Musical Instrument Synthesis Using Genetic Algorithms
Chris Eade
Computer Science BSc
2002/2003

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)___________________________
Summary

This project investigates the possibilities of using genetic algorithms in the synthesis of musical instruments. A number of different techniques are explored and evaluated.

Finally, a simple musical instrument synthesis engine incorporating a genetic algorithm is developed, and evaluated using subjective assessment.
Acknowledgements

I’d like to thank Derek Magee for his valuable advice and stimulating discussions. Without Derek’s guidance this project would have ended up down a very dark alley. I’d also like to thank Haiko Muller for his patience as my supervisor and Seth Bullock for his ideas and views. All much appreciated.
# Contents

1 Introduction ................................................. 1  
  1.1 Motivations for Synthesis ............................ 1  
  1.2 Motivations for Genetic Algorithms .............. 2  
  1.3 Motivations for Synthesis with Genetic Algorithms 2  

2 Literature Review ....................................... 3  
  2.1 Digital Representation of Sound .................... 3  
  2.2 Synthesis Techniques ................................. 3  
     2.2.1 Granular Synthesis .............................. 4  
     2.2.2 Wavetable Synthesis ............................ 4  
     2.2.3 Additive Synthesis ............................. 5  
  2.3 Genetic Algorithm Techniques ....................... 5  
     2.3.1 Markov Chains .................................. 7  
  2.4 Synthesis with Genetic Algorithms ................ 8  
     2.4.1 Additive Synthesis with Genetic Algorithms 8  
     2.4.2 Granular Synthesis with Genetic Algorithms 8  
     2.4.3 Granular and Wavetable Synthesis with Genetic Algorithms 8  
  2.5 Software Tools and Libraries ...................... 9  
     2.5.1 Sox .............................................. 9  
     2.5.2 Audio File ....................................... 9  

3 Method ...................................................... 10  
  3.1 Summary ................................................. 10  
  3.2 Instrument Choice .................................... 10  
  3.3 Grain Construction .................................. 12  
  3.4 Genetic Algorithm Outline ......................... 14  
  3.5 Measuring Population Fitness ...................... 15  
     3.5.1 Probability Matrix with Waveform Comparison 15  
     3.5.2 Static Array with Waveform Comparison ........ 16  
     3.5.3 Static Array with Amplitude Comparison ......... 16
Chapter 1

Introduction

1.1 Motivations for Synthesis

Musical instrument synthesis has long been used as a technique to recreate the sound of real musical instruments. Musical instrument synthesis is a highly desirable method of producing instrument sounds as it allows for the sounds of hundreds of real instruments to be stored and produced from the same device. This device is usually referred to as a Synthesizer. Using a Synthesizer to produce sounds has many advantages over real instruments, but also many disadvantages. Physical size and portability are two of the main advantages to using a Synthesizer. In theory, a Synthesizer can produce all the sounds of a whole 100 instrument orchestra, but at the same time taking up a small fraction of the space required to store such an ensemble. Now imagine this orchestra performing at 10 different venues over 10 days and the obvious portability advantages of a Synthesizer become clear. Using a Synthesizer has another major advantage. A person can reproduce the sound of any musical instrument without having the knowledge of how to play it. Synthesizers also open up new and exciting ways of creating synthetic sounds that have no basis in organic real instrument sound.

On the downside, taking away the unique way in which real instruments are played can remove a dimension of creativity from the instrument. But the main disadvantage of using a Synthesizer over a real instrument is that, due to the synthetic nature of the sound they produce, Synthesizers can struggle to produce sound natural enough to convincingly replicate real musical instruments. Natural musical instruments by their very nature have a randomness to their sound which means they never sound exactly the same twice. It is this disadvantage of Synthesizers that this project aims to look into, using one specific technique to try and produce a solution.
1.2 Motivations for Genetic Algorithms

Evolution in nature is controlled by a process called Natural Selection. Natural Selection is the mechanism that allows populations of life forms to adapt to their surrounding environment. Its works by individuals most suited to their environment living longer and breeding more than less well suited individuals, so passing on their well suited characteristics to the next generation. This fundamental idea was first proposed by Charles Darwin in his classic text, *On the Origin of Species* [6]. Simply, a genetic algorithm (GA) is an algorithm that uses the idea of natural selection to solve problems. A population is evolved by combining 'parent' properties (commonly known as genes in natural life forms) to form the next generation based on the fitness of individual population members. A goal is set for the population to achieve and the idea is for the population to evolve towards this goal and produce a solution to the set problem. GAs were first seriously considered by Holland in [11].

1.3 Motivations for Synthesis with Genetic Algorithms

Due to the organic nature of the sound produced by real instruments, it seems a valid concept to use GAs to synthesize musical instruments. By applying genetic algorithms to synthesis techniques this project aims to introduce natural randomness into musical instrument synthesis, thereby convincingly synthesizing the sound of real musical instruments to a better degree than current synthesis techniques. There exist many distinctly different types of sound synthesis, some more suited to GA application than others. Suitable synthesis techniques are identified and considered in Chapter 2.
2.1 Digital Representation of Sound

Efford [7] describes the sound output from a traditional microphone and amplifier as an analogue electrical signal in which voltage varies continuously with time. This analogue data must be represented digitally before it can be processed by a computer. A technique known as Sampling is used to do this.

Gabor [8] laid the foundations for sampling by considering the idea of individual sound events being perceived by the human ear as a constant sound signal.

Sampling is simply the process of taking measurements (each referred to as a ‘Sample’) of analogue signal amplitude at regular intervals [7]. The Sampling Rate is defined as the number of samples taken in one second, given in Hertz (Hz).

A phenomenon known as Aliasing is identified by Efford [7] whereby the sampling rate must by at least double the highest frequency present in the analogue signal being sampled to accurately represent that signal. A common sample rate, as used by CD audio, is 44,100Hz. This enables frequencies up to 22,050Hz to be accurately represented. As the range of human hearing is typically between 15 and 18,000Hz [7], a sampling rate of 44,100Hz is adequate for most applications.

2.2 Synthesis Techniques

There are 7 distinct sound synthesis techniques that can be drawn from the literature [9] [19] for consideration.

- Subtractive
• Additive
• Wavetable
• Physical Modelling
• Frequency Modulation
• Granular
• Amplitude Modulation

All these techniques are described in detail in the literature. Here three of the techniques, Granular synthesis, Wavetable synthesis and Additive synthesis, will be considered further. GAs require a population of individuals to work on and this is the reason for considering Granular and Wavetable synthesis further. Granular synthesis holds the possibility of using sound grains in some way to populate the GA. The loop, or sustain segment of Wavetable synthesis holds the possibility of populating the GA with the different sound samples that are used. Additive synthesis is considered further as it can be used to demonstrate another possible technique for synthesis with GAs suggested later in this chapter.

2.2.1 Granular Synthesis

Walling [19] describes Granular synthesis as synthesizing a sound by combining smaller grains of sound together. These sound grains can be any kind of sound, from simple sinusoids to complex samples. The duration of these sound grains is fundamental to the success of the technique. Gabor [8] came up with the idea of individual sound events being perceived by the human ear as a constant sound signal, laying the foundations for sampled sound. Traux [18] applies this idea to grain duration in Granular synthesis, suggesting a threshold of about 50ms, above which the produced sound is no longer perceived as continuous by the listener, causing the technique to fail.

2.2.2 Wavetable Synthesis

Walling [19] describes Wavetable synthesis as using sampled sound fragments to represent different parts of a synthesized sound. These parts usually consist of an attack phase, a loop (or sustain) phase, and a decay phase, as shown in figure 2.1. In traditional wavetable synthesis, a single sound sample, of the order of several seconds in duration, is repeated over and over again in the loop phase until the synthesized sound is no longer required, at which point the decay phase (represented by a different single sample) is initiated. The loop sample must have special characteristics so that when it is looped the individual loop segments cannot be heard. This is achieved by ensuring that the sample boundaries (beginning and end of the sample) form a continuous waveform when combined together. If this continuity is not ensured, often a very short blip can be heard at the sample boundaries. Obviously, this continuity must also be ensured at the boundaries between the different phases of wavetable synthesis.
In practical applications of wavetable synthesis, many additional techniques are employed to increase the feasibility of the method for practical synthesis applications. Walling [19] describes a technique called linear interpolation, which is often used to reduce the number of samples that need to be stored, by enabling each set of samples to represent a range of differently pitched sounds. Linear interpolation is an example of a technique that is used to enhance the real-life practicality of a synthesizer. Producing a practical synthesizer is usually a question of finding a balance between sound quality and cost, harnessing clever techniques to achieve this balance.

2.2.3 Additive Synthesis

Additive synthesis is one of the most basic forms of synthesis. Walling [19] describes additive synthesis as the summation of various sinusoids of different frequencies and amplitudes to create a more complex waveform. This technique works on the principle developed by Fourier that all waveforms are made up of simple sinusoids at different frequencies. Walling [19] points out that it is therefore possible, but in reality completely impractical, to synthesize every conceivable waveform using additive synthesis.

2.3 Genetic Algorithm Techniques

Ladd [12] describes a simplified process that GAs go through in terms of the following pattern of steps.

1. The GA is given a population and each member of this population is initialized with a random possible solution to the problem being solved.

2. Each population member is evaluated against the problem being solved to determine its suitability as a solution. A fitness value is assigned to the population member based on this suitability.

3. ‘Parents’ are chosen based on the fitness values of the population members, meaning that population members with higher fitness values have a greater chance of becoming a parent.

4. A new population is produced by reproducing the current population based on the parents chosen in step 3.
5. This new population replaces the current and a generation is complete. The GA then continues at step 2.

Of course some kind of termination mechanism needs to be added to this process otherwise the algorithm would just loop to infinity, even if it had found a perfect solution.

In reality a number of other stages need to be implemented in the process in order to ensure that the GA evolves, or converges, toward an optimal solution. The nature and extent of these additional stages are highly dependant on the problem being solved. Some of the most widely used additional stages are now presented.

*Crossover* is not explicitly mentioned in the above simplified process and is worth describing in more detail. Crossover basically defines the way in which parents are combined to form new population members [12]. The most common way of performing crossover is to choose a random place to split the parents problem solutions and combine these solution parts to form a new solution. A variety of other, often more complex crossover techniques exist, notably Holland’s approach of using small bits of the parent solutions, called schemata, mixed up to form a new solution [11]. Other techniques are considered in detail by Pawlowsky in [15].

*Selection* is mentioned in the above simplified process and is worth describing in more detail. It is simply the selection of ‘parent’ individuals based on their fitness. A number of different ways exist for performing selection, as analyzed by Blickle [3]. Ladd [12] describes one of the simplest forms of selection, *roulette wheel* selection. Roulette wheel selection is based on the concept of a gambler’s roulette wheel. A roulette wheel is constructed whereby individuals of a greater fitness have more entries in the ‘wheel’ than less fit individuals. A random number is generated as an index in the wheel and the corresponding individual at that index selected, as shown in figure 2.2. Roulette wheel selection is used throughout this project.

![Figure 2.2: Roulette wheel selection example.](image)

*Fitness Scaling* is a simple technique that can have a dramatic effect on the performance of a GA, greatly decreasing the number of generations required to find an optimal solution with certain types of problems, as observed by Ladd [12] in his Blackbox example. Fitness scaling increases the relative fitness of higher fitness individuals to give them a greater chance of reproducing. Ladd [12] observes
that the technique has the greatest effect when a close race phenomenon is present in the GA. Close race is when fitness values in a population vary within a narrow range of values, therefore making it difficult for higher fitness characteristics to gain a foothold in the population.

**Mutation** is used to introduce new characteristics into the population. Mutation is commonly implemented in crossover by defining a probability that each transferred bit of a solution will be randomly modified [12]. Ladd [12] suggests that mutation is most important where the initial random population may represent only a small subset of all possible solutions to the problem being solved. In this case, without mutation, it is highly likely that the GA will struggle (or even find it impossible) to converge on an optimal solution, as there is no way for it to discover new solution possibilities.

**Elitism** is the explicit copying of the most fit individual from the current generation to the next without any kind of mutation of its solution. In a conversation with Bullock [4], elitism was described as the product of the GA community’s paranoia of losing their most fit individual. In most cases, by using elitism a GA will always progress or remain static and never start evolving in the opposite direction of an optimal solution. However, Ladd [12] notes that it may be essential, with some problems, for a population to evolve away from an optimal solution in order to rid itself of unfit characteristics, before heading for an optimal solution. This must be kept in mind when using elitism, and may or may not be observed in the course of this project.

Ladd [12] identifies premature convergence, or the plateau phenomenon, as a common problem among GAs. This occurs when the population becomes homogeneous before an optimal solution has been found. Ladd suggests using a combination of elitist selection, fitness scaling and high mutation rates to ensure a robust algorithm and avoid premature convergence. It remains to be seen whether or not this combination will be effective with the problem presented by this project.

Ladd [12] also identifies the way in which fitness is determined by a GA as highly important to its success. This aspect is possibly the hardest part of this project; determining what makes natural sounds good, and then turning this into a fitness function to evaluate the fitness of the sounds produced by population individuals.

### 2.3.1 Markov Chains

Markov Chains are described by Carter [5] as a set of states linked by probabilistic transitions. They have no direct relation to GAs, but are sometimes utilized within GAs to represent problems as populations. There is a whole host of theory about Markov Chains, but most is not important for this project as only very basic Markov Chain concepts will be considered and used. More information can be found in [14], among others.
2.4 Synthesis with Genetic Algorithms

2.4.1 Additive Synthesis with Genetic Algorithms

After conversation with Magee [13] a possible method of synthesis with a GA is proposed. The technique involves utilizing additive synthesis controlled by a GA. The GA consists of a Markov chain representing the properties of different components of the additive synthesis such, as frequency, amplitude and phase. The GA is used to evolve the Markov chain transition probabilities between these components until desired characteristics are achieved.

One major downside to this method is that it provides no way of changing the sound dynamically within the time domain. After a sound is synthesized, that sound will not change whether it is played for one second or one hundred seconds. One of the distinctive characteristics of natural musical instruments, identified in the Chapter 1, is the randomness of the sound they produce. They never sound exactly the same twice. This method therefore holds possibilities for increasing the quality of additive synthesis, but will never be able to tackle the task of convincingly synthesizing a real musical instrument.

2.4.2 Granular Synthesis with Genetic Algorithms

One of the main advantages of using granular synthesis is that it allows for the variation in the time domain required. Roads [17] presents a method of granular synthesis grain regulation with GAs. Roads’ method uses a population individual to hold a grains parameters, such as frequency and amplitude. This limits the grains to being simple sinuosoids unless another form of synthesis, such as additive synthesis, is used to create the grains from more complex parameters stored within each population individual. This could create significant processing overheads. Roads [17] describes the technique as not needing to converge like a traditional GA, suggesting that it has been developed as a method for producing new synthesized sounds and not for the purpose of synthesizing an already present sound source, such as a musical instrument. For this reason the technique is unsuitable for this project, but some of the ideas may still be useful.

2.4.3 Granular and Wavetable Synthesis with Genetic Algorithms

After further conversation with Magee [13], a possible method of synthesis with a GA, combining concepts of granular and wavetable synthesis is proposed. A set of sound grains of a specified length are constructed from a sound sample of a real musical instrument. Grains are constructed in such a way that they begin and end at a zero crossing point of the instrument waveform. This technique, drawn from wavetable synthesis, should partly get rid of audible grain boundaries when grains are arranged out of their original order, by making the produced waveform continuous. A side effect of this will be to cause a slight variation in grain length between grains.

The next stage is to construct a probability matrix of the probability that a certain grain will be played in a certain time slot, as shown in table 2.1. One of these probability matrices constitutes an individual
in the GA population. The fitness of each matrix is calculated by recombining a sound sample using the probabilities stored in the matrix, and comparing this sample in some way with another different real instrument sample of equal length. This method of fitness determination may hold problems, as comparing two real waveforms mathematically can be very difficult. A population of matrices is then evolved using GA techniques as described previously, hopefully producing a matrix that reliably creates a sound sample that is indistinguishable from its real life counterpart.

This synthesis technique has none of the problems identified in the two previous techniques. This is only a rough outline and therefore the technique is subject to refinement. The project will concentrate on this technique.

### 2.5 Software Tools and Libraries

#### 2.5.1 Sox

In [2], Sox is described as “The swiss army knife of sound processing programs”. Sox stands for Sound Exchange, and is used for converting between various audio file formats and changing the properties of an existing audio file (its sample rate for example). Sox is used within this project to ensure all audio data files are of the same type and sample rate. Another useful feature of Sox is its ’text data file’, or ’.dat’ output, which represents audio data in a simple uncompressed textural format. This can be used to provide another form of visualization, and is useful for the plotting of graphs. Descriptions of the other file formats supported by Sox can be found in [1]. The version used for this project is 12.17.1.

#### 2.5.2 Audio File

The Audio File library is developed by Silicon Graphics [10]. A Linux port of the library is available from [16]. Audio File is a C library used for working with audio files. It allows a program to open, read and write audio files stored in a variety of different formats, with simple functions. Audio File is used within this project to read and write audio data files in the Microsoft ’.wav’ format, a description of which can be found in [1]. The version used for this project is 0.1.11.

<table>
<thead>
<tr>
<th>grain 1</th>
<th>time slot 1</th>
<th>time slot 2</th>
<th>...</th>
<th>time slot n</th>
</tr>
</thead>
<tbody>
<tr>
<td>P11</td>
<td>P21</td>
<td>Pn1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grain 2</td>
<td>P12</td>
<td>P22</td>
<td>Pn2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>grain m</td>
<td>P1m</td>
<td>P2m</td>
<td>Pnm</td>
<td></td>
</tr>
</tbody>
</table>

*Table 2.1: Example of a probability matrix.*
Chapter 3

Method

3.1 Summary

The method undertaken involves several main tasks. First of all, an instrument is chosen on which to base the synthesis. Then grains are constructed from a sample of the chosen instrument. These grains are used to develop a GA. Several different ways of determining fitness are proposed, and another simpler way of representing grain order is investigated.

3.2 Instrument Choice

During conversation with Webb [20], the Flute is described as the instrument whose waveform most closely resembles a pure sine wave. This fact means that the flute produces a waveform of low complexity, and therefore low harmonic content compared to other musical instruments, as can be seen in figure 3.1. The flute waveform is therefore easier to analyze and operate on mathematically than a complex waveform with lots of different harmonics. The flute is used during development for this reason.

Two samples of a live flute are made at a set pitch, one for use in grain construction and the other for use in fitness determination by comparison, described later in this chapter. A sample rate of 44,100Hz is used and the resulting audio data stored in ‘.wav’ format. The two samples are normalized to the same volume level to ensure compatibility. The first sample is cropped to one second in length, while the second sample is cropped to 1.5 seconds in length to account for uneven grain length during comparison.
Figure 3.1: The waveforms of three different musical instruments.
3.3 Grain Construction

As described in Chapter 2, wavetable synthesis relies on continuity between the waveform boundaries of separate phases and within the loop phase. This idea is utilized here to construct the grains which are used within the GA. Obviously it is not desirable to be able to hear the boundaries between grains within the synthesized sound. The grain construction technique proposed here is not intended to totally eliminate audible grain boundaries. This will be the main task of the GA.

Based on research by Truax [18], a grain length of 50ms is chosen. In order to ensure waveform continuity, grains begin and end at a zero crossing point from negative to positive sound pressure (the 'zero cross' or 'zero crossing point'). Zero crossing points do not necessarily lie at exactly every 50ms. Therefore, grains are not exactly 50ms long but between 50ms and 51.4ms, as can be seen from figure 3.2, where the length of one waveform repetition is shown to be 1.4ms. Zero crossing point determination is made more complex due to the fact that a flute waveform sometimes performs more than one zero cross from negative to positive sound pressure within one repetition, as can be seen in figure 3.3. One zero crossing point is chosen to be taken as the start of all repetitions, and is defined to be the zero crossing point directly before the greatest peak in a repetition (the correct zero crossing
point). The correct zero crossing point can be determined by looking at the amplitude of the peak directly after the zero cross in question. This is possible due to the nature of a flute waveform. At the beginning of the repetition shown in figure 3.3, the sound pressure rises to its highest peak directly after the zero cross, and then drops into the negative before rising again into a smaller positive peak. This pattern can be seen in every repetition the flute waveform makes, although sometimes the second peak does not reach a positive value. In addition to this, the first peak nearly always reaches a value higher than every second peak in the entire waveform. This allows for a threshold value to be set, above which a peak is considered to be the first in a repetition (the 'first peak threshold value'). It is important that grain boundaries be located at the correct zero crossing point to avoid possible situations as in figure 3.4. Such instant change in the flow of the waveform is clearly audible.

![Figure 3.4](image)

**Figure 3.4:** Situation caused by use of the wrong zero crossing point during grain construction.

An algorithm is developed to perform the process of splitting the sampled flute waveform into grains of between 50ms and 51.4ms in length.

Firstly, the algorithm locates the first zero cross from negative to positive sound pressure. It does this by searching, from the start of the waveform, for a sample value where the previous sample has a value less than zero and the next sample has a value greater than zero. A look ahead buffer is then used to determine the value of the peak directly after the zero cross found previously. If that peak has a value less than the first peak threshold value, then the buffer is discarded, and a search for the next zero cross initiated from the sample after the last sample in the buffer. This process is repeated until a correct zero crossing point is identified.

Secondly, the algorithm starts constructing the first grain from the zero crossing point identified. The waveform is added to the current grain until the boundary of 50ms is reached. 50ms at a sample rate 44,100Hz contains $44,100 \times 0.05 = 2,205$ samples. So each grain, at the very least, will contain 2,205 sample values. At this point, the algorithm starts searching for the next correct zero crossing point, employing the same technique used for finding the first correct zero cross of the waveform, with the following differences. When an incorrect zero cross is identified, the buffer is added to the current grain instead of being discarded. Also, when a correct zero cross is identified, the buffer is added to the start of the next grain, and construction of that grain started from the sample after the last sample in the
Thirdly, the algorithm repeats its grain construction process, described in the paragraph above, until the entire waveform is split into grains. The last grain is discarded, as it does not end on a correct zero cross due to the fact that the waveform is not specially processed to end on a correct zero crossing point. As the flute sample used in grain construction is of one second in length, \((1 ÷ 0.05) − 1 = 19\) grains will be produced in total.

Grains are stored in a two dimensional array of the form \([n][1+m]\), where \(n\) is the number of individual grains, and \(m\) is the number of sample values in the longest grain. The first element of every second dimension is used to store the total number of sample values in that particular grain, because the length of each grain varies. Every other element is used to store a sample value.

### 3.4 Genetic Algorithm Outline

The GA must have a population to work on. The population is represented as a three dimensional array, \([p][n][m]\), based on table 2.1. Here, \(p\) is the size of the population, \(n\) the total number of time slots and \(m\) the total number of grains. Every element of the array is used to store a probability. Another array identical in form is created to store the new population determined during breeding. The population is randomly initialized with probabilities between 0 and 1 and the process of evolution started.

Firstly, each population individual’s fitness is assessed to determine a fitness value. The way in which fitness is determined is central to this project. Various forms of fitness determination are investigated in the next section. These fitness values are stored in a separate array with one entry for each population individual. If a population individual is found to have a fitness value above a preset termination value, the GA stops evolving and outputs the population individual’s current waveform before terminating.

Secondly, the fitness values of all population individuals are scaled so that the least fit individual has a fitness of 0, then normalized so they add up to 1. A roulette wheel is constructed based on the normalized fitness values.

Thirdly, a new population is constructed by choosing two ‘parents’ for every new population individual, and performing crossover on the parents probability arrays. A random crossover point is chosen and crossover performed as demonstrated in table 3.1. The array section above the horizontal line is taken from the first ‘parent’, and the array section below the horizontal line is taken from the second. Both these sections are combined, with mutation, to form a new population individual. Mutation is performed by using a mutation probability that dictates if a probability will be changed when it is copied from the ‘parent’ into the new population member. If a probability is randomly chosen to be mutated, then a new random value between 0 and 1 is assigned to it.

Fourthly, elitism is performed by copying the most fit individual from the population into the first element of the new population. The population is replaced by the new population and the GA continues by iterating from its fitness assessment stage.
Table 3.1: Performing crossover on two probability arrays. Horizontal line represents random crossover point.

### 3.5 Measuring Population Fitness

As stated previously, fitness determination is central to this project. Here, four ways of performing fitness determination are described and investigated.

#### 3.5.1 Probability Matrix with Waveform Comparison

Probability matrices, of the form described in the genetic algorithm outline section, are used to represent a population of individuals. A second sound sample of the flute, as described in the instrument choice section, is used to set a standard by which every population individual is assessed. This second sample is initialized to its first correct zero crossing point, as previously described in the grain construction section, and read into a single array.

The following steps are performed with each population individual to assess their fitness.

1. An output array is initialized to hold the output waveform.

2. For the first time slot, the individuals probabilities are normalized so that they all add up to 1. A roulette wheel is constructed based on these normalized probabilities, and an index in the roulette wheel chosen by generating a random number. The grain corresponding to this index is added to the end of the output array. The process is repeated for all time slots to obtain a complete output waveform.

3. The output waveform is then compared with the second sample waveform. As both of these waveforms start from a correct zero crossing point, they should be in phase. Being in phase means that, as the waveforms are from the same source and of the same pitch, they should repeat with the same overall frequency. Comparison is performed by finding the cumulative squared difference between every sample for the length of the output waveform. This cumulative squared difference is then divided by the amount of samples in the output waveform and square rooted. This statistical technique is known as calculating the root mean squared difference, and helps give a better estimate of overall difference by ironing out large insignificant differences.
4. Fitness is then calculated as the inverse of the root mean squared difference.

A potential problem with this approach is that of a single population individual creating outputs that are of both high and low fitness. Individuals of higher fitness will produce a greater number of more fit outputs, but the potential for them to also produce less fit individuals may mean that their good characteristics are seen as unfit, and therefore discarded. This may cause the GA to converge very slowly or not at all.

3.5.2 Static Array with Waveform Comparison

A similar approach is taken to the previous technique, but with the population individuals taking a different form. Instead of each individual being a probability matrix, a static array approach is taken. Each individual now holds a single grain identifier in each time slot instead of a probability for every grain. This aims to greatly reduce the amount of computation required at each GA iteration, and also reduce the amount of generations required to reach a solution, by removing the possibility for the same matrix to provide outputs of both high and low fitness.

Output waveforms are constructed by adding the grain identified for each time slot consecutively to the end of an initialized output array. Fitness is then determined by comparison with another flute waveform, in exactly the same way as the previous technique.

Using waveform comparison, as in this and the previous technique, may be problematic due to the difficulty of comparing two real waveforms mathematically, as identified in Chapter 2. The next two techniques suggest two alternative methods of determining fitness.

3.5.3 Static Array with Amplitude Comparison

This technique is based on the idea that grain boundaries will be less audible if the dominant peaks at the very end and very beginning of every neighbouring grain pair are close to each other in value. Figure 3.5 shows an extended sample of a flute waveform. It can clearly be seen that the dominant peaks are close to their neighbours in value.

The grain array is altered to hold not just the grain’s length in the first element of every grain, but the value of the grain’s first and last dominant peak in the second and third elements respectively. Fitness is determined for each population individual by calculating the cumulative difference at every grain boundary within that individuals grain pattern, between the last dominant peak value of the grain before the boundary, and the first dominant peak value of the grain after the boundary. This cumulative difference is then divided by the amount of grain boundaries, and fitness calculated as the inverse of the result.

Fitness determination will be much faster than the previous two techniques.
3.5.4 Static Array with Wavelength Form Comparison

By looking at figure 3.5, and figures 3.2 and 3.3, it can be seen that not just the peak value, but the overall shape of the waveform changes between repetitions. This technique aims to take this observation into account when determining fitness, as it could be a contributing factor to making grain boundaries audible. This technique is loosely based on the waveform comparison used in the first two proposed techniques.

At every boundary, the last and first repetition of the grain before the boundary and the grain after the boundary respectively, are compared. By observation, waveform repetitions of the flute sample occur at 700Hz. This means that the last and first \((1 \div 700) \times 44,100 = 63\) samples are used in comparison to represent a single repetition. By using this value the technique becomes highly instrument specific, as different instruments will have different waveform repetition frequencies.

The following steps are performed with every boundary within a population individual’s grain pattern.

1. Two temporary arrays of 63 elements in length are initialized. One is filled with the last 63 samples of the grain before the boundary, and the other with the first 63 samples of the grain after the boundary.

2. Comparison is performed by finding the cumulative difference between every sample in the two temporary arrays.

The sum of cumulative difference at every boundary is then calculated, and divided by the number of grain boundaries. Fitness is calculated as the inverse of this result.

Fitness determination will be slightly slower than the previous technique, but hopefully more reliable. Analysis and evaluation of all the proposed techniques is carried out in Chapter 4.
Chapter 4

Evaluation and Analysis

4.1 Summary

Evaluation of the techniques implemented in Chapter 3 involves several steps. Firstly, a number of evaluation criteria are formulated. These evaluation criteria are then assessed for each technique, and the results of this assessment presented and analyzed.

4.2 Sound Output

See Appendix A for the location of `.wav` files of sound output from the various techniques.

4.3 Evaluation Criteria

A number of different issues are analyzed with relation to the techniques implemented in Chapter 3.

1. The convergence characteristics of each GA are analyzed, to identify the correctness and suitability of the technique being used. If a GA consistently fails to converge or appears to display random convergence characteristics, then the technique may be incorrect or unsuitable in some way for finding a solution to the problem. If the GA takes a large amount of time to converge, then the technique may be unsuitable for the computing resources available.

2. The sound output of each GA is subjectively assessed to determine if any audible grain boundaries exist. This is done several times for each GA at different levels of fitness. If grain boundaries
become less and less audible as fitness increases, then the fitness determination technique being used is more likely to be correct and suitable for the problem being solved.

All the GAs, unless stated otherwise, are executed with 19 time slots, a population of 50 individuals, a mutation rate of 10%, and with both fitness scaling and elitism enabled. Using 19 time slots and 19 grains gives $19^{19} = 1.978 \times 10^{24}$ possible grain arrangements. The machine used to execute the GAs is an unloaded Linux workstation with a processor running at 1.3GHz.

For each of the examples given, the GA is executed several times. The output that represents the overall output most well is chosen to represent that example.

The fitness values computed by the GAs are not expected to have a maximum 'perfect' value. The original grain order will not return a perfect fitness value using any of the techniques. Fitness values are dimensionless quantities, and not comparable between the different GAs unless exactly the same fitness determination technique is being used.

### 4.4 Convergence and Sound Output Analysis

#### 4.4.1 Probability Matrix with Waveform Comparison

A potential problem with this technique is identified in Chapter 3 as a single individual being able to create outputs of both high and low fitness, causing individuals of higher fitness to have their good characteristics discarded. This problem appears to hold true as the GA fails to converge in an acceptable amount of time, as shown in figure 4.1. Keep in mind that each generation takes a relatively large amount of time to compute with this technique, due to the overhead of selecting a grain for each time slot within each population individual, and the overhead of comparing two waveforms for each individual.

The comparison of two waveforms requires at least $(19 \times 0.05) \times 44,100 = 41,895$ sample comparisons. With a population size of 50, comparison must be performed 50 times, giving a minimum of

![Figure 4.1: Convergence of probability matrix with waveform comparison, with population size 50 and mutation rate of 10%.

19
41,895 × 50 = 2,094,750 sample comparisons at each generation. This is a significant computational overhead. To compute 50,000 generations takes approximately 227 minutes using this technique.

Increasing the population size and mutation rate has no positive effect on the convergence of the GA, as demonstrated in figure 4.2. The GA still displays the random maximum fitness variation observed in figure 4.1, and does not converge on a solution.

![Figure 4.2: Convergence of probability matrix with waveform comparison, with population size of 100 and mutation rate of 50%.](image)

It is possible that a population size of 50 is too high, diluting the good characteristics within the population. But decreasing the population size and slightly increasing the mutation rate, to compensate for a possible lack of robustness within the population, has no positive effect on the convergence of the GA, as demonstrated in figure 4.3.

![Figure 4.3: Convergence of probability matrix with waveform comparison, with population size of 20 and mutation rate of 30%.](image)

It is worth noting that, even with elitism enabled, the GA is able to evolve away from a solution, in the direction of decreasing maximum fitness. This is due to the nature of the probabilistic technique
being used. Each population individual holds, for every time slot, a probability for every grain. This means that each time a population individual produces an output, that output will most likely be different to other outputs produced by the same population individual, giving a different fitness value. This is also the reason why the GA fails to converge in an acceptable amount of time. The GA is most likely suffering from close race conditions within the population individuals themselves. If the GA is run for many generations then convergence may possibly be observed. This is, of course, highly impractical given the rate at which the GA computes generations.

The sound output produced by this technique appears random. Grain boundaries can clearly be heard at all fitness levels achieved by the GA. This is not surprising, due to the narrow range of fitness levels achievable by the GA. Grain boundaries are perceived as blips in the sound output, and are caused by rapid change in the shape of the waveform.

This technique is obviously unsuitable for synthesizing a natural flute sound.

4.4.2 Static Array with Waveform Comparison

The technique evaluated here aims to greatly improve on the above technique, by removing the probabilistic nature of the population individuals. The fact, stated in Chapter 3, that the amount of computation required by the GA would be greatly reduced by doing this, does not have much effect on the overall running speed of the GA. To compute 50,000 generations takes approximately 213 minutes using this technique. This is a very small improvement over the previous technique. This leads to the conclusion that the comparison of two waveforms for every population individual is the overriding factor in determining the overall running speed of the GA, with this and the previous technique.

Elitism in this technique works as expected, ensuring that the GA always converges in the direction of increasing fitness, as shown in figure 4.4. The fitness values between this and the previous technique

![Figure 4.4](image)

**Figure 4.4:** Convergence of static array with waveform comparison, with population size 50 and mutation rate of 10%.

are comparable due to the fact that fitness is determined in exactly the same way. The maximum fitness
of 0.015 achieved by this technique, is significantly higher than the previous technique’s maximum fitness of 0.0117, suggesting a slight improvement. More likely, this improvement is due to random variation between runs of the GA.

The maximum fitness of the GA plateaus at about generation 25,000. This may be due to many reasons, but most likely the technique provides little improvement in suitability to the problem over the previous technique.

This conclusion is backed up when the sound output is subjectively assessed. The sound output produced by this technique appears random. Grain boundaries can clearly be heard at all fitness levels achievable by the GA. The most fit sound output, 0.015, shows no improvement over any other outputs, from this or the previous technique.

This technique also appears unsuitable for synthesizing a natural flute sound.

### 4.4.3 Static Array with Amplitude Comparison

As predicted in Chapter 3, the technique evaluated here runs much faster than all the others. To compute 50,000 generations takes approximately 13 seconds using this technique. This is due to the fact that only one comparison for every grain boundary must be performed for each population individual. For 19 time slots there are 18 grain boundaries. With a population size of 50, only $18 \times 50 = 900$ comparisons need to be performed at each generation in order to determine the fitness of every population individual. In comparison, the probability matrix with waveform comparison technique requires at least 2,094,750 comparisons at each generation.

Using this technique, the GA converges in a positive way, as shown in figure 4.5, reaching a plateau at about 50,000 generations. This plateau is not necessarily a problem of premature convergence, de-

![Figure 4.5: Convergence of static array with amplitude comparison, with population size 50 and mutation rate of 10%.

scribed in Chapter 2, as it may be close to the absolute maximum fitness the GA can achieve. Executing the GA several times produces an absolute maximum fitness of 0.076923 once, with the other executions
reaching a maximum fitness of 0.045455, all within about 50,000 generations. The GA takes only 13 seconds to compute 50,000 generations, so this seems promising. But looking at the order in which the grains are arranged at different fitness levels, as shown in table 4.1, and subjectively assessing the sound output, shows this technique to be fundamentally flawed.

<table>
<thead>
<tr>
<th>Fitness</th>
<th>ts1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.006711</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>0.007092</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0.045455</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>0.076923</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

**Table 4.1:** Grain arrangements output by the static array with amplitude comparison technique. Time slots are labeled in the top row.

The technique appears to produce strings of the same grain within its output, favouring grains 12 and 14 among others. This is a highly unexpected result, and appears to be a fundamental flaw with the technique. The favoured grains must begin and end with similar valued dominant peaks to be arranged in this fashion.

The sound output produced by the repeating nature of the grain arrangement with this technique does not sound natural. Grain boundaries can clearly be distinguished, becoming more prominent as fitness increases. As fitness increases, the sound produced begins to sound more and more 'electronic'. This is clearly undesirable, and makes this technique unsuitable for synthesizing a natural flute sound.

### 4.4.4 Static Array with Wavelength Form Comparison

This technique runs slightly slower than the previous technique, but not as slow as the two waveform comparison techniques. To compute 50,000 generations takes approximately 210 seconds using this technique.

The GA converges in the direction of increasing fitness, as shown in figure 4.6, and starts to plateau at around generation 10,000. The convergence displayed shows a good balance between progress by evolution and progress by mutation. Good characteristics quickly propagate through the population, as shown by the slope of the graph where fitness is increasing (for example, around generations 2000 and 5000). After the current good characteristics have been propagated, the convergence levels off (for example, around generation 8000), until new good characteristics are introduced by mutation.

Running the GA several times produces an absolute maximum fitness of 0.000858, within a reasonable amount of time. It is therefore likely that 0.000858 is very close to the maximum fitness the GA will ever achieve.

It is possible that the GA is not converging at its maximum possible rate, or reaching its absolute maximum fitness value with a population size of 50 and a mutation rate of 10%. A number of different combinations of population size and mutation rate are now tried, in order to 'tune' the GA to achieve
**Figure 4.6:** Convergence of static array with wavelength form comparison, with population size 50 and mutation rate of 10%.

better convergence.

Figure 4.7 shows the convergence achieved with a population size of 50 and a mutation rate of 50%. The GA does not converge to anywhere near the maximum fitness level achieved with a population size of 50 and mutation rate of 10%. Fitness levels can clearly be seen to increase by a dramatic jump between two generations. This is a sign that the GA is only converging by mutation, and not by evolution, considerably slowing the convergence rate, and indicating that the mutation rate is too high.

Figure 4.8 shows the convergence achieved with a population size of 20 and a mutation rate of 30%. Again, the GA does not converge to anywhere near the maximum fitness level achieved with a population size of 50 and mutation rate of 10%. This is a sign that a population size of 20 is not large enough to represent a wide enough variety of characteristics to make the population robust. Similar dramatic jumps in fitness seen with a population size of 50 and mutation rate of 50%, can also be
observed here. This indicates that a mutation rate of 30% is too high.

Figure 4.9 shows the convergence achieved with a population size of 30 and a mutation rate of 10%. The GA converges much more steadily this time, but takes over 12,000 generations to reach a maximum fitness close to the one achieved with a population size of 50 and a mutation rate of 10%. This increase in the number of generations required is, however, outweighed by the speed with which 12,000 generations are computed with a population size of 30. 12,000 generations take 32 seconds to compute, compared to 42 seconds for 10,000 generations with a population size of 50, both achieving the same level of maximum fitness.

Figure 4.10 shows the convergence achieved with a population size of 80 and a mutation rate of 10%. The GA converges rapidly, taking about 4,000 generations to reach a maximum fitness close to the one achieved with a population size of 50 and a mutation rate of 10%. The robustness of the population has
Figure 4.10: Convergence of static array with wavelength form comparison, with population size 80 and mutation rate of 10%.

increased with a population size 80, producing faster convergence. Computing 4,000 generations takes 28 seconds with a population size of 80, in comparison to the 42 seconds required to compute 10,000 generations with a population size of 50. Both achieve the same level of maximum fitness.

Figure 4.11 shows the convergence achieved with a population size of 100 and a mutation rate of 10%. Increasing the population further has a detrimental effect on the convergence of the GA. The GA takes 6,000 generations to reach a maximum fitness close to that achieved with a population size of 50 and a mutation rate of 10%. Computing 6,000 generations takes 53 seconds with a population size of 100. A population of 100 is too large, diluting the good characteristics and causing the GA to converge more slowly than with a population size of 80.

The best choice of GA parameters, given these experiments, is a population size of 80 with a mutation rate of 10%.

Figure 4.11: Convergence of static array with wavelength form comparison, with population size 100 and mutation rate of 10%.
The sound output from this technique has slightly audible grain boundaries at lower fitness levels, but with fitness above 0.0008, audio boundaries are practically inaudible. Further subjective assessment of sound output from this technique is performed in the next section.

This technique clearly outperforms all the other suggested techniques, and is therefore chosen as the most successful sound synthesis technique developed by this project.

4.5 Further Analysis of Wavelength Form Comparison Technique

Due to the subjective nature of evaluating sounds produced by real instruments, a number of opinions from different people are sort to evaluate the sound produced by the chosen synthesis method, wavelength form comparison.

A survey of eight people was carried out, to determine if the original sample used in grain construction can be distinguished from the synthesized sounds produced by the wavelength form comparison technique. Each person was played four pairs of sounds, one the original sample and one a synthesized sample of different fitness. The task was to identify, for each pair of sounds, which is synthesized and which is real. Four sound outputs of different fitness were used to perform the survey. The results of the survey are shown in table 4.2.

<table>
<thead>
<tr>
<th>Fitness of synthesized sound output</th>
<th>Number of people who correctly identified the synthesized output out of 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000602</td>
<td>6</td>
</tr>
<tr>
<td>0.000813</td>
<td>5</td>
</tr>
<tr>
<td>0.000826</td>
<td>4</td>
</tr>
<tr>
<td>0.000858</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4.2: Survey results to assess sound output from the wavelength form comparison technique.

Three of the people who participated in the survey are semi-professional musicians. It is interesting to note that two of these musicians correctly identified all of the synthesized sound outputs, with the other identifying all but the one of 0.000813 fitness correctly. The wavelength form comparison technique is therefore not quite good enough to convince a musician that the sound of a real flute is actually being played. But with fitness levels above 0.0008, nearly half of all the people surveyed thought the sound they were listening to was that of a real flute.
Chapter 5

Conclusions and Further Development

5.1 Project Summary

A survey of only eight people does not produce statistically valid data. No firm conclusions can be drawn from such a small sample size. Ideally, an extensive survey of more than thirty people from different backgrounds should be performed. But due to time constraints this was not feasible. The results obtained by performing the survey to evaluate the sound output from the wavelength form comparison technique, as shown in table 4.2, do however give an insight into the success of the project. The original aim, to convincingly synthesize a real musical instrument, has proved to be feasible when applied to the synthesized sound output from the developed synthesis technique. The technique is not quite good enough to convincingly synthesize a real musical instrument to a musician, but the average listener is convinced by the technique that they are listening to the sound of a real flute.

In its current state, the technique is unsuitable for anything except generating a very short sustained sample of a flute, at a single pitch. In order to achieve sustained samples of longer than one second, a greater number of time slots must be considered by the GA, adding an increased number of possible grain arrangements, and therefore decreasing convergence performance. A different possibility for producing sustained samples of longer than one second is proposed in the next section, for further development.

The technique is obviously unsuitable in its current state for real-time synthesis. To produce a sound sample of one second requires about 28 seconds of computation. Possible enhancements to the technique to enable real-time synthesis are proposed in the next section.

Another problem with the technique is a lack of an attack and decay phase at the beginning and end of a produced sound sample. This could be solved by using an idea from wavetable synthesis. This
possibility is discussed in the next section.

## 5.2 Further Development

### 5.2.1 Probability Matrix with Waveform Comparison

The probability matrix with waveform comparison technique could be developed further, to attempt to make it successful. Instead of just assessing the fitness of one waveform output produced by each matrix, to determine the fitness of that matrix, a number of waveform outputs could be produced and assessed, and their average fitness taken as the fitness of that individual matrix. A potential problem with doing this would be the computational overhead of assessing the fitness of several waveforms for every population individual. This overhead may be outweighed if good convergence is achieved, but this seems unlikely, due to the close race conditions identified within each matrix in Chapter 4. An attempt to solve these close race conditions could be made, by using fitness scaling within each matrix.

### 5.2.2 Range of Instruments

Currently, the synthesis technique is only capable of producing an instrument sound at a single pitch. The technique of linear interpolation, as used in wavetable synthesis and described in Chapter 2, could be applied to the sound output to enable it to represent a variety of different pitches within a pitch range.

The possibility of synthesizing instruments other than a flute exists. In order to do so, the number of samples used to represent a single waveform repetition will need to be defined for each instrument. For example, this number can be roughly determined for a distinct bass clarinet and saxophone pitch from figure 3.1. The bass clarinet waveform repeats about every 4.3 ms. Therefore, about \(0.0043 \times 44100 = 190\) samples will have to be used to represent a single waveform repetition. The saxophone waveform repeats about every 3.9 ms, around \(0.0039 \times 44100 = 172\) samples. In addition, a way of determining a correct zero crossing point within one waveform repetition will need to be developed for each instrument, so that grain construction may be performed correctly.

### 5.2.3 Application Considerations

The possibility of using a hierarchy of longer and longer grains exists, to produce sound samples of much greater length than one second. A number of one second samples could be produced by the existing GA, and then used by another identical GA as grains. These long grains could then be arranged by the GA and an output generated, of much greater length than the first sound samples. If the first sound samples are stored in some kind of static grain bank, then the possibility of real-time synthesis becomes a reality.

The lack of an attack and decay phase at the beginning and end of a produced sound sample, as highlighted in the previous section, could be solved in the following way. Separate attack and decay samples could be appended to the beginning and end of every produced sound sample. Further more,
the GA could be extended to determine which attack and decay samples would best suit the current grain arrangement, from a range of different attack and decay samples. This could be done in the same way as the fitness of grain boundaries are assessed with the present technique, by wavelength form comparison between the attack sample and the first grain, and between the decay sample and the last grain.
Bibliography


   URL: http://sox.sourceforge.net/ [27th April 2003].


Appendix A: Personal Reflection

This project has been the largest single piece of work I have ever attempted. The sheer size of the programming and development tasks to be completed often felt overwhelming in themselves, without even considering the amount of effort required to write a report of the size and content required. The amount of satisfaction achieved by completion of the work is all the greater because of this. The scale of this project has enabled me to learn many lessons, about myself and the way in which I tackle work, and about research, development, report writing and time management.

Before starting this project, time management was not one of my strong points. The project forced me to strictly organise my time, and set realistic goals, to be met at various points throughout the project. This enabled me to tackle the project in small chunks, and therefore mostly avoid the feeling of being way out of my depth. This helped to improve productivity. I strongly recommend strict time organisation and the setting of achievable goals to all future final year project students, independant of project area.

The coding and development of the synthesis engine was by far the biggest aspect of this project. Turning all the research and ideas into a program that actually produces a realistic sound turned out to be a huge undertaking. It would have been nice to have enough time to tidy up and optimise the program’s code at the end of the project, but just producing a working program took far more time than originally anticipated. I recommend to future students undertaking any project that involves coding, to set aside significant amounts of time for code development and debugging. When starting out coding something new, it is all too easy to falsely assume that because good research and ideas are in place, turning them into working code will be trivial.

The amount of time spent writing the report was also considerable. It is easy to forget that the report is, in most cases, the only part of the project that will be assessed. I would warn future final year project students to ignore this fact at their peril. The amount of effort required to put the content of any project into words is considerable. I greatly admire those who have the skill to do this well.

I am generally pleased with how the project has gone. All the stress and hard graft has been worth it, and many lessons learnt along the way. Performing the survey to subjectively assess the final sound
output was a nice way to end the project. It was very nice to hear people complimenting my work without realising it, by saying things like, "Umm, err, I don’t really know. They both sound so similar!".
Appendix B: Sound Output

Listed here are the file names and location of .wav files output by the GAs at various stages in the project. They are provided purely to make reading this report more interesting, and are not essential in order to understand any aspect of this project.

All files located on the Leeds University School of Computing Linux network in the following directory:

/home/cserv1_a/student/csxc/csche/fyproj/

- **originalFlute.wav** - The real flute sample used in grain construction.
- **random.wav** - Sound output with a completely random grain arrangement.
- **pmwc012124.wav** - Probability matrix with waveform comparison technique, output with a fitness of 0.012124.
- **sawc013070.wav** - Static array with waveform comparison technique, output with a fitness of 0.013070.
- **saac006711.wav** - Static array with amplitude comparison technique, output with a fitness of 0.006711.
- **saac076923.wav** - Static array with amplitude comparison technique, output with a fitness of 0.076923.
- **sawfc000602.wav** - Static array with wavelength form comparison technique, output with a fitness of 0.000602.
- **sawfc000813.wav** - Static array with wavelength form comparison technique, output with a fitness of 0.000813.
- **sawfc000826.wav** - Static array with wavelength form comparison technique, output with a fitness of 0.000826.

- **sawfc000858.wav** - Static array with wavelength form comparison technique, output with a fitness of 0.000858.
Appendix C: Code Listing

The C++ code for the best technique developed, static array with wavelength form comparison, is presented here. The code was written for the purpose of research, and therefore not very neat or pretty.

/*
 * Final Year Project (Computer Science BSc)
 * School of Computing, University of Leeds
 *
 * Musical Synthesis Engine
 *
 * This synthesizer uses a genetic algorithm to evolve a set
 * of sound grains into an order such that they reproduce an
 * approximation to the sound of a real musical instrument.
 * The grains are constructed from an input sound sample of
 * a real musical instrument and the output evolved by
 * comparison of the first and last 63ms of each grain.
 * The engine is currently optimised for a flute.
 *
 * Author: Chris Eade
 * Date : March 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <audiofile.h> // Linux port of SGI Audiofile library
#include <math.h>

int main(void)
{
    // Input file (MS RIFF WAV, 44100Hz, 16-bit, 1 channel (mono))
    char* infile = "fluteIn.wav";
    char* outfile = "fluteOut.wav";

    // Audiofile file handles
    AFFilehandle inSndfile;
    AFFilehandle outSndfile;
    // Audiofile file setup definitions
    AFFilessetup setup;

    // Declare variables
    int popSize = 80; // Size of the population to run
    int mutationRate = 10; // Mutation rate in percent
    float termination = 0.00081; // Termination criteria

    int sampCount = 63; // The length of one waveform repetition
    int firstBit[sampCount]; // Buffer to store first bit of grain
    int lastBit[sampCount]; // Buffer to store last bit of grain
    int tempBit[2000]; // Temp Buffer

double sampleRate = 44100.0; // sample rate. Tested for 44100Hz
short int inSample=0; // Buffer to store input sample
int n,m,x,y,z,p,o,s,t; // A few counters/storage variables
short int i,j,a,b,c; // A few more temp storage variables
int cross=0, top=0; // Used to test crossover in wav file
int framesRead=0; // Another counter

float totalProb=0.0; // Normalize counter
int roulettePop[1099]; // Roulette wheel for selection
int totalRoul=0, lastx=0; // More counters
int probTemp, leastFrames; // More temp variables

int mostFit = 0; // Used to store the most fit pop member
int fitP=-1; // Used to store the fit pop member
int iter=0; // Generation counter
float fitness, fitemp, error, errorTemp=0;  // Temp storage
float totalFitness=0, minFitness = 0, maxFitness = 0;
int totalDiff, temp;

int male, female, crossover;

// Seed the random number generator
srand(time(0));

/*
  * Open wav files and construct grains
  */

setup = afNewFileSetup();
// Open the input wav file
inSndfile = afOpenFile(infile, "r", setup); // setup ignored for "r"
if (inSndfile == AF_NULL_FILEHANDLE) {
    fprintf(stderr, "Cannot open input wav file: %s", infile);
    return 1;
}
afFreeFileSetup(setup);

// Init the properties of the output wav file
setup = afNewFileSetup();
afInitFileFormat(setup, AF_FILE_WAVE);
afInitChannels(setup, AF_DEFAULT_TRACK, 1);
afInitSampleFormat(setup, AF_DEFAULT_TRACK, AF_SAMPFMT_TWOSCOMP, 16);
// Open the output wav file
outSndfile = afOpenFile(outfile, "w", setup);
if (outSndfile == AF_NULL_FILEHANDLE) {
    fprintf(stderr, "Cannot open output wav file: %s", outfile);
    return 1;
}
afFreeFileSetup(setup);

// Get the number of frames and the frame size of the input file
int frames = afGetFrameCount(inSndfile, 1);
double frame_size = afGetFrameSize(inSndfile, AF_DEFAULT_TRACK, 1);
int framesPerGrain = 2205;
int numOfGrains = frames / framesPerGrain;
short int frameArray[frames];

// The first element of grains is used to store its size
int grains[numOfGrains+10][framesPerGrain+1000];

// Initialise the file pointer to the first zero crossing
afReadFrames(inSndfile, AF_DEFAULT_TRACK, &inSample, 1);
i = inSample;
for(n=0; n<frames; n++)
{
    if (framesRead == frames) break;
    afReadFrames(inSndfile, AF_DEFAULT_TRACK, &inSample, 1);
    framesRead++;
    j = inSample;
    if(i<0 && j>0) break;
    i = j;
}

// Read in data for each grain
s=0;
for(n=0; n<numOfGrains; n++)
{
    x=s+1;
    while(x<framesPerGrain || cross == 0)
    {
        grains[n][x] = j;
        cross = 0;

        if (framesRead == frames) break;
        afReadFrames(inSndfile, AF_DEFAULT_TRACK, &inSample, 1);
        framesRead++;
        i = inSample;

        if(j<0 && i>0)
        {

        
}
a = j;
b = i;
y = 0;
while (top == 0)
{
    afReadFrames(inSndfile, AF_DEFAULT_TRACK, &inSample, 1);
    framesRead++;
    c = inSample;
    if (b > a && b > c)
    {
        tempBit[y] = b;
        break;
    }
    tempBit[y] = b;
    a = b;
    b = c;
    y++;
}
if (b > 10000 && x >= framesPerGrain)
{
    // append to next grain and start on next grain
    for (m = 0; m <= y; m++)
    {
        grains[n+1][m+1] = tempBit[m];
    }
    s = y + 1;
    i = c;
    cross = 1;
} else
{
    // append to current grain and continue current grain
    for (m = x + 1; m <= x + y + 1; m++)
    {
        grains[n][m] = tempBit[m-x-1];
    }
    x = x + y + 1;
    i = c;
cross = 0;

} // if cross
j = i;
x++;
}

grains[n][0] = x-1; // Length

int totalGrains = n-1; // total grain count discarding last grain
int lengthInGrains = totalGrains;
afCloseFile(inSndfile); // Close the input file

/*
 * 2D Probability matrix construction and evolution
 */

int pop[popSize][lengthInGrains]; // 2d population matrix
int newPop[popSize][lengthInGrains]; // 2d population matrix
float popFitness[popSize]; // Fitness of each population indiv
// sized to ensure array doesnt overrun into neighbouring memory
int outFrames[frames*2];
int fitFrames[frames+2000];

// Assign every grain to slot at random
for(p=0; p<popSize; p++)
{
    for(n=0; n<lengthInGrains; n++) // time
    {
        newPop[p][n] = rand()%totalGrains-1;
        while (newPop[p][n] == -1)
            newPop[p][n] = rand()%totalGrains-1;
    }
}

while(fitP == -1)
{
    // Generation counter
    iter++;
}
// Set the old population to equal the new
for(p=0; p<popSize; p++)
{
    for(n=0; n<lengthInGrains; n++)
    {
        pop[p][n] = newPop[p][n];
    }
}

for(p=0; p<popSize; p++)
{
    // Calculate the fitness
    errorTemp = 0;
    for(n=0; n<lengthInGrains-1; n++)
    {
        a = pop[p][n];
        b = pop[p][n+1];

        y = grains[n][0] - sampCount+1;
        for(x=y; x<grains[n][0]+1; x++)
            lastBit[x-y] = grains[a][x]; // Last few ms of current grain
        for(x=0; x<sampCount; x++)
            firstBit[x] = grains[b][x+1]; // First few ms of next grain

        // Compare the last and first bit
        totalDiff = 0;
        for(x=0; x<sampCount; x++)
        {
            i = lastBit[x];
            j = firstBit[x];

            temp = 0;
            if (i > j && j > 0) temp = i - j;
            else if (i < j && i > 0) temp = j - i;
            else if (i > j && i < 0) temp = j*-1 -i*-1;
            else if (i < j && j < 0) temp = i*-1 -j*-1;
            else if (i > j && i > 0 && j < 0) temp = i - j;
        }
    }
}
else if (i < j && j > 0 && i < 0) temp = j - i;
else temp = 0; // 4
totalDiff += temp;
}
errorTemp += totalDiff / (float)sampCount;
}
error = errorTemp / (float)lengthInGrains;

if(error > 0) fitness = (1/error);
else fitness = 1;

// Keep track of minimum fitness for scaling
if(fitness < minFitness) minFitness = fitness;

popFitness[p] = fitness;

// Assess the termination condition
if(fitness > termination)
{
   fitP = p; // Will cause the program to stop evolving
}

// Scale fitness to window
for(p=0; p<popSize; p++) {
   popFitness[p] -= minFitness;
totalFitness += popFitness[p];
// Keep track of fittest individual for elitism
if(popFitness[p] > maxFitness) {
   mostFit = p;
   maxFitness = popFitness[p];
}
}

// Print out the average fitness of this generation
if(iter % 10 == 0 || fitP != -1)
{
   printf("Average / Most Fit at Generation %d = ", iter);
printf("%f / %f\n", totalFitness / popSize, popFitness[mostFit]);
}
// Reset a few variables
totalFitness = 0;
minFitness = 0;
maxFitness = 0;

// Re-seed the random number generator
srand(time(0));

/*
 * Breeding of the population
 */

// Normalise population probabilities
totalProb = 0.0;
for(p=0; p<popSize; p++)
{
    totalProb = totalProb + popFitness[p];
}
for(p=0; p<popSize; p++)
{
    popFitness[p] = popFitness[p]*(1/totalProb);
} // Finish normalise

// Construct a roulette wheel for whole population
lastx = 0;
totalRoul = 0;
for(p=0; p<popSize; p++)
{
    probTemp = (int)(popFitness[p]*1000);
totalRoul = totalRoul + probTemp;
    for(x=lastx; x<lastx+probTemp; x++)
        roulettePop[x] = p;
    lastx = x;
}

// Breeding Season!

z=5;
for(p=0; p<popSize; p++)
{
    male = roulettePop[rand() % totalRoul];
    female = roulettePop[rand() % totalRoul];
    // Choose a random crossover point
    crossover = rand() % lengthInGrains-1;

    for(n=0; n<crossover; n++)
    {
        newPop[p][n] = pop[male][n];
        // Mutation
        if((rand() % 100) < mutationRate)
        {
            newPop[p][n] = rand()%totalGrains-1;
            while (newPop[p][n] == -1)
                newPop[p][n] = rand()%totalGrains-1;
        }
    }
    for(n=crossover; n<lengthInGrains; n++)
    {
        newPop[p][n] = pop[female][n];
        // Mutation
        if((rand() % 100) < mutationRate)
        {
            newPop[p][n] = rand()%totalGrains-1;
            while (newPop[p][n] == -1)
                newPop[p][n] = rand()%totalGrains-1;
        }
    }
    // Elitism
    newPop[0] = pop[mostFit];
} // while not fitP

/*
 * Output a wav file if a fit individual is found
 */
printf("\nFit Population Member = %d\n", fitP);

if(fitP != -1) // If a fit one exists then output a wav
{
   // Output to a wav file
   short int outSample;
   y=0;
   printf("\n\n Grain Order: ");
   for(n=0; n<lengthInGrains; n++)
   {
      x = pop[fitP][n];
      printf("%d ", x);
      // Use the corresponding grain for this time gap
      for(m=1; m < grains[x][0]+1; m++) // grain
      {
         outFrames[z] = grains[x][m];
         z++;
      }
      y = y+z; // Keep track of the total frames
   }
   printf("\n\n");

   for(n=0; n<z; n++)
   {
      outSample = outFrames[n];
      // Write the sample into the output file
      afWriteFrames(outSndfile, AF_DEFAULT_TRACK, &outSample, 1);
   }
}

afCloseFile(outSndfile); // Close output file

return 0;
} // main