clear tonal motion like a cadence making it clear whether its E minor or G major.

08 (DIY)

Rhythm: Good variety of rhythm.

Melody: Three-bar phrases are unusual. Some disjunct leaps e.g. bars 2 and 3 but nothing unperformable.

Canon: Some unprepared dissonances e.g. bar 2 beat 2 (fourth is a dissonance). Still some crossing parts. Unison at the end of bar 2 makes the texture feel thin. Sometimes harmony could be further clarified e.g. bar 3 beat 4 rather than doubling the third (presumably).

09 (Canon Creator)

Rhythm: Good varied rhythm. Nothing wrong with it at all.

Melody: Could probably benefit from more stepwise movement rather than all of the leaps by thirds and fourths.

Canon: Some strong dissonances e.g. beat 3 of the bar when the second part enters or beat 2 of bar 5. Still a lot of crossing parts. Good palindromic canon subject.

09 (DIY)

Rhythm: Good driving rhythm - gives it a forward direction.

Melody: Three-bar phrases are again unusual. Other than that, perfectly good melody.

Canon: Second entry on a pair of unisons hides the entry completely. First bar of the second entry is almost completely in octaves - very bare harmony. Bars 3 and 4 are full of dissonances.

Appendix C

Project Proposal

Emily Fox
Churchill College
ef337

Computer Science Part II Project Proposal

Automated Canon Composition

May 10, 2016

Project Originator: *Dr Samuel Aaron*

Project Supervisor: *Dr Samuel Aaron*

Director of Studies: *Dr John Fawcett*

Overseers: *Dr Stephen Clark* and *Prof. Alan Mycroft*

Introduction and Description of the Work

Sonic Pi is a piece of software which has been developed to enrich the teaching of computing and music in schools, and also to lower the barrier to entry by encouraging learning through experimentation and play. It is primarily built as a live coding environment to facilitate making music algorithmically and in real time. Using an internal Ruby domain specific language (DSL), it enables the user to use code to express a piece of music using standard programming control structures such as loops, threads and conditional statements, in addition to providing new mechanisms such as live-loops, cues and synchronisation.

Currently, automatically generating melodies in Sonic Pi amounts to choosing notes and durations pseudo-randomly, perhaps from a given subset specified by the user. However, there is currently no way of setting any constraints or including any logic in this. Consequently, this means that getting new melodies simply reduces to trying different seeds, from which you cannot predict any properties. To do this in practice would therefore be a long and arduous process which almost certainly outweighs any advantages from having the computer generate the sequence in the first place.

This project aims to develop a new feature of Sonic Pi which uses logic constraints in order to generate pseudo-randon canons, tailored to a user's requirements. This could then be used in a live coding environment as another layer to a composition. It will use the pseudo-random functionality built into the Sonic Pi already and guidance from heuristics to give these canons musical form.

The form of a canon has been chosen to give the project a focus, however in the future other genres could be implemented using the ideas, or even lead to more general techniques in the same field. Broadly, canons are piece of music which have multiple voices (or instruments) which play the same tune but with a time offset between them. The interesting thing here then, is that because the same melody is played at different offsets, there have to be constraints to ensure that the overlapping notes sound 'nice' (in a musical sense) alongside each other.

The main computer science challenges for this project are that of converting musical and aural constraints into those able to be specified by formal logic in order to be solved. Also, a semantically meaningful way for the user to be able to interact with Sonic Pi must be created in order for them to generate these sequences. Conversely, understanding the limits of what has been created will also be important; by its nature, things that are expressible in musical terms will not necessarily be expressible as constraints for the program.

Resources Required

No extra resources are required except my own laptop which has Sonic Pi installed and LaTeX for creating documents. A description of my laptop, and the procedures I will

follow to ensure that I avoid any data loss due to failures are detailed below.

Laptop specs: Optimus Series, 2.60GHz CPU, 8GB RAM, 1.75TB hard disk space, Linux Mint Debian Edition

Contingency plans against data loss

For written documents, I will work with the .tex file in a Dropbox directory on my computer which will continually sync to the web, as well as use Git to version it. For the code, I will host my code on GitHub and push changes after each commit, or at least after every few hours worth of work. I will also make daily backups of my home directory onto an external hard drive.

Contingency plans against hardware/software failure

In the event of my laptop breaking down, I would use my netbook instead. Although this is less powerful, it would be adequate until I could borrow a suitable Raspberry Pi, repair my laptop, retrieve another (unused) one from home, or buy a new one, as appropriate. I would use an external mouse/keyboard/monitor to make it fit for purpose.

Netbook specs: Samsung NC10 Netbook, Atom N270, 1.6GHz, 1GB RAM, 160GB HDD, CrunchBang Linux

Declaration of responsibility

"I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure."

Starting Point

Experience

Prior to the start of the project I have next to no experience with Ruby and none with miniKanren (which will be used for the logic part of the project). However, I do have some experience with Prolog and some knowledge of language theory which I have gained through parts IA and IB of the Cambridge Computer Science Tripos, which will help the learning process. I have a few hours of experience using Sonic Pi, so enough to understand the basics but no thorough knowledge. All these factors will need to be taken into account when scheduling timings for the project.

Software tools

- *Sonic Pi*: the software which will be extended. I will use v2.8-dev and add to it as applicable. `http://sonic-pi.net/`
- *miniKanren*: a DSL for logic programming which I will need to use in order to set and solve the constraints. I will use the Ruby implementation by Scott Dial and Sergey Pariev which is found at `https://github.com/spariev/mini_kanren`. `http://minikanren.org/`
- *LilyPond**: this will be needed should I export the music to score notation. I will use v2.18 (the latest stable version as of writing). `http://www.lilypond.org/`

* Only if certain extensions are carried out.

Substance and Structure of the Project

Core project

The project will involve extending Sonic Pi to generate canon melodies from a set of user-specified parameters, which are converted to constraints. Some basic constraints will have to be hard coded, for example intervals that might be used or general patterns in note sequences that will result in music which has some musical structure. This will require research as to what kind of constraints would be appropriate and how to represent these. Other constraints will be dependent on what the user wants, for example key, length and number of repeats, and it should be possible to specify these when doing live coding.

A key part of the project will be the user's interaction with the software, so the first part will be defining and implementing how the user of Sonic Pi can specify these constraints and use it in their program. This will involve defining a Ruby function that takes the constraints as arguments and then outputs them as an intermediate data structure containing all the information about the melody, which can then be fed into another function which will make the relevant calls to play the melody. Thought will therefore have to go into how best to represent the melody internally.

Extensions

Extensions to this project involve adding further user defined constraints such as: setting allowable intervals where the melodies overlap, being able to hard code certain notes (for example, the start note) and setting coefficients for the variation allowed in note length or pitch. A construct which would allow the user to create canons with distinct sections, could also be implemented. For example, the key could be different, or the average note length or pitch could be varied between sections.

Other methods of interacting with the software could also be developed, for example allowing the user to 'draw' the melody. The user would be able to draw a single line or line sequence in an image file which represents the general shape of the melody, and then this could be turned into constraints. This would bring in computer vision techniques to interpret what the user had drawn into logic constraints which can be solved.

There are also many different types of canon[1], so a further addition could be to allow different types to be generated. For example, one for which each voice is adjusted by a certain interval, or an inverted canon where one melody is essentially played 'upside down'.

Finally, the ability to have the function output the music in traditional music score form would be very useful. The internal representation of the melody would aim to be readable mainly by the machine and not by a human, so being able to have it export to traditional score notation would be beneficial for human understanding. A function would convert the internal representation to a LilyPond one in order to transcribe the music.

Evaluation

Success will be measured on two user studies and some measurements made over various runs of the program.

User study 1: This will aim at testing the quality of what is produced. Some people will be asked to listen to some melodies that have been generated using the Sonic Pi and some that have been converted to play within the Sonic Pi that were composed by an actual musician, and give a judgement on whether they think that the canon was created by the computer or composed by a human.

There is scope for asking different groups of people (some music experts, and some non-experts) to see whether there is any significant difference in the results; being able to convince a non-expert that a melody was generated by a machine would be a lesser achievement than convincing a musician, so this would be a good measure of the general musicality of what is produced.

User study 2: This will aim at testing the usability of the new functionality. It will get people with different backgrounds to try to generate melodies using the canon generator within Sonic Pi and measure how they find the experience, including what they think of the output.

Since Sonic Pi is aimed at people with a wide range of ability in this area, the group of people tested will also vary to give a good representation of how successfully people can use what has been created, and to what level.

Other measures: In order for this to be useful in a performance context, the melodies will have to be generated within a bounded time, therefore an investigation into the timings will also be done. Different constraints can be provided and computation time measured

for each. I will also determine whether there is a practical upper limit to how many parts each canon can have, and other such limits.

Summary of the project's main phases

1. Learning how to use miniKanren and Ruby.
2. Finding a good representation for the musical constraints which the user will be able to pass to the function.
3. Creating an internal representation for a canon which can be used to pass them around between functions.
4. Developing functionality for Sonic Pi which will take in the user's parameters, specifying basic properties of the melody and then will produce a canon in the internal representation.
5. Producing a function which will take this representation of a canon and make the appropriate calls to 'play' and 'sleep' so that the user can hear it.
6. Adding any extensions, such as alternative ways to set constraints or adding further parameters (see 'Extensions').
7. Evaluating the success of the implementation (see 'Evaluation').
8. Writing the dissertation.

Success Criteria

The following should be achieved:

- Create a program which can generate canon melodies from given constraints. Specifically, the program must be able to output a melody (in some format, whether transcribed score or the internal data structure form) that satisfies the requirements of a 'round', which is a limited type of canon. Namely, it must:
  - Consist of at least three voices
  - Have all voices sing the same melody at the unison
  - Have each voice starting at different times[2].
- Be able to specify properties of the generated melody:
  - Number of parts
  - Specified key
  - Specified length
  - Number of repeats

and have the generated music conform to these (this can be easily verified by analysing at the melody created).

- Interface this with Sonic Pi by introducing a well-defined way of combining available constraints in a call to the function.
- Users are able to use Sonic Pi to generate canons using the new functionality. (This will be determined by the second user study.)

References

[1] *Harvard Dictionary of Music*, Willi Apel, Harvard University Press, 1969
ISBN: 9780674375017
Canon, page 125

[2] *Round (music)*, Wikipedia, 2015
`https://en.wikipedia.org/wiki/Round_(music)`

Timetable and Milestones

Weeks 1 and 2: 23rd October - 5th November

Research characteristics of canon melodies and how these might be translated to logic constraints. Look into miniKanren and practise its use by finding an implementation for Ruby. Learn Ruby.

Milestones:

- Have some simple test code which demonstrates an understanding of Ruby and miniKanren.
- Complete a document containing the theory of canonic musical structures and how they translate to logic constraints.
- Detail the workflow which will have to be followed to implement these constraints using miniKanren.
- Send these documents to the supervisor for reference and feedback.

Weeks 3 and 4: 6th November - 19th November

Decide on an internal representation of the canon. Write some Ruby code which is able to generate a melody given a key and a length, and output it in the internal representation.

Write a function which will take the internal representation and play it using Sonic Pi.

Milestones:

- Write up the structure of the internal representation of the canons and discuss with the supervisor.
- Be able to demonstrate some sample melodies being created by the program which conform to the constraints. Different sets of constraints must be given, and different outputs generated for those inputs. Be able to see the internal representation and also hear it being played by Sonic Pi.

Weeks 5 and 6: 20th November - 3rd December

Develop the code so that the melodies generated are actually canons in two parts. This will involve constraining the notes which overlap to make sure that they sound good together, using music theory knowledge. Also, the way in which the user will input their constraints must be decided upon.

Milestones:

- Write up details of how the user can specify constraints, and discuss it with the supervisor.
- Be able to demonstrate two part canons being created by the program. When different parameters are given, the output must change according to those parameters, and objectively satisfy them. Be able to see the internal representation and also hear it being played by Sonic Pi.

Weeks 7 and 8: 4th December - 17th December

Extend the functionality so that canons can be created with arbitrarily many voices. If this does not take the whole time, the rest of this time can be used as a buffer to catch up if any parts of the project have fallen behind schedule.

Complete and send off the form to the ethics committee for testing on humans, so that there is plenty of time for it to be returned.

Milestones:

- Declaration form for the ethics committee submitted.
- Be able to demonstrate some sample canons with at least three voices (internally represented and aurally).

Weeks 9 and 10: 18th December - 31st December

Christmas week, so this time is set apart to enjoy the festivities! If more buffer time is needed then this time can be used.

Milestones:

- None.

Weeks 11 and 12: 1st January - 14th January

Buffer time (if needed), otherwise time to implement one of the extensions: exporting the music to LilyPond to get the canon in score notation.

Milestones:

- Document some examples of constraints used to generate canons, and the exported music in score notation. Send to supervisor.

Weeks 13 and 14: 15th January - 28th January

Preparation for the user study. Design the questions for testing and decide on how test data will be generated for other measures.

Complete a progress report, and prepare for the progress report presentation.

Milestones:

- Produce a document detailing the test methods that will be used, and the exact questions that will be asked.
- Have a list of the people who will take part in the tests confirmed.
- Send details of testing to supervisor for feedback.
- Complete and submit a progress report.
- Have completed slides and notes for the presentation. Send to supervisor for feedback.

Weeks 15 and 16: 29th January - 11th February

Buffer time (if needed), otherwise an extension to allow users to draw the melody. This will involve reading the image file and converting it to constraints which will be able to influence the melody created.

Milestones:

- Demonstration of being able to use an image file to constrain a melody.
- Document some canons and the images used to create them.

Week 17 and 18: 12th February - 25th February

Carry out the evaluation.

Milestones:

- Produce a document containing all the raw data collected that was specified in the evaluation preparation.
- Produce graphs of the data that has been collected, from both the user study and other metrics.
- Produce a general summary of the results collected and send to supervisor for reference.

Week 19 and 20: 26th February - 10th March

Start writing the dissertation.

Milestones:

- Have a completed draft of the introduction and preparation sections.
- Send draft to the supervisor to read and give feedback.

Week 21 and 22: 11th March - 24th March

Continue writing the dissertation. (More time is given here since this is the Easter vacation, so more time is allowed for revision and time for a break.)

Milestones:

- Have more than half of the implementation section complete.

Week 23 and 24: 25th March - 7th April

Continue writing the dissertation. (More time is given here since this is the Easter vacation, so more time is allowed for revision and time for a break.)

Milestones:

- Have a completed draft of the implementation section.
- Have a second draft of the introduction and preparation sections.
- Send both to the supervisor to read and give feedback.
- Make a start on the evaluation section.

Week 25 and 26: 8th April - 21st April

Continue writing the dissertation. (More time is given here since this is the Easter vacation, to more time is allowed for revision and time for a break.)

Milestones:

- Have a completed draft of the evaluation and conclusion sections.
- Have a second draft of the implementation section.
- Send both to the supervisor to read and give feedback.

Week 27 and 28: 22nd April - 5th May

Make final tweaks to get the final draft of the dissertation.

Milestones:

- Complete the final draft of dissertation, ready for printing.

Week 29: 6th May - 13th May

Print and hand in the dissertation at least a few days before the deadline to leave plenty of time for revision and avoid penalties for late submission. Deadline for submission is 13th May.

Milestones:

- Dissertation handed in.