

**“Automatic generation of musical
Species counterpoint using genetic algorithms”**

Final Report

Submitted for the BSc in Computer Science

May 2017

By

Jack Alan Taylor

Word Count: 16118

Table of Contents

1	Introduction	3
2	Aim and Objectives	4
3	Background	6
	3.1.1 – Music Theory.....	6
	3.1.2 – Species Counterpoint	7
	3.1.3 – Computer Aided Composition.....	7
	3.2 - Problem Context	8
	3.2.1 – Genetic Algorithms	8
	3.2.2 – Individual Species of Species Counterpoint	9
	3.3 – Alternate solutions	11
	3.4 – Comparison of Technologies	12
	3.5 – Discussion of Libraries	13
	3.6 – Processes and Methodology.....	15
4	Technical Development	16
	4.2 – System Implementation and design.....	17
	4.3 – Interaction and UI Design	17
	4.3.2a – Playback and MIDI Design	20
	4.3.2b – Playback and MIDI Implementation	21
	4.3.3 – Counterpoint Design and Fitness Function Implementation.....	23
	4.4 – Genetic Algorithm	43
	4.4a – Genetic Algorithm Implementation Design	43
	4.5 – System Implementation	46
	4.6 – Experiments and Results.....	51
	4.7 – Test Strategy	59
5	Evaluation	60
	5.2 – Project Achievements	61
	5.3 – Further Work	63
6	Conclusion	64
	Appendix A - Original Task-list.....	65
	Appendix B - Algorithm Flowchart.....	66
	Appendix C - Original Time-Plan	67
	Appendix D – Music Intervals.....	678
	Appendix E – Risk Analysis.....	679
	References -	72

1 INTRODUCTION

In this report, I will show how Genetic Algorithms (GA) that mimic the process of Darwin's Theory of Evolution can be used to generate music that increases in musicality across the number of generations that the algorithm has run for. The GA will create contrapuntal music called *counterpoint*, a musical technique where at least 2 independent melodies play simultaneously to create *harmony*.

Specifically, *species counterpoint* will be generated. This is a composition method that has been used for centuries to teach students how to compose formal counterpoint. The students are introduced to counterpoint in five stages (or species), each stage has a formalised set of rules, which describe the *pitch* and *rhythmic* relationships between the melodies, each stage increasing in complexity. The student must then write an accompanying melody that fits these rules.

Each generated counterpoint will have two voices:

- *Cantus firmus*, this is a simple melody part, based on medieval *plainchant* and is used in species counterpoint as a base melody. This will be provided as input to the student or program.
- The other melody is the *counterpoint*; the computer or student will write this by looking at rules from that specific species. There are two main sets of rules here; one governs the *intervals* between the notes and the other governs how the melody should move over time.

Computationally the task can be visualized as a large search problem, which aims to find an optimal combination of notes to make up a melody. There are many potential results from this search and only a small number of results would satisfy the counterpoint rules. By using genetic algorithms, we can generate an initial population of melodies, apply the species counterpoint rules to the objective function and use evolutionary processes to improve the gene pool iteratively and in theory find musical solutions relatively quickly.

Later in the paper, we will consider the effectiveness of using genetic algorithms for this task and look at how that compares to using an interactive human objective function and how other machine learning techniques could handle the same task.

2 AIM AND OBJECTIVES

The aim of this project is to use genetic algorithms to generate species counterpoint that will become more “musical” over time. The program will look at a given *cantus firmus* (“fixed melody”) and then generate another separate melody known as the *countermelody*; this countermelody will be represented as a ‘genome’. By using a Darwinian survival-of-the-fittest approach we can use genetic algorithm techniques like selection, crossover and mutation to evolve the melody over time, so the produced result will become more musical in respect to the counterpoint rules, defined in the counterpoint rule book “Gradus ad Parnassum”, which outlines 5 different *species* of counterpoint - each species’ rules gaining more complexity.

Objective 1 – Implement a system that generates first species counterpoint.

This is the major objective of the project; research into generating a melody that follows the rules of ‘first species counterpoint’ needs to be done. The GA will need to generate a melody that fits the composition rules for the first species. The output will be an entirely valid counterpoint but the rhythms for the cantus firmus and countermelody will be in *unison*, so we do not necessarily need to look at encoding rhythms in the genome for the first species. This gives a solid base to develop more species of counterpoint.

Objective 2 – Generate second and third species counterpoint.

As well as generating first species counterpoint, the program should be able to generate 2nd and 3rd species. Many of the same rules apply to this countermelody type, but to achieve the objective we will also need to encode our rhythms. *Second species* generation will need to create two *half* notes per one note of the cantus firmus and *third species* generation will need to create four *quarter* notes per cantus firmus note. This objective will allow us to create counterpoint that is more complex and will lead to a more enjoyable listening experience. The extra notes added creates more complexity in the melodies and will introduce some musical dilemmas like *passing notes*.

Objective 3 – Generate fourth species counterpoint and florid counterpoint

Like Objective 2, this will create a more complex counterpoint but these species of counterpoint are much more complicated. *Fourth species* generation will need to create a countermelody whose notes offset against the cantus firmus notes. In music, this is known as a *suspension*, computationally this is different to the other species as for each counterpoint note we need to make it be correct in terms of the previous cantus firmus note as well as the future cantus firmus note. We will also aim to create *florid counterpoint* or *fifth species counterpoint*, this will combine all the previous species rules with added rules, this can be massively complex so we will look at the extent we can generate “true” florid counterpoint. We are likely to run into problems as the search is so wide so this should give some evidence to apply different machine learning techniques to the problem.

Sub-Objective 1 – Compare effectiveness of GA

We will compare the effectiveness of using the developed GA against using a user critic or using other evolutionary algorithms and searches. Later, when trying to generate the advanced species of counterpoint we may need to consider different machine learning methods, which can be used to make conclusions on the different implementations. Related to this we will also compare effectiveness of specific genetic algorithms so I can cross validate across all the parameters and find out what is the most effective genetic algorithm and conclude why.

Sub-Objective 2 – Create output for the generated music

We will look at generating audio output of the melodies to use for tests throughout the project or some other musical format, a large amount of work here will be getting the individual notes to play for the right duration and time. To be able to add figures easily to this report, we will also discuss the steps necessary to create musical scores of our produced counterpoint allowing us to analyse the melody visually and have a clear format to show the melodies.

Personal Objective 1 – Practical development experience in natural computation

My main inspiration for choosing this topic was to gain practical experience developing and working with the natural computing aspects of Computer Science. These are becoming increasingly useful in technology and interest me greatly. I hope that at the end I will have gained practical knowledge in developing for the natural computation techniques used in this project. This should give me a solid foundation to continue further study and research into the field of machine learning.

3 BACKGROUND

3.1.1 – MUSIC THEORY

In this report, a few music terminologies are used, so to have some understanding of the project we need to discuss these terms.

Music is composed by creating a sequence of *notes*, notes are the individual components of any piece of music and represent the duration and pitch of a single sound. The figure below (**Figure 1**) shows a sequence of notes; the shape of the note indicates its *duration*. In the first bar, the semibreve indicates that the note sounded will last one whole bar duration. In bar two we can see that two (minim) half notes fit into that same space, so two notes are played in the same time one note was played in the previous bar, with each note being half as long in duration. This pattern continues for the next two bars. The heights of the notes indicate the *pitch*; the higher the note is on the score the higher the pitch is.

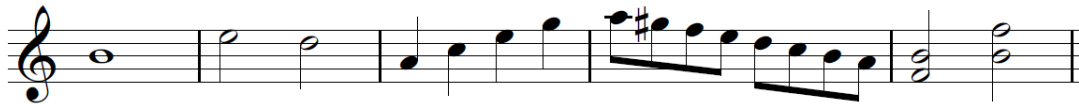


Figure 1: Example of a melody

An *interval* is the distance between the heights (itches) of the notes. We would say that in the second bar the *melodic interval* between the two minims is a second as there is only a difference of one line in the score. In the final bar, we can see that two notes are on top of each other, indicating that these notes are played together. We can then work out the *harmonic interval* of these notes by looking at the difference in heights between them - which would work out as a 4th as there is a 4 note difference between the two notes. This is shown further in Figure 2.

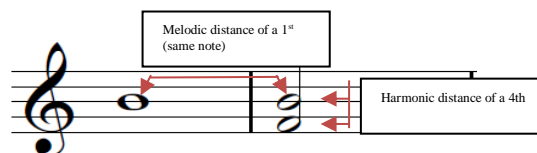


Figure 2: Example showing melodic and harmonic intervals

The above example (**Figure 1**) show a *monophonic texture* meaning that there is only a single line with no accompanying parts. For this project we will look at generating *polyphonic* or *contrapuntal* music with multiple melodic voices (two or more) playing at the same time as shown below (**Figure 3**).



Figure 3: 5th species counterpoint melody

3.1.2 – SPECIES COUNTERPOINT

In 1725, Johann Joseph Fux invented *species counterpoint*. His book “Gradus ad Parnassum” outlines a set of formalised rules that dictate the valid *pitch* and *duration* relationships, between contrasting melodic parts. These rules are introduced in five separate stages (species) that each increase in complexity with the aim of the student of music being able to reproduce complex “free counterpoint”, by the time they have reached the fifth species. In our program, the computer will play the role of the student.



Figure 4: Short example of 5th species counterpoint

There are two separate melody parts in counterpoint; the *cantus firmus* is a simple “fixed” melody part that is limited in rhythm and melodic direction and is provided to the student. The student creates their *countermelody* by following the rules provided. After completing this the student then goes on to work with 3 and 4-part counterpoint rather than the two-part countermelody the examples show.

3.1.3 – COMPUTER AIDED COMPOSITION

The research area that encompasses this report is *Computer aided composition* (CAC) which utilises computer technology to generate scores that can assist composers or even to generate music automatically.

There have been many different approaches to automatic music generation. Some use *neural nets*, in 1994 a “Creative Machine architecture” generated over 10,000 musical passages by training a neural net with 100 popular melodies (Tijus, 1988) and in 1991 “NEUROGEN” used neural networks to create four-part harmony compositions (Gibson et al., 1991). *Genetic algorithms* have also been used; in 1999, a system for harmonising user-specified melodies using a genetic algorithm to evaluate the generated harmonies was developed (Löthe, 1999). In 2000 “CONGA” was developed to enable users to evolve rhythmic patterns using genetic algorithms (Dostál, 2012) and more recently the *DarwinTunes* project has used genetic algorithms to create music by using volunteers to rate the generated sound loops (MacCallum et al., 2012). There have also been developments in generation of species counterpoint. Rule-based systems have been used like Schottstaedt’s Automatic Species Counterpoint (Farbood, 2001), Machine Learning has been applied (Dannenberg, 1997), Markov models has been used and some approaches fuzzy logic have been developed (Cádiz, 2006) and genetic programming has been used (Polito et al., 1997).

Like DarwinTunes’ approach, we will look at how Genetic Algorithms can be used in automatic species counterpoint generation. In this report we will investigate the application of defining some algorithmic evaluation criteria for the fitness function that would not require any user intervention like the volunteers in DarwinTunes.

3.2 PROBLEM CONTEXT

3.2.1 – GENETIC ALGORITHMS

A genetic algorithm has been used to generate the music in this project. Genetic algorithms are heuristic search algorithms that borrow ideas from evolutionary processes of natural selection. Rather than a gradient descent approach, that aims to find the minimum error by getting the gradient of the function and *sliding* down to minimize the error (*steepest descent* search approach), which is often used in back propagated feedforward neural networks. Genetic Algorithms are more similar to a random walk where the directions to slide down the surface are determined at random which is more likely to converge to a global minima or optima.

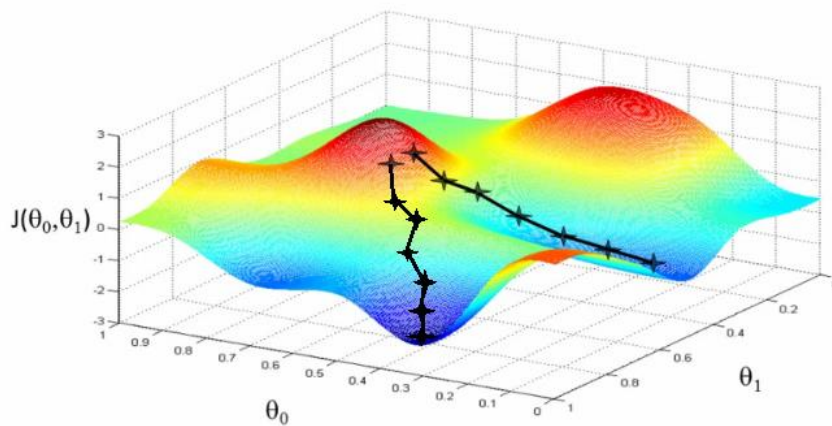


Figure 5: Gradient Descent approach (Learning, Learning and Shaikh, 2017)

Genetic Algorithms are based on *Neo-Darwinism*, which collects the ideas from Darwin's classical theory of evolution, Weismann's theory of natural selection and Mendel's interpretation of genetics (Daida et al., 1999). The basis to this approach is that a population needs to be "fit" to survive in its local environment and pass on its genes into the next generation. Natural mutation and propagation acts to provide increasing fitness in the generation. Genetic Algorithms consist of a *population* of *chromosomes*, which are the individual components of the population; in normal GA applications, this would be working with bit strings and in this project our chromosomes will be our individual musical notes. There also needs to be an evaluation method that will work out the fitness of the individual melodies in the population. Genetic Algorithms also need several *genetic operators*; a *selection* operator, which chooses how the individual genomes are chosen from the entire population, a *crossover* operator which swaps patterns of individual chromosomes to create new solutions or offspring and a *mutation* operator is used to replace chromosomes to provide variety into the population (Back et al., 2000).

3.2.2 – INDIVIDUAL SPECIES OF SPECIES COUNTERPOINT

More research was needed to work out how to create counterpoint in our program. As stated earlier - one melody, the *cantus firmus* is provided, then the student will create another melody called the *countermelody* that can be played contrapuntally over the given melody and there are five distinct species of counterpoint music.

In **first species counterpoint**, the student will aim to produce a smooth singable melodic line which can be played with the given melody to create a musically correct texture and harmony. The generated melody will provide one note for every note in the given melody; if there are 11 semibreves (whole notes) in the cantus firmus then the written counterpoint should have 11 semibreves, that are sounded at the same time as the cantus firmus notes.

In this species only *consonance* harmony is used which in simple terms means the differences between each note sound “nice” together as opposed to *dissonant* harmony which is generally a harsher, less nice sound (Cazden, 1945).

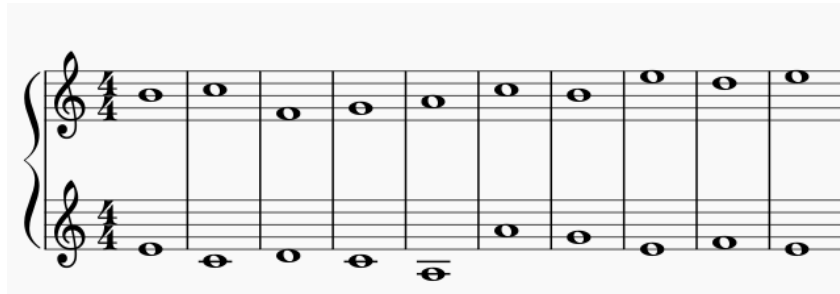


Figure 6: Example of First species counterpoint

Once the student can perform first species counterpoint, they can move onto **second species**, which increases in complexity. This counterpoint will move in half notes against the cantus firmus' whole notes, meaning that for every note in the cantus firmus two notes will be written that take up half of the cantus firmus note's duration each. Usually this counterpoint will start with a half note rest and end on a semibreve so the length of the countermelody will be twice the amount of the cantus firmus minus one for the rest and minus one for the semibreve at the end. This species will start to use some dissonant harmony but is still mainly consonant harmony as in first species counterpoint.



Figure 7: Example of 2nd species

The **third species** increases in complexity again and instead the countermelody moves in quarter notes against a cantus firmus of whole notes this means generally there will be a 4:1 ratio of countermelody notes against cantus firmus notes. Musically the introduction of 4 notes in a bar adds in some more musical theory to think about; we now must consider the *strength* of the individual beats inside a bar. A 4/4-time signature means 4 crotchet (quarter note) beats per bar; the first is a *strong beat* (downbeat), the second and fourth notes are *weak beats* and the third note lands on a *moderately strong beat* this makes it much more complex to compose for. There is also much more dissonant harmony here; that aims to resolve onto a nicer sounding consonance by using *passing notes*, where the first note will be a consonant harmony, which moves to a dissonant harmony and resolves back to a consonant harmony, which is more satisfying and interesting to listen to.



Figure 8: Example of third species counterpoint.

In fourth species counterpoint both the countermelody and the cantus firmus move once per bar like first species but the countermelody is offset by a half note. If we say each bar of music is subdivided into four beats, then the cantus firmus notes will start on the first beat of each bar and end on the fourth but the countermelody to be written needs to start on the third beat and last until the end of the second beat of the next bar. Musically this is known as a *suspension* and makes composing more complex as this means every countermelody note will play over two cantus firmus notes and must fit harmonic rules for both notes rather than just one previously.

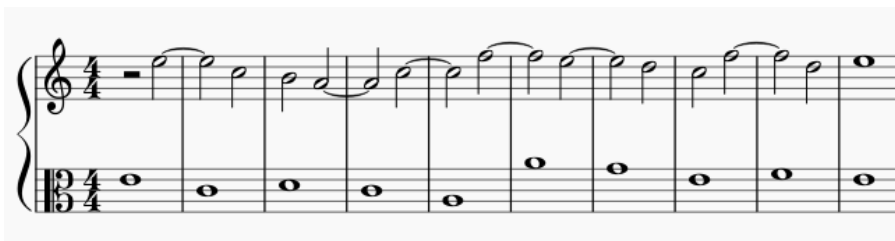


Figure 9: Example of fourth species counterpoint

The student then reaches fifth species or *floral* counterpoint this is a combination of the first four species so encoding rhythms is much more complex as there are less strict rules. This is shown in (Figure 4).

3.3 – ALTERNATE SOLUTIONS

There have been some attempts that aim to tackle the problem of generating species counterpoint. The strict rules and large search potential make it a good target for all types of evolutionary programming and algorithms. The other solutions mainly approach this problem by using variable neighborhood search algorithms.

Herremans and Sorenson (2012) used a variable neighborhood search algorithm is used to generate cantus firmus and first species counterpoint melodies. This approach has an algorithm, which starts with an initial solution, and iteratively improves the solution over time. The set of all solutions that can be reached from the current solution is named the *neighborhood*, and the optima of the search is reached when no better solution in the current neighborhood can be found and then can escape further local optima by changing the neighborhood. In some other approaches when no improvements exist a part of the solution is randomly changed to allow continued searching.

Other implementations that use the rules of counterpoint to automatically generate and evaluate music use *probabilistic logic* to develop music (Aguilera et al., 2010), in this implementation the probabilistic logic network aims to provide accurate probabilistic inference. In (Komosinski & Szachewicz, 2014) paper, *dominance relation* is used to analyse counterpoints, which allows analyzing of the fitness function without making assumptions on the importance of the individual criteria to avoid data loss.

Machine learning and neural networks have also been used in counterpoint composition. Here, a large collection of artificial neurons, based on the biology of the brain is used to connect to each other in a network. The network can be linked to other networks and each one will have its own summing and activation function that creates some form of output based on the input and weights that go into the network. This is tackled by providing the network with a series of correct species counterpoint examples and a gradient descent approach is used to minimize the error cost function of the produced output until an appropriate solution is generated.

This study shows how a genetic algorithm could be used to generate all species of counterpoint and specifically how well it can approach the very complicated fifth species counterpoint and show if this is effective.

3.4 – COMPARISON OF TECHNOLOGIES

To implement this program, we will have to use a technology or set of technologies that allow code creation and can perform algorithms quickly. A genetic search can take a long time so this is important, the choice of programming language and IDE must also support many libraries, potentially there is a lot of scale to the project with working with audio files, XML files, graphing tools and we should consider whether to use a language that supports evolutionary algorithm libraries. These libraries could potentially allow us to implement solutions more quickly and allow focus on a comparison of the different approaches statistically towards the end of the project instead of spending more time just developing the initial genetic algorithm approaches. It would also eliminate a lot of extra timely re-coding and testing.

3.4.1 – Java

A modern high-level language that can be very powerful and easy to implement solutions, with its wide range of IDEs. It also has specific evolutionary-algorithm libraries that are very powerful but perhaps have too much functionality for this type of project. I have some experience coding in Java but primarily in mobile app development and in a specific IDE for app development. Due to the potential large-scale of this project, it would make more sense to use a language that I am more familiar with and has libraries that are more lightweight.

3.4.2 – Python

Python is now an extremely popular language with a massive range of support and libraries that would be useful for this project it is great for data visualization, which could be useful in this project. Although it tends to be simple to develop for Python, I would feel more comfortable working in a language that I have worked longer in due to the scale of the project. Potentially I could look at python later with libraries like BOKEH and Pandas to visualize the data from the comparison of our solutions. There are also tools like TensorFlow and SciKit-Learn that aim to allow simpler machine learning implementations.

3.4.3 – MATLAB

MATLAB is very useful linear algebra technology that is very popular in generating machine-learning solutions; it can also produce plots and figures very easily and could be used for further neural network approaches.

3.4.4 – C#

C# is the language I have most experience with, as well as this I have worked extensively in the Visual Studio IDE and I think it would be very suitable for this project. It supports a wide range of libraries, its intermediary CIL code makes it very quick and portable with other .NET languages and I feel like I could program the solutions quicker than any other language due to my experience. C# also has a genetic algorithm framework library available as a free download, unlike some of the others, this still takes a good amount of coding to implement and is lightweight as it just supports genetic algorithms, which will be useful in the first stage of the project.

For these reasons, I will use C#, Visual Studio and the Genetic Algorithm Framework to develop the encoded solutions, at a later point I will research into more technologies depending on what I focus on in the later stage of the project and potentially revisit the technologies mentioned above. Python would seem to be the best choice if a different machine learning approach was introduced later in the project.

3.5 – DISCUSSION OF LIBRARIES

Coming into the project, I had very little knowledge on any natural computation and no experience working with evolutionary algorithm programming. At the start, I had to do a lot of research to learn how genetic algorithms work. After discussing with my supervisor, to help ease the learning curve and allow me to program earlier I concluded that working in a library could help me add more scale to the project that encompassed all the species of counterpoint rather than focusing on just a single species. I also have needed to research into ways to generate MIDI files and music from the output of the genetic algorithm so I have researched into libraries to make that process more straightforward as it's not the main objective of the project.

3.5.1 – Cartesian Genetic Programming

This is a genetic programming software that allows easy creation of genetic algorithms and AI techniques. This is inspired by LISP and is very efficient but is not as code heavy like some other libraries. This could be useful if one variation of the solutions does not converge very well in the actual written code using another library.

3.5.2 – ECJ

The evolutionary computation research system is a popular library for JAVA that encompasses a lot of natural computation techniques. If this project was implemented in Java this would be the library of choice.

3.5.3 – NNTool

This is a powerful neural network library built into MATLAB. Although artificial neural networks are not in the primary scope of this project, it may be useful to use later on to analyse the effectiveness of neural network technology, for music generation as compared to genetic algorithm approaches.

3.5.4 – TensorFlow

TensorFlow is an open source machine learning library targeted for Python and developed by Google. Although NNTool is very good at handling artificial neural networks it doesn't natively support some other machine learning techniques related to deep learning. Once research is done on what machine learning techniques we will try we can discuss whether we will use NNTool or TensorFlow.

3.5.5 – Genetic Algorithm Framework

To stick with my chosen language to develop with I have looked for C# libraries that help with implementing with genetic algorithm. The Genetic Algorithm Framework (gaframework.org) is a .NET assembly that can be downloaded through NuGet and has good documentation available. I have tested using this in an application, it still requires a large amount of coding and I have got good results so I will be using this library in my solution.

3.5.6 – MIDIUtil

Creating and playing the music programmatically created many issues and did not work correctly in C#. This would take a lot of research and time to program which would slow down my project. When the melodies are created by the algorithm it's very beneficial to also hear the melody so you can evaluate the problems more quickly but trying to implement the code to do this has slowed down the project so I've looked for a library to simplify this slightly. MIDIUtil is a Python library that aids in creating MIDI files from Python programs and this will be used throughout testing and may stick through the entire project. I used this library later on than I should and development overall has been much easier after I could easily hear the solutions my algorithm created.

3.6 – PROCESSES AND METHODOLOGY

This project will be approached iteratively, with each new species counterpoint that is implemented into the project, the design process and algorithms must be revisited and should provide more insight into how to improve the previous implementations. The use of a library that helps with the genetic algorithm implementation will allow this to be more manageable and would mean that substantial amounts of code will not need to be rewritten to test different approaches to the generation.

When the algorithm and fitness function are implemented the project becomes relatively test heavy, a fair amount of work is optimizing the parameters. As more functionality is added to the fitness function, it can show flaws in the current setup where the genes variation may fall and no better solutions can be made. It can also mean that a low fitness level is maintained. To overcome this a lot of work is done on optimizing the code, to make the algorithm run more efficiently so a “good” solution will be made.

Whilst working on the algorithm I have also been working on another application in Python. In C#, many issues occurred when trying to play notes simultaneously so I instead made a separate application that processes the output from the C# application and encodes and plays a MIDI from this. As a new species counterpoint is developed work is needed to be done so that the notes are played at the right times.

A large amount of study was needed in this project to understand the musical rules that govern the melodies to allow me to create accurate rules in the fitness function and will allow me to personally evaluate the produced melody and make my own decision on what further rules are needed to create a more enjoyable melody.

After creating an efficient genetic algorithm that works on each species, we will consider what other ways there are to implement the solution. There have been heuristic approaches, hybrid genetic searches and virtual neural network approaches to this problem so we can look at other implementations and compare the effectiveness for each one and perhaps re-implement our solutions with a hybrid of other techniques. For example, we could use reinforcement learning to look at our generated melodies and then can create melodies by training a neural network with our genetic algorithm. To support this, I’ve been developing my skills in MATLAB to potentially increase the scope of this project.

4 TECHNICAL DEVELOPMENT

This chapter will detail the steps that are needed as part of the system's development as well as the design process.

This section of the report deals with 4 primary areas of implementation and design. First, we will detail the overall design of the genetic algorithm and the different methods that can be used to get different results. We have the general UI & program design of the program which details how the user will interact and use the software. There is a separate program that aims to create a MIDI track whose use and implementation will be detailed and finally for design and implementation we look at how each species of counterpoint is designed and programmed to create the required outputs.

We will also look at the different experimental design and implementations that have appeared throughout the implementation process and how that has changed as well as looking at the performance and output difference when changing the parameters in the genetic algorithm.

We will also document the testing strategy and overall testing of this software.

4.2 – SYSTEM IMPLEMENTATION AND DESIGN

4.3.1 – INTERACTION AND UI DESIGN

This section will look at how the main system is developed. First, we will look at the process of how the program runs and then go into more detail on how each stage of counterpoint is handled by the program.

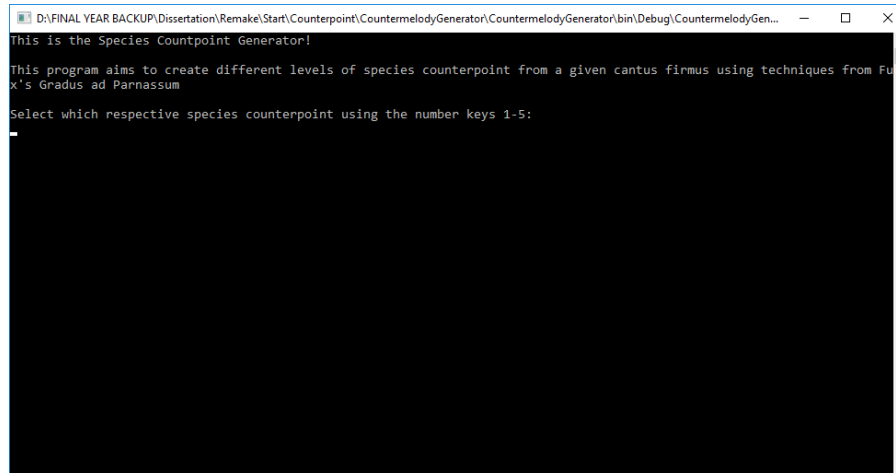


Figure 10: Introductory Screen of software

The UI of the system is very minimalist; the program launches and prompts the user about what species of counterpoint they want to generate. Once the user selects which species then the data for that species is prepared. The appropriate cantus firmus (given melody) is read into the program from a file, which is created externally. The file contains the notes names of the notes that make up the cantus firmus. An object of the respective stage of counterpoint is created then the steps are taken to generate the counterpoint.

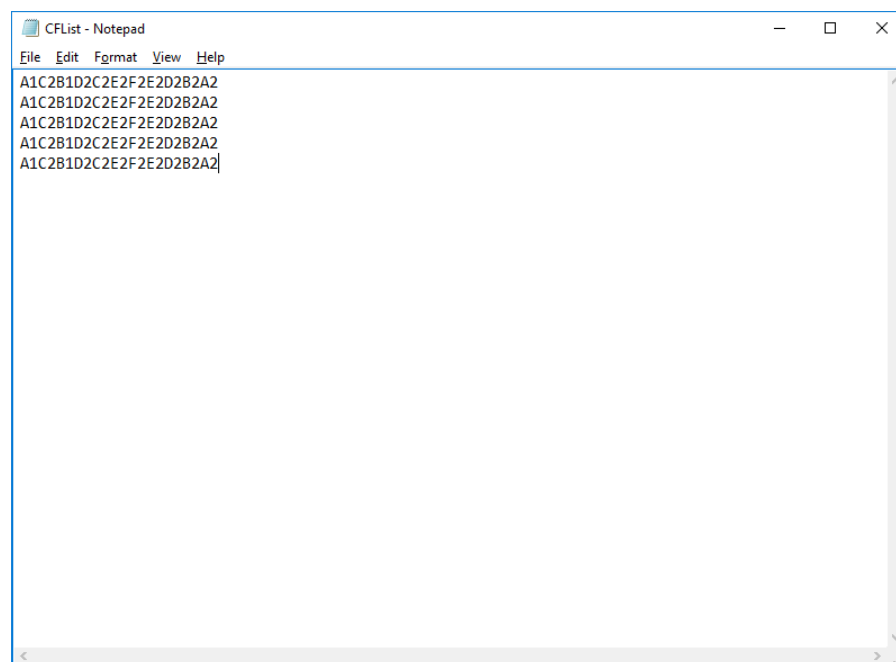


Figure 11: Melody structure in file

We then print what the species of counterpoint is, along with the actual notes of the cantus firmus that the counter melody is being generated for and details on the individual note lengths and other Boolean musical features like whether the note is sharpened flattened or tied are printed.

As the genetic algorithm is developing we are printing out to the console, the generation of the algorithm, the fittest individual melody in the population (which has the highest score in that generation) and then the overall average fitness score for the population so we can see how well that generation is suited and how it changes over time.

```
Species 1 counterpoint:
Generated Cantus Firmus:
Note:      A1, C2, B1, D2, C2, E2, F2, E2, D2, B1, A1,
Leng:      01, 01, 01, 01, 01, 01, 01, 01, 01, 01, 01,
Flat:      00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00,
Shar:      00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00,
Tied:      00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00,

Generation: 1, Fittest: 0.04999999999999998, Avg Fitness: 4.999999999999998E-05
Generation: 2, Fittest: 0.04999999999999998, Avg Fitness: 6.999999999999998E-05
Generation: 3, Fittest: 0.04999999999999998, Avg Fitness: 6.999999999999998E-05
Generation: 4, Fittest: 0.04999999999999998, Avg Fitness: 0.0001199999999999999
Generation: 5, Fittest: 0.09999999999999997, Avg Fitness: 0.0002699999999999999
Generation: 6, Fittest: 0.25, Avg Fitness: 0.0006999999999999999
Generation: 7, Fittest: 0.25, Avg Fitness: 0.00142
Generation: 8, Fittest: 0.3, Avg Fitness: 0.0026299999999999999
Generation: 9, Fittest: 0.75, Avg Fitness: 0.0044299999999999999
Generation: 10, Fittest: 0.75, Avg Fitness: 0.0063999999999999999
Generation: 11, Fittest: 0.75, Avg Fitness: 0.01041
Generation: 12, Fittest: 0.75, Avg Fitness: 0.0171
Generation: 13, Fittest: 0.75, Avg Fitness: 0.02711
Generation: 14, Fittest: 0.75, Avg Fitness: 0.0449199999999999999
Generation: 15, Fittest: 0.8, Avg Fitness: 0.0635999999999999999
Generation: 16, Fittest: 0.8, Avg Fitness: 0.08171
Generation: 17, Fittest: 0.8, Avg Fitness: 0.1089
Generation: 18, Fittest: 0.85, Avg Fitness: 0.14665
Generation: 19, Fittest: 0.89, Avg Fitness: 0.18527
Generation: 20, Fittest: 0.9, Avg Fitness: 0.22701
Generation: 21, Fittest: 0.9, Avg Fitness: 0.27438
Generation: 22, Fittest: 0.9, Avg Fitness: 0.33677999999999998
Generation: 23, Fittest: 0.9, Avg Fitness: 0.39012999999999999
Generation: 24, Fittest: 0.94, Avg Fitness: 0.44842999999999999
Generation: 25, Fittest: 0.94, Avg Fitness: 0.49896999999999999
Generation: 26, Fittest: 0.94, Avg Fitness: 0.57018
Generation: 27, Fittest: 0.94, Avg Fitness: 0.60151000000000002
Generation: 28, Fittest: 0.95, Avg Fitness: 0.64645000000000002
Generation: 29, Fittest: 0.95, Avg Fitness: 0.67367
Generation: 30, Fittest: 0.95, Avg Fitness: 0.70106999999999999
Generation: 31, Fittest: 0.95, Avg Fitness: 0.71997000000000004
Generation: 32, Fittest: 0.95, Avg Fitness: 0.72836000000000004
Generation: 33, Fittest: 0.95, Avg Fitness: 0.73446000000000004
Generation: 34, Fittest: 0.95, Avg Fitness: 0.74950000000000004
Generation: 35, Fittest: 0.95, Avg Fitness: 0.75474000000000003
Generation: 36, Fittest: 0.95, Avg Fitness: 0.76026000000000003
Generation: 37, Fittest: 0.95, Avg Fitness: 0.76504000000000001
Generation: 38, Fittest: 0.95, Avg Fitness: 0.76532000000000001
Generation: 39, Fittest: 0.95, Avg Fitness: 0.77681000000000002
Generation: 40, Fittest: 1, Avg Fitness: 0.78091999999999999
```

Figure 12: Console printing on algorithm training

Once an individual in the population has a perfect fitness score of 1.0 then the produced counterpoint melody will be printed to the screen. The durations of the notes to be played will be printed also. After the user is then prompted again with the initial screen asking for a new species to generate for.

```
Generation: 1000, Fittest: 0.82, Avg Fitness: 0.81
2 - 1
2 - 0.5
2 - 0.5
2 - 1
2 - 0.5
2 - 0.25
1 - 0.25
2 - 1
2 - 1

This is the Species Counterpoint Generator!
This program aims to create different levels of species counterpoint from a given cantus firmus using techniques from Fux's Gradus ad Parnassum
Select which respective species counterpoint using the number keys 1-5:
-
```

Figure 13: Final output of melody notes and rhythms

At this point the generated countermelody and given cantus firmus are converted into MIDI format and written into a new file in the local system. MIDI (Musical Instrument Digital Interface) is a protocol that allows digital instruments to communicate to a computer; this formed a standard that allows you to carry digital information as event messages, which represent the notes pitches, velocity, length and other parameters. We simply convert the note names (e.g. “A2”) as shown in the above screenshot into a MIDI pitch. The full musical note to MIDI pitch is shown below – The letter A stands for the musical note of A and the number dictates the octave that A is sounded at so A1 and A4 would be the same note but would A4 would be much higher pitch. The written file is then used to later to play music in a separate application which is detailed in the next section.

Note	Octave										
	-1	0	1	2	3	4	5	6	7	8	9
C	0	12	24	36	48	60	72	84	96	108	120
C#	1	13	25	37	49	61	73	85	97	109	121
D	2	14	26	38	50	62	74	86	98	110	122
D#	3	15	27	39	51	63	75	87	99	111	123
E	4	16	28	40	52	64	76	88	100	112	124
F	5	17	29	41	53	65	77	89	101	113	125
F#	6	18	30	42	54	66	78	90	102	114	126
G	7	19	31	43	55	67	79	91	103	115	127
G#	8	20	32	44	56	68	80	92	104	116	
A	9	21	33	45	57	69	81	93	105	117	
A#	10	22	34	46	58	70	82	94	106	118	
B	11	23	35	47	59	71	83	95	107	119	

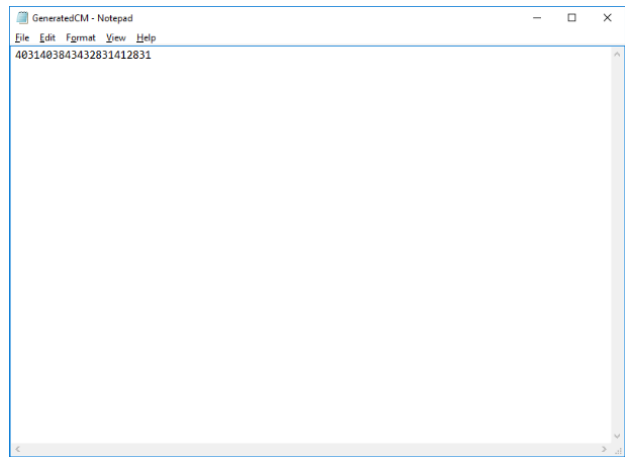


Figure 14: Produced midi output in file and MIDI table of notes
(Andymurkin.wordpress.com, 2012)

4.3.2A – PLAYBACK AND MIDI DESIGN

A simple program has been written in Python to create a MIDI file from the written files from the main program. This application is another console-based application that reads in the inputs and splits the notes from the file into individual notes. Based on what stage of counterpoint it is, we can programmatically create a MIDI track that fits the rhythm of that species of counterpoint.

We open the application by entering a CMD command prompt and executing the python file which then prints out what its read from the files and prompts the user to write what species of counterpoint that is, and the program makes a new midi track with a certain tempo.

```

C:\Users\Jack> python example.py
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Jack> cd desktop
C:\Users\Jack\Desktop> python example.py
[ 33, 30, 28, 38, 50, 66, 42, 40, 32, 36, 23 ]
[ 45, 52, 56, 53, 45, 43, 40, 43, 47, 43, 46 ]
C:\Users\Jack\AppData\Local\Programs\Python\Python35\lib\site-packages\winutil\WinFile.py:899: FutureWarning: Please explicitly set adjust_origin. Default behaviour will change in a future version.
  FutureWarning)

which species? 1

```

Figure 15: MIDI text output from python program

```
print(cantusfirmus)
print(countermelody)

countermelody2 = ['33', '28', '35', '29', '43', '40', '38', '36', '30']
track = 0
channel = 0
time = 1 # In beats
duration = 1 # In beats
tempo = 30 # In BPM
volume = 100 # 0-127, as per the MIDI standard

MyMIDI = MIDIFile(1) # One track, defaults to format 1 (tempo track)
MyMIDI.addTempo(track,time, tempo)

ij = 0
ji = 0

print("\n")

# Ask for the number and store it in userNumber
userNumber = input('Which species? ')

# Make sure the input is an integer number
userNumber = int(userNumber)
```

Figure 16: Python code showing MIDI structure

4.3.2B – PLAYBACK AND MIDI IMPLEMENTATION

First Species

In first species counterpoint, audio should be played note against note – two individual channels of music are needed and we simply iterate through each note in the cantus firmus and counter melody and sound them at the same time. At each iteration we increment the time factor of the channel to ensure it doesn't play all notes at the same time.

```
if(userNumber == 1):
    for pitch in counter melody:
        if(ij < (len(counter melody))):
            MyMIDI.addNote(track, 0, int(cantusfirmus[ij]) + 12, time, duration, 80)
            MyMIDI.addNote(track, 0, int(counter melody[ij]) + 12, time, duration, 70)
            time = time + 1
            ij += 1
    with open("species1.mid", "wb") as output_file:
        MyMIDI.writeFile(output_file)
```

Figure 17: Code for writing first species midi

Second Species

For the second species of counterpoint, we need to play two counter melody notes in the time we played one counter melody note before, so the code is not too different we just half the duration of the counter melody notes in the channel and play two notes on each iteration. On the second counter melody, we add 0.5 to time to ensure it plays after the first counter melody note that was added. We check if the counter melody is the last and if it is one counter melody note is played as permitted by the species (like the first species again).

```
if(userNumber == 2):
    for pitch in counter melody:
        if(ji < (len(counter melody) - 1)):
            MyMIDI.addNote(track, 0, int(cantusfirmus[ij]) + 12, time, duration, 80)
            MyMIDI.addNote(track, 0, int(counter melody[ji]) + 12, time, duration / 2, 70)
            ji += 1
            MyMIDI.addNote(track, 0, int(counter melody[ji]) + 12, time + 0.5, duration / 2, 5)
            time = time + 1;
            ji += 1
            ij += 1
        else:
            MyMIDI.addNote(track, 0, int(cantusfirmus[ij]) + 12, time, duration, 80)
            MyMIDI.addNote(track, 0, int(counter melody[ji - 1]) + 12, time, duration, 70)
    with open("species2.mid", "wb") as output_file:
        MyMIDI.writeFile(output_file)
```

Figure 18: Code for writing second species midi

Third Species

For the third species of counterpoint we need to play four counter melody notes in the time, we played one in the first species. This is again similar to the previous species of counterpoint but instead we play four notes per one cantus firmus note. We divide the duration by 4 to make each note play for the right length and increment the time by 0.25 of a bar each time we play a counter melody note. We have the final check to ensure both notes at end are played together like the first species again.

```
if(userNumber == 3):
    for pitch in counter melody:
        if(ij < (len(cantusfirmus) - 1)):
            MyMIDI.addNote(track, 0, int(cantusfirmus[ij]) + 12, time, duration, 80)
            MyMIDI.addNote(track, 0, int(counter melody[ji]) + 12, time, duration / 4, 70)
            ji += 1
            MyMIDI.addNote(track, 0, int(counter melody[ji]) + 12, time + 0.25, duration / 4, 55)
            ji += 1
            MyMIDI.addNote(track, 0, int(counter melody[ji]) + 12, time + 0.50, duration / 4, 60)
            ji += 1
            MyMIDI.addNote(track, 0, int(counter melody[ji]) + 12, time + 0.75, duration / 4, 50)
            time = time + 1;
            ji += 1
            ij += 1
        else:
            MyMIDI.addNote(track, 0, int(cantusfirmus[ij]) + 12, time, duration, 80)
            MyMIDI.addNote(track, 0, int(counter melody[ji - 1]) + 12, time, duration, 70)
    with open("species3.mid", "wb") as output_file:
        MyMIDI.writeFile(output_file)
```

Figure 19: Code for writing third species midi

Fourth Species

The fourth species is a bit different; we want to play one note per one and a half cantus firmus notes besides the first and last bar but each counter melody note only plays half a bar after the counter melody note to create a suspension

```
if(userNumber == 4):
    for pitch in counter melody:
        if(ij < (len(counter melody) - 1)):
            MyMIDI.addNote(track, 0, int(cantusfirmus[ij]) + 12, time, duration, 80)
            MyMIDI.addNote(track, 0, int(counter melody[ij]) + 12, time + 0.5, duration, 70)
            time = time + 1
            ij += 1
        else:
            MyMIDI.addNote(track, 0, int(cantusfirmus[ij]) + 12, time, duration, 80)
            MyMIDI.addNote(track, 0, int(counter melody[ij]) + 12, time, duration, 70)
    with open("species4.mid", "wb") as output_file:
        MyMIDI.writeFile(output_file)
```

Figure 20: Code for writing fourth species midi

Fifth Species

The fifth species is a bit different because every counter melody note is not the same length so the main program writes the midi pitch note and then after one number that represents the length. We can split these apart from the program by detecting every multiple of 3 in an index and add that to a 1d vector or array. Then we can just play the notes by instead work out what time it would play by looking at the previous notes and totaling the lengths and then the index of the length array at the same index as the note of the counter melody will give the duration of that note

Audio Playback

Once the MIDI track has been programmatically written the actual midi file can be written to the local system and then played through any modern media player. The file can also be opened in any free musical notation software like MuseScore or Lilypond very easily to look at the produced score and step through individual note playbacks if necessary. Below shows what happens after importing our first species counterpoint into the notation software (semibreves shown as crotchets to fit figure).



4.3.3A – FIRST SPECIES COUNTERPOINT

In first species counterpoint, the algorithm must generate one note for every one note of the cantus firmus - that has been chosen and read in from the file. For this part we will assume that the cantus firmus length is 11 so the output will need to have 11 notes too.

The first species is the simplest version due to each note in the cantus firmus and countermelody being the same length. Below will show what these rules are and how they are implemented inside the fitness function. The figures are created by moving the generated midi into a music notation software, which is a very smooth process and some are made up to show the rules easier.

For the first few examples, pseudocode will be given to get some sense of how the programming work but in later examples this will be omitted as the previous techniques are reused to get different results.

A lot of this section will be on musical *intervals*, the different intervals are shown in (**Appendix D**).

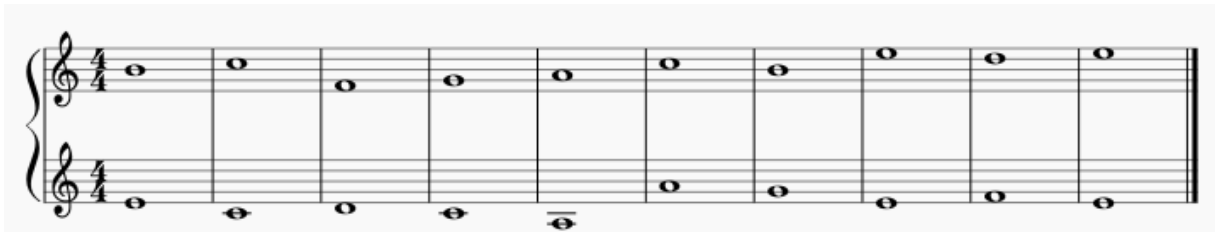


Figure 23: First Species Counterpoint example

Melody

These rules will ensure that the individual generated melody is singable and follows standard music theory without any consideration for the overall *harmony* of the piece. Below will show a list of rules that will be used to dictate the melodic dictation of the melody. We will also show a pseudocode implementation of all these rules.

Rule 1 – Mainly stepwise melody

This rule ensures that the melody is singable by moving in small melodic intervals up and down. The musical score (**Figure 26**) shows an example of a stepwise melody, the vertical difference between the notes is just one line or space for each note.



Figure 24: Stepwise melody

Whereas the figure below shows an example of a melody that is entirely none stepwise where each note moves at a high pitch difference (higher vertical difference in the score).



Figure 25: Disjoint melody

This can easily be implemented by running a loop over all the notes in a melody and tallying the amount of stepwise movements there are. This ensures that at least 60% of the notes move in stepwise motion. This is handled by converting the note names (A2) to a pitch representation where the higher the number, the higher the pitch is. By simply deducting the next note pitch at index ($i + 1$) from the current note at index (i).

```
for note_pitch in counter melody_notes
    if Abs(note_pitch[i] - note_pitch[i + 1]) == 1
        step_movements += 1

if (step_movements > note_total * 60 / 100)
    correct
else
    incorrect
```

Figure 26: Disjoint melody

Rule 2 – Avoid leaps greater than 5th except for octave and ascending minor.

This rule ensures that the melody does not make any large leaps and ensures that if there are any moderate allowed leaps that the melody changes direction afterwards. This can be done using similar approach to Rule 1 but instead checks if the difference between the current pitch and next pitch is greater than four (rule states 5th but is a 4 due to zero-indexing) then if the difference is not a -5 for the ascending 6th or 7th for the octave then it is a bad melody. If it is a correct leap, we ensure that the next note goes in the opposite direction.



Figure 27: Example of leap resolution

We will omit the for loop in pseudo code below and assume it's the same as rule 1.

```
#if greater than 5th
if Abs(difference) > 4
    if(difference != -5 || 7)    #if ascending 6th or octave
        if diff > 0 and note_pitch[i + 2] > 0 #if both move up direction
            badmelody
        else if diff < 0 and note_pitch[i + 2] < 0 #if both move down direction
            badmelody
        else goodmelody
    else
        badmelody
```

Figure 28: Pseudocode of rule implementation

Rule 3 – Avoid too many repeated notes

To promote variety, we need to ensure that there are not many repeated notes in the melody. To do this we have the list of numeric notes sorted from the highest notes to lowest and any ones that appear more than once are tallied. If the count for any individual pitch is greater than 25% of the entire melody then it is a bad melody.

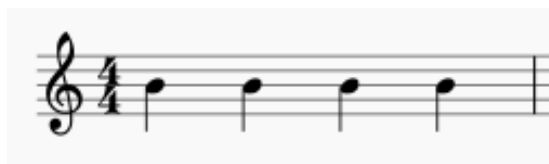


Figure 29: Example of repeated notes in a melody

Rule 4 – Keep melodic range in a singleable range – should not exceed a 10th

This ensures that the difference between the lowest and highest note of a melody is not greater than a 10th. This is very simple with the ordered notes list used above. We can just deduct the zero and final index and work out if an interval is greater than a 10th. The example below shows a melody whose lowest note and highest have a melodic interval of a 10th so this is the highest range of a melody we should encounter.

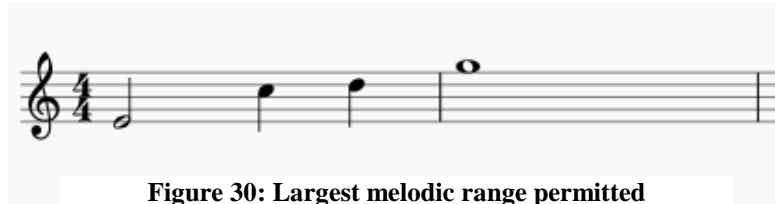


Figure 30: Largest melodic range permitted

```
#Difference between last note and first absolute value greater than 10th?  
if Abs(note_pitches[0] - note_pitches[count - 1]) > 9  
    badmelody;  
else  
    goodmelody;
```

Figure 31: Pseudocode of rule 4

Rule 5 – Ensure penultimate note of melody is not a repeat and is stepwise onto final note

This rule ensures that the last *melodic interval* is a single step onto the final note of the melody. It also ensures that this note is not repeated throughout the melody. The below example shows what this could look like.



Figure 32: Pseudocode of rule 4

```
#If the note at index is equal to penultimate increment variable  
if (note_pitches[i] == note_pitches[count - 2])  
    instances = instances + 1;  
  
#After for loop if instances is more than 1  
if instances > 1  
    badmelody
```

Figure 33: Score and pseudocode showing rule 5

Rule 6 – Don't move at same melodic intervals consecutively

The melody must not move at the same intervals repeatedly. The example below shows a case where an interval of a 4th melodic interval is repeated – this would be a bad melody.



Figure 34: Pseudocode of rule 4

```
for (note in notes)
    if (note[i] - note[i + 1] == note[i + 1] - note[i + 2])
        if (note[i + 1] - note[i + 2] == note[i + 2] - note[i + 3])
```

Figure 35: Score and pseudocode showing rule 6

Harmony

These rules will ensure that the individual generated melody follows musical harmony to ensure that the overall sound sounds “nice” and “musical” rather than at specific points the music sounding harsher.

Rule 7 – Avoid parallel fifths, octaves and perfect fourths

A parallel interval is when both melodies move in the same interval. For 3rd and 6th intervals these are fine but when fifths, fourths and octaves (eighths) are used then this is not allowed. This is similar to the pseudocode in the above rule six but instead we check differences between the other melody too and if both are the same then it's not a valid melody

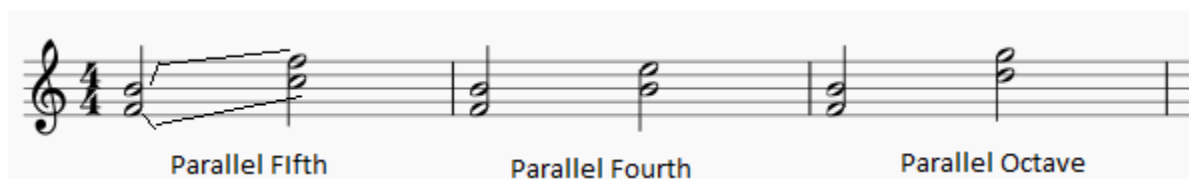


Figure 36: Showing parallel perfect intervals

```
#This example assumes both melodies have same rhythm
for (i < melodylengths, i++)
    if (cantusfirmus[i] - cantusfirmus[i + 1] == counter melody[i] - counter melody[i + 1])
        badmelody
```

Figure 37: Pseudocode to avoid parallel intervals

Rule 8 – Avoid using too many thirds and sixths

To promote variety in the music generally too many thirds and sixths are not allowed. These will sound well together but can be boring if too many of the same harmonic intervals are re-used. The example below show what two melodies that only have 3rds for harmonic intervals would look like.



Figure 38: Showing song of only harmonic third intervals

Pseudocode has been omitted due to how this as the separate components of this are covered above. A count that counts how many thirds and sixths are needed by taking away the countermelody and cantus firmus note pitches at the same index.

Rule 9 – Avoid melodic crossover

In this stage of counterpoint *melodic crossover* is avoided – this enforces that the countermelody and cantus firmus stick to their own separate melodic range, rather than crossing over pitches. At any point, the cantus firmus should stay above or below the pitch of the melody. Below shows an example of melodic crossover in the final bar.



Figure 39: Example of melodic crossover

Programmatically a list of all the harmonic intervals is created (difference between the countermelody and cantus firmus pitches). The list can then be iterated through and if the first note difference is greater than 0 then all of the following harmonic intervals should be greater than 0 and vice versa if the harmonic intervals start below 0.

Rule 10 – Avoid any dissonances

In music, there are *consonance* and *dissonant* harmonies – consonant sounds “nice” and dissonant sounds harsh. There are certain intervals in music like the 2nd and 7th that are dissonant and others that are consonant. In first species counterpoint we must avoid any dissonances. The example below shows a consonant harmony in its first bar and a dissonant harmony in the second bar.



Figure 40: Example showing dissonant harmonies

Rule 11 – Harmonic intervals at the start and end should finish at perfect interval

The first and end intervals of the music should finish on a “perfect” interval so must finish on an octave above or below or just a fifth below. The first bar shows the countermelody being at an octave above the cantus firmus, second bar shows it being a fifth above and the final bar shows an octave below.

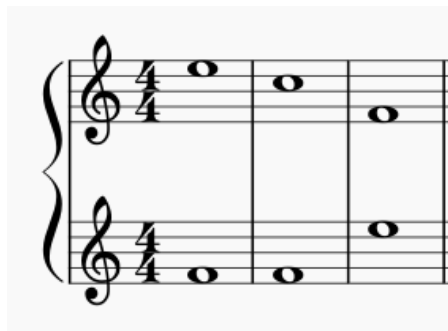


Figure 41: Example showing rule 11

Rule 12 – Ensure there is contrary motion especially at the end of the melody

The melody should use contrary motion. This is where the countermelody will move in the opposite direction preferably at the same intervals as the cantus firmus. This is an important feature of counterpoint composition.



Figure 42: Example showing good contrary motion

This example shows contrary motion as both notes move in opposite directions at the same intervals. This should always occur at the end of the piece.

4.3.3B – SECOND SPECIES COUNTERPOINT

In second species counterpoint the algorithm must generate two notes for every one note of the cantus firmus that has been chosen and read in from the file. For this part, we will again assume that the cantus firmus length is 11. It would be expected that the generated melody would be 22 notes long but in this case, we will generate 20 notes. This is because we don't play any notes for the first half a bar (a rest) and the final bar must end with another whole note like the first species counterpoint.



Figure 43: Example showing second species counterpoint

This becomes a bit more complex as we start to work with dissonances in the harmony but the intervals are still easily calculated as its just 2 notes per 1 besides the first and last bar where its 1 for 1 note.

We also introduce the notion of a weak and strong beat so this must be programmed in.

Melody

Second species counterpoint inherits many melodic rules from the first species but will have to be reprogrammed to fit the new rhythms. As before the melody should not make leaps larger than a fifth unless the octave or ascending minor sixth are used, if there is a leap it should be compensated by moving in the other direction and each melody should be in a singleable range not exceeding a 10th but there are also some newly added rules.

Rule 1 – Avoid same melodic interval twice on same pitches

If the melody consecutively uses the same melodic interval it must not use the same pitches and just repeat the same idea as the example shows below we can use the same melodic intervals just not at the same pitches.



Figure 44: Example following rule 1

Rule 2 – No Leaps on weak beats

In music, there are *beats*; beats are a unit of time that make up a musical *bar*. In a typical 4/4 measure, a bar is made up of 4 quarter notes or quarter **beats** this means there are 4 beats in a bar which is signified by the 4/4 notated in front of all the figures of music included.

This means that one bar in 4/4 can either be made up of one whole note, two half notes, four quarter notes or 8 eighth notes as shown below.



Figure 45: Example showing different rhythms

In terms of rhythm these individual beats have differing *strengths* where down beats are known as strong beats and upbeats are weak beats. The downbeat or strong beat in 4/4 is always at the first beat in the music, and then the following beats are the weak beats in the bar. To get some feel of how this works count “1, 2, 3, 4” repeatedly for the same duration each - but emphasise the “1” and this is how a 4/4 measure works. This introduces more complexities in the music as we have specific rules for the strength of beats of music.

This rule states that no leaps should occur on weak beats – below we’ll show an example of a musical motif that does this. The beats are made up of 2 half notes that take up 2 beats of the bar each. In the second bar, we can see that a leap would be made on third beat in that case which would not be allowed in music.



Figure 46: Example showing leap onto weak beat

To fix this you would compose it with the leap going onto the strong beat (first beat or downbeat) as shown below

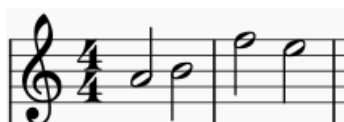


Figure 47: Example showing leap onto strong beat

Rule 3 – A single climax

To create an interesting melody – a single *climax* is allowed this means that in the melody there is one note that is higher pitch than the rest and is not repeated again during the melody.

Dissonance handling

Generally, the harmonic rules are reused from the first species but our list of harmonies needs to be greater as there are more countermelody notes, the inclusion of weak beats and the permission of using more *dissonant* harmonies means more rules need to be added to govern this properly.

Parallel intervals of a fifth and octaves should also still be avoided

Rule 4 – Only dissonances on weak beats

Like rule 2 where leaps can only be made on strong beats, these strong beats should only be of consonance harmony where the harmony sounds pleasant to our ears but on the weak beats, the music is free to play dissonant harmony - which sound harsher and should then be resolved to consonance again on the strong beat. This makes more variety in the music and makes it more interesting and satisfying to listen to but we must still ensure we do not overuse the dissonances.

Readers who have studied in music may know of *accidentals* that can create dissonance but this report only makes use of natural notes rather than using any sharps or flats in the music.

4.3.3C – THIRD SPECIES COUNTERPOINT

In third species counterpoint the algorithm must create 4 notes for every one note of the cantus firmus this means in (nearly) every bar there should be four quarter notes against the whole notes of the cantus firmus. Like second species the final bar will also end with a whole note and start with one bar of rest like 2nd species. So, if we had 11 cantus firmus whole notes we should generate 40 counter melody notes.



Figure 48: Example showing third species

As well as working with strength of beats and dissonances like before, this species utilises passing notes, which means this is more complex to compose and program.

Melody

Again, third species inherits many rules that have been detailed. The melody should not leap larger than a fifth but now it cannot leap with an ascending sixth, leaps should be compensated by opposite movement, the melody should be in a singleable range, should not be any overlapping with the cantus firmus and the same melodic intervals on the same pitches.

Rule 1 – Do not outline a tritone in melody

A tritone is outlined if 3 or 4 adjacent whole tones are used. This can sound very dissonant and should be avoided. To avoid this the melody should not use repeated melodic interval movements.



Figure 49: Example showing tritone outline

This example melody outlines a tritone.

Rule 2 – Double Climax included

In the previous species of counterpoint, one climax, where one note is higher pitch than the others, which is not repeated, was included. Due to an increase of notes, this species requires a ‘double climax’. This means we should have two distinct climax notes as part of the melody.

Harmony

Generally same harmonic rules are in place but here we’ll have many more harmonies to process due to the extra generated countermelody notes so is a lot more computationally heavy

Rule 3 – Dissonances should be approached and left by step

If a dissonance occurs it must be approached and left in stepwise motion where the melodic distance or interval between the notes are a 2nd and moves in steps after and before the dissonance.

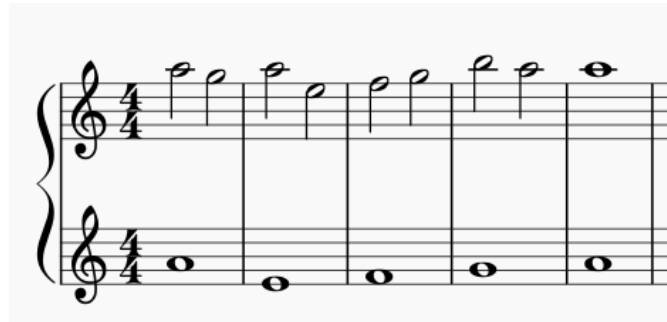


Figure 50: Example showing rule 3

4.3.3D – FOURTH SPECIES COUNTERPOINT

The first three species of counterpoint were relatively similar but this species is unique – I will aim to address the new musical techniques in an understandable way.

The music in this species becomes *syncopated* – this is when weaker in the bars are emphasised unlike before when only the first beat was emphasised.

The first bar will start with a 2-beat rest – then note will be sounded that will be played or ‘tied’ over the bar and is continued to be played into the first 2 beats and a new note is played on the third beat which is then tied over again. The example below shows what a typical fourth species counterpoint would look like. The rules should show more detail

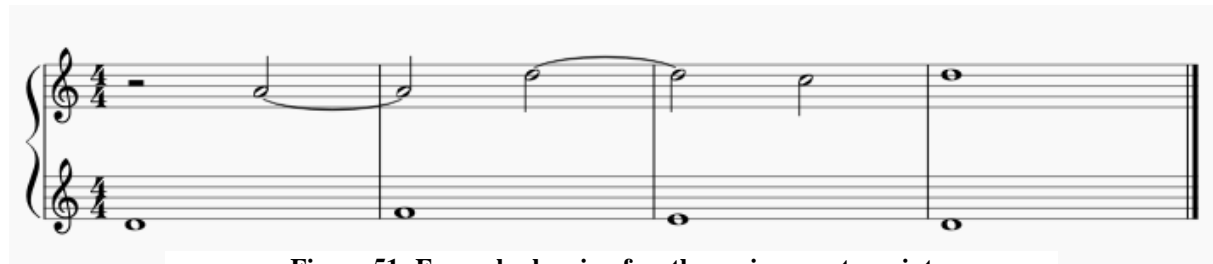


Figure 51: Example showing fourth species counterpoint

Melody

The idea of tying or playing a note over 2 bars but the melody underneath still changes is known as a *suspension* in music. Other than that, generally the melodic rules are the same and handles leaps similarly but if the countermelody crosses over the cantus firmus this is now permitted

Rule 1 – Countermelody must step down at end

There is only one way to approach the finish of the countermelody in fourth species counterpoint. The melody must form a *suspension* and step down onto the octave or unison. The next page will describe suspensions in detail.

Harmony

Rule 2 – Unisons are now acceptable on second note that goes over cantus firmus

Unisons are where the same note and pitch is played at the same time. The other species did not allow this but in fourth species if it is the second note that starts on the weak off beat, this is permitted and is suspended over the bar. The 2nd note below is the tied note and is also a unison with the bottom melody. This is now permitted.



Figure 52: Example showing good unison use

Dissonances

Note that there should still be no dissonances of the strong beat of the bar as defined by the other species

Rule 3 – Suspensions are permitted

Specifically, the suspension in music is an accented dissonance that occurs on strong beats and it must be followed by a harmonic consonance. A suspension can be subdivided into three separate parts musically:

1. The *preparation* of a suspension is a note on a weak beat of the bar that is at a harmonic consonance with the cantus firmus note. This note will tie over the bar so the same note lasts over one bar.
2. Then the *suspension* is the second part which is the part described above that is played over the bar which will be the same note as the preparation but this time it should create a dissonance harmony with the cantus firmus note.
3. The *resolution* will then be on the weak beat of the same bar as the suspension but it should be consonant and will always be stepwise from the previous suspended note.

There are several types of suspensions that can occur the **7-6**, the **4-3** and the **9-8**. The first of each pair of these listed numbers lists the harmony that should occur on the suspended part of the figure – so this should be a 7th, 4th or 9th then the resolution part should be the same harmonic interval as listed by the second number of these pairs.

So, if the suspension is a 7th then it should resolve onto a 4th, if the suspension is at a 4th it should resolve onto the 3rd and if the suspension is the 9th then this should resolve onto an 8th. The figure below shows an example of all these.

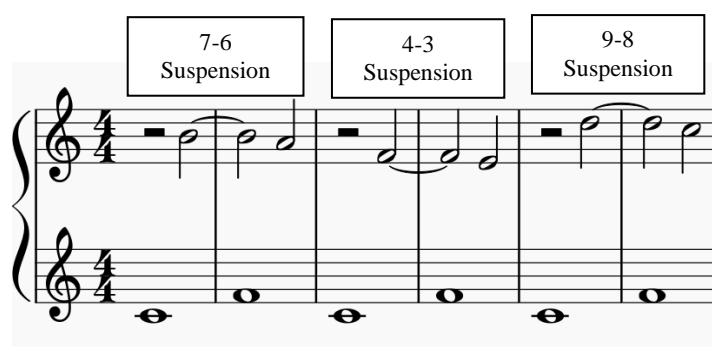


Figure 53: Example of good use of suspensions

These suspensions should only be used if the countermelody is at a higher pitch than the cantus firmus, but similarly if the countermelody is below the cantus firmus a **2-3**, **4-5** or **5-6** suspension can be used which is the same idea but the suspension steps up instead of going down.

The pseudocode for the process to handle these suspensions is documented below:

```
#If dissonant note
if(note[i] == dissonant_note)
    #If above cantus firmus
    if(harmonic_interval[i] > 0)
        #If on strong beat of bar
        if(beat == 0 | 1)
            badmelody

        #If weak beat ensure at is correct 7-5/4-3 or 9-8 suspension
        else if(harmonic_interval[i] == 7)
            if (harmonic_interval[i + 1] != 6)
                badmelody
        else if(harmonic_interval[i] == 4)
            if (harmonic_interval[i + 1] != 3)
                badmelody
        else if(harmonic_interval[i] == 9)
            if(harmonic_interval[i + 1] != 8)
                badmelody
        else
            badmelody

#If below cantus firmus
if(harmonic_interval[i] < 0)
    #If on strong beat of bar
    if(beat == 0 | 1)
        badmelody

    #If weak beat ensure at is correct 2-3/5-6 or 4-5 suspension
    else if(harmonic_interval[i] == 2)
        if (harmonic_interval[i + 1] != 3)
            badmelody
    else if(harmonic_interval[i] == 5)
        if (harmonic_interval[i + 1] != 6)
            badmelody
    else if(harmonic_interval[i] == 4)
        if(harmonic_interval[i + 1] != 5)
            badmelody
    else
        badmelody
```

Figure 54: Pseudocode for suspensions

4.3.3D – FIFTH SPECIES COUNTERPOINT

Fifth species counterpoint combines all the rules and ideas from all the previous four species of counterpoint. This is a much harder task computationally and to compose for. Rather than before where every individual note in the counter melody would be the same duration (same rhythm), fifth species instead can use whole notes, half notes, quarter notes and now eighth notes.

The implementation of this is much more difficult –we must create a population of configurations of rhythms. In doing this each individual melody will be of a different length and the library utilises for the genetic algorithms had massive problems with this. The way this is handled will be discussed in the following section of the report on the genetic algorithm implementation.

Shown below are a few examples of a correct fifth species counterpoint.



Figure 55: Example of fifth species counterpoint

Rhythm

Rule 1 – No leaps for very quick notes

No leaps should be made on eighth or quarter notes this is generally unpleasant to hear and will be avoided completely. The example bellows shows what an appropriate use of a leap in a melody with shorter notes could look like.



Figure 56: Shows good leaping

Rule 2 – Do not start with short-short-long rhythm

In most bars of music, a (short-short-long) rhythm should be avoided in some cases it is fine but definitely not in the first part of the melody. An example of a short-short long rhythm is shown below. Note that it does not necessarily need to just be two short rhythms then one long, as it would appear; the second bar shows another example of this.



Figure 57: Shows short short rhythm

Rule 3 – Do not use more than 2 eighth notes in a bar

Now that the note of an eighth has been introduced, we need to ensure that it is not overused. At most, there should be no more than two eighth notes in a bar. The first four notes in the bar below are eighth notes so this would not be permitted as there are four eighth notes.



Figure 58: More than two eighth notes in a bar

Melody

Rule 4 – Unisons acceptable at start and ending and passing

In this species, unisons are only permissible at the start/end or as a passing note. This is the same in third species but is reiterated as the other species have conflicting rules about unisons.

4.3.3E – DESIGN PROCESS CONCLUSION

Audio

Initially I tried to implement the audio playing in a C# console application, I tried to use native .NET beep() functions to play a note at a certain frequency for a certain amount of time. This worked okay with one melody besides having massive issues with timing as it is running based on processor speed but when I made it multithreaded to play multiple notes at the same time the windows media functions in .NET couldn't handle it and would sometimes just play random note or wouldn't play at all. From this I left the audio playing capability to very late in the project.

Counterpoint Generation

The first four species of counterpoint implement nicely and give good output but 5th species had a lot of issues using the library. It works by creating a population of notes and configurations of note equal the correct number of beats that should be in the music.

This was a problem because that means each melody would have a different number of notes so the mutation and crossover operators did not really work. Instead, I took the melody with the max amount of notes and filled the others with “null” notes to reach the same melody length. This approach was not efficient.

4.4 – GENETIC ALGORITHM

4.4A – GENETIC ALGORITHM IMPLEMENTATION DESIGN

Genetic algorithms are flexible and there are many ways to approach developing the initial genetic algorithm. Here we will speak about them conceptually and later compare the effectiveness of each approach. Each method below will run iteratively until the terminating criteria is reached and then it will return the best solution, which will then have converted to midi format, written to a file and played by another program that has been developed.

4.4.1 – Creating random population of solutions

The first step is to create an initial set of random melodies (solutions), in the natural selection metaphor this is the same as having a population of animals that are instead randomly generated. The overloaded constructor for the ‘Population’ class is shown below.

```
Public Population (int populationSize, int chromosomeLength,  
Bool reEvaluateAll, bool useLinearlyNormalisedFitness,  
ParentSelectionMethod selectionMethod, evaluateInParallel);
```

Figure 59: Constructor code

A population size must be decided on, as this will generate a set of solutions of this length, we want to have variety in our population so we do not lose any important musical motifs.

We must select a way of encoding our solutions (chromosomes) to give the output that is required; most genetic algorithms use a binary string approach but here we will create list of all the possible notes for the species, then for each individual in the population we will create a chromosome that has a random sequence of these permitted notes. Specifically, we do this by first encoding the chromosomes with every possible note (gene), we then use the threaded RandomProvider function given by the genetic algorithm library to shuffle each chromosome. The solutions only have to be 11 (or any length) notes long so then we can just remove the extra notes from the melody, this will loop until we have the full population of individuals set by population size.

4.4.2 – Evaluating the chromosomes

A fitness function will allow us to evaluate each melody in our population to give each a ‘score’ that shows how suited the melody is to its environment. In the project our counterpoint rules are the *environment*, however well they abide by those explicit rules the better the score is they receive. We must create a function that’s passed to the framework as a delegate which is called when needed to evaluate the solution, then will return a real value between 0 and 1.0 (0 – not suited at all to environment, 1 – perfectly suited). If returned value is 1.0 then we can terminate the genetic algorithm as an optimal solution as defined by our function has been reached we can do this using another function as a delegate called ‘TerminateFunction’ when this returns true the algorithm will terminate. The counterpoint design sectioned detailed these compositional rules that are used for evaluation.

4.4.3 – Selecting parents & creating child solutions

After each run of evaluating all the melodies, they must be sorted them in terms of the best and worst and decide on the consequences of this. We will create genetic operators that operate on the evaluated population to decide how new solutions should be created and selected for the next generation. The program will use three separate operators; Elitism, Crossover and Mutation.

Elitism

This operator will allow a certain percentage of the best solutions to pass through to the next generation, without any modification to their individual components. Doing this could corrupt a unique optimal part of the melody; this ensures that we keep the features that make the top part of the population better which in this example equate to musical *motifs* (small musical ideas).

Crossover

This operator will create new solutions (children) from their parents in the last generation. Which can replace their parents in the new population; while the better melodies go untouched due to *elitism*. We set what percentage of the solutions will create new children; this is just a random probability and not based on how great their fitness score is. This is the reason why we use the elite operator to maintain the top solutions in case they do not pass through the generation, while still giving a chance for the worse scored solutions with perhaps rare motifs that are unique to that one individual in the population. We set this to 80% initially but more testing needs to be done on its impact to the population. We also set a replacement method here; we can use generational replacement, which will create a new population entirely from the children of the population including the solutions that are 'elite'. A 'Delete Last' approach can be used also, this is a mechanism which replaces the weakest solutions in the current population with their children.

Mutation

This operator will aim to provide some variety in the solutions; it traverses each individual in the population and based on a specified probability will swap one note with another to hopefully provide some more variety in the population.

We will discuss the effectiveness of using the different methods in the later analysis section of the report.

Selection

This operator controls how the members in the population are used. We can use *fitness proportionate selection (FPS)* or also known as *roulette wheel selection (RWS)* to locate potentially useful solutions for recombination this dictates how likely each individual is to be selected to crossover. In FPS, the more fit or highest potential to combine and make a fit solution means that they have more likely to be chosen whereas the less fit individuals are more likely to be eliminated.

Similarly, we can use *Stochastic universal sampling (SUS)* which selects solutions in the population by repeated random sampling by using a single a single random value that is used to sample all the solutions. This is useful if we want the weak solutions to get a chance to be chosen. It also reduces the *bias* that appears in fitness proportionate selection.

Another option is to use *tournament selection*, which is different to the two other approaches. Instead this selection mechanism runs “tournaments” of a few melodies inside the population that are chosen at random. The melody with the highest fitness in that tournament is selected for crossover. We can control a *selection pressure* operator which selects how large the tournaments are. The larger the tournament then it’s more likely the less fit individuals will be phased out of the next population and its phenotypes or genes will be less likely to be passed on.

4.5 – SYSTEM IMPLEMENTATION

Note

The note class is how we have developed the notion of an individual musical note in this software. The primary overloaded constructor for this class looks like this:

```
public Note(string note, bool tied, bool sharp, bool flat, double length)
{
    NoteName = note;
    Tied = tied;
    Sharp = sharp;
    Flat = flat;
    Length = length;
}
```

Figure 60: Note Constructor code

We can see that are notes are made up of different musical features:

- a **string** variable **note** for them name of the pitch of the note.
 - E.g. A1, B1, C1, C3, C4 – first letter is the note, second is the octave
- a **double** variable **length** which states the ‘length’ of that note:
 - E.g. 1 = whole note, 0.5 = half note, 0.25 = quarter note, 0.125 = eighth note
- A series of **Boolean** variables are also included these will state if there are any extra details to add to the note
 - **Tied** – dictates whether the note is tied over a bar in the music or not
 - **Sharp** – dictates whether the note is a sharp (No use in this stage of program)
 - **Flat** – dictates whether the note is a sharp (No use in this stage of program)

This class also has a list of functions or methods that can be used

- **Sharpen ()** – will take the note and set the sharp bool member of the note to true
 - Will also set flatten to false if it is true
- **Flatten ()** – will take the note and set the flat bool member of the note to true
 - Will also set sharp to false if it is true
- **Increment ()** – takes the current note and ‘increments’ by one, so it looks for what the note the next step up would be (which is defined explicitly) and assigns that note to the new note that is one step up.
 - E.g. A2 would be incremented to B2
- **Decrement ()** – takes the current note and ‘decrements’ by one, so it looks for what the note the next step down would be (which is defined explicitly) and assigns that note to the new note that is one step down.
 - E.g. C3 would be decremented to B2
- **ToMIDI()** – converts that current note into a midi pitch
 - E.g. E2 = 40
- **ToNumericPitch()** – converts note name into a pitch as defined in program
 - E.g. F1 = 4, G1 = 5, A1 = 6, E2 = 10

Cantus Firmus

This class does the reading of the Cantus firmus from the text file. Will split each string into individual notes and pitches and create **Note** objects for all them, which is added into an array of notes.

This can then be used in the program to print this cantus firmus and pass into the counterpoint generation classes.

Program

The program class is the main class of the piece of software. The main function is the first function that will be executed by the software.

This class will mainly be writing the initial text to prompt the user to select which species of counterpoint to be generated. The actual use of this was documented in the earlier UI section. This is implemented with an **Initialise()** function which is the first thing called by the main function, this has a series of **Console.WriteLine()** and **Console.Read()** statements which will print to the console and retrieve the user input for the selections.

The input from the user is taken and passed into a **CreateSpecies()** function which takes that input, reads in the cantus firmus note array, and splits it into a form that can be easily readable. When the cantus firmus prints to the console, an object of the specific type of counterpoint to be generated is created. This can be an object of either of these classes: **FirstSpecies**, **SecondSpecies**, **ThirdSpecies**, **FourthSpecies** or **FifthSpecies** and the cantus firmus array of notes is passed into those objects.

Species Counterpoint

We have 5 similar classes for each species of counterpoint but a lot of the same ideas are reused so this section will cover the overall development and process and then go into specific ideas for each species of counterpoint in the following ‘Experiments and results’ section.

This part is the large majority of the development of the project – this is where the music is generated and the genetic algorithm itself is created. This section may seem rather small for being the main part of the project but a lot of detail was covered in the species counterpoint design and implementation earlier and the above section on the overview of the genetic algorithm.

Genetic Algorithm

This part will cover the process of all individual species; In the **GenerateMelody()** function a list of notes are created one for the passed in Cantus firmus melody and an empty list for the to be generated counter melody.

```
internal void GenerateMelody(Note[] cf)
{
    //Assign cantus firmus melody
    cantusFirmus = cf;

    //Melody constants - zero indexed
    const int melodyLength = 11;

    //Default GA constants
    const int populationSize = 1000;
    const int elitePercentage = 10;
    const double crossoverRate = 0.8;
    const double mutationRate = 0.8;

    //Get available notes
    var notes = CreateNotes().ToList();

    //Create Population
    var population = new Population();

    for(var p = 0; p < populationSize; p++)
    {
        var chromosome = new Chromosome();
        foreach(var note in notes)
        {
            chromosome.Genes.Add(new Gene(note));
            chromosome.Genes.Add(new Gene(note));
            chromosome.Genes.Add(new Gene(note));
            chromosome.Genes.Add(new Gene(note));
            chromosome.Genes.Add(new Gene(note));
        }

        var rnd = GAF.Threading.RandomProvider.GetThreadRandom();
        chromosome.Genes.ShuffleFast(rnd);
        chromosome.Genes.RemoveRange(0, 129);

        population.Solutions.Add(chromosome);
    }
}
```

Figure 61: Genetic Algorithm initialisation

The figure above shows how we set some defaults at the start of this function. We set a population size, elite percentage, crossover rate and mutation rate that will be used to create the **population** of chromosomes (melodies).

A **CreateNotes()** function is then called to get every available note that can be used and then the population is created by adding 5 duplicates of each of this available note. This is to ensure that each initial melody doesn't just have one version of each note as if we ever need repeated notes it may take a long time for crossovers to create melodies with the notes we need.

Now after adding these to the population we can shuffle the list randomly using the genetic algorithm frameworks native shuffle functions and then remove a range of these notes so only 11 remain. This process repeats until there are enough melodies in the population to match the population size defined earlier.

```

        population.EvaluateInParallel = true;
        population.ParentSelectionMethod = ParentSelectionMethod.StochasticUniversalSampling;
        population.LinearlyNormalised = true;

        //Initialise elite operator
        var elite = new Elite(elitePercentage);

        //Initialise crossover operator
        var crossover = new Crossover(crossoverRate,
                                     true,
                                     CrossoverType.SinglePoint,
                                     ReplacementMethod.GenerationalReplacement
        );

        //Initialise mutation operator
        var mutate = new SwapMutate(mutationRate);

        //Initialise Genetic Algorithm object
        var ga = new GeneticAlgorithm(population, CalculateFitness);

        //Subscribe events to genetic algorithm
        ga.OnGenerationComplete += ga_OnGenerationComplete;
        ga.OnRunComplete += ga_OnRunComplete;
        ga.Operators.Add(elite);
        ga.Operators.Add(crossover);
        ga.Operators.Add(mutate);

        ga.Run(Terminate);

        Console.Read();
    }

```

Figure 62: Genetic Algorithm initialisation

The above code shows how we then initialise the population object and tailor its “settings”. We can either set **evaluateinparallel** which evaluates each chromosome in population at the same time to true or false and we can also change the parent selection method as described in the previous genetic algorithm section – here we’ve set it to Stochastic Universal Sampling but can also use tournament selection or fitness based selection.

We then create objects for the mutation operators so we can apply the elite, crossover and mutation rates that we set previously. Note in the crossover object declaration, we can also set it to double point crossover instead of single or use a delete last replacement method approach as opposed to generational replacement – the details of these were shown in the previous GA section also.

We then subscribe certain events to the genetic algorithm object so that they are automatically called; when an optimal solution is created for **OnRunComplete** and then one that will be called after every iteration of generation called **OnGenerationComplete**. In addition, a **CalculateFitness** and **Terminate** delegate is used.

CalculateFitness()

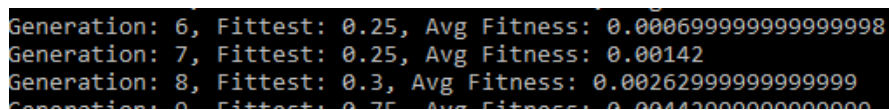
This is the fitness function of the genetic algorithm, all the rules discussed in the species counterpoint section will be implemented here and if any rules are broken then the ‘score’ of that melody is punished.

We implement these rules by creating lists of all the individual harmonic and melodic intervals and then assessing if they meet the rules by looking and comparing certain indexes. Examples of this in pseudocode were shown in the species counterpoint section and the code for the first species counterpoint fitness function will be included in the appendices of the report.

After a melody is assessed a score is returned which is then assigned to that specific individual in the population, and all the other individuals will then be assessed also.

OnGenerationComplete()

After each individual in the population is assessed the program prints what generation has been processed, it gets the top individual and prints its score and then prints the average fitness for the whole population.



```
Generation: 6, Fittest: 0.25, Avg Fitness: 0.0006999999999999998
Generation: 7, Fittest: 0.25, Avg Fitness: 0.00142
Generation: 8, Fittest: 0.3, Avg Fitness: 0.0026299999999999999
Generation: 9, Fittest: 0.75, Avg Fitness: 0.0044399999999999999
```

Figure 63: Console writing

Terminate()

This delegate returns true if a “stopping condition” is met, this can be explicitly set, so when a fitness score of 1 (perfect score) is reached, terminate will return true and no further evolution will occur. The functionality also exists to stop at a certain generation.

```
public static bool Terminate(Population population, int currentGeneration, long currentEvaluation)
{
    return (population.GetTop(1)[0].Fitness >= 1 || currentGeneration > 99);
}
```

Figure 64: Terminate delegate

OnRunComplete()

This is run when terminate returns true, it will simply get each note in the cantus firmus and countermelody, convert them into MIDI format using the methods in the Note class and then this will be written to a file ready to use for the audio playing application discussed earlier.

The program is then reset and the user can choose another species to generate.

4.6 – EXPERIMENTS AND RESULTS

For this project, a lot of time has been spent on ensuring the algorithm creates an optimal solution. With genetic algorithms, this is done by modifying the parameters of the algorithm until a solution converges.

Effect of mutation rate

The mutation rate is a measure of probability of how likely a gene (note) in an individual will be swapped with another random gene (or note).

In the genetic algorithm, the mutation rate can be modified. The graph below shows how changing the mutation rate can affect the speed of convergence. This was documented by running the algorithm 100 times and taking an average.

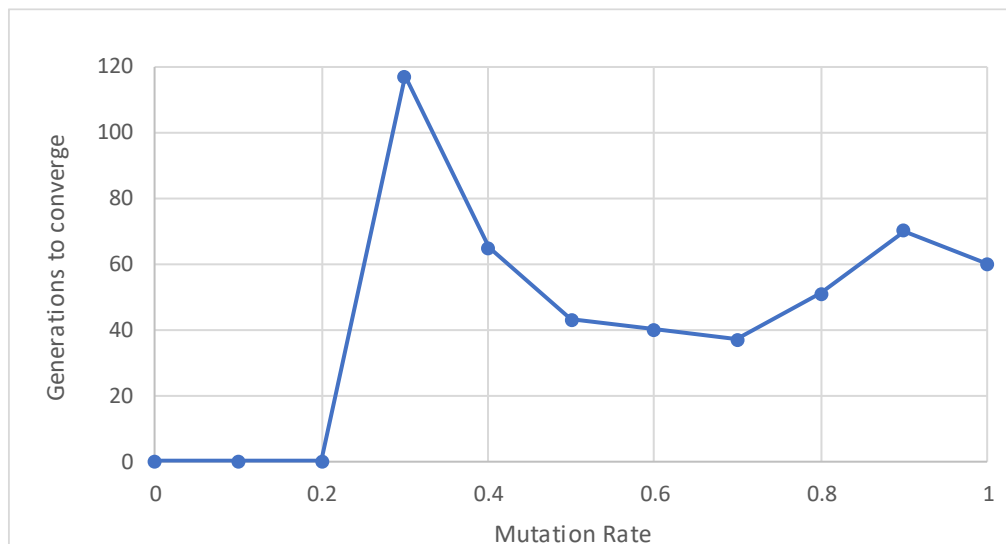


Figure 65: Mutation rate against convergence

This clearly shows that a low mutation rate means that a solution will not be found, and then as the mutation rate increases the solution is found fast until after a mutation rate of 70% where it starts to take longer for a solution to converge, with 100% mutation rate then the solution is found a bit quicker. There is a clear minimum at 70% so this is the rate we should use.

To conclude a higher rate means that the algorithm searches more areas of the search space but the very high mutation rates do allow the population to converge to an optimum. The lower mutation rates also may not converge to an optimal solution as well as having a less varied population generally.

Effect of crossover rate

The crossover rate is a probability that is inspired from sexual breeding; this is a measure of how likely two solutions can breed and swap around their genes in crossover.

The genetic algorithm's crossover rate may be altered also, we can see that with a low crossover rate then an optimum is usually never reached. After we set the probability to over 60% then a solution converges, the higher the crossover rate the faster a solution is found but using 100% crossover probability results in worse results even though a solution converges.

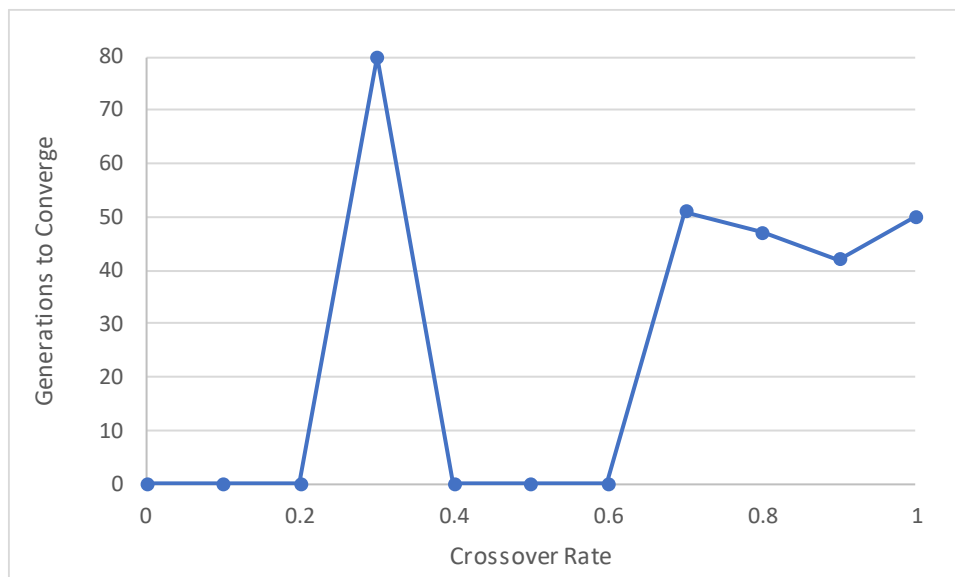


Figure 66: Crossover rate against convergence

Single crossover is a type of crossover is where a random point in the melody is selected and split; one of the split parts is exchanged with another melody's split part. *Double point* is where two points are selected to split off and be exchanged instead.

The table below shows how these two types of crossover result in a different performance. The numbers of the vertical axis state what number of species counterpoint is used and then the horizontal axis shows how many generations it takes on average for a solution to converge. Note that in this example some of the melody lengths are different in the species so we only look at the two bars for each individual species rather than compare them to others.

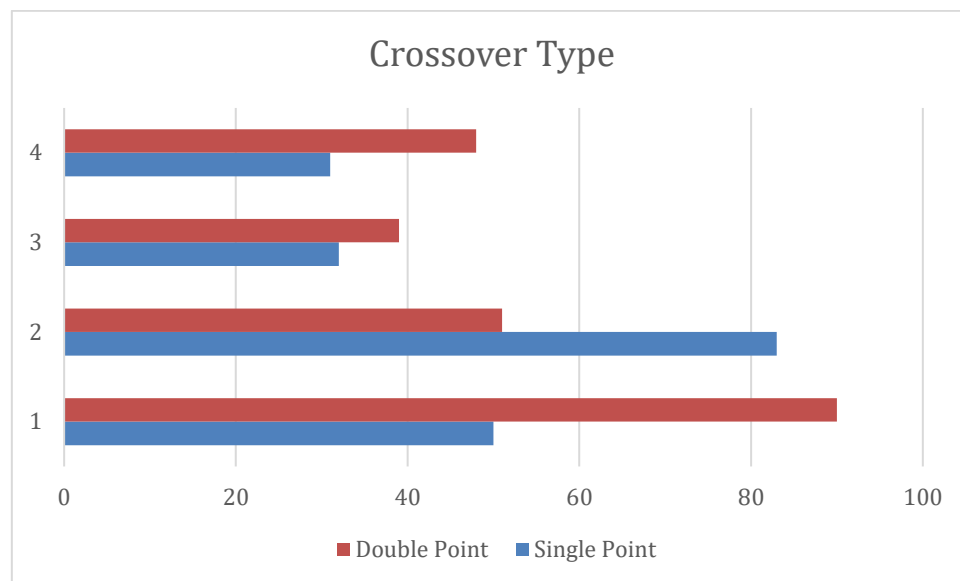


Figure 67: Crossover type against convergence

We can see in most cases, that single point crossover will result in a solution converging faster but that this doesn't seem to be the case for every species. In species, 1, 2 and 3 the single point is better but in 2 and 5 (occluded from the graph as is very different to other species) then double point seems to result in better performance.

Generally, single point is better as because it only takes one point, the sequence of notes are not necessarily 'broken' instead it just swaps around from different melodies. This means that in the species where a satisfactory solution takes longer to converge then crossover could ruin the good sequences of notes that occur. The complexity in 5th species means that this is the case always so we want either a low double point crossover or a normal rate single crossover.

To conclude the graph here is not overly useful as if we choose a cantus firmus with more notes then double point will be worse as it will be harder for a solution to converge but it can very powerful in solutions that converge more quickly.

Effect of elitism operator on convergence

Elitism is a process where a proportion of the fittest candidates enter the next generation unchanged. The graph below supports the conclusion that generally the higher the elitism rate the longer it takes to converge unless the elitism percentage is below 10%,

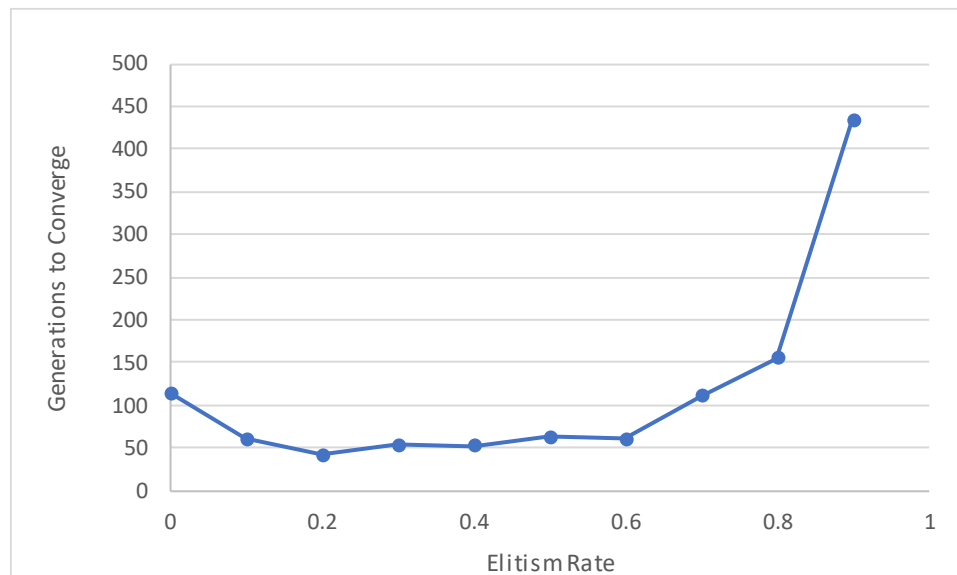


Figure 68: Elitism rate against convergence

These results occur as if the elitism rate is high then less new solutions are being made, providing less variety in the population. A very low elitism rate will mean there is a lot of variety in the population but the best solutions are not ensured to enter the next generation so there's a chance good solutions or "rare" genes may not survive through the process so a solution does not tend to converge.

This graph can be a bit misleading because the higher the elitism rate then the faster each generation will go through this is because only a few new solutions are made in each generation and need to be processed but the graph results still seem to be a show of what rate is good.

Effect of population size on convergence

The population size dictates how many individuals exist in any given generation. This rate can be modified. The graph below shows that the higher number of individuals in a population mean that a solution converges in less generations but this is misleading. The more individuals in a population then even though there are less generations needed to converge it takes a much longer time as more crossovers and mutations need to be processed.

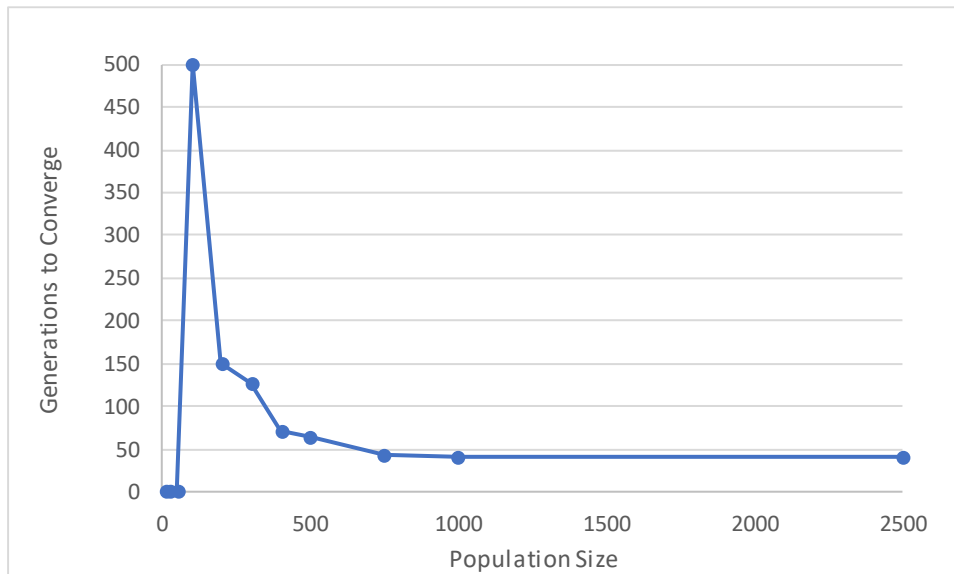


Figure 69: Population size against convergence

If working on a high-performance computer then this wouldn't be much of an issue but on a standard commercial computer anything above 1000 takes a very long time to process. 750 seems to normally work out with satisfactory results but in species like 5th species population and any long cantus firmus then this may take longer so in general we want to ensure the population is above 100 but below 1000.

Effect of parent selection methods

Earlier sections mentioned the techniques used in parent selection. The graph below shows how each of these techniques result in differing performance across the species.

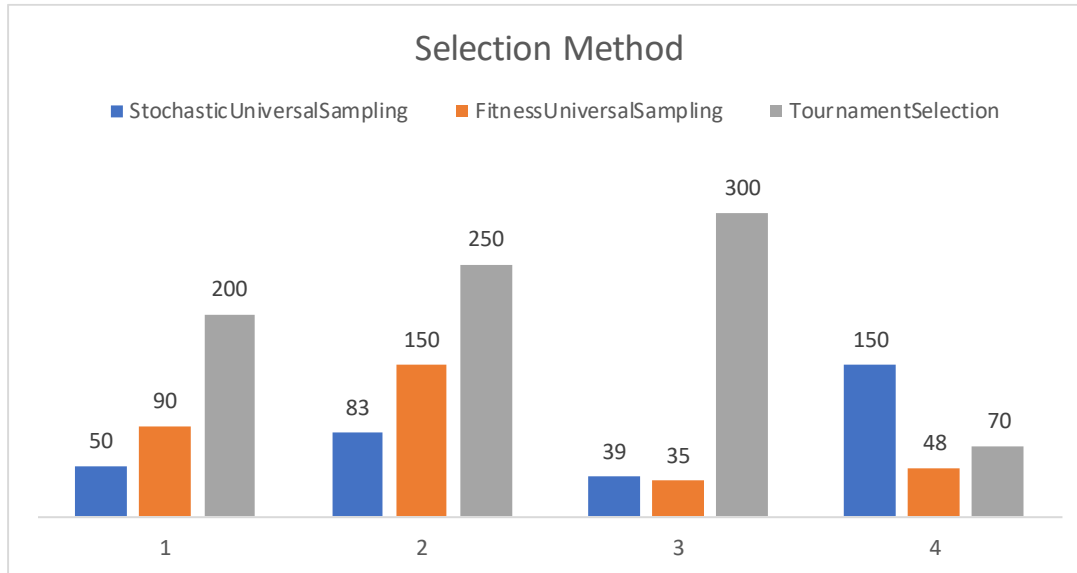


Figure 70: Selection method against convergence

It shows that in most cases Stochastic universal sampling (SUS) performs the greatest, this is expected as it gives low scoring individuals a chance to breed which could have interesting melodic devices that would go unnoticed in the other variants – generally Fitness Proportionate Selection (FPS) has bad performance when one member has high fitness in comparison to the others and this is mostly the case besides in the more complicated species. When we use a smaller population size or have closer rated individuals then FPS or tournament selection can perform well.

Effect of automatic fitness function

I created a prototype, which allowed the user to create the fitness score for each melody generated. The main problem in this approach is that the population is usually at about 1000 melodies and converges anywhere from 70-1000 epochs in an automatic fitness function.

This would take a long time to do with a user critic, reducing the population size just increases the number of epochs needed for a solution to converge massively and any increase in population would also take a long time.

I removed the score editing parts of the fitness function and set them as triggers in the program so it can easily be seen if anything improves, the odd individual would have one characteristic that's needed but this approach takes too long to do any meaningful testing.

Neural Network approach

To tackle the issues that came from generating 5th species counter, research was put into considering an alternative solution.

An artificial neural network (ANN) is a processing device used in *supervised* learning problems. It consists of interconnected layers; the *input layer* which contains the input features of the artificial neuron, the *hidden layer* does computation on the inputs and their respective weights and then an *output layer*. The networks' adaptive weights are then trained using the *back-propagation* algorithm, which calculates an error and manipulates the weights using the delta rule. The network aims to predict an accurate output for the specific inputs without knowing any specific domain knowledge or rules like in the GA used in this project.

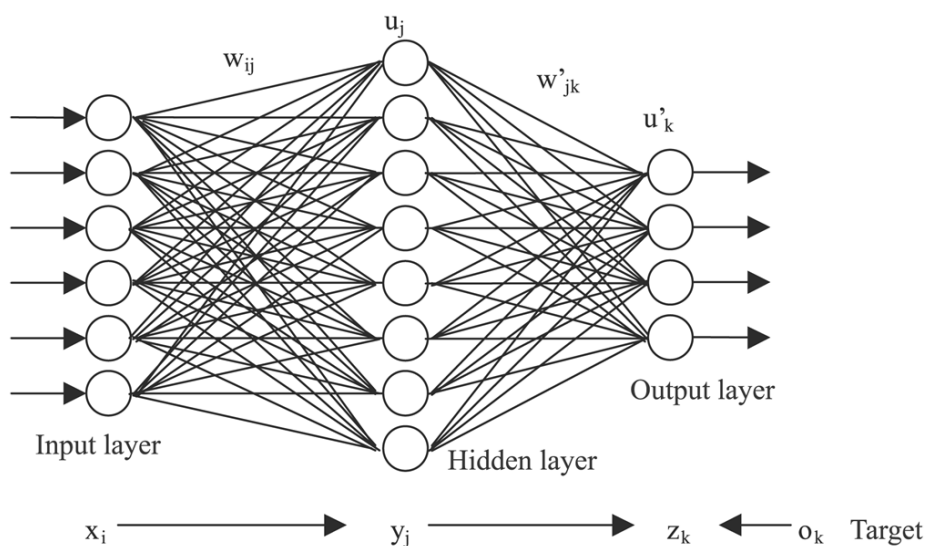


Figure 71: Neural Network Architecture (<https://www.extremetech.com/wp-content/uploads/2015/07/NeuralNetwork.png>, 2017)

I researched into an implementation using a *Restricted Boltzmann machine* which is a two-layer neural network consisting of a *visible* and *hidden layer*. Nodes in one layer are connected to the next layer but not connected to the nodes in the same layer. I came across the approach developed by Dan Schiebler (2017).

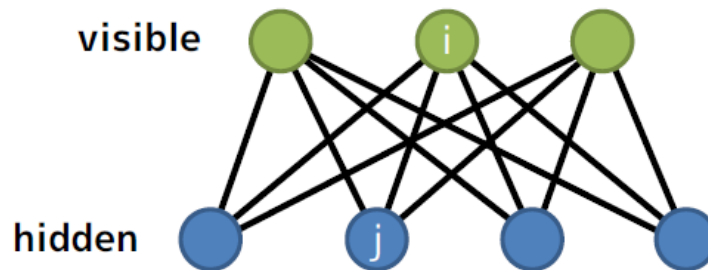


Figure 72: Restricted Boltzmann Machine Architecture (Saito, 2017)

Each node ‘decides’ whether to transmit the data it has received by making a random decision depending on the nodes that it is connected to.

The program created from this project allows the user to set a series of *hyperparameters* which dictate the size of the visible and hidden layer, the number of epochs that training goes through and the number of training examples needed to be trained on.

A matrix for the inputs and weights are then created along with two bias vectors for each layer that adjust the probabilities.

Gibbs Sampling is used for learning, it creates sample of input based on a multivariate probability distribution, which constructs a statistical model where each state depends on its previous state then randomly obtains a sample of the probability distribution created.

A sample of the original input and the generated output is taken and the weight matrix is updated based on their differences. The model is then trained for each example one at a time, and then the Gibbs chain is run to create a sample from the sample which is then shaped into a midi file for playback.

I edited this code and gave it examples of counterpoint, which allowed the network to generate small snippets of counterpoint music. Training took a long time, 100,000 epochs over 20 training examples seemed to create something close to counterpoint but still with a lot of issues melodically – the music would take rests in strange positions and would not use “nice” sounding rhythms most of the time.

This approach may have taken longer and still quite far away from true 5th species counterpoint but this approach seemed to generate a music more natural sounding sample of music and further research into better training samples and other neural network architectures could produce impressive results.

4.7 – TEST STRATEGY

This project used a huge amount of incremental testing – any change to the fitness function could bring up problems with the way the genetic algorithm was set out prior to that and was highly temperamental as shown in the previous section of the report. Sometimes when having an inefficient design of the genetic algorithm it could take a long time to converge so the main thing I looked for when testing this project was whether this configuration of parameters was quicker than the previous one, this seemed to be an effective way to judge whether the genetic algorithm was better than the previous one.

Towards the end of the project, I realized that I could fully implement the fitness function and then focus on changing the parameters but sometimes this would throw up other issues. There could be bugs in the melody so I would have missed them so I would not know where the specific problem lied. The problem could be from the parameters or it was impossible for a solution to converge as there are opposing rules or if it is difficult to meet those rules.

To solve this I figured a test-driven approach to this project was the best, I would make slight changes and test to see if a solution could converge if not then I'd discover why – is there opposing rules? Alternatively, is it impossible to reach this condition or I'd discover if making changes to the initial population would help. This obviously took a long time and at certain times it could take a long time to make changes to the algorithm that would increase the efficiency but this allowed gave me the chance to make conclusion on when to use certain genetic algorithm techniques and when not to as discussed in the experiments section prior to this.

When audio was implemented black-box testing was performed very easily, I could easily remove rules, generate a melody and listen to see if the rules were actually adhered to and then with all rules I could generate the melody and self-assess whether I thought the solution was good enough else I could make changes to musical problems I could hear.

5 – EVALUATION

In this project, my aim was to create a working genetic algorithm process for developing some form of music. I focused specifically on countermelody generation for the type of music and hoped to create acceptable countermelodies at differing stages of complexity. Overall, I believe I met my objectives and each objective's success criteria will be documented below in the subsection, as well as showing any difficulties that occurred during that objective implementation and how they could have been better managed and improved. This section should critically review the general process for achieving these objectives.

5.2 – PROJECT ACHIEVEMENTS

Objective 1 – Implement a system that generates first species counterpoint.

This first development objective was highly important to the project development. The aim was to create a genetic algorithm that would generate simple counter melody that moved note against note against the cantus firmus, this would then serve as a base to develop all the other stages of counterpoint. This objective was successfully reached and an accurate depiction of first species counterpoint was able to be generated easily and relatively quickly. A lot of iterations took place to test different ideas of implementing counterpoint programmatically but in the end, I believe the solution that was developed worked elegantly and provided a good base to develop future species of counterpoint so this objective was successfully met.

Objective 2 – Generate second and third species counterpoint.

This objective was more complex but the base provided from the first species counterpoint has allowed the creation of similar genetic algorithms that would successfully generate second and third species counterpoint. The algorithm has more complex rules and had to generate more notes to fit these composition rules. The results are fairly accurate for both species and produce generally good results but more work could be done in the future on creating more interesting third species counterpoint. Implementing these species led me to the conclusion that developing automatic music generation software will be limited by the knowledge and understanding of the developer's music theory knowledge. The developer can implement their own bias on how the music should sound so spending more initial time talking to more developed musicians could have allowed me to create more interesting results for this part. My initial decision to limit the musical notes to natural notes and not include any notion of key signatures and accidentals can sometimes make the music seem a bit 'unfamiliar' but this is still correct in music theory. Later I could easily create more tonal and approachable music for a wide audience – still I feel like this is still great progress and any listener should be able to hear the melodies generated are still quite nice.

Overall I believe this was a successful implementation of the objective, the output is largely accurate and more time spent on thinking about the project musically would be able to be implemented very easily with the projects current implementation.

Objective 3 – Generate fourth species counterpoint and florid counterpoint

Fourth species counterpoint generation worked very well, I was successfully able to introduce the idea of suspensions and syncopated rhythms into the generated music. Not only this but the program creates solutions very quickly and accurately. As always with music this can be expanded on, there are many complexities and advanced approaches to evaluate suspensions and the current state would give me a platform to approach these more advanced functions.

Fifth species counterpoint generation as expected earlier on wasn't so much of an easy task, a lot of work was put in to create any accurate output as the inclusion of varied rhythms created a lot of complexity. At the start of the project I didn't really expect to be able to create very nice fifth species counterpoint so this was more a stretch goal for the project. I think although the solutions are generally accurate that this species is probably best to not be approached with genetic algorithms, the inclusion of these extra rhythms makes it a much more difficult computational process and modern machine learning ideas utilizing neural networks and deep learning processes should be able to generate much more accurate and enjoyable results.

This objective was still successful as I successfully created an implementation that generates output for both species, although I did think the approach would be difficult and wouldn't work too well it's allowed me to explore why this was the case and research further into other implementations.

Sub-Objective 1 – Compare effectiveness of GA

This was not really a huge development project; I set out to compare my implementation with a user critic. These findings are outlined in the experiments section along with statements on the advantages of using genetic algorithms with a fitness function rather than a user critic. I believe this objective was a success and gave me the results I expected. Later I wrote about some research of how a neural network could handle the same task. I would have liked to have made some of my own implementations of techniques and perform some statistical analysis on them but I didn't get them time.

Sub-Objective 2 – Create output for the generated music

One aspect that has slowed down the project is that the actual audio generation of the melodies took a long time to get working. This means that when a melody is created I could not listen to the melody to be able to use my hearing to find out what is wrong with the melody more easily. Although I wasn't able to get a solution that creates the midi track for 5th species counterpoint working in time.

This objective took a lot longer to implement than expected but I have successfully created a solution that generated a MIDI file that can be played. The overall process of doing this isn't very robust or fluid, the user will have to use file systems and the other python application to do this but the user will still be able to listen to the generated output so I believe this objective was successfully met. But, there is plenty of room to improve this process; implementing the midi creation in one application and I could produce some visual output for seeing the structure of the melodies, the user can easily drag the file into any free notation software but it would be nice to have visualization for this inside the program.

5.3 – FURTHER WORK

Although I am happy with the work produced, many improvements could be made. I have contacted former music teachers and I hope to study counterpoint more and get a consensus across these musicians to create a better rule set to create counterpoint – Fux’s rules are great but there are more complex interpretations of counterpoint that I could use to create a better solution. Taking knowledge from other musicians stops me creating a biased few of creating counterpoint. One lesson learned from this project is that the rules dictate exactly how the music is formed so getting more viewpoints leads to less biased solution.

I plan to write this code without a library now I have a better understanding to give myself more freedom and have it web-enabled by either writing it in JavaScript or by continuing to use C# and writing an ASP.Net server that can dynamically produce counterpoint from the users input. This will allow me to showcase my project easier and I could easily write visualisations that show the produced output in visualisation libraries like BOKEH rather than the user just using a command line. As well as this, I could possibly make a mobile app that allows the user to add or sing their own input as a cantus firmus for the program.

I am inspired by the neural network approach and have done more research in recurrent neural network and convolution neural networks like WaveNet (van den Oord et al., 2016) that produces music by looking at the raw audio waves rather than MIDI format. There are not enough interfaces and libraries to easily incorporate MIDI into a program so working directly with waves will solve this. I have already started to work on prototypes of this in TensorFlow and intend to carry on my research in the area.

I also want to take what I have learned from this project and work at a larger scale project incorporating music. One idea is a tool that allows a musician to write a solo or melodic line and a piece of software could generate a backing track and play back the full piece of music. I believe this technology will be great for musicians who want to write their own music but do not know enough theory and should lower the skill gap for creating great music.

6 CONCLUSION

My personal objective coming into this was to gain experience developing genetic algorithms. I have successfully done this and provided a fundamental knowledge that would allow me to implement this project again without the libraries that limited some parts of fifth species counterpoint. This experience has also given me a base to learn about other areas of natural computation and machine learning.

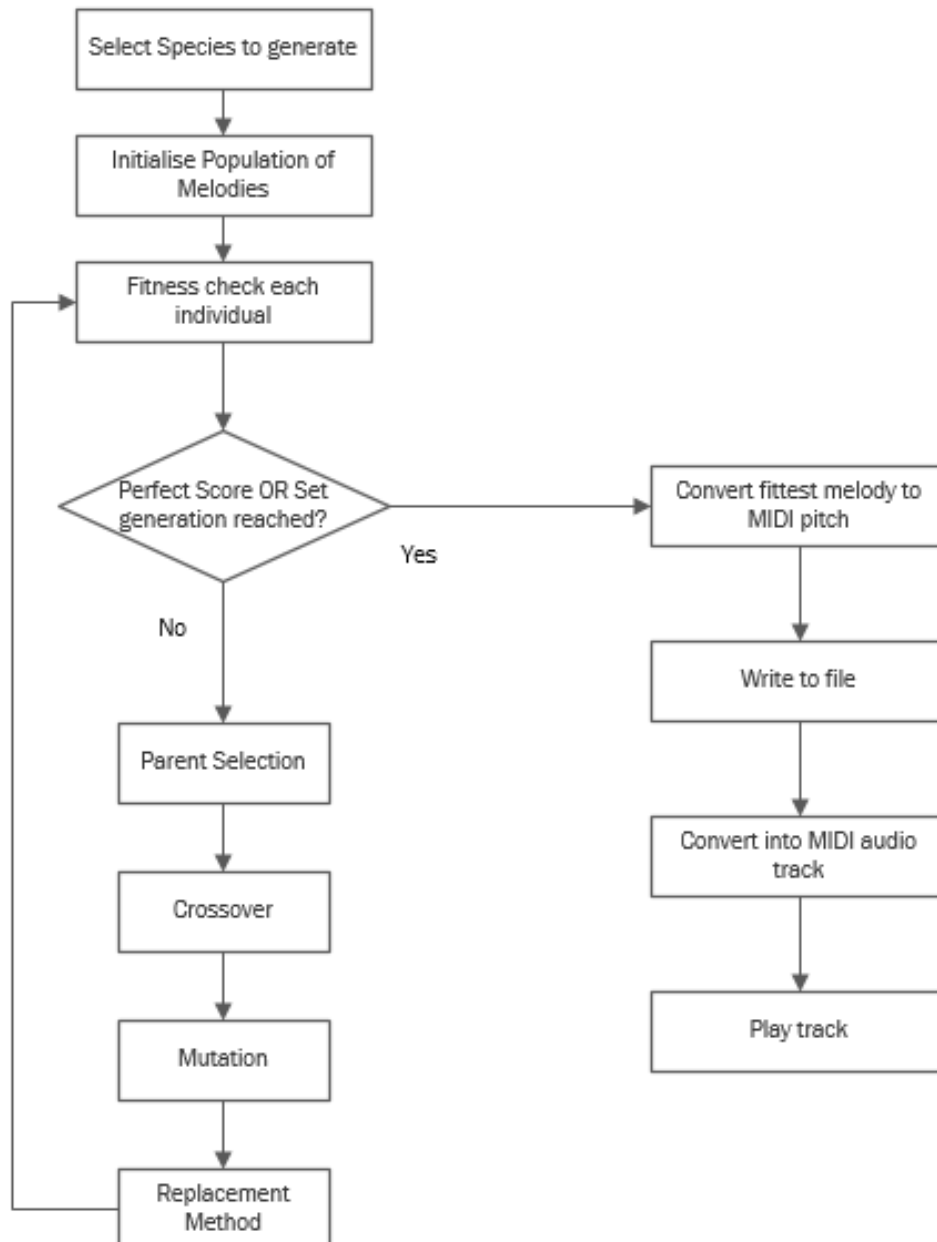
I have plenty of knowledge to create more projects in the realm of machine learning and computational creativity as discussed in the further work section and I am very glad I chose this project for my final year dissertation. I now plan to further study in this area of computer science and directly apply my knowledge gained from doing this course.

There are parts and processes I could have improved but this has only added to my understanding and has given me a practical knowledge of how to approach doing similar projects more efficiently.

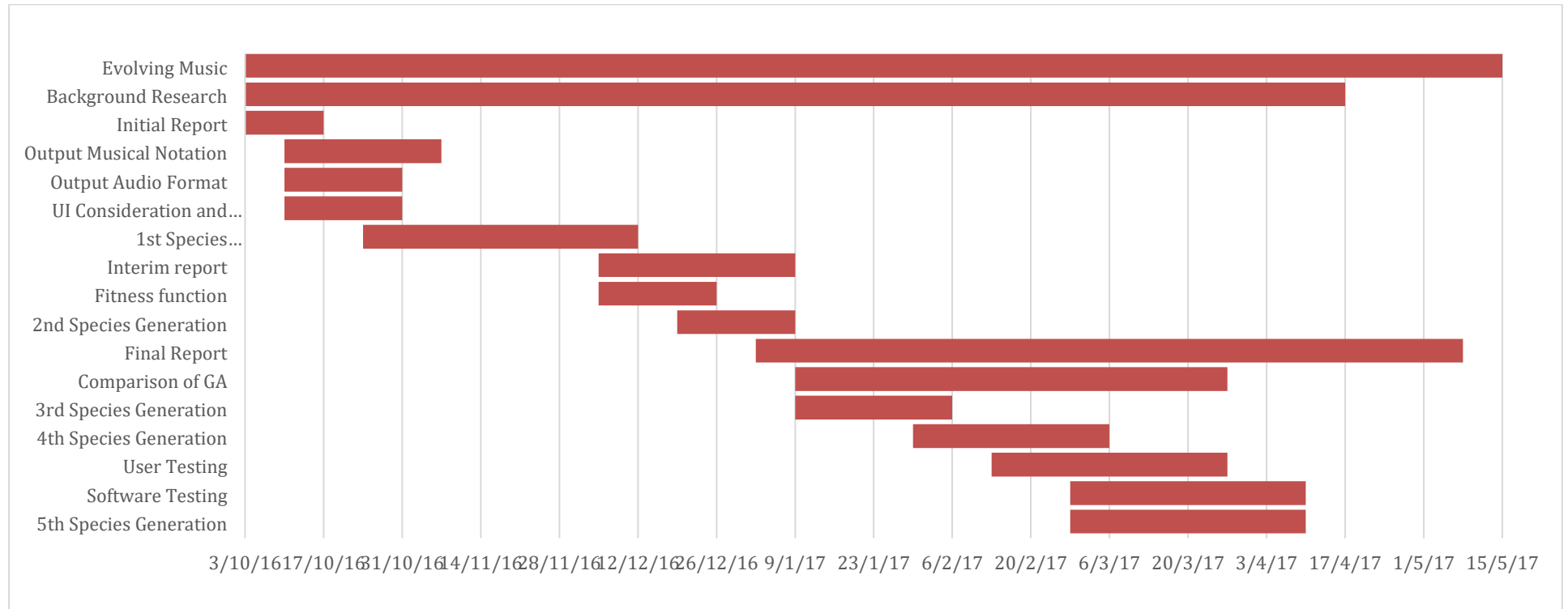
Appendix A: Original Task-list

#	Task Name	Description	Duration (Weeks)
1	Initial report	Write the interim report deliverable	2
2	UI considerations	Look into and decide whether any UI needs to be built, if so decide on the technology and create design diagrams for the UI.	2
3	Audio Output	Consider and create a system that will take a series of musical notes and create an audio-listenable file based on the notes that can be run on the computer or the program itself	2
4	Music Representation	Look into how the data must be stored to be able to create a notated musical score that show what the countermelody looks like. See if this can be made automatically programmatically.	3
5	1 st Species Counterpoint (without objective function)	Use GA techniques to generate a 1 st species countermelody. This will include making a base initial population, breeding, mutation and selection that we'll be able to use and modify for the next species' but excluding using an automatic fitness check. This will give us a good base to work on and allow us to compare the results when we implement the automatic fitness check.	7
6	Implement fitness check	Now we'll actually implement the objective function into our GA to generate 1 st species counterpoint	2
7	2 nd species counterpoint	Here we'll use the GA to generate 2 nd species of counterpoint, following the rules for the second species.	3
8	3 rd species counterpoint generation	Here we'll generate the 3 rd species of counterpoint, following the rules for the third species.	3
9	4 th species counterpoint generation	This should be more complex than the previous 3 species of counterpoint.	4
10	5 th species counterpoint generation	Here we'll implement "free-counterpoint" encompassing techniques from all 4 species which should be much more complex than the other species generated prior.	6
11	Comparison	We will write up about the effectiveness of using an objective function rather than user critics and look at how other evolutionary techniques and algorithms could be used differently for counterpoint generation.	6
12	User testing	We will get some students to test the programs results to see if they can tell a difference between generated countermelodies and real countermelodies over the same cantus firmus.	5
13	Software Testing	At this point we will set a content freeze on the project and focusing on testing, debugging and fixing bugs in the program.	3

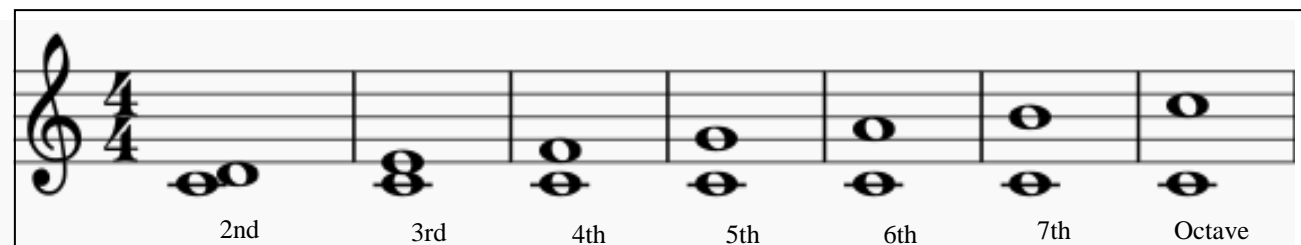
Appendix B: Algorithm Flowchart



Appendix C: Original Time-Plan



Appendix D: Musical Intervals



Appendix E: Risk Analysis

	Current Risk				Residual Risk		
Risk	Severity (L/M/H)	Likelihood (L/M/H)	Significance (Sev. x Like.)	How to Avoid/Recover	Severity (L/M/H)	Likelihood (L/M/H)	Significance (Sev. x Like.)
Loss of Source Code	H	M	HM	Regularly commit to source control.	L	M	LM
Hardware failure	H	L	HL	Make sure committing to SVN and have another computer available with required software installed.	L	L	LL
Multiple deadlines	H	H	HH	Must effectively manage time, allow for additional time to handle tasks at exam period, gantt chart to help keep track of what tasks need to be completed and when.	M	M	MM
Difficulty in learning how to implement genetic algorithms	H	M	HM	Methodically learn about genetic algorithms and carry on background research throughout the project	H	L	HL

End I/O will not work in audio/notation export	M	L	ML	Make sure to prototype the exporting of musical information before implementing the genetic algorithms	L	M	LM
Bugs/Shut down of software/APIs needed to export audio information	H	M	HM	Research into alternative ways to export the musical information before implementing one way	M	M	MM
Feasibility of Software	H	M	MM	Don't set goals that couldn't be completed in the time frame, evaluate the objective if its taking too long or is more complex than expected.	M	M	MM
Not enough time	H	L	HL	Research the individual tasks before allocating a time block for them, reprioritize objectives to recover.	M	L	ML
Not enough time to complete documentation/reports	H	M	HM	Write report throughout entire time frame and create a plan for the report, to recover would need to reallocate allotted time.	H	S	HL
Not enough time for software testing	M	H	MH	Make sure that objectives are reached earlier to ensure you have testing time	M	L	ML
Difficulty fixing bugs	M	M	MM	Design all algorithms before implementing them to minimize risk of bugs, must research into the bugs to recover	M	L	ML
'Gene Pool' hits a fitness 'cap'	M	M	MM	Ensure the less fit genomes can still breed and mutate to allow for a good amount of variation in the gene pool	M	L	ML

Not being able to code the program	H	M	HM	Use technologies that I have had experience developing in and do background research regularly, to recover I will look at the approach generally rather than specific to one language so I could implement the solution I had if I didn't know enough knowledge of the language	M	S	MS
Source Control loss	H	M	HM	Keep regular physical copies too, use university SVN	H	L	HL

References

- Aguilera, G., Galán, J.L., Madrid, R., Martínez, A.M., Padilla, Y. and Rodríguez, P., 2010. Automated generation of contrapuntal musical compositions using probabilistic logic in derive. *Mathematics and Computers in Simulation*, 80(6), pp.1200-1211.
- Andymurkin.wordpress.com. (2017). Software/MIDI | Music Electronics | Page 2. [online] Available at: <https://andymurkin.wordpress.com/category/softwaremidi/page/2/> [Accessed 29 Apr. 2017].
- Bäck, Thomas, David B. Fogel, and Zbigniew Michalewicz, eds. *Evolutionary computation 1: Basic algorithms and operators*. Vol. 1. CRC press, 2000.
- Back, Thomas. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.
- Cádiz, R.F., 2006. Compositional control of computer music by fuzzy logic.
- Cazden, N., 1945. Musical consonance and dissonance: A cultural criterion. *The Journal of Aesthetics and Art Criticism*, 4(1), pp.3-11.
- Dannenberg, R.B., Thom, B. and Watson, D., 1997. A machine learning approach to musical style recognition.
- Daida, Jason M., et al. "Of metaphors and Darwinism: Deconstructing genetic programming's chimera." *Evolutionary Computation*, 1999. CEC 99. Proceedings of the 1999 Congress on. Vol. 1. IEEE, 1999.
- Dostál, M., 2012. Genetic Algorithms As a Model of Musical Creativity--on Generating of a Human-Like Rhythmic Accompaniment. *Computing and Informatics*, 24(3), pp.321-340.
- Farbood, M. and Schoner, B., 2001, September. Analysis and synthesis of Palestrina-style counterpoint using Markov chains. In *Proceedings of the International Computer Music Conference* (pp. 471-474).
- Gibson, P.M. and Byrne, J.A., 1991, November. Neurogen, musical composition using genetic algorithms and cooperating neural networks. In *Artificial Neural Networks*, 1991., Second International Conference on (pp. 309-313). IET.

- Herremans, D. and Sörensen, K., 2012. Composing first species counterpoint with a variable neighbourhood search algorithm. *Journal of Mathematics and the Arts*, 6(4), pp.169-189.
- Komosinski, M., Kups, A., Leszczyńska-Jasion, D. and Urbański, M., 2014. Identifying efficient abductive hypotheses using multicriteria dominance relation. *ACM Transactions on Computational Logic (TOCL)*, 15(4), p.28.
- Learning, I., Learning, I. and Shaikh, F. (2017). *Introduction to Gradient Descent Algorithm along its variants*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/> [Accessed 3 May 2017].
- Löthe, M., 1999, September. Knowledge based automatic composition and variation of melodies for minuets in early classical style. In *Annual Conference on Artificial Intelligence* (pp. 159-170). Springer Berlin Heidelberg.
- MacCallum, R.M., Mauch, M., Burt, A. and Leroi, A.M., 2012. Evolution of music by public choice. *Proceedings of the National Academy of Sciences*, 109(30), pp.12081-12086.
- Polito, J., Daida, J.M. and Bersano-Begey, T.F., 1997, April. Musica ex machina: Composing 16th-century counterpoint with genetic programming and symbiosis. In *International Conference on Evolutionary Programming* (pp. 113-123). Springer Berlin Heidelberg.
- Saito, M. (2017). Restricted Boltzmann Machineの学習手法についての簡単なまとめ. [online] Mglab.blogspot.co.uk. Available at: <http://mglab.blogspot.co.uk/2012/08/restricted-boltzmann-machine.html> [Accessed 29 Apr. 2017].
- Shiebler, D. (2017). Musical TensorFlow, Part 1 - How to build an RBM in TensorFlow for making music. [online] Danshiebler.com. Available at: <http://danshiebler.com/2016-08-10-musical-tensorflow-part-one-the-rbm/> [Accessed 29 Apr. 2017].
- Strong, A. (2017). Guitar Rhythm Explained. [online] Classicguitarexperience.com. Available at: <http://www.classicguitarexperience.com/2016/09/guitar-rhythm-notation.html> [Accessed 29 Apr. 2017].
- Tijus, C.A., 1988. Cognitive processes in artistic creation: Toward the realization of a creative machine. *Leonardo*, pp.167-172.
- van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. and Kavukcuoglu, K., 2016. Wavenet: A generative model for raw audio. *CoRR* abs/1609.03499.

<https://www.extremetech.com/wp-content/uploads/2015/07/NeuralNetwork.png>. (2017).
[image].