SOUND SYNTHESIS USING THE GENETIC ALGORITHM

PREPARED FOR: The Department of Electrical and Electronic Engineering at the University of Cape Town

PREPARED BY: David Johnson, fourth-year student in the Department of Electrical and Electronic Engineering

7 November 1995

Thesis prepared in partial fulfilment of the requirements for Degree of BSc in Electrical and Electronic Engineering

ACKNOWLEDGEMENTS

Firstly I would like to thank Professor John Green for providing me with such a stimulating thesis topic which combined my interests in music and electronics so well. He is also thanked for helping me discern what was relevant to the thesis and for providing me with the stimulating visual Genetic Algorithm program called ImoGen. The many smiles, when I came to his room to play new sounds I had generated, will also not be forgotten.

Near the beginning of the thesis I had I number of useful conversations with Newton at Paul Bothners in Claremont. He helped me understand current trends in music synthesis and demonstrated some of the capabilities of latest keyboards on the market.

A number of useful conversations were held with students in the department. I thank Mauro for showing me why the ear is impervious to phase information in waves and a student doing masters in Genetic Algorithms for his help with allocating far memory in C.

Finally I want to acknowledge my brother Steven Johnson, for always being there to help both with software problems and with problems of the human kind. His laser printer which is going to be used to print this thesis is also greatly appreciated.

SYNOPSIS

This thesis report describes an investigation into using a genetic algorithm to guide a sound search using FM synthesis models.

There were two objectives in this thesis. The first was to explore the possibility of producing new unheard of sounds by rating a population of FM generated sounds in each generation that is produced by a genetic algorithm. The second was to test the robustness of FM synthesis models by trying to match real sounds.

Current trends in music synthesis were firstly discussed to ascertain what has been achieved and what areas of music synthesis could still be investigated. It was shown that all synthesizers used one of three types of synthesis, these being Additive synthesis, subtractive synthesis and FM synthesis. Of these FM synthesis is still known to be the most powerful in that a vast array of possible sounds is achievable with very few oscillators and wave manipulation.

The power behind FM synthesis was shown to be the fact that "negative" harmonics reflect onto the positive frequency axis creating a final mix with a complex harmonic envelope. The FM models investigated were Chowning FM and Double FM. Both consisted of an array of common FM functions known as carriers each multiplied by an individual envelope and finally added together to produce the final waveform. The Double FM model was capable of producing very complex harmonic envelopes with very few carriers whereas the Chowning FM model needed a large combination of carriers to achieve intricate harmonic envelopes.

The genetic algorithm was combined with the FM models by creating a fitness function and encoding FM parameters into the gene. Real sound matches were done by using a genetic algorithm to search for FM parameters and using a lease-mean-squares algorithm to find a set of closest weights which when used in the model caused it to match harmonics of the original sound as close as possible. It was found that the Chowning FM model produced the best initial match over a large number of generations. DFM however always produced the best initial match and also made good high order harmonic matches. New sounds were searched for by using parabolic envelopes and encoding their peak point together with all the other FM parameters of

the gene. Each gene mapped to a unique sound was given a fitness value by the user depending how close it was to the sound wanted. DFM synthesis produced the greatest variety of sounds and was therefore the preferred model for searching for new sounds.

Important conclusions drawn were: The FM models could provide a large enough search space to very closely approach all the sounds tested in this thesis which were Piano, Voice, Guitar, Cello, Organ and Recorder. Standard GA ratings with a crossover rate = 60%, mutation rate = 0.1% and a population size of 50 worked the best for real sound matches with fitness improvements levelling off after about 100 generations. For new sound searches GA parameters with crossover rates close to 100% and mutation rates as high as 10% worked well with a population size of 10 to allow easy inter-comparison of sounds.

LIST OF ILLUSTRATIONS

TABLE OF CONTENTS

1. INTRODUCTION

Today's music synthesis market is dominated by digital keyboards which have very little scope to allow the musician to mould new sounds to their needs. Paradoxically it was the keyboards of the late 70's and early 80's that were far more flexible and powerful for generating interesting new waveforms. The only trouble was that they needed people with a deep understanding of sound waves and programming to understand how to use many of these early keyboards. In particular the Yamaha DX-7 FM synthesizer left many a musician in a complete quandary when faced with its multitude of controllers and operators for creating a particular sound. Most of today's electronic keyboards have opted for the easy way out and store high quality sampled waveforms of pre-selected instruments that can only be partially manipulated by the musician.

Recent research in the area of function optimization and its possible application to sound searches could revolutionize the way many of the old synthesis techniques such as FM synthesis were used. The genetic algorithm has proved itself to be a particularly robust function optimizer for even the most difficult noisy, high dimensional and multi-model functions. FM synthesis is known to be the most powerful but least predictable forms of synthesis and it therefore forms a good suite with the genetic algorithm.

The objective of this thesis was to investigate the feasibility of using a genetic algorithm as a tool to search the sound space of FM producible sounds. This objective was split into two phases:

1. The possibility of producing new unheard of sounds by giving a population of sounds individual fitness ratings based on their subjective closeness to a wanted sound.
2. Testing the robustness of FM synthesis models by matching real sounds.

The experimental apparatus that were to be used were simply a relatively fast IBM-compatible workstation, some form of digital to analogue and analogue to digital converter, an amplifier, a pair of speakers and a compiled language together with algorithms for the sound search written in that language.

The possible thesis scope was very large with many spin-off applications once particular tasks were accomplished. For example once real sounds had been matched there was the possibility of using these sounds as an initial population in a sound search but these applications needed a lot more coding that couldn't be done in the time allocated for the thesis.

The thesis begins with a description of the complexity of musical sound and the attempted techniques at synthesising it. It then moves on to show why FM synthesis is the best and most powerful model to be used for sound searching and matching. Two FM models are developed in this chapter together with the equations which fully describe their capabilities. The next chapter gives a detailed description of why the genetic algorithm is so useful and how it fits hand in glove with FM synthesis models. The in-depth methods for matching real sounds and searching for new sounds are also presented in this chapter. The results for this chapter are then presented for real and new sound searches together with the spectral plots of real instruments and model instruments. A brief description of the equipment and software is given next followed by conclusions and extensions on FM synthesis using the genetic algorithm.

I long for instruments obedient to my thought and which, with their contribution of a whole new world of unsuspected sounds, will lend themselves to the exigencies of my inner rhythm

Edgard Varese, 1917

We also have sound-houses, where we practice and demonstrate all sounds, and their generation. We have harmonies which you have not, of quarter sounds and lesser slides of sounds. Diverse Instruments of Musick likewise to you unknown, some sweeter than you have together with Bells and Rings that are dainty and sweet. We represent small sounds as well as Great and Deep. We make diverse tremblings and Warblings of sounds. We have certain helps, which set to the ears doe further hearing greatly. We have strange and artificial echoes, reflecting the voice many times, as if it were tossing it. We have also means to convey sounds in trunks and pipes, in strange lines and distances.

Francis Bacon
New Atlantis, 1624

2.1 The enigma of sound

Music is one of the most information rich forms of human communication. A compact disk for instance uses nearly 1.5 megabits per second to faithfully transmit a stereo recording. What is it that gives music this complexity? And more importantly, what is it that gives the individual instruments creating the music their characteristic sound or timbre?

Every instrument has a basic timbre, a method of tone production and a style intimately linked to its tone generating mechanisms. When playing an instrument, one is usually building tone around its basic timbre with the source of energy which creates the sounds and the nuances which are unique to the instrument. The energy source is for example the cello bow and one of the cello's nuances is the sound it makes when the bow is bounced off the string, commonly known as spiccato.

2.1.1 The basic timbre

Any sound, over a particular period in time, can be broken up into a sum of sine waves as was first shown by Fourier. Viewing the sound in the frequency domain, a spectral plot is obtained and for most musical instruments these spectra occur at integral harmonics. What this means is that for a fundamental frequency of, for example, 440Hz some of the following frequency components may be present in the sound: 880Hz, 1320Hz .... n*440Hz. where n is the harmonic number. This is not always the case however - percussive sounds like cymbals, drums and certain bells have non-harmonic spectra. Instruments can therefore all be classified according to the overall spectrum present in the sound. The classification of instruments is summarized in **Table 2.1** (De Furia 1986):

OVERALL SPECTRUM	ACOUSTIC SOURCE	INSTRUMENT TIMBRE
ALL HARMONIC PARTIALS	Open Air columns Vibrating strings	Brass Strings Guitar Bass Woodwinds
ODD HARMONIC PARTIALS	Closed air columns	Clarinet Recorder Organ Whistles
NON HARMONIC PARTIALS	Membranes Suspended bars	Electric piano Bells Vibes Cymbals Tympani
RANDOM HARMONIC PARTIALS	Transients Rushing Air Snares	Breath Drums Wind Thunder Mechanical action

Table 2.1 Classification of instruments

2.1.2 Tone production

The tone production mechanism also affects how we perceive the instrument as it affects the evolution of the instruments harmonics over time. In fact John M. Chowning in his pioneer paper on FM synthesis in 1973 wrote that the character of the temporal evolution of the spectral components of sound is of critical importance in the determination of timbre (Chowning 1973). A piano sound is very different when the hammer hits the string compared with when the string

is merely vibrating. It is thus important to classify three stages of an instruments tone production. They are the attack, decay, sustain and release of a note. In synthesiser terminology it is known as the ADSR envelope function. A typical envelope for a brass instrument is shown below in **Figure 2.1**

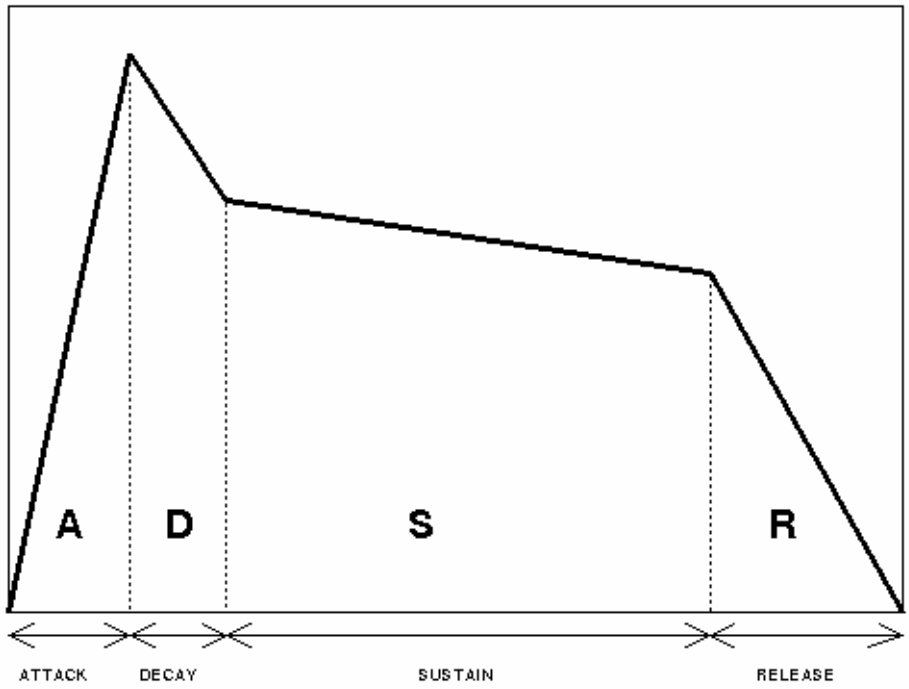

Figure 2.1 Envelope showing 4 stages of tone production

The attack portion of any instrument is always the most chaotic and complex. This is associated with the wind rushing into the organ pipe, the hammer hitting the piano string or the wave front of air rushing down a flautist tube before resonance. The attack usually has a rapid increase in amplitude in the fundamental frequency but other harmonics may have a slower attack as is the case with an organ where it takes time for the harmonics to build up in the pipe.

The decay is the short portion just after the attack and may be absent in some instruments. It is very common in a wind instrument especially when over blowing a note.

The sustain is the portion of the note when the sound has reached equilibrium and the timbre

and volume stay fairly constant. This too can be absent as is the case with a guitar.

Finally there is the release which occurs when the source of excitation of the instrument is removed. This could be very rapid for example when the damper of a piano comes down on the string or more gradual as is the case with a trumpet. In a trumpet it takes some time for the air to stop vibrating in the horn.

Two basic types of instruments need to be defined to generalise their envelopes. These are: continuous excitation instruments and momentary excitation instruments. **Table 2.2** shows instruments which fit into one of these 2 categories.

CONTINUOUS EXCITATION INSTRUMENTS	MOMENTARY EXCITATION INSTRUMENTS
Brass	Plucked strings
Woodwinds	Pianos
Bowed strings	Percussion
Voices	
Whistles	
Organs	

Table 2.2 Types of instruments

In general the two types of instruments will have amplitude envelopes which look similar to the ones in **Figure 2.2** and **Figure 2.3**

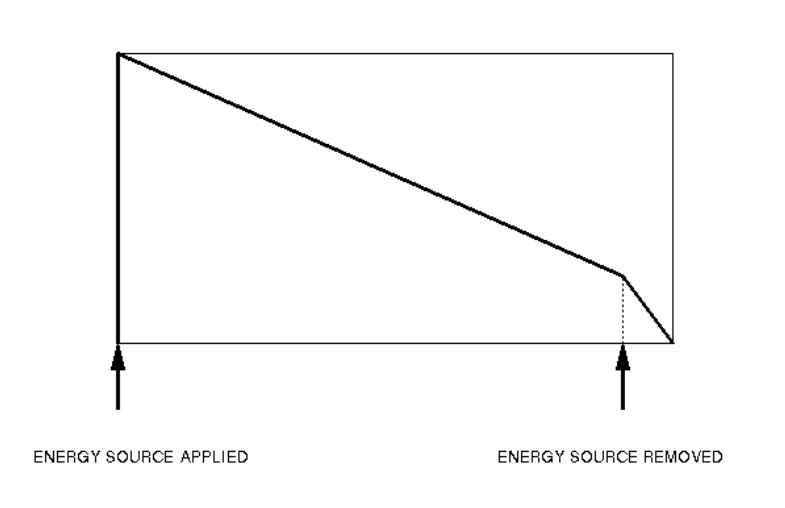

Figure 2.2 Momentary excitation envelope

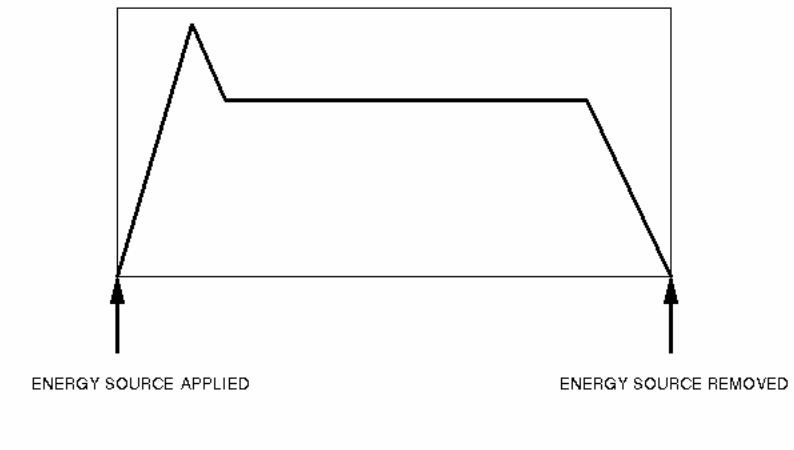

Figure 2.3 Continuous excitation envelope

In order for and instruments timbral and amplitude evolution to be fully described, however, a plot of each harmonic's envelope is necessary. For example **Figure 2.4** shows the harmonic envelopes of a cello.

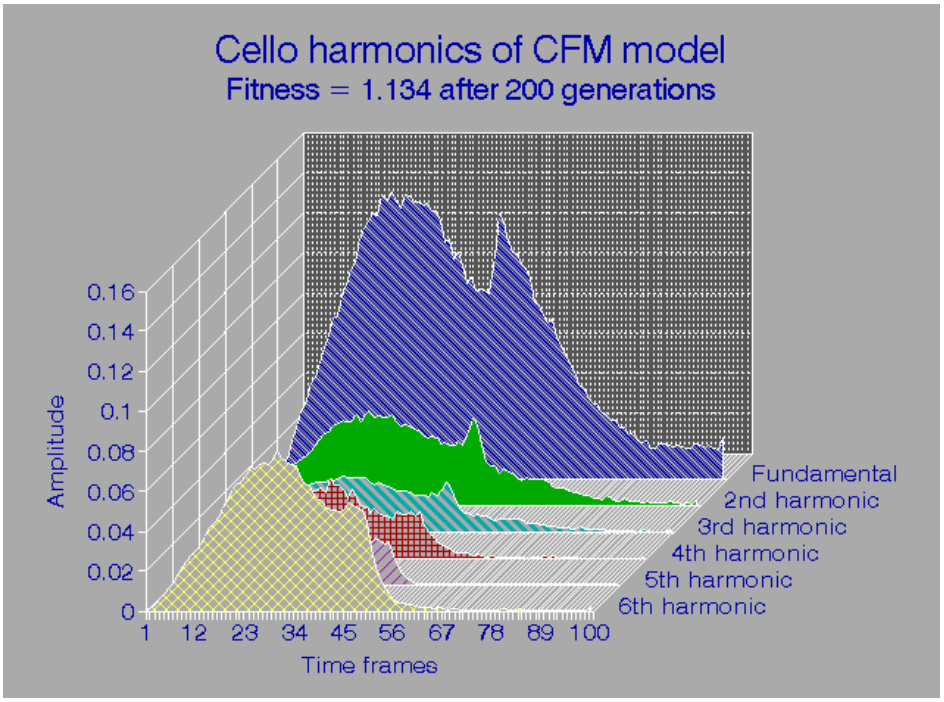

Figure 2.4 Cello harmonics

2.1.3 Style of instrument

When listening to the sound of a piano and guitar they can often sound quite similar played at the same volume. It is only when a pianist begins using the sustain pedal and plays thundering notes in its bass or when a guitarist plays quiet harmonics that the true style of the instrument comes into being. Articulation, phrasing and subtlety of expression are also crucial in determining the quality of an instrument.

2.2 Overview of music synthesis

The challenge for the synthesist as shown in the previous section is indeed substantial. Can instruments be created using modern electronics or computer software that will closely match instruments that are available today? Or that will go beyond this and explore new worlds of sound that will fulfil Verese's dream.

2.2.1 Brief history of music synthesis

Modern electronic music grew out of a French musical movement known as *musique concrete* developed by acoustical engineer Pierre Schaeffer and composer Pierre Henry in the late 40's. The primary aim of *musique concrete* was the alteration of the listener's perception. This was done through tape manipulation, splicing, filtering and electronic alteration of pre-recorded sounds. Although it couldn't be classified as pure music synthesis, some of the techniques used were important to the developments in electronic music.

The first true commercial synthesisers not committed to dedicated studios were the Minimoogs developed by Robert Moog in about 1969. These were analogue synthesisers using VCO's to produce wave shapes such as sawtooths and square waves linked to various audio filters to change the harmonic content of the sound. Analogue synthesisers dominated the market in the 70's with developments such as polyphonic and velocity sensitive synthesisers.

In 1978 an American company, Sequential Circuits, brought out the Prophet 5 which had five

separate built in analogue synthesisers with a microprocessor and memory storage for preset and programmable voices. This design technique was to be known as digital/analogue hybrid synthesis. As processors and memory began to become smaller and cheaper pure digital synthesis started to look like an attractive option. Yamaha moved on this idea and built a fully digital FM synthesiser in about 1982 called the DX7 which became enormously popular among electronic keyboardists. Other companies such as Casio also jumped on the digital bandwagon and built the Casio CZ101 which was a four voice multi-timbral instrument.

Today, the music synthesis market is dominated by digital synthesisers with some of the key brand names being Roland, Korg, Fairlight and Yamaha. Most synthesizers today employ what is known as PCM wave synthesis where high quality sampled sounds are stored in wave tables and manipulated. There are actually very few synthesisers available today were new sounds can be built up from basic waveforms.

2.2.2 Music synthesis techniques

All music synthesisers are analogue, digital or a hybrid thereof and employ one of three synthesis techniques, these being additive synthesis, subtractive synthesis and FM synthesis.

In additive synthesis basic waveforms such as sine waves or saw-tooth waves are added together to make a unique sound. If the number of sine wave generators equals the number of harmonics in additive synthesis an exact match of a real sound can be made. This is done by making sure that the amplitude of each sine wave generator is equal to the real sounds harmonic at a particular point in time. The only problem is that up to 100 sine wave generators would be needed, each multiplied by a time varying envelope to match a sound with 100 significant harmonics. So instead combinations of high spectral content waveforms such as square waves are combined with sine waves to form new instruments.

Subtractive synthesis makes use of pre-existing waveforms and sends them through a series of filters, thus "subtracting" spectra off the sound. These pre-existing waveforms could be the basic set such as square waves or complex samples of real sounds. The filters used are high-pass, low-pass, band-pass and notch filters which usually have tuneable cut-off frequencies, roll-off's and resonances. The original Minimoogs made use of subtractive synthesis with four pole

(24dB/Oct) filters which gave them there typical "fat" sounds.

FM synthesis is the process whereby one oscillator is used to change (modulate) the pitch (frequency) on another. On most FM synthesisers such as the DX-7 one or more oscillators are pre-wired as the "control oscillators" with the performer switching output of these oscillators on and off as desired. These oscillators then control the signal oscillators. As will be shown in the remainder of the thesis, FM synthesis can produce a large amount of possible sounds for a relatively simple amount of electronics in the analogue domain or data manipulation in the digital domain. FM synthesis has become standard on many sound kits that one buys today. For example the Sound Blaster comes with the Yamaha OPL-2 chip with hundreds of pre-selectable instruments. One should in no way, however, judge the flexibility of FM-synthesis on these simple sounds, because no envelope functions are employed and usually only up to 4 oscillators are used. FM synthesis is an art that still challenges the audio specialist. Recent papers in Computer Music Journal (Beauchamp et al 1994) and the Journal of Audio Engineering Society of America (Tan et al 1994) confirm this.

3 THE POWER OF FM SYNTHESIS

FM music synthesis was first investigated by John Chowning in 1973 (Chowning 1973). He showed that very convincing matches of many natural music sounds could be made with a relatively simple synthesis equation. The mathematical equation was one which was very familiar in the telecommunications field: The common broadcast FM equation with a carrier frequency a modulation index and a modulating frequency. As was shown in chapter 2, the challenge of music synthesis was creating temporal evolution of the spectra up to substantially high order harmonics for complex instruments. FM synthesis achieves this with relative ease and simplicity.

3.1 Chowning FM

To unlock the secrets of the FM equation, one firstly needs to look how it create its spectra. The FM equation looks as follows:

$$fm(t) = \sin(2\pi f_c t + I_m \sin(2\pi f_m t)) \qquad \textbf{(1)}$$

Obtaining its frequency domain plot has to be done using Bessel functions as there is no closed-form Fourier transform of the above equation. After some mathematical manipulation, the FM equation can be expressed as a sum of sine waves spaced at the modulating frequency from the carrier frequency.

$$fm(t) = \sum_{k=-\infty}^{\infty} J_k(I_m) \sin(2\pi(f_c + kf_m)t) \qquad \textbf{(2)}$$

Note this a short hand way of writing the true frequency domain plot which has 2 dirac deltas of the same amplitude at (fc+kfm) and -(fc+kfm) for each sine term. But the positive frequency terms are fully descriptive as the negative frequency terms are just the negative mirror image.

The Number of side frequencies that occur is related to the modulation index in such a way that as I increase, energy is stolen from the carrier and distributed among an increasing number of side frequencies.

As was stated in the introduction, the purpose of this thesis was only to produce harmonic instruments. For harmonic instruments the ratio of the carrier to the modulating frequency must obey the following relation.

$$\frac{f_c}{f_m} = \frac{N_1}{N_2} \qquad \textbf{(3)}$$

The fundamental frequency of the modulating wave will be

$$f_0 = \frac{f_c}{N_1} = \frac{f_m}{N_2} \qquad \textbf{(4)}$$

For example **Figure 3.1** shows the spectrum for f_c = 800Hz, f_m = 100Hz and I = 2.

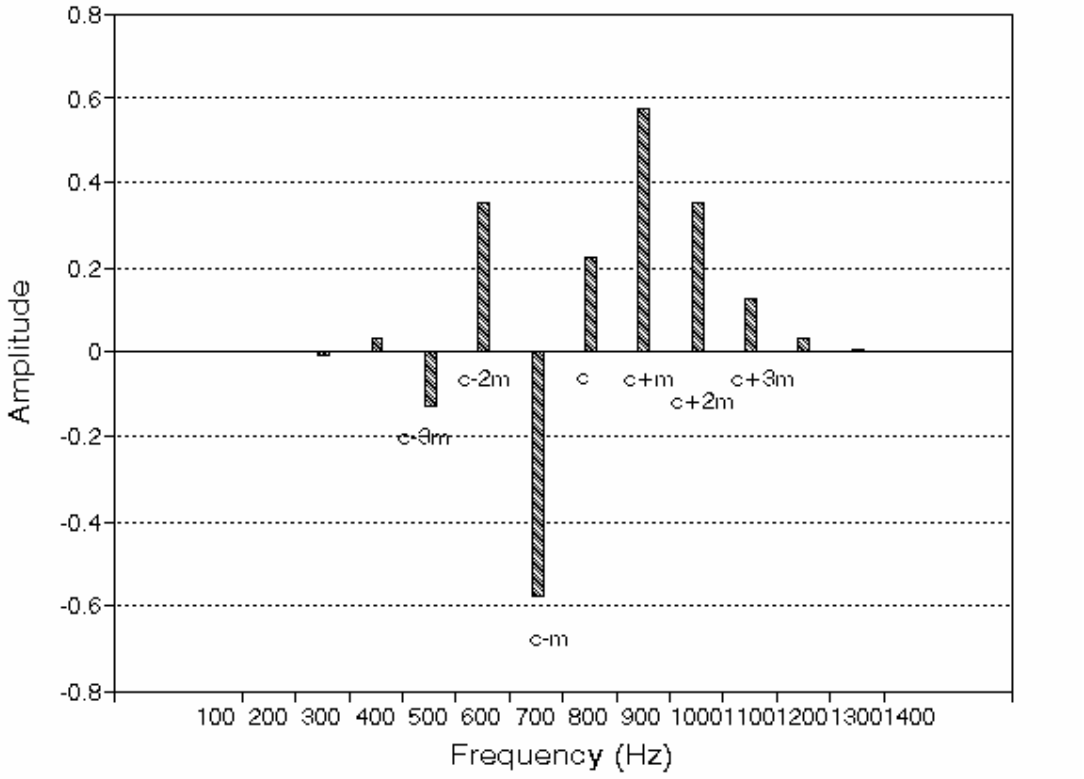

Figure 3.1 Chowning FM spectral plot with fc=800Hz, fm=100Hz and Im=2

Note that the spectra amplitudes are the same on either side of the carrier. The spectra with bars extending downwards have a phase which differs by 180 deg.

If the modulation index is increased further to I=4, an interesting phenomenon occurs. Some of the spectra now appear in the negative frequency domain. Negative frequencies are merely a notation and need to be interpreted in a real world situation. It turns out that the "negative" frequencies mix with the positive frequency spectra in such a way that they subtract from their corresponding positive frequency domain pairs. This is due to the simple trigonometric mapping of a sine function: sin(-2pift) = -sin(2pift).

Figure 3.2 shows a normal spectral plot representing negative frequencies and phase information. **Figure 3.3** shows how the "negative" frequency harmonics are reflected and **Figure 3.4** shows the final amplitudes of the real world harmonics. It is this harmonic mixing that allows FM synthesis to have such complex harmonic envelopes.

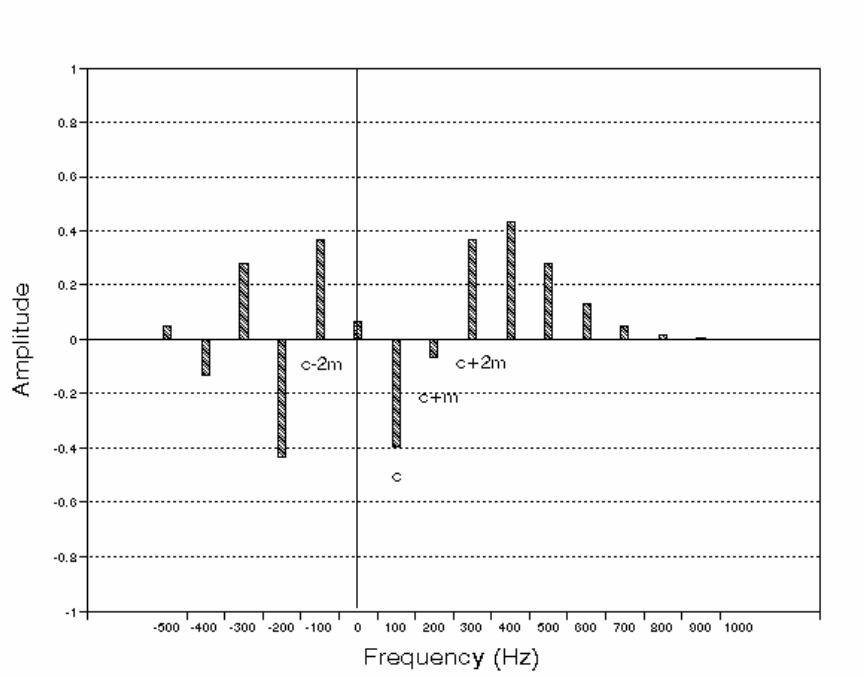

Figure 3.2 Chowning FM spectral plot with fc = 100Hz, fm = 100Hz and Im = 4

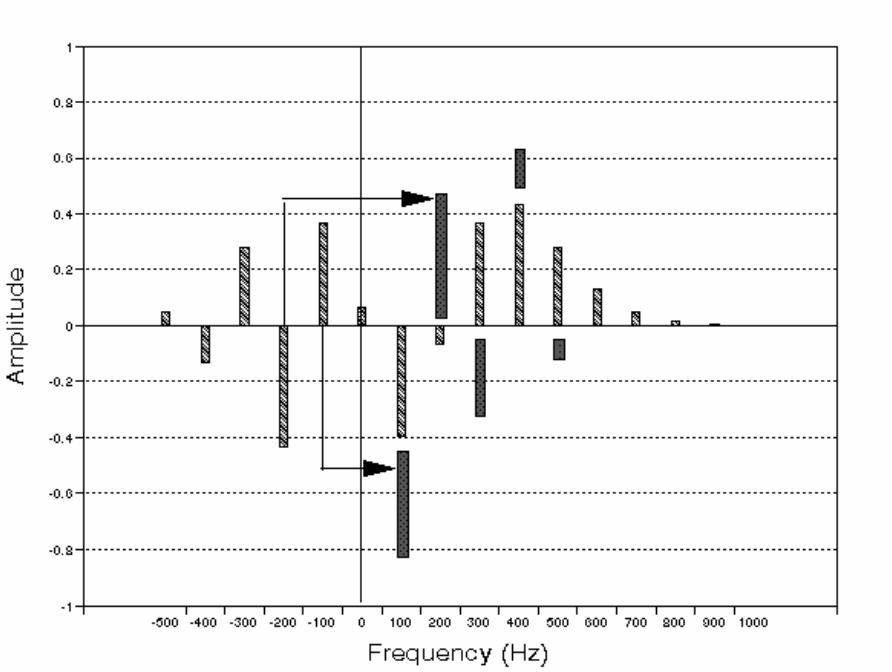

Figure 3.3 Chowning FM showing negative frequency reflection

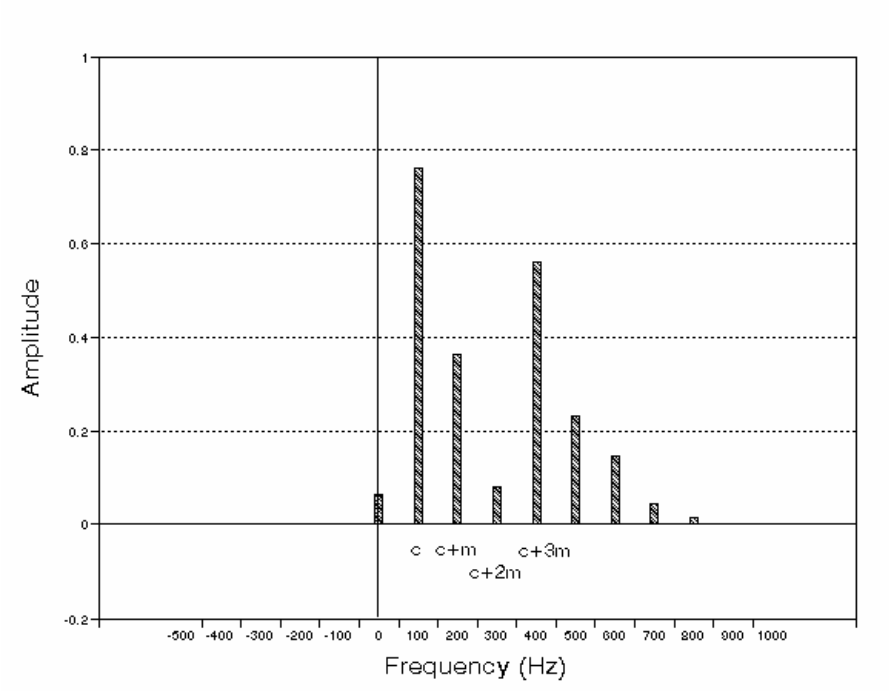

Figure 3.4 Chowning FM showing final amplitude of positive frequencies

3.2 Double FM

Variations on the theme of FM synthesis have been explored in recent years. One of the most successful of these is Double Frequency Modulation (Tan et al 1994) h will be referred to as DFM. The equation for DFM is as follows:

$$dfm(t) = \sin(I_a \sin(2\pi f_a t) + I_b \sin(2\pi f_b t)) \qquad \textbf{(5)}$$

Expressed as a sum of sine waves:

$$dfm(t) = \sum_{i=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} J_i(I_a) J_k(I_b) \sin(i(2\pi f_a t) + k(2\pi f_b t)) \qquad \textbf{(6)}$$

Simple chowning FM synthesis had a limitation in that unless a high modulation index was used, the spectrum was arranged symmetrically on either side of the carrier. In order to obtain complex harmonic envelopes it is necessary to employ complex combinations of several FM carriers.

DFM synthesis can provide sounds with far more complex harmonic envelopes with relatively little extra computational load. The only additional calculations necessary are one extra sine function and one extra multiplication. The reflected side frequency phenomena discussed for Chowning FM also applies to DFM.

It proves a little more difficult to predict what the spacing between the harmonics will be. Again the assumption is made that harmonic sounds will be generated, so the following rule must be obeyed.

$$\frac{f_b}{f_a} = \frac{N_2}{N_1} \qquad \textbf{(7)}$$

If N1 and N2 are nonzero integers then the following 3 empirical rules for the frequencies generated can be applied to DFM (Tan et al 1994), they are:

1. N1 odd, N2 odd

$$f = f_a \pm n\frac{2f_a}{N_1} \quad \textbf{(8)}$$

2. N1 odd, N2 even or N1 even, N2 odd

$$f = f_a \pm \frac{nf_a}{N_1} \quad \textbf{(9)}$$

3. f_a or f_b is Zero

$$\begin{aligned} f &= \pm nf_a \ \ if \ \ f_b = 0 \\ f &= \pm nf_b \ \ if \ \ f_a = 0 \end{aligned} \quad \textbf{(10)}$$

Example spectra for the 3 cases are shown in **Figure 3.5** , **Figure 3.6** and **Figure 3.7**

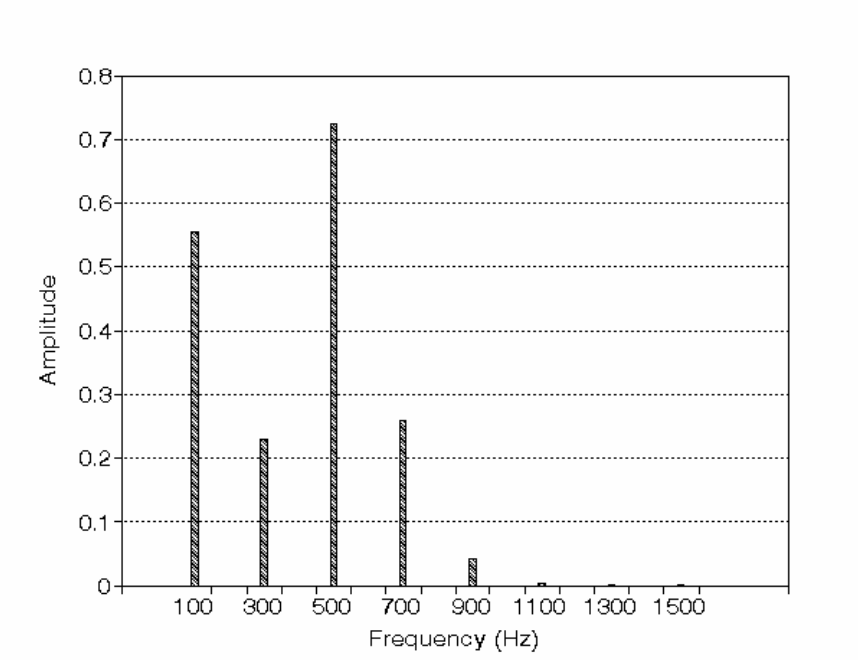

Figure 3.5 DFM with N1 odd and N2 odd (fa=100Hz, fb=100Hz, Ia=Ib=3)

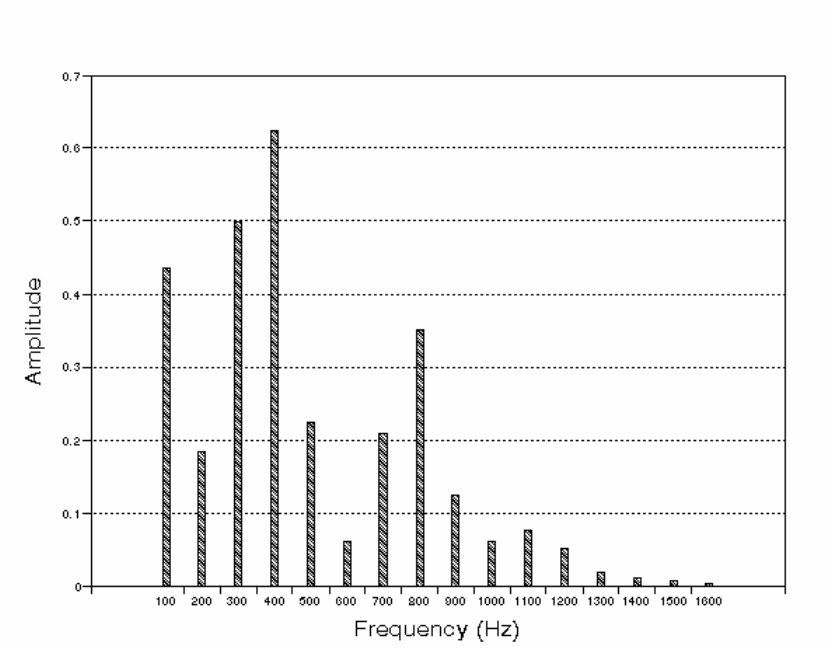

Figure 3.6 DFM with N1 off and N2 even (fa=100Hz,, fb=200Hz, Ia=Ib=3)

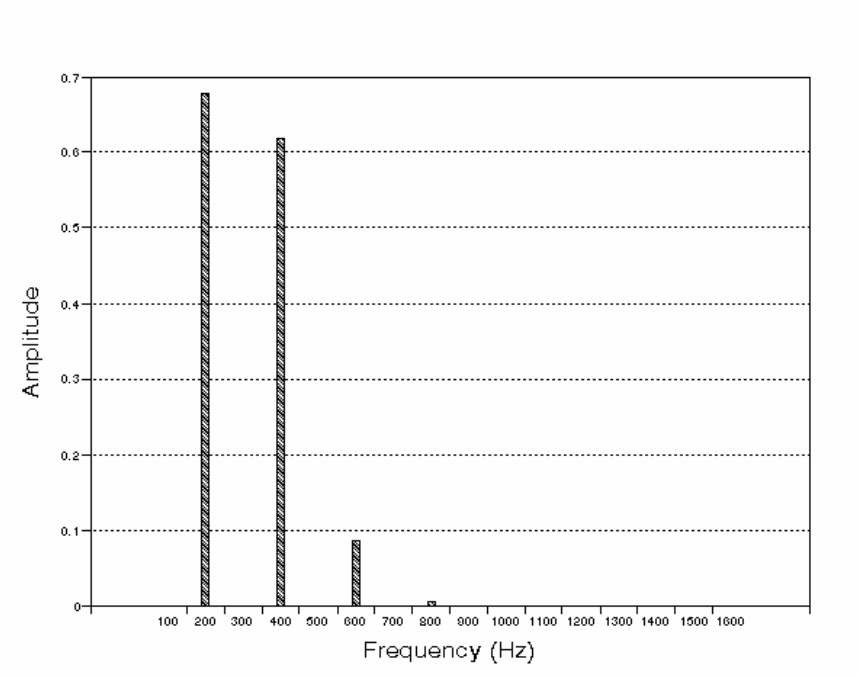

Figure 3.7 DFM with fa=100Hz, fb=0, Ia=Ib=3

3.3 Synthesis models

Up until now the basis behind 2 FM synthesis equations has been presented. These equations, however, are not complete enough in themselves to synthesise any natural musical sound. There are 2 reasons for this. The first is that evolving spectra are needed to convincingly match real sounds and the second is that a determined number of spectral permutations will occur for the set of all possible harmonic parameters in the CFM and DFM equations. Even DFM with its considerable improvement on the number of spectral permutations will have a very small chance of producing a harmonic spectrum that will match that of a real instrument at a particular point in time.

To solve this problem a set of FM equations are added together each with it's own FM parameters. Each of these equations is known as a carrier and each carrier is multiplied by a weight function. There are a number of ways to create evolving spectra, one of them is to apply an envelope function to the modulation index (Chowning 1973), another method is to use non-linear envelopes as the weights for each of the carriers. (Beuchcamp et al 1993)

Using envelope functions on the modulation index tends to create very unrealistic sounds. The reason for this is that changing the modulation index causes rapid chaotic changes in the spectra. This is one of FM strengths and weaknesses: small changes in the parameters cause unpredictable changes in the spectra yet one often comes across a change which will be pschoacoustically very satisfying. The only known factor when increasing the modulation index is that it increases the bandwidth of the sound. The sounds created by modulation index envelopes are interesting in their own right, but I wouldn't call them realistic as Chowning likes to suggest in his paper.

A far more robust and controllable way of creating evolving spectra is using envelope weights for each of the carriers. Using this method smooth transitions and combinations of the spectral basis plots for each of the carriers are obtained. This was the final model settled on and two diagrams are shown in **Figure 3.8** and **Figure 3.9** for the CFM and DFM model counterparts.

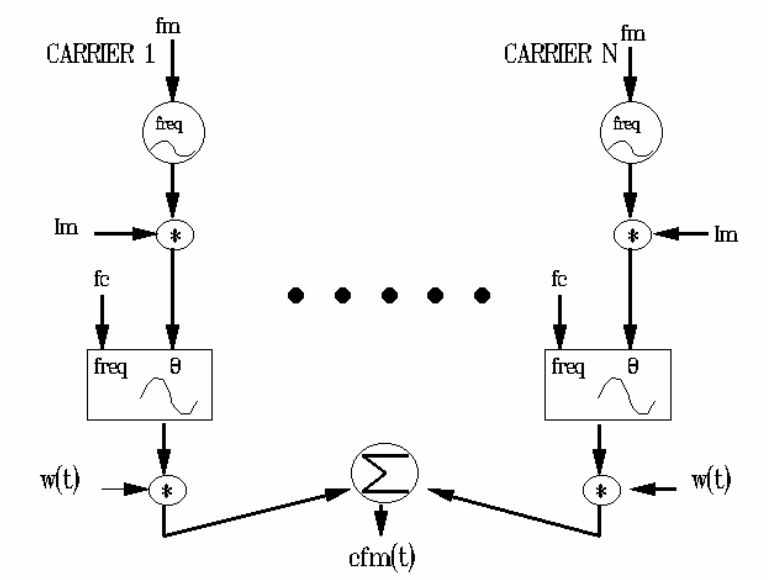

Figure 3.8 CFM model

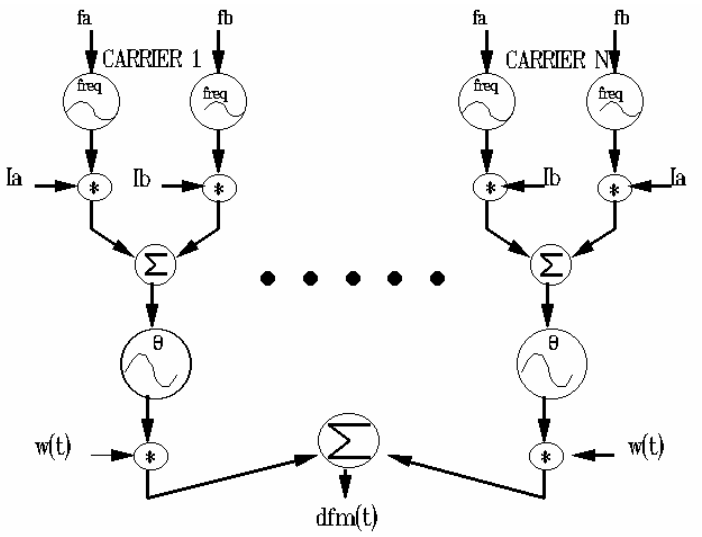

Figure 3.9 DFM model

Choice of good weighting envelopes is crucial in determining the quality of the final sound. Each carrier will have a fixed basis spectrum that now needs to be carefully combined through time varying envelopes. It is almost like having the ingredients for a delicious cake but not knowing how much of each ingredient put in or at what temperature to bake it.

The two classes of instruments discussed in chapter 2 are of crucial importance in choosing weight functions. Continuous excitation instruments such as a trumpet will have weight envelopes with a slow attack, a constant sustain and a relatively fast release. Momentary excitation instruments such as a piano will have a rapid attack, no sustain and a slow release. These envelope weights will be revealed in the following two chapters when weights are derived from real instrument matches and implicit envelopes are used for guiding a sound search.

The two FM synthesis models discussed so far have the ability to produce an enormous range of possible sounds. Guidelines which tell you which parameters to use to produce particular genres of instruments are available (Chowning 1973)(Tan et al 1994)] but by no means cover all the permutations of sounds in the models sound-space. It is possible to simply experiment with frequency ratios, modulation indexes and envelopes and hope to come across an interesting instrument but once an instrument is found, refinement of the sound is impossible. This is due to the unpredictable nature of Fm sounds when changing its parameters.

What is needed is a tool to guide the sound search, in the same way that a potter needs a potter's wheel to make beautiful pottery. The genetic algorithm has become a very attractive function optimization technique, especially in the artificial intelligence research domain. It consistently outperforms gradient techniques on difficult problems such as optimizations involving noisy, high dimensional and multimodal functions. [Grefenstette 1990). One of their other important strengths is the fact that they are general purpose function optimizers and are therefore not problem specific.

4.1 Basic outline of how GA works

The Genetic algorithm is a combinatorial optimization technique which evolves solutions to problems using processes analogous to natural selection and breeding. The standard GA uses selection, crossover and mutation as its genetic operators. These operators work on a population of candidate solutions which are represented as encoded genes. All the pertinent information for the function you are trying to optimize must be encoded in the gene, the most popular method and the one used in this thesis is binary encoding where all the parameters are mapped to a bit strings which are concatenated together.

The GA begins by either by creating a random initial population or getting an initial population from the user. Each individual gene in the population is then evaluated using a fitness function.

Selection of the fittest genes to be used for mating is then done. Typically about 60% of the genes are selected for mating although this percentage is often increased when smaller populations are used. The selection strategy chosen for this thesis is known as proportional selection which is best understood using a roulette wheel analogy. Each gene is assigned a wedge which is proportional in area to its fitness value. If the population size was 100 and the selection percentage 60%, the wheel would be spun 60 times. Whenever a ball lands on a wedge, the corresponding gene would be stored in the list of genes to be used for mating.

In order to search other points in the search space variation is introduced into the new population by the next GA operator known as crossover. Under the crossover operator, two genes in the mating population exchange portions of their binary representations. This is implemented by choosing a point at random in the genes called the crossover point and exchanging segments to the right of this point. As an example, if the two genes looked as follows:

x1 = 11001|100
x2 = 10110|001

After crossover the resulting genes would be:

y1 = 11001|001
y2 = 10110|100

Crossover is understood using what is known as schema theory. A schema is a set of possible genes eg. 100##### is the set of all genes starting with 100, where a # means "don't care". Crossover serves two complementary search functions. Firstly it provides new points for testing within schemas already present in the population. eg. x1 and y1 are representatives of the schema 11001###, and by evaluating y1, the GA gathers further information about this schema. Secondly the GA introduces representatives of new schemas into the population. In the above example y2 is a representative of the schema ##1101## which is not represented in either parent.

After all the crossovers have been done, the new children replace the parents in the old population and a new population is formed. To prevent premature convergence to a local maximum an auxiliary operator called mutation is used. If the mutation rate is set at 0.1% then 1 out of every 1000 candidates will have a random bit flipped from a 1 to a 0 or a 0 to a 1.

Each individual in the new population is then evaluated again using the fitness function and the algorithm repeats. The GA terminates after a certain set number of trials have been done or when a termination condition is reached.

The GA process is summarised in the **Figure 4.1**

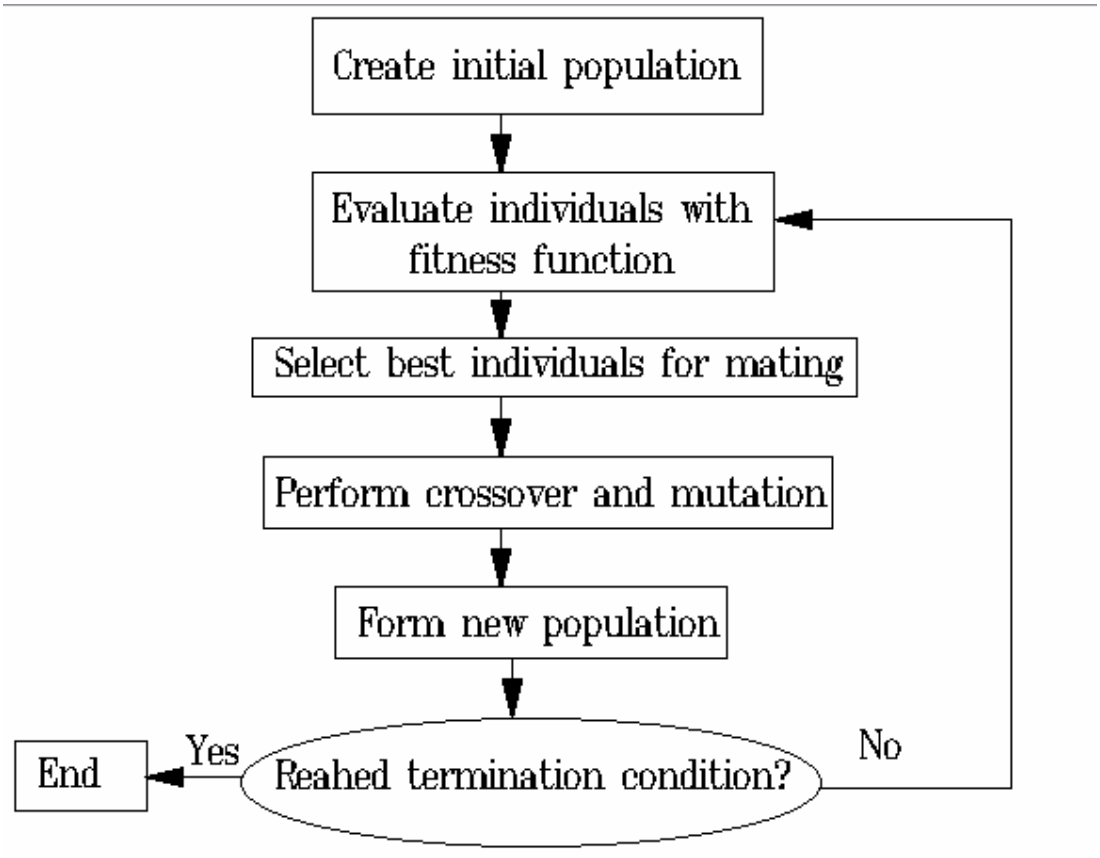

Figure 4.1 GA flow chart

4.2 GA method for Matching real sounds

(Note much of the work in this section is based on ideas obtained from a paper written in Computer Music Journal entitled: Machine Tongues XVI: Genetic Algorithms and their application to FM Matching Synthesis. It must be stressed though that no source code was given for algorithms used, only one type of synthesis model was tried out and many of the equations given were actually incorrect)

A good starting point to test the flexibility of the synthesis models and the power of the Genetic Algorithm is to try and find all the FM parameters and envelopes that will match a real sound. There are however a number of complications that need to be ironed out first before the Genetic Algorithm can be used successfully.

It is considerably easy to encode all the parameters of the FM equation for each carrier into the gene, but how would one represent the envelope functions? The answer to this lies in understanding the interdependency of the basis spectra which each fm carrier function produces and the envelopes which these spectra are multiplied by. Once the basis spectra are known there must exist a set of envelope functions which will cause the synthesis model to match the evolving sound you're trying to match as closely as possible. The converse of this statement is also just as valid. Finding the envelopes, once the basis spectra are determined, turns out to be a simple least mean squares matrix solution.

The matching process ends up being a two-edged problem which is described in **Figure 4.2**. The GA creates a new population of genes, which create a unique set of basis spectra for each carrier. These spectra are stored in a harmonic matrix A. A least mean squares algorithm is then used to solve for W in the equation AW = B. where B contains successive frames of the original discrete-time spectra and W contains the harmonic weights for each carrier. The successive frames of the model discrete-time spectra are created by multiplying the harmonic matrix A with the weight matrix W (Bmodel = AW). The model and original spectra are then compared and a fitness value is obtained which is scaled according to the closeness of the match. This fitness is obtained for each gene , the GA takes over, forms a new population and the process repeats itself until a termination condition is reached.

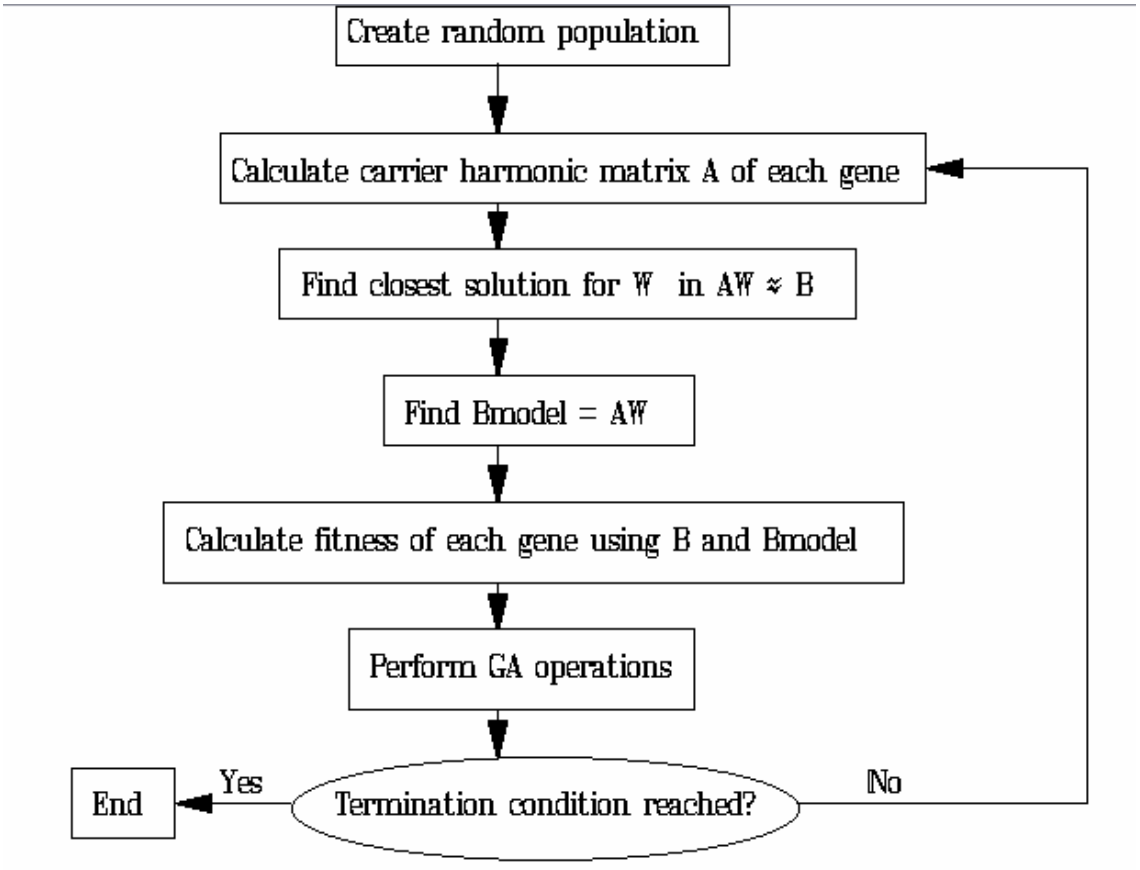

Figure 4.2 Real sound matching process

4.2.1 Obtaining the original discrete-time spectra Matrix B

A standard FFT algorithm was used on a sampled file of the instrument trying to be matched. 220Hz, which is A below middle C, was chosen as the frequency that most of the instruments would be sampled at. The radix-2 number N, which is be the number of data points analysed and the number of frequency bins produced, was chosen next. If the sampling rate equals 220*N then the FFT will analyse exactly one period of the wave and the fundamental will be stored in the 1st frequency bin, the 2nd harmonic in the 2nd and so on. This is a very useful result, because any harmonic up to the Nth can be used in the matching process. The highest sampling rate below the sound blasters cut-off of 44kHz must now be found. With the sampling rate equal to 220*N where N must be a power of 2, the best sampling rate is

28160Hz with N=128. All the instruments were sampled at 28160Hz for a period of about 2 seconds and stored in a WAV file. (See Appendix B for format of a WAV file)

The WAV file was analysed over a number of discrete time frames and the harmonics for each time frame were stored in a harmonic matrix which will be called B. Typically 10 time frames where used for the matching process. In instruments with difficult attack, 10 spectra equally spaced in time between the beginning of the sound and the point where the highest amplitude of the sound occurs were selected. Whereas in continuous excitation instruments, 10 spectra over their whole period where used. The structure of matrix B looks as follows:

$$B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1N_{frames}} \\ b_{21} & b_{22} & \dots & b_{2N_{frames}} \\ . & & & \\ . & & & \\ . & & & \\ b_{N_{hars}1} & b_{N_{hars}2} & & b_{N_{hars}N_{frames}} \end{bmatrix} \qquad \textbf{(11)}$$

Where N_{frames} = Number of frames and N_{hars} = Number of harmonics.

The code for the FFT process in Appendix A

4.2.2 Obtaining the carrier harmonic matrix A

This is the only step which is dependent on the type of FM synthesis being used. The harmonic equations for CFM and DFM synthesis will be derived in turn.

4.2.2.1 CFM synthesis

In chapter 3, Equation (2) showed the sine wave decomposition of the CFM function. To find the harmonic amplitudes the negative frequency components need to be reflected onto the positive frequency components. For harmonic tones, fc and fm are related by a rational multiplier. When f_c is an integer multiple of f_m we have a special case of this result and the fundamental frequency will be $f_1 = f_m$ as predicted by equation (4). This limitation in no way hinders the convergence of the model on a real sound because all the harmonics are still generatable by the model. This is due to the fact that the harmonics space themselves at integral multiples of the modulation frequency around the carrier frequency. The modulation frequency happens to be the fundamental, so all the harmonics are accessible.

Letting $f_c = n*f_m$, equation (2) becomes

$$cfm(t) = \sum_{k=-\infty}^{\infty} J_k(I_m) \sin(2\pi(n+k) f_1 t) \qquad \textbf{(12)}$$

Spectral reflection of this equation becomes easy and ignoring the DC component equation (12) becomes

$$cfm(t) = \sum_{k=1}^{\infty} [J_{k-n}(I_m) - J_{-(k+n)}(I_m)] \sin(2\pi k f_1 t) \qquad \textbf{(13)}$$

If the modulation frequency is made the same for all the carriers the kth harmonic of the jth carrier is given by

$$a_{kj}(I_{mj}) = J_{k-n_j}(I_{mj}) - J_{-(k+n_j)}(I_{mj}) \quad \textbf{(14)}$$

$n_j = f_{cj}/f_m$ is the ratio between the jth carrier frequency f_{cj} and the common modulator frequency f_m. I_{mj} is the modulation index of the jth carrier.

4.2.2.2 DFM synthesis

The sine wave decomposition of the DFM equation is given by equation (6). Simplifications can again be made to the model by making f_a constant for all carriers and making f_b an integral multiple of f_a. So $f_b = n*f_a$. If the empirical rules of equations (8), (9) and (10) are analysed one sees that again no severe limitations are put on the search space as all harmonics can be generated in each carrier. Equation (6) becomes:

$$dfm(t) = \sum_{i=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} J_i(I_a) J_k(I_b) \sin(2\pi(i+kn) f_a t) \quad \textbf{(15)}$$

Unfortunately due to the lack of symmetry of this equation, an implicit equation for the harmonics cannot be directly found. A combinatorial solution is however easy to obtain. A complete description of this is relegated to Appendix F.

Either of the two synthesis models will produce a carrier harmonic matrix A with the following structure.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N_{cars}} \\ a_{21} & a_{22} & \dots & a_{2N_{cars}} \\ . & & & \\ . & & & \\ . & & & \\ a_{N_{hars}1} & a_{N_{hars}2} & & a_{N_{hars}N_{cars}} \end{bmatrix} \quad \textbf{(16)}$$

Where N_{hars} = Number of harmonics used in analysis and N_{cars} = Number of carriers.

4.2.3 Calculating the Weight Matrix W

Once the harmonic matrix A and the discrete-time spectra matrix B are known, there must exist a weight matrix which will solve AW = B in the least mean squares sense. It is useful firstly to take a closer look at the whole structure of this equation (17), it looks as follows:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N_{cars}} \\ a_{21} & a_{22} & \dots & a_{2N_{cars}} \\ . & & & \\ . & & & \\ . & & & \\ a_{N_{hars}1} & a_{N_{hars}2} & & a_{N_{hars}N_{cars}} \end{bmatrix} x \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1N_{frames}} \\ w_{21} & w_{22} & \dots & w_{2N_{frames}} \\ . & & & \\ . & & & \\ . & & & \\ w_{N_{cars}1} & w_{N_{cars}2} & & w_{N_{cars}N_{frames}} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1N_{frames}} \\ b_{21} & b_{22} & \dots & b_{2N_{frames}} \\ . & & & \\ . & & & \\ . & & & \\ b_{N_{hars}1} & b_{N_{hars}2} & & b_{N_{hars}N_{frames}} \end{bmatrix} \quad \textbf{(17)}$$

If the number of carriers equals the number of harmonics then the solution of the above equation is implicit using direct matrix inversion of A. What we want, however, is a reduced set of carriers, otherwise straight additive synthesis could have been used where each carrier represents a sine wave at a harmonic frequency. For a reduced set of carriers equation (17) is what is known as over determined and a process called single value decomposition must be done on A to solve for W. The solution can be seen in appendix D.

4.2.4 Finding the fitness

Once the closest weight matrix is calculated, a closest fit model can be calculated by multiplying the carrier harmonic matrix A by the weight matrix W (Bmodel = AW). The structure of Bmodel is identical to B.

The most sensible fitness function should have a bearing on how the closest weight matrix was determined. That is the sum-squared difference between the reconstructed spectrum the original spectrum over all the time frames. This telescopes down to the following equation for the fitness function.

$$fitness = \sum_{i=1}^{N_{frames}} \left[\frac{\sum_{k=1}^{N_{hars}} (B_{ki} - Bmodel_{ki})^2}{\sum_{k=1}^{N_{hars}} bmodel_{ki}^2} \right]^{1/2} \qquad \textbf{(18)}$$

4.2.5 A complication with harmonic signs

The carrier harmonic matrix of equation (16) is a little simplistic because the signs of many of the harmonic amplitudes in FM-generated spectra are negative reflecting a 180° phase shift and the B matrix only contains spectral magnitudes. The ear is impervious to this phase shift as can be seen in appendix E so it is not crucial to the recreation of the sound but it does mean that our equation AW ≈ B needs to be modified slightly. The simplest way to do this is to construct a diagonal square matrix S with each dimension equal to the number harmonics. The diagonal elements are unknown and can take on a value of plus or minus one. The adapted matrix equation now looks as follows:

$$AW \approx SB \qquad \textbf{(19)}$$

Minimization now involves finding both W and S. An easy way to calculate S is to include it in the gene where the sign of each harmonic is a 1 bit parameter. With D = SB, for a given individual S, least mean squares will be used minimize AW ≈ D. So the genetic algorithm will be trying to find the set of modulation indices, carrier ratios and amplitude signs that make the best match.

4.2.6 Gene encoding

4.2.6.1 CFM Gene

Each carrier has an FM function defined by a carrier to modulation frequency ratio, n and a modulation index I_m. These parameters are translated into binary equivalents and placed adjacent to one another in the gene. There are also the sign bits discussed above which are tagged on at the end of the gene. The fully descriptive gene is shown below in **Figure 4.3** with the number of bits used to represent the parameters shown above.

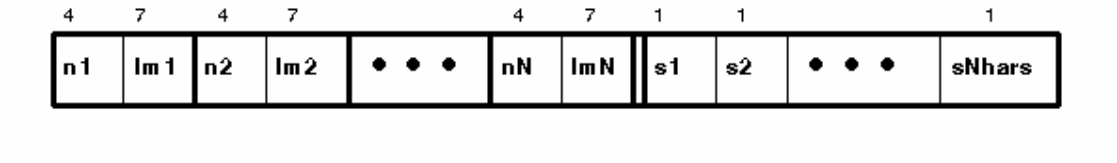

Figure 4.3 CFM Gene for real sound matching

4.2.6.2 DFM Gene

Each carrier has an FM function defined by a Frequency b to a ratio, n and two modulation indexes Ia and Ib. These are also translated to binary equivalents and placed in the gene. Again sign bits are tagged on at the end of the gene. The gene is shown below in **Figure 4.4** with the number of bits used to represent the parameters shown above.

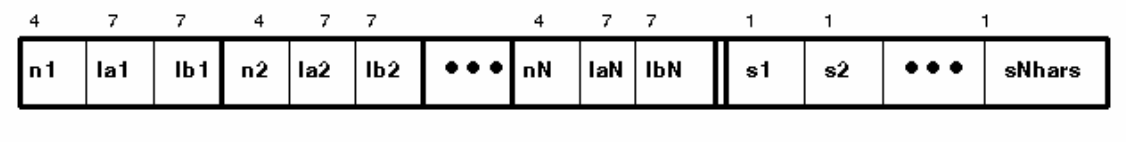

Figure 4.4 DFM Gene for real sound matching

4.2.7 Final match of sound

Once the best possible parameters to the FM equations are found using 10 frames of harmonics, a more rigorous least-mean-squares match with 100 harmonic frames is finally done to create the sound wave. When the final sound is created, linear interpolation between weight frame points is done. For example, if the final wave is created using a sample rate of 30 kHz with a period of 2 seconds, 60000 sample points are used. Each of the 100 frames of weights for each carrier represents 600 data points. To find out the weight value at a particular data point, a straight line is drawn from one frame weight to the next and the weight value is read off this straight line.

4.2.8 GA parameters used

Traditional values for the GA parameters were found to give the best results. These are a population size of 50, crossover rates of 60% and a mutation rate of 0.1%. A total number of 10000 trials which is 200 generations were typically used after which the fitness improvements were very small.

4.3 GA method to create new instruments with human guide

Now that the synthesis models have been taken through their paces with impressive matches to real sounds as will be seen in the next chapter, an exciting concept follows from this. If the genetic algorithm can search the sound space for real instruments, surely it can search the sound space for instruments that lie between these real sounds if the user enters fitness values for the sounds generated by FM parameters. There is however, a fundamental subjective

riddle in this proposition, and that is that the only way we can relate to what we define as a "good" sound is by comparing a new sound with pschoacoustically pleasing sounds that we have become accustomed too. This means that when searching the sound space there will be a natural tendency to find new sounds that will cluster around sounds we already know. When trying to envisage a new sound we would tend to think of an interesting bell-like piano or a organ-like marimba rather then a completely foreign unheard-of sound.

The sound search algorithm is outlined in **Figure 4.5**. The GA firstly creates an initial random population. At point A each of the genes in the population is interpreted and corresponding FM sound waves are generated. Each sound is initially assumed to have a fitness of zero and at point B the sound numbers are displayed on the screen together with their current fitness. The user then enters the number of the sound to be heard and the corresponding sound is played, after which the computer prompts for a fitness value. The algorithm then goes back to point B until the user is finished evaluating the sounds. The computer then prompts the user to see if more sounds need to be searched for and the algorithm ends if the user is content with the current population. If not, All the GA operators act on the population and the algorithm goes back to point A.

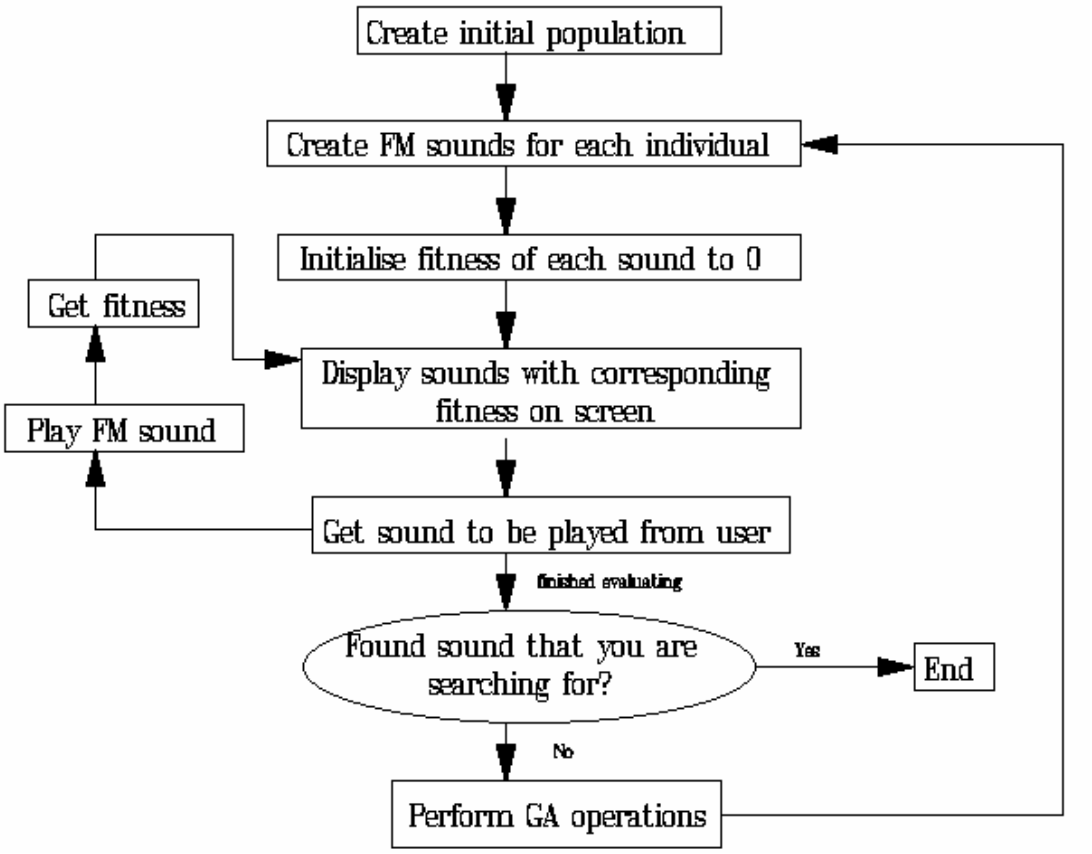

Figure 4.5 Sound Search algorithm

4.3.1 Envelopes used

The modelling approach needs to be changed a little from the real-sound matching process to do the sound search. Firstly the envelopes or weights as they were called in the previous section need to be encoded in the gene. It is obviously out of the question to store each discrete value in the envelope in the gene as this will result in genes thousands of bits long. If the weight function results from the real sound matches are studied in the next chapter one sees that they often tend to follow a $-x^2$ parabolic rise and a $+x^2$ parabolic fall with peaks at different positions for each weight.

Using parabolic envelopes with its amplitude and peak position encoded in the gene turns out to produce rich and varied sounds.

Two types of envelopes which correspond with momentary excitation instruments and continuous excitation instruments were used. These are shown in **Figure 4.6** and **Figure 4.7**. With the peak of the attack for each carrier's envelope falling at different points, complicated attacks were generated due to the non-linear interaction of the spectra of each carrier.

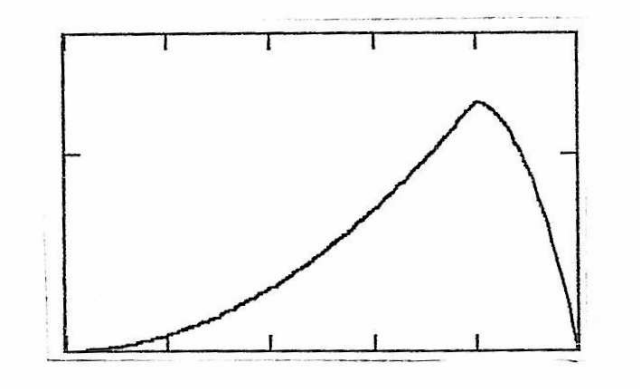

Figure 4.6 Momentary excitation parabolic envelopes

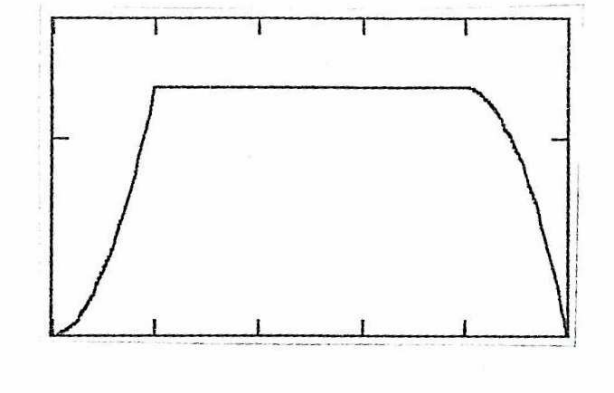

Figure 4.7 Continuous excitation parabolic envelopes

4.3.2 Generation of waves

Waves were created in standard PCM WAV file format outlined in Appendix B. An initial problem was the amount of time it took to generate the WAV files for each population. Evaluating the sine function at a particular point was one of the key causes of the lack of speed and high sampling rates of 30kHz, which were initially thought to be needed, consumed time. Storing the sine wave in a lookup table was used to solve the first problem. The sampling rate could also be lowered to as little as 5kHz without too much degradation in

the quality of the sound. This is understandable when considering that the sound was generated at 220Hz and harmonics up to the 11th are still significant. The 11th harmonic is at 2420Hz which is less than half the sampling rate, so the Nyquist theorem is satisfied. The time to generate a WAV file was brought down from about 25 seconds to 3 seconds using these techniques. The code to generate WAV files is shown in Appendix B.

4.3.3 Playing waveforms

Reliable C code was found which could play WAV files through a Sound Blaster's D/A (Morgan 1993). The C code had the ability to load all the WAV files into expanded memory so that quick inter-comparison of sounds was possible without searching for data on the hard-drive.

4.3.4 Gene encoding

The first carrier had its amplitude set to maximum and the modulation frequency in CFM or frequency B in DFM set to the fundamental frequency. This ensures that all the sounds are created at the same pitch. All possible variables in the model are exploited short of the sound becoming non-harmonic to create as much diversity as possible.

4.3.4.1 CFM gene encoding

Figure 4.8 shows how the CFM gene was encoded. For each carrier an envelope amplitude a_i, a envelope peak position $envp_i$, a carrier-to-base frequency ratio f_{ci}, a modulation index I_m and a modulation-to-base frequency ratio f_{mi} must be determined. The first carrier has less information in it because f_m is set to the fundamental frequency and the envelope amplitude is set to its maximum. The number of bits used to represent the frequency ratios directly reflects the number of possible frequencies it can produce. Typically 4 bits were used for the frequency ratios which will allow f_m or f_c to contain 16 possible multiples of the base frequency. 7 bits were used to represent the modulation index I_m and with a range [-10, 10], the resolution is about 0.156. The number of bits used to represent the parameters is shown above their respective parameters.

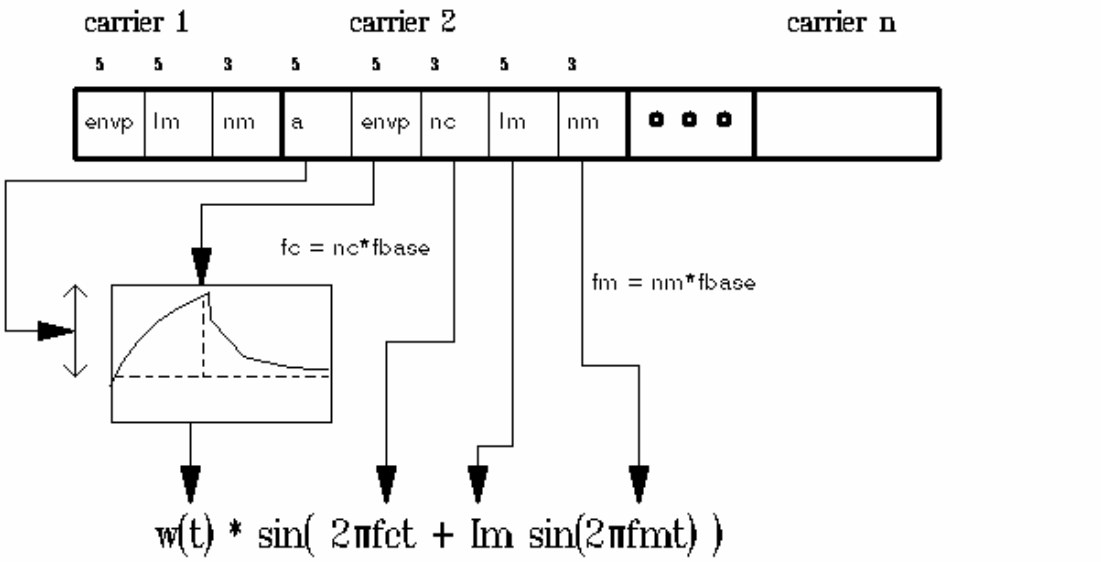

Figure 4.8 CFM Gene

4.3.4.2 DFM gene encoding

Figure 4.9 shows how the DFM gene was encoded. For each carrier an envelope amplitude a_i, a envelope peak position $envp_i$, Modulation indexes I_{ai} and I_{bi}, frequency a-to-base frequency ratio n_{ai} and frequency b-to-base frequency ratio n_{bi} must be determined. Again the

first carrier has less information in it because f_a is set to the base frequency and the envelope amplitude is set to its maximum. The same bit values and parameter ranges that were used in CFM were used for DFM and the number of bits used to represent each parameter is shown above their respective parameters.

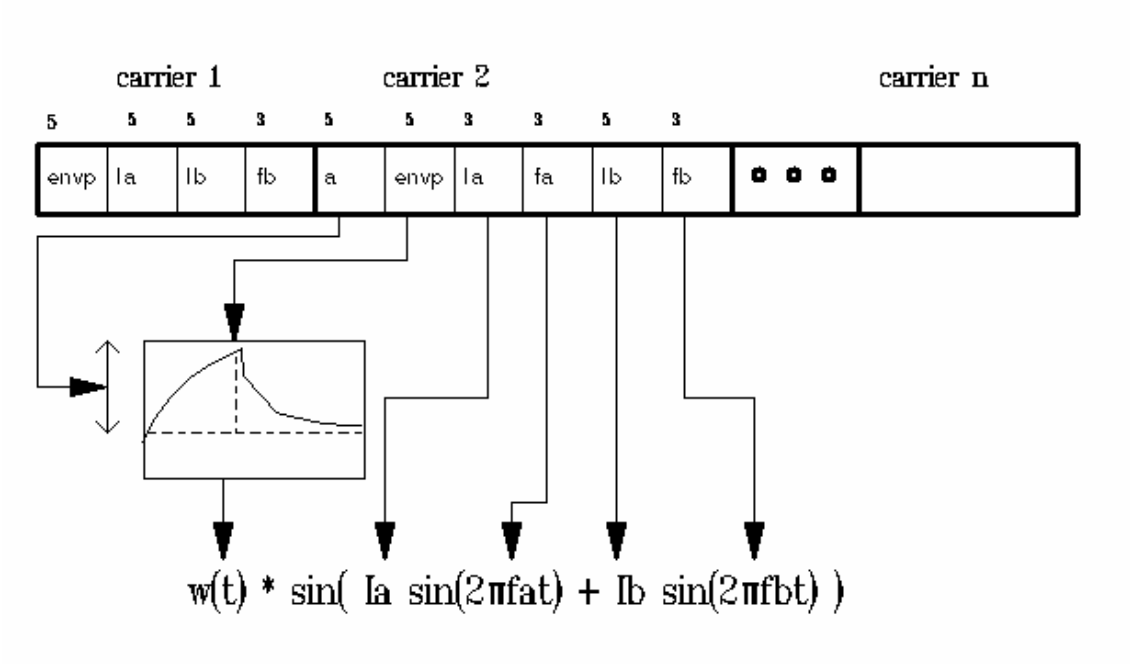

Figure 4.9 DFM Gene

4.3.5 GA parameters used

A population size of 10 was about the largest that could be used before inter-comparison of sounds become too difficult. Crossover rates of 100% and mutation rates of 10% were typically used to create as much diversity as possible in the search. These controversial values are however understandable when considering that finding a pschoacoustically pleasing new sound in a considerable large search space is a very unexacting science.

4.4 GA algorithm used

The GA code that was used for all the experiments was Greffenstettes GENESIS version 5.0. (Greffenstette 1990). The code was written in a very accessible modular fashion, so that any fitness function could easily be written and used by the GA. The complete real sound matching process described in section 4.2 could be condensed into the fitness function and is shown in appendix C. Similarly the human fitness tester code which creates and plays WAV files could all be placed in the fitness function.

5 RESULTS OF SOUND SEARCHES

5.1 Matching real sounds

The matches performed to date have been done on cello, voice, guitar, piano, organ and recorder. Each instrument had its own nuances which had to be taken into account when running the matching process. For instance the guitar and piano had a very complicated attack, so the harmonic matrix B stored spectral time frames from the start of the sound until its amplitude peak for matching whereas the recorder, organ, voice and cello had relatively simple attacks, and spectral time frames were used over their complete period. 10 spectral time frames were used for the parameter search

The number of carriers needed to perform a good match depended on the complexity of each instrument. It was found, however, that 4 carriers was usually always sufficient for a convincing match. If the number of carriers equals the number of harmonics of the original sound, an exact match can be achieved through additive synthesis but defies the whole purpose of FM synthesis, which is carrier reduction.

The number of harmonics used for the matching process was a difficult parameter to pick. Doing an FFT on all the instruments that were going to be matched showed that most only had more than 30 harmonics on the attack, so 30 harmonics was chosen as a good average value.

Increasing the number of carriers, harmonics or spectral time frames increases the amount of time taken to do an evaluation proportionally to any one of these parameters. So doubling the number of harmonics doubles the amount of time it takes to find the fitness value.

5.1.1 Comparison of CFM and DFM

Many experiments were run to compare DFM and CFM synthesis convergence properties. The results as seen in **Figure 5.1** show the fitness of over 200 generations for a cello match. One sees that DFM resulted in good initial matches but CFM improved very quickly, overtaking DFM synthesis after the 10^{th} generation. After 140 generations the CFM improvements begin to slow down and DFM starts closing on CFM. Although after 200 generations CFM had a slightly better fitness than DFM it does not necessarily reflect the overall quality of the resulting sound. In fact DFM tended to match the harmonics above 15 with far greater accuracy than CFM, but because these are of such small magnitude it didn't reflect in the fitness value. This meant that DFM matches tended to be more "bright" which is a good characteristic for certain instruments such as piano.

DFM would also be an advantage if a very rapid initial approximate match was wanted of an instrument. The great disadvantage with DFM, however, is that it takes about 4 times the amount of time to calculate the fitness value because of the complication in predicting the harmonics. Because of this time constraint all the resulting matches shown are done using CFM with 200 generations.

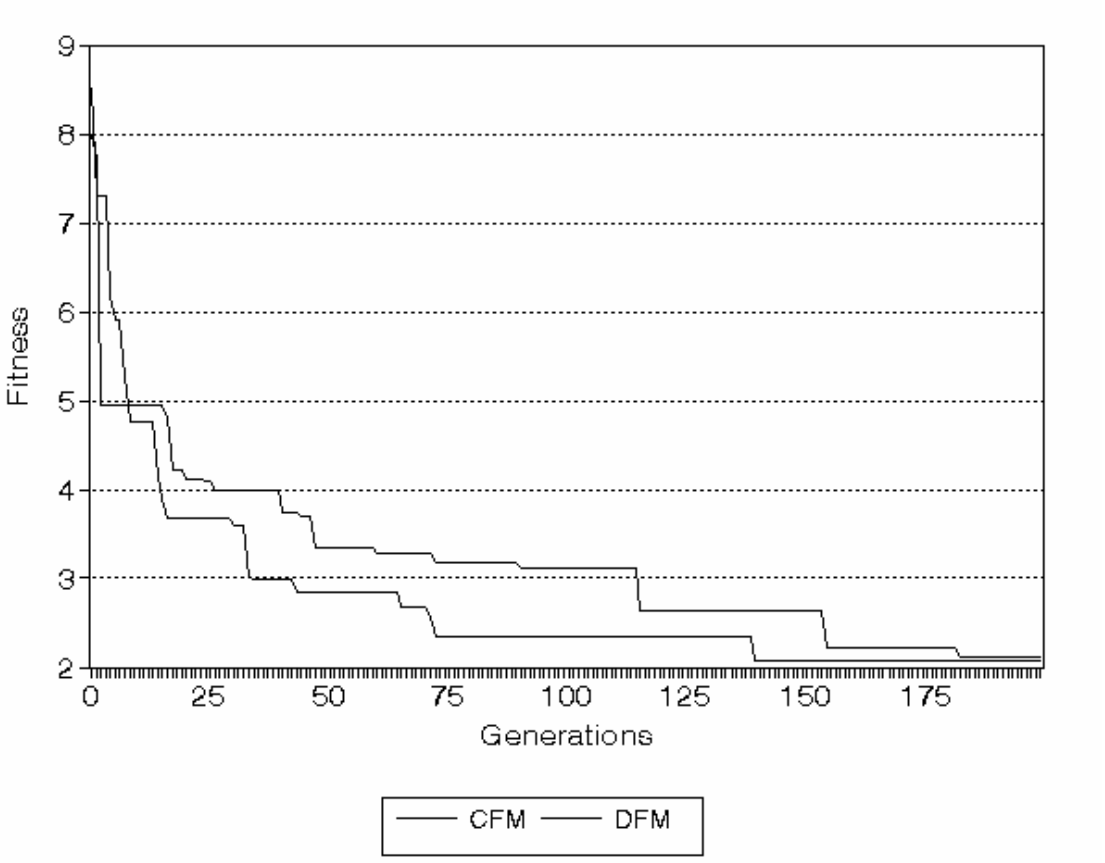

Figure 5.1 Fitness comparison of DFM vs. CFM over 200 generations

5.1.2 The instrument matches

The time-varying harmonics of 4 instrument matches are given for each instrument with its original harmonics and the model harmonics. These are **Figure 5.2** and **Figure 5.3** for the piano, **Figure 5.4** and **Figure 5.5** for the voice, **Figure 5.6** and **Figure 5.7** for the guitar and **Figure 5.8** and **Figure 5.9** for the cello. Only 6 out of the 30 harmonics matched are shown on the graph and there is a general trend for the matches to get worse as the harmonics get higher. The plots for each instrument will be discussed in turn in the subsequent text. On the cassette which accompanies this thesis Samples 1 to 4 contain these sound matches. The samples always start with the original sound followed by the sound match and then repeat this pattern once to allow careful comparison.

5.1.2.1 The Piano match (Sample 1 on tape)

The first 6 harmonics are shown. The fundamental and 2nd harmonics are almost perfectly matched. The 3rs harmonic in the model loses some of the initial peakiness of the original and the 6th harmonic is slightly over0exagerated. Listening to the result, one finds a slight lack of brightness on the attack which is really due to a lack of good high-order harmonic matches and too few time frames for the attack.

5.1.2.2 The Voice match (Sample 2 on tape)

The first 6 harmonics are shown. An initial recognisable difference between the original and model is the smoothing of the wavy amplitude variations. This waviness is essentially a tremolo which is present in most human voice. Listening to the result, one immediately recognises that the match has a classic computer voice synthesiser sound too it. It's a very difficult sound to match because it has such a harmonically full sound; nevertheless it was a convincing match.

5.1.2.3 The Guitar match (Sample 3 on tape)

The 1st, 2nd, 3rd, 8th, 9th and 10th harmonics are shown. The 4th to 7th harmonics were too

insignificant to display on the graph, so they were left out. Looking at the 8th harmonic, the parabolic rise and fall of a momentary excitation instrument is very clear. A lot of resolution was lost in the 3rd harmonic but it did excellently in matching the chaotic up and down motion of the fundamental and 2nd harmonic.

5.1.2.4 The Cello match (Sample 4 on tape)

The first 6 harmonics are shown. This I believe was the most convincing match out of the four even though its fitness rating was not as good as the rest. As was discussed earlier, the fitness rating is not the ultimate test of how close the match is. Again we see flattening of the peaks as was the case with the guitar. An interesting phenomenon can be seen after about the 70th time frame. At this point the bow of the cello left the string and it became a pure vibrating string. A vibrating string will produce a waveform that will be very close to a sine wave and this is seen in the diagram, only the fundamental becomes significant after about the 70th time frame. This effect was very well matched as can be heard in the sample.

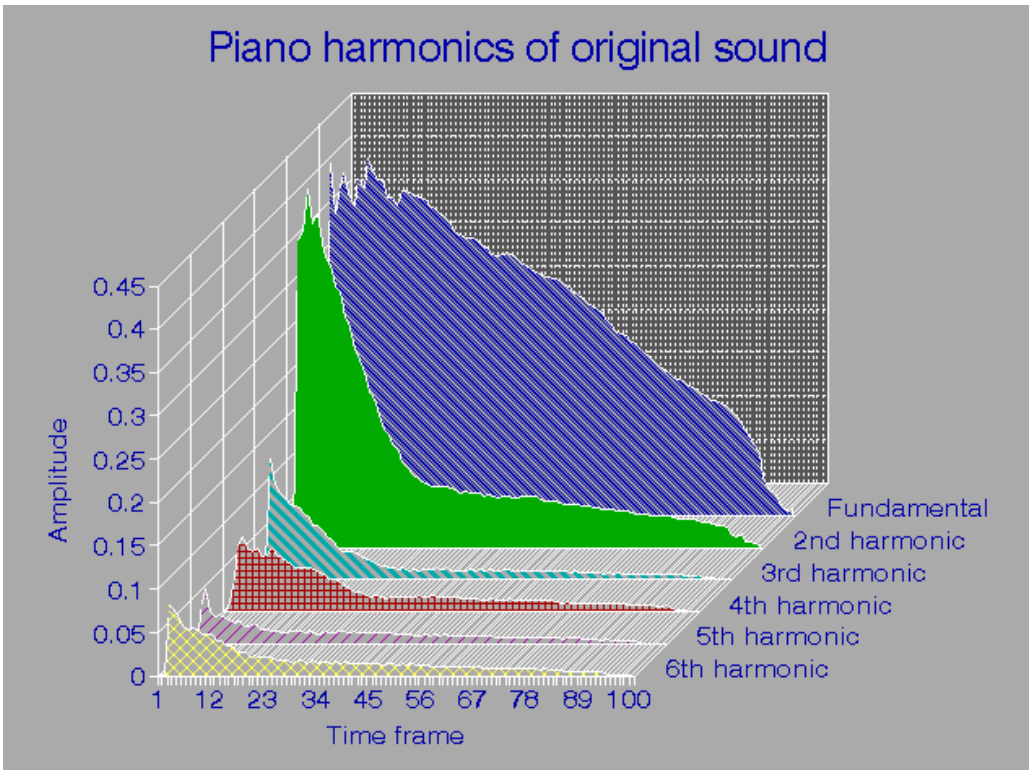

Figure 5.2 Original Piano Harmonics

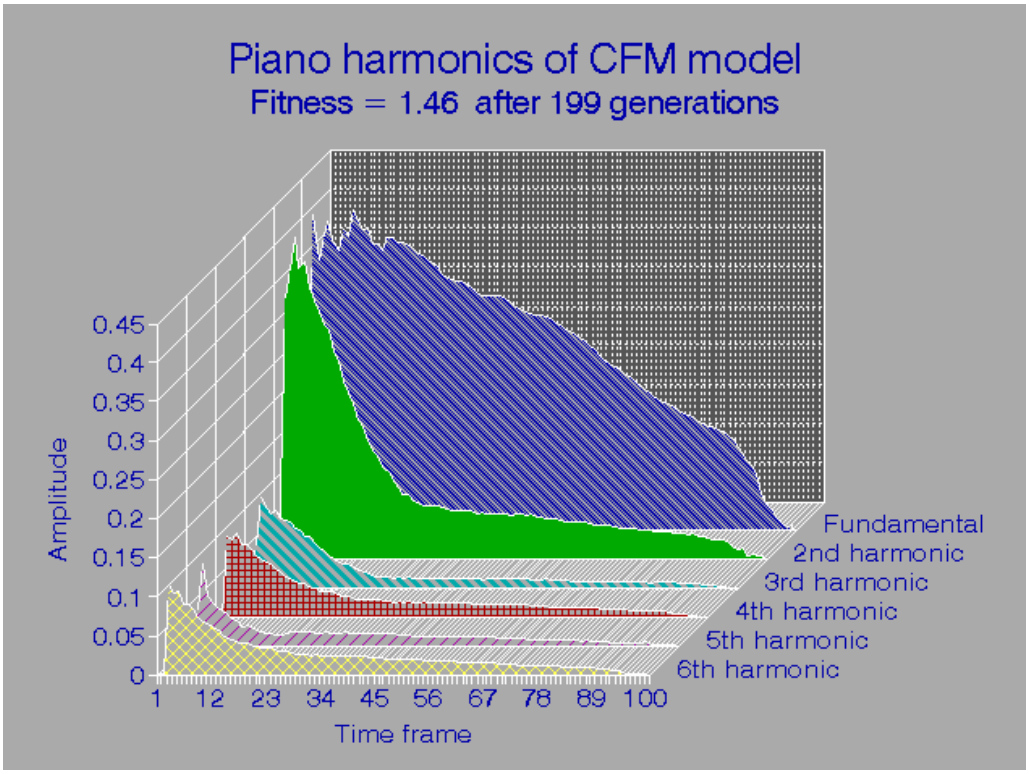

Figure 5.3 Model Piano Harmonics

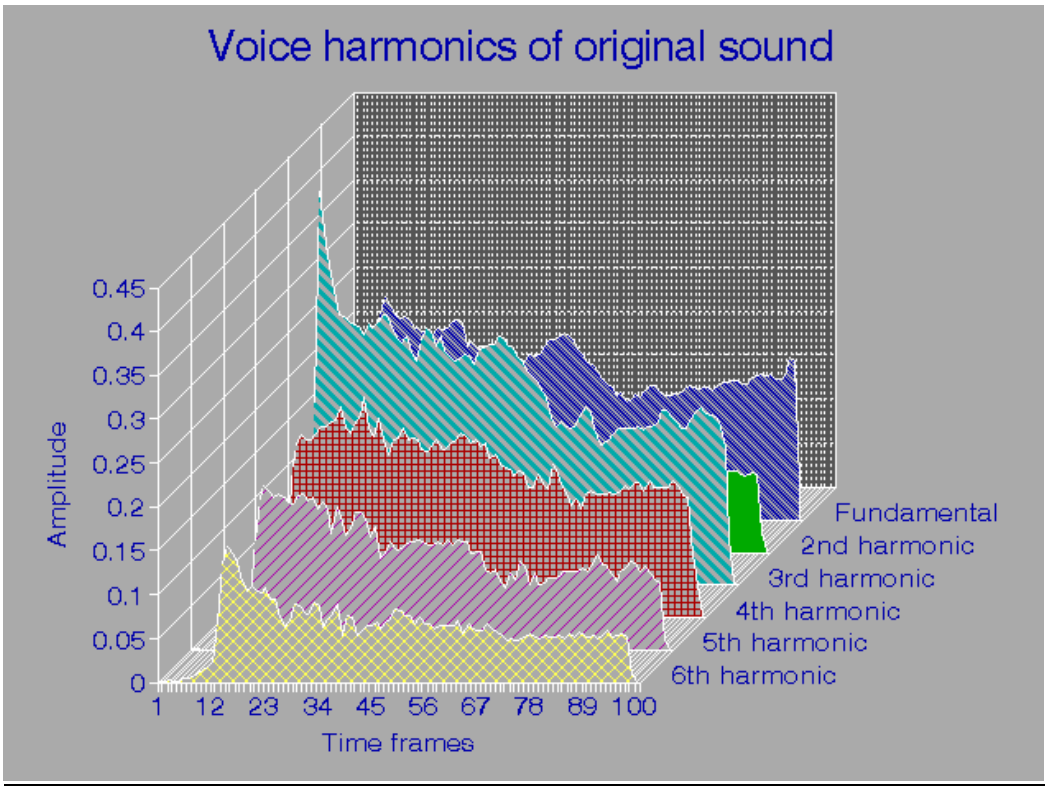

Figure 5.4 Original voice harmonics

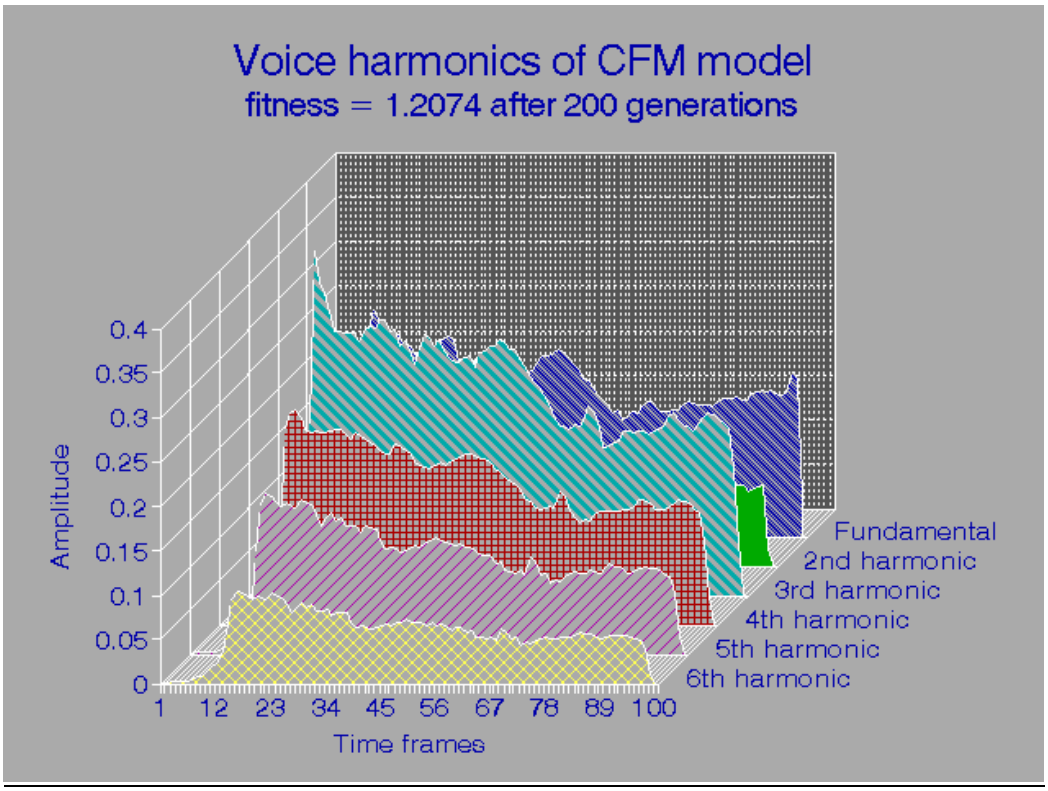

Figure 5.5 Model Voice harmonics

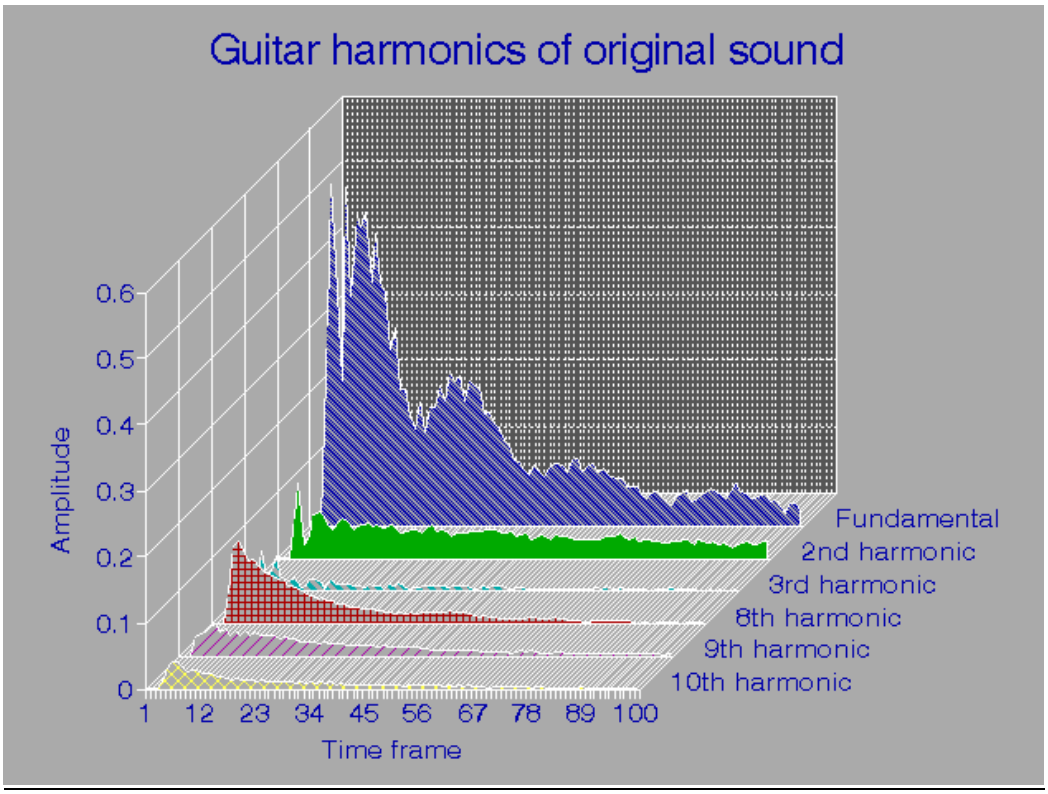

Figure 5.6 Original Guitar harmonics

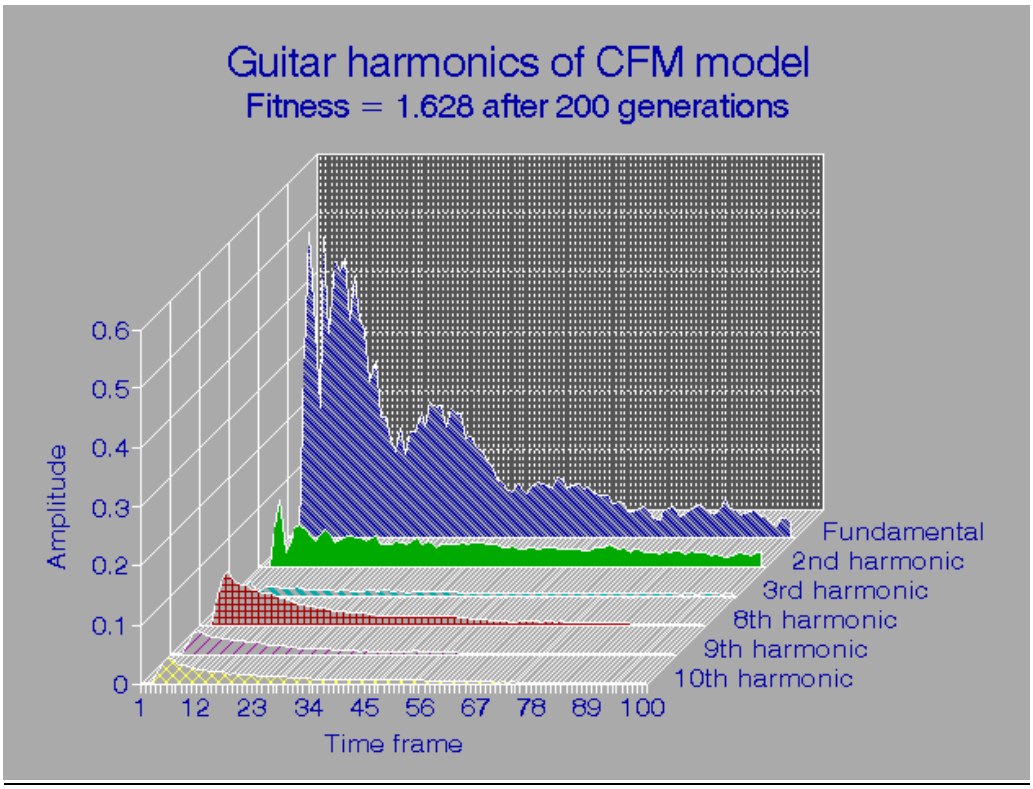

Figure 5.7 Model Guitar harmonics

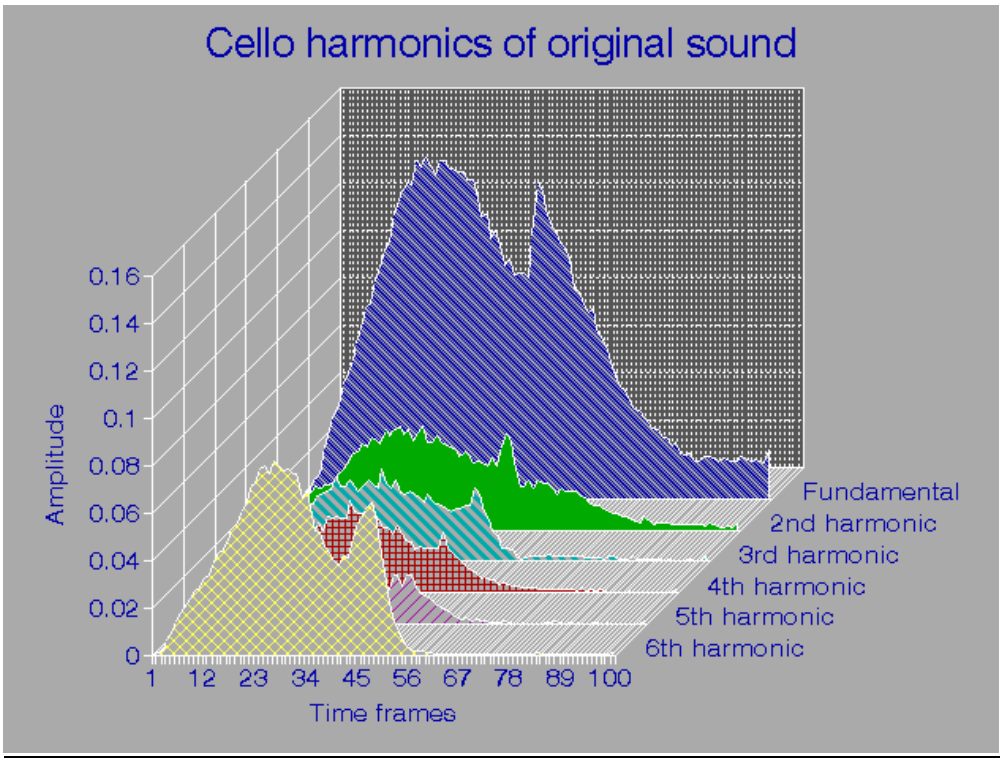

Figure 5.8 Original Cello harmonics

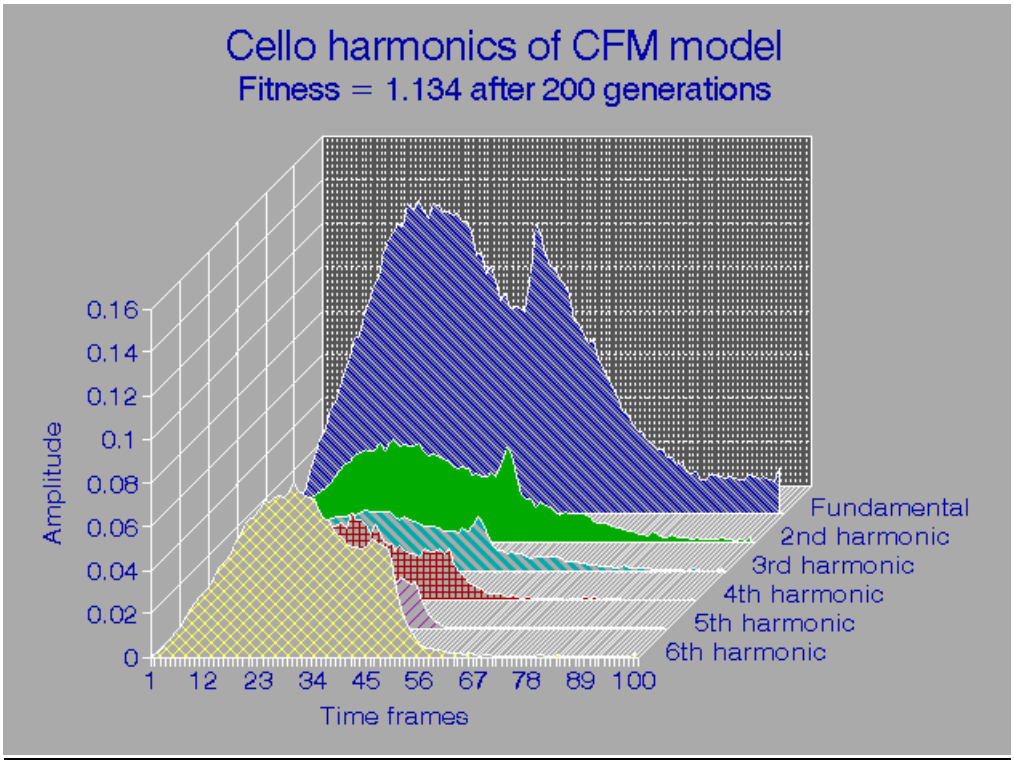

Figure 5.9 Model Cello harmonics

5.1.3 Model parameters for the piano

The complete results for a four-carrier match to the piano sound are shown in **Figure 5.10**. It shows a plot of each amplitude envelope, the carrier-to-modulator radios n_j and the modulation indices, I_j for this match. Carrier 4 has a modulation index of 0 which means that it will produce a sine wave at the fundamental frequency. Analysing the envelope of the carrier and comparing it with the fundamental in Figure5.3 one sees that they are basically identical. Carrier 1's high modulation index and high amplitude envelope is compensated for by carrier 2 and 3.

This example demonstrates the delicate balancing act that takes place to combine the carrier harmonics. The complexities of this process increase as more carriers are added which is why FM matching is such a difficult problem.

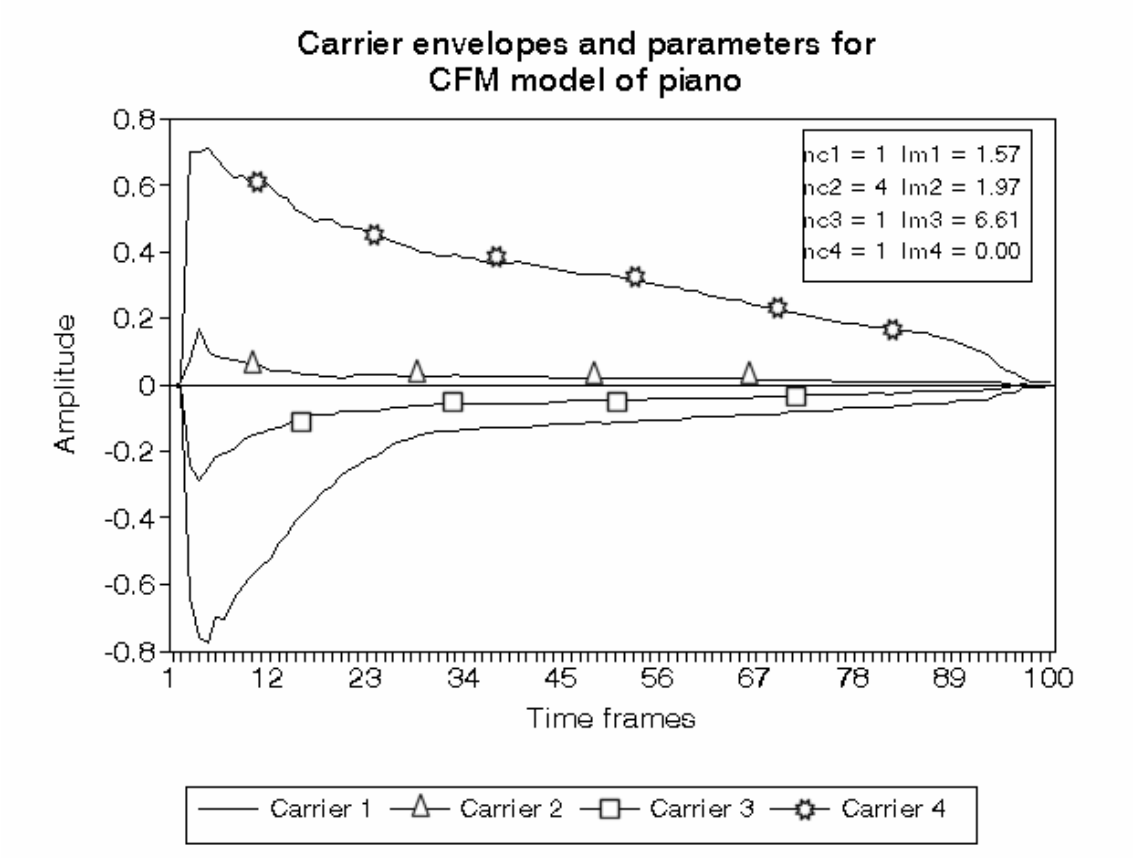

Figure 5.10 Parameters for the piano match

5.2 Creating new sounds

Describing results of new sounds discovered is very difficult to do in written form. To overcome this problem, some of the sounds produced have been placed on the cassette that comes with this thesis.

The majority of interesting sounds were created using DFM synthesis which makes sense when considering how DFM performed when matching real sounds. As discussed in the last section it was good at making initial matches, but CFM outdid it in the long run. With the limited small population and number of generations that can be spanned before ones patients runs out, DFM offers the most diverse set of small generation sounds. When CFM is used many hours will be needed to approach a particular type of sound you are searching for. This is because CFM relies more on subtle balances between each carrier's set of harmonics than DFM does.

5.2.1 Random sounds to a piano sound

To give the results more of an objective edge, a piano that was produced from random sounds is given below in **Figure 5.11** and should be compared with **Figure 5.2** in the last section. One sees that although the attack shape is oversimplified the harmonic proportions are very closely approximated. This piano is sample 5 on the tape and is played after to a real piano. The piano was created using the DFM synthesis model.

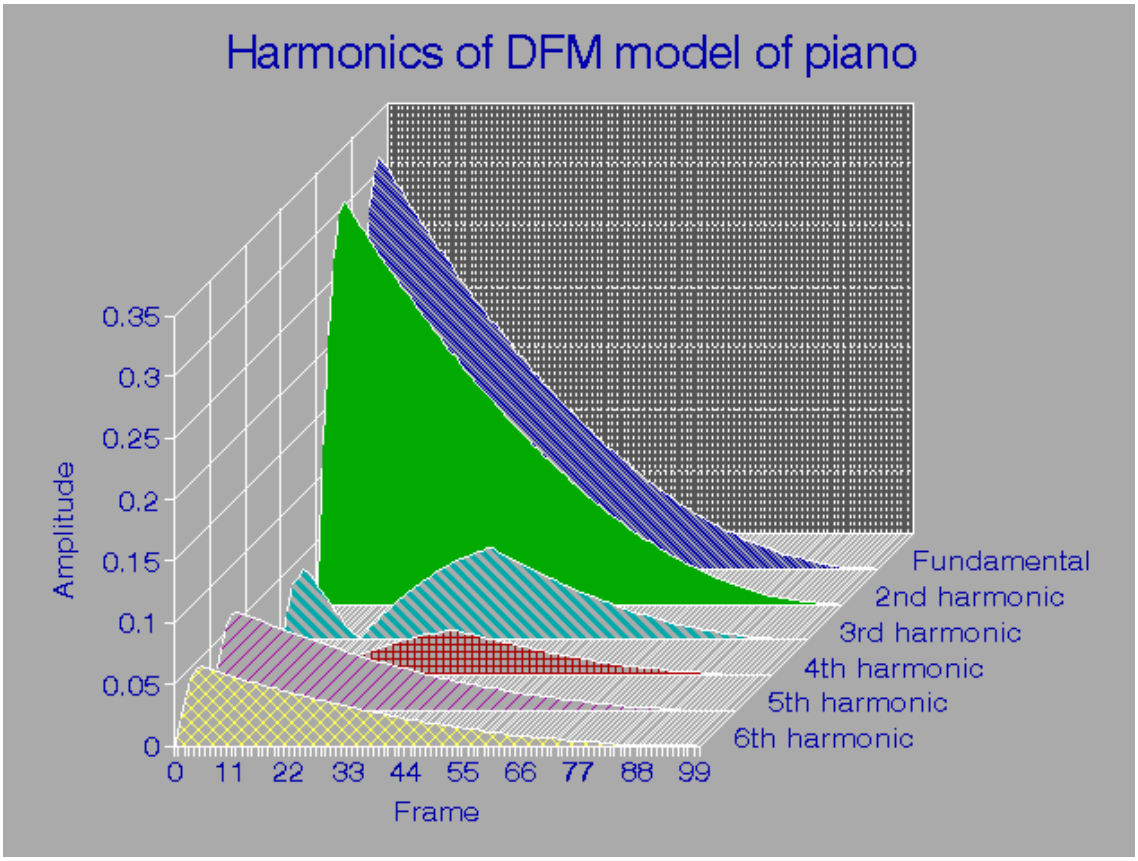

Figure 5.11 Piano produced from random sounds

5.2.2 Bell sounds

A great deal of interesting bell-like sounds were discovered. Bell sounds are known to be one of FM's strong points (Deutsch 1985) and these results confirmed this. The bells sounds can be heard as samples 6 to 7 on the cassette and end with a short piece of music written for each of them (samples 8 to 9). A spectral plot of bell sample 6 is given below in **Figure 5.12**. This bell was generated using DFM synthesis.

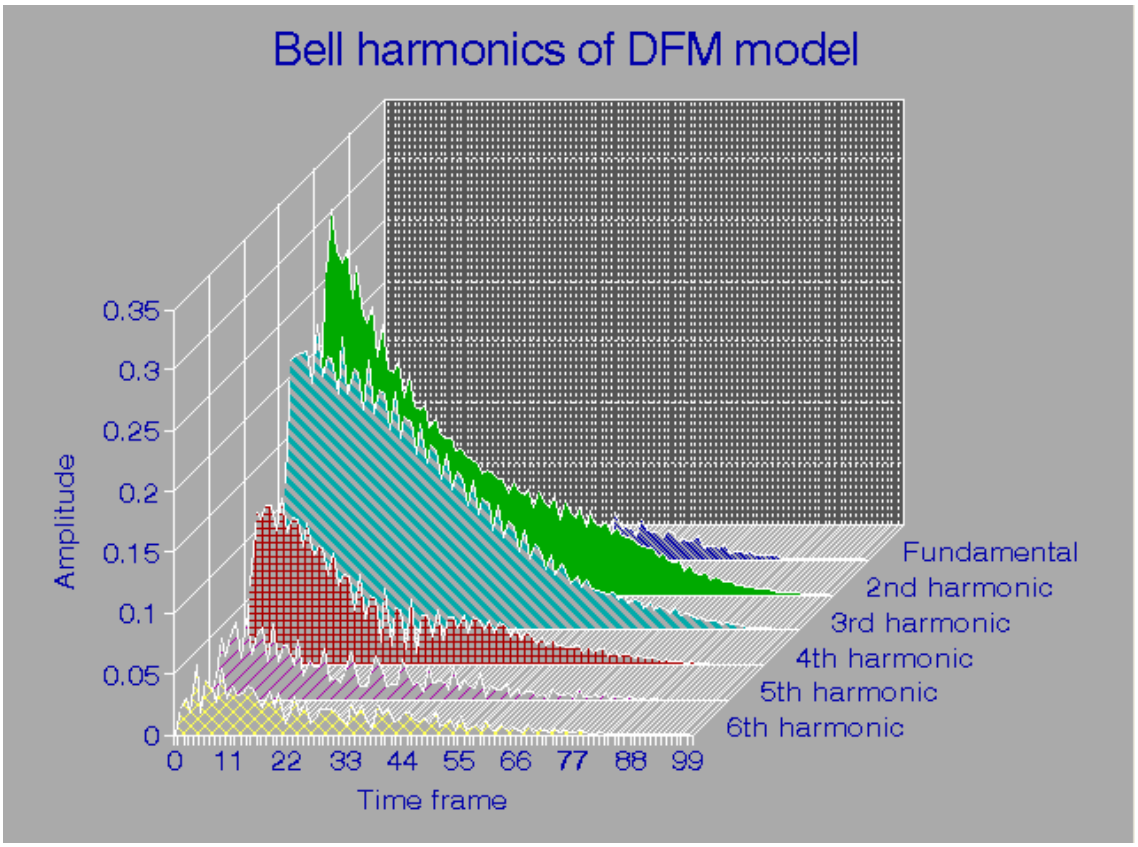

Figure 5.12 Bell harmonics

5.2.3 Rapid sound evolution

To show how the bell in **Figure 5.12** was generated a rapid evolution of the highest rated individuals from each generation are played on the tape. These are heard as Sample 10 and span 17 generations.

5.2.4 Other interesting sounds

An assortment of interesting sounds generated with the sound search is included as Sample 11 on the cassette.

The work station used for all the simulations was an IBM 386DX-33Mhz based machine. All the software was written in ANSI C and compiled with Borland's Turbo C ver 3.0. The wave files were stored in standard PCM WAV file format and played through a Sound Blaster SB16 16 bit 44kHz D/A. Real sounds were also sampled using the Sound Blasters 16 bit A/D with 44kHz sampling capability.

The real sound match and new sound search code was controlled by two parameter files, one to specify the Genetic algorithm parameters and another to specify the FM wave producing code's parameters. Examples of these are given in Appendix G with explanations of each parameter.

With real sound matches, the algorithm begins by reading in an FFT of the original sound. On completion it writes files with matching FM parameters, weights and evolving harmonics of the sound match. A separate program is then run which finally finds the best weights using a 100 frame FFT of the original sound and creates the sound match file. The matching procedure is relatively efficient with 200 generation, 4 carrier CFM matches taking about half an hour.

When doing a new sound search sound files are created on the run and played out through the sound blaster D/A for evaluation. When a low sampling rate such as 5kHz is used with 3 carriers one only has to wait about half a minute for a new population of 10 individuals to be formed.

Looking at the results, one sees that FM synthesis is very powerful when it works together with the genetic algorithm. The objectives of matching real sounds and searching for new sounds have certainly been met.

The following conclusions can be made about using the genetic algorithm to match real sounds.

1. Using up to four carriers for either DFM or CFM synthesis provides a large enough sound space to very closely approach all the sounds tested in this thesis.
2. Matching up to 30 harmonics in all the tested sounds is sufficient, although more could be used on instruments such as the guitar with high harmonic attacks.
3. Using 10 harmonic time frames of sound was sufficient to achieve good matches. These frames should be placed in the attach phase for momentary excitation instruments and over the whole instrument period for continuous excitation instruments.
4. The DFM model produced good initial matches of a sound but was overtaken by the CFM model after about 10 generations. It did however produce better matches for very high order harmonics.
5. Standard GA settings with crossover = 60%, Mutation rate = 0.1% and a population size of 50, produced very good rates of convergence with the fitness usually beginning to level off after about 100 generations.
6. The time taken to match a sound depended on the number of harmonics used, the number of frames used and the total number of carriers for the model. The amount of time taken increases in proportion to the increase in any one of these parameters.

The following conclusions can be made about using the genetic algorithm to search for new sounds:

1. Close matches to real world sounds could be made using an initial random population of sounds.
2. A population size of 10 was about the largest that could be used before inter-comparison

of sounds became too difficult.

3. Genetic encoding of the peak point of envelope weights for each carrier was a very effective way of creating evolving timbres and amplitudes of waveforms.
4. All variables available in each synthesis model should be encoded in the gene to create large sound space to search.
5. DFM synthesis tended to create the richest and varied sounds with good subjective matches to sounds one was searching for within about 5 generations.
6. Fitness values given to sounds gave the best results when they were between 1 and 20. Elitism resulted in a lack of genetic diversity in subsequent populations so even poor sounds could be given ratings as high as 5.
7. GA parameters with crossover rates close to 100% and mutation rates as high as 10% should be used to create as much diversity in subsequent generations.

7 EXTENSIONS

Now that FM synthesis has been shown to have a massive sound space which is pliable enough to represent both interesting new sounds and convincing real sounds, the next step would be to build a real-time synthesiser. This could be done using a fast dedicated DSP chip linked to high speed static ram which would be built onto a slot-in computer board. This plug in-board would become a completely flexible sound-canvas which could have a midi-in and midi-out for connection to a keyboard. The sound canvas would then be controlled by software with the genetic algorithm as its heartbeat. Newly made sounds could be stored on disk, as a set of envelopes and FM parameters for each carrier which could be loaded into the sound canvas and used.

Another important extension in the area of reproduction of real sounds is to allow the synthesiser to synthesize the complete set of tones belonging to a particular instrument over its ranges of pitch and dynamics. Ideas for this are finding basis FM parameters which are optimum over a large collection of spectra from the instrument and then finding a mapping function which transforms the weights to optimum values for various pitches and dynamics of the instrument.

DFM synthesis matching needs to be made more efficient and work needs to be done to see if an explicit solution for the harmonics is obtainable instead of using the inefficient technique in Appendix F.

A very interesting possibility exits for searching new sounds by using real sound matches as an initial population. When crossover is done in the GA, it is proposed that the complexed real-match envelopes for each gene are averaged to produce a new unique envelope.

REFERENCES

Beauchamp J. 1982. "Synthesis by Spectral Amplitude and Brightness Matching of Analysed Muiscal Instrument tones" *Journal of the Audio Engineering Society* 30(6): 396-406

Beauchamp J., A. Horner., and L. Haken 1993. "Machine Tongues XVI: Genetic Algorithms and their application to FM matching synthesis" *Computer Music Journal* 17(4): 17-29

Beauchamp J., A.Horner., and L.Haken 1993. "Methods for multiple wavetable synthesis of Musical Instrument Tones" *Journal of the Audio Engineering Society* 41(5): 336-356

Chowning J.M. 1973. "The synthesis of complex audio spectra by means of Frequency Modulation" *Journal of the Audio Engineering Society* 21(7): 526-534

De Furia, S. 1986. *The secrets of Analogue and Digital Synthesis*. Rutherford, New Jersey: Ferro productions

Deutsch, R. 1985. *Synthesis, and introductions to the History, Theory and Practice of Electronic Music*. Sherman Oaks, California: Alfred Publishing Co., Inc

Freedman M. 1966. "Analysis of Musical Instrument Tones" *Journal of the Acoustical society of America* 41(4): 93-806

Grefenstette J. 1990 *A User's Guide to GENESIS version 5.0.* (software documentation)

Grey. J., J.Moorer. 1977. "Perceptual evaluations of synthesized musical instrument tones" *Journal of the Acoustical society of America* 62(2): 454-463

Moog R. 1986. "Digital Music Synthesis" *BYTE* June 1986: 155-168

Morgan D. 1993 RUCKUS C library code for playing sound data on the Sound Blaster (Source Code)

Press, W., B. Flannery, S. Teukolsky, and W. Vetterling. 1989. *Numerical Recipes*. Cambridge, UK: Cambridge University Press

Tan B., S.Gan., S.Lim., S.Tang. 1994. "Real-time implementation of Double Frequency Modulation (DFM) Synthesis" *Journal of the Audio Engineering Society* 42(11): 918-926

APPENDIX A FFT CODE

```
/*
 * FFT.C
 *
 * Written by D.L. Johnson for undergraduate thesis
 * "FM synthesis and the genetic algorithm"
 *
 * purpose: To produce time varying harmonic plot of a WAV file
 *
 * modified:3 November 95
 */

/* Includes */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <alloc.h>

/* Defines */

#define TWO_PI  ((double)2.0 * M_PI)

typedef struct {
   unsigned char bitsize;
   unsigned int sample_rate;
   unsigned long tot_samples;
} WINFO;

/* Globals */

unsigned int samples;
int power;
double far *real, *imag;
double max;
FILE *fpi, *fpo;

/* Prototypes and forward declarations */

void fft(void);
int  permute(int);
double magnitude(int);
void create_out_file(int f, int hars);
void get_header(FILE *fp, WINFO *wi);

/* ============================================================ */
/* Main file has command                                        */
/* FFT <in> <out> <N> <frames> <period> <Harmonics>             */
/* ============================================================ */

void main(int argc, char *argv[]) {
   WINFO wi;                          // Wave file information
```

```
unsigned char buf8;               // Storage buffer for 8 bit info
int buf16;                        // Storage buffer for 16 bit info
unsigned int start;               //
unsigned int i;                   //
int frames;                       //
int frame_length;                 //
int frame_no;                     //
int freq_increment;               //
int harmonic;                     //
unsigned long pos;                // Current position in WAV file
unsigned int startpos;            // Starting position of WAV file data
float time_frac;                  // Fraction on WAV file to analyse
unsigned long samples_analysed;   //
int hars;                         // Number of harmonics to print
int ox,oy;                        // x and y co-ordinates on screen
```

```
  if (argc != 7)  {
err1:    fprintf(stderr, "\n\nUSAGE: FFT <in> <out> <N> <frames> <period>
<Harmonics>\n");
    fprintf(stderr, "<in> is a .WAV file with extension\n");
    fprintf(stderr, "<out> is a .prn with extension\n");
    fprintf(stderr, "<N> is a power of 2 number\n");
    fprintf(stderr, "<period> is the fraction of time to be analysed in WAV
file eg. 0.1 for 10%\n");
    fprintf(stderr,  "<Harmonics>  is  the  number  of  harmonics  to  be
printed\n");
    exit(1);
  }

/* Extract variables from command line */

  samples = abs(atoi(argv[3]));
  frames = abs(atoi(argv[4]));
  time_frac = fabs(atof(argv[5]));
  hars = abs(atoi(argv[6]));

  if (hars>samples) {
    printf("\n\nNumber of harmonics must be less than N\n\n");
    exit(1);
  }

/* Calculate power value for FFT */

  power =  log10((double)samples) / log10((double)2.0);

/* Allocate memory for real and imaginary data */

  if ((real = (double far *)farmalloc(samples * sizeof(double))) == NULL)
err2:    fprintf(stderr, "Out of memory\n");

  if ((imag = (double far *)farmalloc(samples * sizeof(double))) == NULL)
    goto err2;

/* Open WAV input file and data output file */

  if ((fpo = fopen(argv[2], "w")) == (FILE *)NULL)
    goto err1;

  if ((fpi = fopen(argv[1], "rb")) == (FILE *)NULL)
    goto err1;

/* Get header information from WAV file */

  get_header(fpi, &wi);

/* Do preliminary calculations */

  samples_analysed = time_frac*wi.tot_samples;
  frame_length = samples_analysed/frames;
  freq_increment = wi.sample_rate/samples;

  printf("\n\nFFT ANALYSIS is being done\n\n");
```

```
  printf("total samples = %ld\n",wi.tot_samples);
  printf("total frames = %d\n",frames);
  printf("Frame length = %d\n",frame_length);
  printf("Freqency increment = %d\n",freq_increment);
  printf("Samples used in each frame = %d\n\n",samples);

/* Write first line of output file which is harmonic bins */

  fprintf(fpo,"%d",0);
  for (harmonic=1; harmonic<=hars; harmonic++)
    fprintf(fpo,",%d",harmonic);
  fprintf(fpo,"\n");

/* Find current file position and begin alaysis frame by frame */

  startpos = ftell(fpi);
  for (frame_no=0; frame_no<frames; frame_no++) {
    if (frame_no == 0) {
      ox = wherex();
      oy = wherey();
    }
    /* Get chunk of data samples long */
    if (wi.bitsize==8) {
      pos = startpos+(long)frame_no*frame_length;
      gotoxy(ox,oy); printf("Analysing pos-> %ld in file",pos);
      fseek(fpi, pos, SEEK_SET);
      for(i=0; i<samples; i++) {
        fread(&buf8, sizeof(char), 1, fpi);
        real[i] = (double)buf8*2/255-1;
        imag[i] = 0;
      }
    }
    else {
      pos = startpos+2*(long)frame_no*frame_length;
      gotoxy(ox,oy); printf("Analysing pos-> %ld in file",pos);
      fseek(fpi, pos, SEEK_SET);
      for(i=0; i<samples; i++) {
        fread(&buf16, sizeof(int), 1, fpi);
        real[i] = (double)buf16/32768;
        imag[i] = 0;
      }
    }

/* Perform FFT on data chunk: results returned in *real and *imag */

    fft();

/* Write frame of spectral information to OUT file */

    create_out_file(frame_no,hars);
  }
  printf("\n\n");

/* Clean up */

  fclose(fpo);
  fclose(fpi);
```

```
  farfree(real);
  farfree(imag);
}

/* ============================================================ */
/* Extracts all information in header of WAV file               */
/* ============================================================ */

void get_header(FILE *fp, WINFO *wi) {

  struct riff_header {
    char riff_code[4];
    unsigned long file_size;
    char wav_code[4];
  } rh;

  struct format_chunk {
    char fmt_ckid[4];
    unsigned long fmt_cksize;
    unsigned short fmttag;             // 1=PCM
    unsigned short channels;           // 1=mono, 2=stereo
    unsigned long samples_sec;
    unsigned long average_bytes_sec;
    unsigned short block_align;
    unsigned short bits;               // 8 -> 16 bit
  } fc;

  struct data_chunk_header {
    char data_ckid[4];
    unsigned long data_cksize;
  } dch;

  fread (&rh,sizeof(rh),1,fp);
  fread (&fc,sizeof(fc),1,fp);
  fread (&dch,sizeof(dch),1,fp);

  wi->bitsize = fc.fmt_cksize;
  wi->sample_rate = fc.samples_sec;
  if (wi->bitsize == 8)
    wi->tot_samples = dch.data_cksize;
  else
    wi->tot_samples = dch.data_cksize/2;
}

/* ============================================================ */
/* The heart of the program -> The radix-2 FFT algorithm        */
/* Time domain data is passed to it in *real and *imag          */
/* and frequency domain data is returned in *real and *imag     */
/* ============================================================ */

void fft()
{
  unsigned i1, i2, i3, i4, y;
  int   loop, loop1, loop2;
  double  a1, a2, b1, b2, z1, z2, v;
```

```
  /* Scale the data */

  for (loop = 0; loop < samples; loop++)  {
    real[loop] /= (double)samples;
    imag[loop] /= (double)samples;
  }

  i1 = samples >> 1;
  i2 = 1;
  v = TWO_PI * ((double)1.0 / (double)samples);

  for (loop = 0; loop < power; loop++)  {
    i3 = 0;
    i4 = i1;

    for (loop1 = 0; loop1 < i2; loop1++)  {
      y = permute(i3 / i1);
      z1 =  cos(v * y);
      z2 = -sin(v * y);

      for (loop2 = i3; loop2 < i4; loop2++)  {
        a1 = real[loop2];
        a2 = imag[loop2];

        b1 = z1*real[loop2+i1] - z2*imag[loop2+i1];
        b2 = z2*real[loop2+i1] + z1*imag[loop2+i1];

        real[loop2]      = a1 + b1;
        imag[loop2]      = a2 + b2;

        real[loop2 + i1] = a1 - b1;
        imag[loop2 + i1] = a2 - b2;
      }

      i3 += (i1 << 1);
      i4 += (i1 << 1);
    }

    i1 >>= 1;
    i2 <<= 1;
  }
}

/* ============================================================ */
/* Write a frame of spectral magnitude information to the       */
/* OUT file. Only the number of harmonics specified by hars     */
/* will be written to the file                                  */
/* ============================================================ */

void create_out_file(int f, int hars) {

     int  loop;
     double x;

     fprintf(fpo,"%d",f);

     for (loop = 1; loop <= hars; loop++)  {
```

```
        x = magnitude(loop)*2;
        fprintf(fpo,",%lf",x);
      }
      fprintf(fpo,"\n");
}

/* ========================================================== */
/* Calculate Power Magnitude                                  */
/* ========================================================== */

double magnitude(n)
int  n;
{
   n = permute(n);
   return (sqrt(real[n] * real[n] + imag[n] * imag[n]));
}

/* ========================================================== */
/* Bit reverse the number                                     */
/* Change 11100000b to 00000111b or vice-versa                */
/* ========================================================== */

int permute(index)
int  index;
{
   int  n1, result, loop;

   n1 = samples;
   result = 0;

   for (loop = 0; loop < power; loop++)  {
      n1 >>= 1;      /* n1 / 2.0 */
      if (index < n1)
         continue;

      result += (int) pow((double)2.0, (double)loop);
      index -= n1;
   }

   return result;
}
```

APPENDIX B FM-WAVE CODE

```
/*  FM SYNTHESIS CODE
 *  Written by David L. Johnson 1995 for undergraduate THESIS
 *  "FM synthesis and the genetic algorithm"
 *
 *  file: fmmake.c
 *
 *  purpose: Creates .WAV file based on information in waveinfo
 *           structure -> WINFO. Can create Chowning FM waves or
 *           Double FM waves and will use either rise-fall parabolic
 *           envelopes or flat-top envelopes.
 *
 *  calling function : create_wav_file(char *filname, int sw[], WINFO wi)
 *
 *  modified:      18 oct 95
 *
 */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <alloc.h>
#include "fmmake.h"

/*=========================================================================*/
/* Header of wavefile is created here, it contains all the information    */
/* on the wave file such as sampling rate, file_size, bit resolution etc. */
/* ======================================================================= */

void create_header(FILE *fp, WINFO wi) {

   struct riff_header {
     char riff_code[4];
     unsigned long file_size;
     char wav_code[4];
   } rh;

   struct format_chunk {
     char fmt_ckid[4];
     unsigned long fmt_cksize;
     unsigned short fmttag;       // 1=PCM
     unsigned short channels;   // 1=mono, 2=stereo
     unsigned long samples_sec;
     unsigned long average_bytes_sec;
     unsigned short block_align;
     unsigned short bits;       // 8 -> 16 bit
   } fc;

   struct data_chunk_header {
     char data_ckid[4];
     unsigned long data_cksize;
   } dch;
```

```
  strcpy(rh.riff_code,"RIFF");
  if (BIT_RESOLUTION == 16)
    rh.file_size = wi.tot_samples*2+44;
  else
    rh.file_size = wi.tot_samples+44;
  strcpy(rh.wav_code,"WAVE");

  strcpy(fc.fmt_ckid,"fmt ");
  fc.fmt_cksize = 16;
  fc.fmttag = 1; // PCM format
  fc.channels = 1; // Mono
  fc.samples_sec = wi.sample_rate;
  if (BIT_RESOLUTION == 16)
    fc.average_bytes_sec = wi.sample_rate*2;
  else
    fc.average_bytes_sec = wi.sample_rate;
  if (BIT_RESOLUTION == 16)
    fc.block_align = 2;
  else
    fc.block_align = 1;
  fc.bits = BIT_RESOLUTION;

  strcpy(dch.data_ckid,"data");
  if (BIT_RESOLUTION == 16)
    dch.data_cksize = wi.tot_samples*2;
  else
    dch.data_cksize = wi.tot_samples;


  fwrite (&rh,sizeof(rh),1,fp);
  fwrite (&fc,sizeof(fc),1,fp);
  fwrite (&dch,sizeof(dch),1,fp);
}

/* ========================================================================= */
/* Envelopes for each carrier are given here. the rise and fall of        */
/* both envelopes are parabolic                                           */
/* ========================================================================= */

/*
/* env_parabola has the following shape

                *
          *     |*
        *       |  *
       *        |     *
     *          |          *
     *          |                *
     0----------p----------------1----> time as a fraction of sample time

/* env_flat looks as follows


       * * * * * * * * * * *
    *  |                   |*
   *   |                   | *
  *    |                   |   *
```

```
  *    |                    |       *
  0----p------------------0.8------1-----> time as a fraction of sample time

 */


float env_parabola(unsigned int sample, float p, unsigned int ts) {
   float point;
   point = p*ts;

   if (sample < (point))
     return (-pow((sample/point),2) + (2/point)*sample);
   return( (float)(1/pow(((float)point-ts),2)) * \
          pow(((float)sample-ts),2) );
}

float env_flat(unsigned int sample, float p, unsigned int ts) {
   float point1, point2;
   point1 = p*ts;
   point2 = 0.8*ts;

   if (sample < point1)
     return (-pow((sample/point1),2) + (2/point1)*sample);
   if (sample < point2)
     return (1);
   return (  (1/pow((point2-ts),2)) * pow((ts-sample),2) );
}
```

```
/* ====================================================================== */
/* Wave code to produce chowning fm waves with the following mathematical */
/* structure :  SIN( 2π*fc*t + Im*SIN(2π*fm*t) )                          */
/*                                                                        */
/* See FMMAKE.H  to see structure of WINFO                                */
/* ====================================================================== */

void chowning_fm(WINFO wi, int sw[], unsigned char buf8[]) {
  unsigned int q2[NUM_CARS];    // Index of sine wave array
  float at q3, q4;              // Intermediate variables
  unsigned int sample;          //
  register int i;               //
  int tot_amp;                  // Total amplitude of all carriers
  unsigned char nb[NUM_CARS];   // Freq Multipliers
  int sine_points;              // Total number of points in sine-table
  float t;                      // Time

  sine_points = wi.sample_rate/wi.base_freq;

  /* Preliminary Fast FM calculations */

  /* Calculate total amplitude of each carrier */
  tot_amp = 0;
  for (i=0; i< wi.carriers; i++)
    tot_amp += abs(wi.a[i]);

  /* Calculate frequency multipliers */
  for (i=0; i<wi.carriers; i++) {
    nb[i] = wi.fb[i]/wi.base_freq;
    q2[i] = 0;
  }

  if (wi.env == flat_top) {
    for (sample=0; sample<wi.tot_samples; sample++) {
      q4 = 0; // Accumulative value for carrier results
      for (i=0; i<wi.carriers; i++) {
        t = (float)sample/wi.sample_rate;
        q3 = 2*M_PI*wi.fa[i]*t + env_flat(sample,wi.envp[i],\
             wi.tot_samples)*wi.Ib[i]*sw[q2[i]]/10000;
        q4 += wi.a[i]*env_flat(sample,wi.envp[i],\
              wi.tot_samples)*sin(q3);
        q2[i] += nb[i];
        if (q2[i]>=sine_points)
          q2[i] = q2[i]%sine_points;
      }
      q4 = 0.5*q4/tot_amp;
      buf8[sample] = ((float)255/2)*(q4+1);
      if ( (sample % 1000) == 0) printf(".");
    }
  }
  else {
    for (sample=0; sample<wi.tot_samples; sample++) {
      q4 = 0;
      for (i=0; i<wi.carriers; i++) {
        t = (float)sample/wi.sample_rate;
        q3 = 2*M_PI*wi.fa[i]*t + wi.Ib[i]*env_parabola(sample,\
```

```
            wi.envp[i],wi.tot_samples)*sw[q2[i]]/10000;
        q4 += wi.a[i]*env_parabola(sample,wi.envp[i],wi.tot_samples)*\
             sin(q3);
        q2[i] += nb[i];
        if (q2[i]>=sine_points)
          q2[i] = q2[i]%sine_points;
      }
      q4 = 0.5*q4/tot_amp;
      buf8[sample] = ((float)255/2)*(q4+1);
      if ( (sample % 1000) == 0) printf(".");
    }
}
```

```
}

/* ========================================================================= */
/* Wave code to produce double fm waves with the following mathematical   */
/* structure :  SIN( Ia*SIN(2π*fa*t) + Ib*SIN(2π*fb*t) )                   */
/*                                                                         */
/* See FMMAKE.H  to see structure of WINFO                                 */
/* ========================================================================= */


void double_fm(WINFO wi, int sw[], unsigned char buf8[]) {
  unsigned int q1[NUM_CARS], q2[NUM_CARS];  // Sine-wave index values
  float q3, q4;                             // Intermediate values
  unsigned int sample;                      //
  register int i;                           //
  int tot_amp;                              // Total amplitude of carriers
  unsigned char na[NUM_CARS], nb[NUM_CARS];  // Freq Multipliers
  int sine_points;                          // Number of sine-table points

  sine_points = wi.sample_rate/wi.base_freq;

  /* Preliminary Fast FM calculations */
  tot_amp = 0;
  for (i=0; i<wi.carriers; i++)
    tot_amp += abs(wi.a[i]);

  /* Calculate frequency multipliers */
  for (i=0; i<wi.carriers; i++) {
    na[i] = wi.fa[i]/wi.base_freq;
    nb[i] = wi.fb[i]/wi.base_freq;
    q1[i] = 0;
    q2[i] = 0;
  }

  if (wi.env == flat_top) {
    for (sample=0; sample<wi.tot_samples; sample++) {
      q4 = 0;
      for (i=0; i<wi.carriers; i++) {
        q3 = (wi.Ia[i]*sw[q1[i]]+wi.Ib[i]*sw[q2[i]])/10000;
        q4 += wi.a[i]*env_flat(sample,wi.envp[i],wi.tot_samples)*\
              sin(q3);
        q1[i] += na[i];
        if (q1[i]>=sine_points)
          q1[i] = q1[i]%sine_points;
        q2[i] += nb[i];
        if (q2[i]>=sine_points)
          q2[i] = q2[i]%sine_points;
      }
      q4 = 0.5*q4/tot_amp;
      buf8[sample] = ((float)255/2)*(q4+1);
      if ( (sample % 1000) == 0) printf(".");
    }
  }
  else {
    for (sample=0; sample<wi.tot_samples; sample++) {
      q4 = 0;
```

```
        for (i=0; i<wi.carriers; i++) {
          q3 = (wi.Ia[i]*sw[q1[i]]+wi.Ib[i]*sw[q2[i]])/10000;
          q4 += wi.a[i]*env_parabola(sample,wi.envp[i],wi.tot_samples)*\
                sin(q3);
          q1[i] += na[i];
          if (q1[i]>=sine_points)
            q1[i] = q1[i]%sine_points;
          q2[i] += nb[i];
          if (q2[i]>=sine_points)
            q2[i] = q2[i]%sine_points;
        }
        q4 = 0.5*q4/tot_amp;
        buf8[sample] = ((float)255/2)*(q4+1);
        if ( (sample % 1000) == 0) printf(".");
      }
    }

}

/* ============================================================*/
/* Main Function to create WAV file with filename *filename  */
/* sw[] contains the sine-wave lookup table                   */
/* wi contains all the information about the wave to be made */
/*                                                            */
/* See FMMAKE.H  to see structure of WINFO                    */
/* ============================================================*/

void create_wav_file(char *filename, int sw[], WINFO wi) {
  register int i;                       //
  char wavfile[13], inffile[13];        // WAV file and info file
  unsigned char *buf8;                  // 8-bit buffer storage area
  FILE *fp1, *fp2;                      // File pointers

  /* Allocate memory for 8-bit buffer */

  buf8 = (unsigned char *) malloc(wi.tot_samples);
  if (!buf8) {
    printf("Not enough memory to allocate buffer\n");
    exit(1);  /* terminate program if out of memory */
  }

  /* Add filename extensions onto filename */
  strcpy(wavfile,filename);
  strcat(wavfile,".wav");
  strcpy(inffile,filename);
  strcat(inffile,".inf");

  fp1 = fopen(wavfile,"wb");
  if (fp1 == NULL) {
    fprintf(stderr,"Error opening file %s \n", wavfile);
    exit(1);
  }

  fp2 = fopen(inffile,"wb");
  if (fp2 == NULL) {
    fprintf(stderr,"Error opening file %s \n", inffile);
    exit(1);
```

```
  }

  /* Create header chunk for WAV file */

  create_header(fp1, wi);

  /* Create information file (fname.inf) */
  switch (wi.fm) {
    case cfm : {
      fprintf(fp2,"Chowning FM\n");

      for (i=0; i < wi.carriers; i++) {
        fprintf(fp2,"Carrier %d :\n",i);
        fprintf(fp2,"env = %d, a=%d, envp = %f, fc=%d, fm=%d, Im=%f
\n",wi.env, wi.a[i],wi.envp[i],wi.fa[i],wi.fb[i],wi.Ib[i]);
      }
    }
    break;
    case dfm : {
      fprintf(fp2,"Double FM\n");

      for (i=0; i < wi.carriers; i++) {
        fprintf(fp2,"Carrier %d :\n",i);
        fprintf(fp2,"env = %d, a=%d, envp = %f, fa=%d, fb=%d, Ia=%f, Ib=%f
\n",wi.env,wi.a[i],wi.envp[i],wi.fa[i],wi.fb[i],wi.Ia[i],wi.Ib[i]);
      }
    }
    break;
    default : printf("Unknown FM!!!");
    exit(1);
  }
  fclose(fp2);

  /* Create .WAV file data */
  move(22,0);
  printf("Creating wave file -> %s", wavfile);

  switch (wi.fm) {
    case cfm : chowning_fm (wi, sw, buf8); break;
    case dfm : double_fm (wi, sw, buf8); break;
    default : printf("Unknown FM!!!");
    exit(1);
  }

  /* Write .WAV file data to file filename.WAV */
  fwrite(buf8,sizeof(char),wi.tot_samples,fp1);

  printf("\n");

  free(buf8);

  fclose(fp1);
}

/* ============================================================*/
/*  Test code to see if FMMAKE.C works                         */
/* ============================================================*/
```

```
void main() {

   int a[3] = {50,20,15};
   float envp[3] = {(float)10/32, (float)5/32, (float)1/32};
   int fa[3] = {220, 660, 1320};
   int fb[3] = {440, 880, 1760};
   float Ia[3] = {1.0, 1.0, 0.68};
   float Ib[3] = {1.2, -0.35, 0.9};
   int i;
   int *sw;
   float res,degree;
   int sine_points;
   WINFO wi;

   wi.sample_rate = 10000;
   wi.sample_time = 2;
   wi.tot_samples = wi.sample_rate*wi.sample_time;
   wi.base_freq = 220;
   wi.fm = dfm;
   wi.carriers = 3;

   for (i=0; i<wi.carriers; i++) {
     wi.a[i] = a[i];
     wi.envp[i] = envp[i];
     wi.fa[i] = fa[i];
     wi.fb[i] = fb[i];
     wi.Ia[i] = Ia[i];
     wi.Ib[i] = Ib[i];
   }

   /* Calculate number of points needed in sine-wave table */
   sine_points = wi.sample_rate/wi.base_freq;

   /* Create sine-wave table */
   sw = (int *) malloc(sine_points*sizeof(int));
   if (!sw) {
     printf("Not enough memory to allocate buffer\n");
     exit(1);
   }

   for (i=0; i<sine_points; i++) {
     degree = 2*M_PI*((float)i/sine_points);
     res = sin(degree);
     sw[i] = 10000*(res);
   }

   /* Create a WAV file called organ1 */
   create_wav_file("organ1",sw,wi);

   free(sw);
}

/* End of fmmake.c */
```

```
/*
 * fmmake.h
 * definitions for fmmake.c */

#define NUM_CARS 5
#define NUM_HARS 30
#define NUM_FRAMES 10

#define NCARS NUM_CARS+1
#define NHARS NUM_HARS+1
#define NFRAMES NUM_FRAMES+1

#define BIT_RESOLUTION 8


enum {cfm, dfm};
enum {flat_top, parabola};

typedef struct {
   unsigned int sample_rate;
   float sample_time;
   unsigned int tot_samples;
   unsigned int base_freq;
   int fm;
   int env;
   unsigned int carriers;
   int a[NCARS];
   float envp[NCARS];
   int fa[NCARS];
   int fb[NCARS];
   float Ia[NCARS];
   float Ib[NCARS];
} WINFO;
```

```
/*
/*  FITNESS FUNCTION CODE FOR MATCHING REAL SOUNDS
 *  Written by David. L. Johnson 1995 for undergraduate thesis
 *  MUSIC SYNTHESIS USING THE GENETIC ALGORITHM
 *
 *  file:  compeval.c
 *
 *  purpose: To calculate the fittness of and individual in a population
 *           To do a least mean squares solution to AW = B and find W
 *
 *  modified: 1 November 95
 *
 */

#include "extern.h"
#include "fmmake.h"
#include "bitinfo.h"
#include <math.h>

/* external functions */

void svdcmp(float a[][NCARS], int m, int n, float w[], float v[][NCARS]);
float bessjn(int n, float x);

/* ================================================= */
/* Calculate new matrix D using sign matrix S with    */
/* Sign values contained in the diagonal              */
/* D = S*B                                            */
/* ================================================= */

void Multiply_sign_matrix(float B[][NFRAMES],float S[],float D[][NFRAMES]) {
   int i,j;

   for (i=1; i<=NUM_HARS; i++)
      for (j=1; j<=NUM_FRAMES; j++)
         D[i][j] = S[i]*B[i][j];
}

/* ================================================= */
/* Find harmonics of each carrier using information   */
/* Stored in WINFO structure                          */
/* ================================================= */

void Get_carrier_harmonics(float A[][NCARS], WINFO wi) {

   int i,j,k;
   int n[NCARS] = {0};
   int na[NCARS], nb[NCARS];
   int index;

   switch (wi.fm) {
```

```
     case cfm : {
        for (i=1; i<=wi.carriers; i++)
          n[i] = wi.fa[i]/wi.fb[i];

        for (i=1; i<=NUM_HARS; i++)
          for (j=1; j<=wi.carriers; j++)
            A[i][j] = bessjn((i-n[j]),wi.Ib[j]) - \
                      bessjn(-(i+n[j]),wi.Ib[j]);
     } break;
     case dfm : {
        for (i=1; i<=wi.carriers; i++) {
          na[i] = wi.fa[i]/wi.base_freq;
          nb[i] = wi.fb[i]/wi.base_freq;
        }
        for (k=1; k<=wi.carriers; k++)
          for (i=-10; i<=10; i++)
            for (j=-10; j<=10; j++) {
              index = i*na[k]+j*nb[k];
              if ((abs(index)>=1) && (abs(index)<=NUM_HARS))
                if (index>0)
                  A[index][k] += bessjn(i,wi.Ia[k])*bessjn(j,wi.Ib[k]);
                else
                  A[abs(index)][k]                                    -=
bessjn(i,wi.Ia[k])*bessjn(j,wi.Ib[k]);
            }
     } break;
     default : {
        printf("Unknown fm");
        exit(1);
     }
   }
}

/* ================================================= */
/* Find carrier weights by solving AW=B in the       */
/* least-mean squares sense. Used Single Value       */
/* decomposition algorithm                           */
/* ================================================= */

void Get_weights(float A[][NCARS], float W[][NFRAMES], float B[][NFRAMES], \
               WINFO wi) {

   float Wsvd[NCARS] = {0};     // Matrix used in svd
   float V[NCARS][NCARS] = {0};    // Matrix used in svd
   float U[NHARS][NCARS] = {0};    // Matrix used in svd
   float T1[NCARS][NCARS] = {0};     // Intermediate results
   float T2[NCARS] = {0};          // Intermediate results
   int i,j,k;
   float Wsvd2;


   for (i=1; i<=NUM_HARS; i++)
     for (j=1; j<=wi.carriers; j++)
       U[i][j] = A[i][j];

   svdcmp(U,NUM_HARS,wi.carriers,Wsvd,V);
```

```
  for (i=1; i<=wi.carriers; i++)
    for (j=1; j<=wi.carriers; j++) {
      if (Wsvd[j] == 0)
        Wsvd2 = 0;
      else
        Wsvd2 = 1/Wsvd[j];
      T1[i][j] = V[i][j]*(Wsvd2);
    }

  for (k =1; k<=NUM_FRAMES; k++) {

    for (i=1; i<=wi.carriers; i++)
      T2[i] = 0;

    for (i=1; i<=wi.carriers; i++)
      for (j=1; j<=NUM_HARS; j++)
        T2[i] += U[j][i]*B[j][k];

    for (i=1; i<=wi.carriers; i++)
      for (j=1; j<=wi.carriers; j++)
        W[i][k] += T1[i][j]*T2[j];
  }
}

/* ================================================== */
/* Find model harmonics using weight matrix calculated */
/* above                                               */
/* ================================================== */

void Get_frame_harmonics(float A[][NCARS],float W[][NFRAMES],\
       float Bm[][NFRAMES], WINFO wi) {

  int i,j,k;

  for (i=1; i<=NUM_FRAMES; i++)
    for (k=1; k<=NUM_HARS; k++)
      for (j=1; j<=wi.carriers; j++)
        Bm[k][i] += W[j][i]*A[k][j];

}

/* ================================================== */
/* Calculate fitness of individual by comparing       */
/* the model frame-harmonics Bm and the real          */
/* frame harmonics B                                  */
/* ================================================== */

double fitness(float B[][NFRAMES],float Bm[][NFRAMES], WINFO wi) {

  double sum1=0;
  double sum2=0;
  double sum3=0;
  int i,j;

  for (i=1; i<=NUM_FRAMES; i++) {
    for (j=1; j<=NUM_HARS; j++) {
      sum1 += pow( (B[j][i]-Bm[j][i]),2 );
```

```
        sum2 += pow( Bm[j][i],2 );
      }
      sum3 += sqrt(sum1/sum2);
   }
   return (sum3);
}

/* ================================================ */
/* Main evaluation function, gets Wave information wi */
/* and original sound harmonics B and returns model   */
/* harmonics Bm and fitness *fit                      */
/* ================================================ */

eval(WINFO wi, float W[][NFRAMES], float B[][NFRAMES], \
     float Bm[][NFRAMES], float S[], double *fit)
{
   int i,j,k;
   float A[NHARS][NCARS] = {0};    // Harmonics for each carrier
   float D[NHARS][NFRAMES] = {0};   // real harmonics with changed signs

   for (i=0; i<=NUM_HARS; i++)
      for (j=0; j<=NUM_FRAMES; j++)
         Bm[i][j] = 0;

   for (i=0; i<=NUM_CARS; i++)
      for (j=0; j<=NUM_FRAMES; j++)
         W[i][j] = 0;

   Multiply_sign_matrix(B,S,D);
   Get_carrier_harmonics(A,wi);
   Get_weights(A,W,D,wi);
   Get_frame_harmonics(A,W,Bm,wi);
   *fit = fitness(D,Bm,wi);
}
```

The method for finding the best Weight matrix in the equation AW ≈ B is outlined below (Press et al 1989)

The structure of the original equation looks as follows:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N_{cars}} \\ a_{21} & a_{22} & \dots & a_{2N_{cars}} \\ . & & & \\ . & & & \\ . & & & \\ a_{N_{hars}1} & a_{N_{hars}2} & & a_{N_{hars}N_{cars}} \end{bmatrix} x \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1N_{frames}} \\ w_{21} & w_{22} & \dots & w_{2N_{frames}} \\ . & & & \\ . & & & \\ . & & & \\ w_{N_{cars}1} & w_{N_{cars}2} & & w_{N_{cars}N_{frames}} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1N_{frames}} \\ b_{21} & b_{22} & \dots & b_{2N_{frames}} \\ . & & & \\ . & & & \\ . & & & \\ b_{N_{hars}1} & b_{N_{hars}2} & & b_{N_{hars}N_{frames}} \end{bmatrix} \quad \textbf{(20)}$$

This equation is actually a group of (N_{frames}) sub problems which can be solved in turn. If corresponding frame columns are extracted from the above equation (eg. the first column of W and the first column of B) then the sub problem can be stated as follows in tableau:

$$\begin{bmatrix} \\ A \\ \\ \end{bmatrix} . \begin{bmatrix} \\ w \\ \\ \end{bmatrix} = \begin{bmatrix} \\ b \\ \\ \end{bmatrix} \quad \textbf{(21)}$$

where (w) = column from W and (b) = corresponding column from B

The first stage of the solution is doing a Single Value Decomposition or SVD on matrix A. It can be shown from linear algebra that A can be broken up into the following matrices.

$$\begin{bmatrix} \\ A \\ \\ \end{bmatrix} = \begin{bmatrix} \\ U \\ \\ \end{bmatrix} \begin{bmatrix} w_1 & & \\ & w_2 & \\ & & w_3 \end{bmatrix} \begin{bmatrix} \\ V^T \\ \\ \end{bmatrix} \quad \textbf{(22)}$$

where A = MxN matrix
U = MxN column orthogonal matrix
W = NxN diagonal matrix
V^T = Transpose of NxN orthogonal matrix V

The solution for (w) now follows:

$$\begin{bmatrix} \\ w \\ \\ \end{bmatrix} = \begin{bmatrix} \\ V \\ \\ \end{bmatrix} \begin{bmatrix} \\ diag(1/w_j) \\ \\ \end{bmatrix} \begin{bmatrix} \\ U^T \\ \\ \end{bmatrix} \begin{bmatrix} \\ b \\ \\ \end{bmatrix} \qquad (23)$$

If $w_j = 0$ in the the jth entry of the diagonal matrix W then simply set $1/w_j$ to 0.

The solution is obtained for each frame in turn and the resulting column vectors (w) are inserted into the final matrix solution W.

If two waveforms are created, one which consists of sine wave A at frequency f_1 added to a sine wave B at $2f_1$ and another which consists of a sine wave 180° out of phase with A added to sine wave B, audio tests show that the ear cannot distinguish between these two waveforms.

These result are shown below in
Figure E.1 and **Figure E.2**

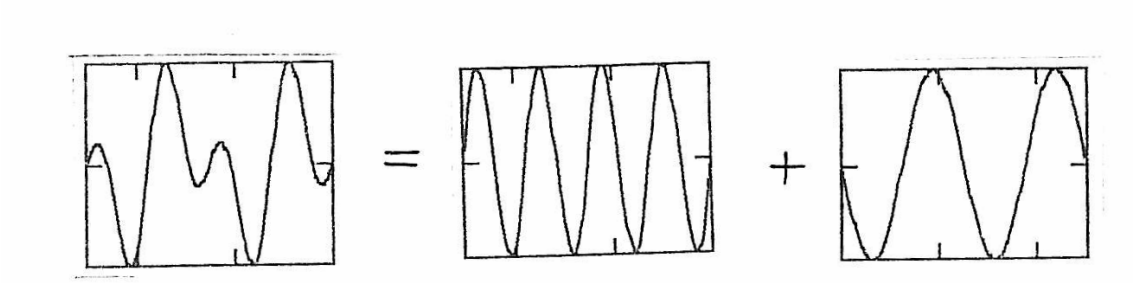

Figure E.1 $\text{Sin}(2\pi f_1 t) + \text{Sin}(4\pi f_1 t)$

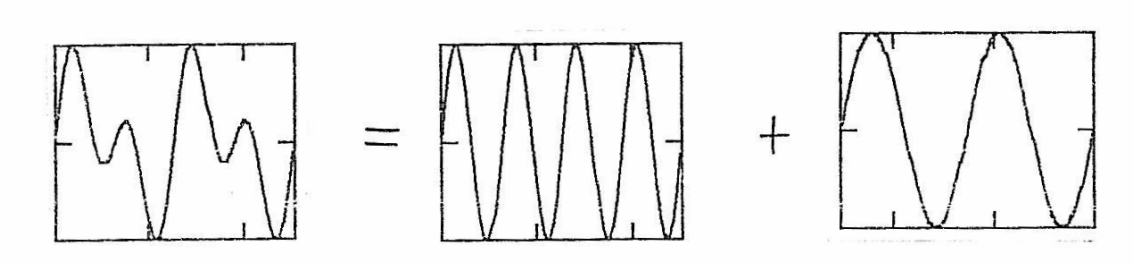

Figure E.2 $\text{Sin}(-2\pi f_1 t) + \text{Sin}(4\pi f_1 t)$

Equation (15) looked as follows:

$$dfm(t) = \sum_{i=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} J_i(I_a) J_k(I_b) \sin(2\pi(i+kn) f_a t) \quad \textbf{(24)}$$

The harmonic number is given by i+kn and various combinations of i and k will map to particular harmonic numbers. The above equation suggests that it is necessary to evaluate an infinite number of combinations. Fortunately Bessel functions have the property that as the number k in $J_k(I_m)$ gets larger the Bessel function gets smaller. So when does the bessel function get insignificant? Well for modulation indexes I_m up up 10 it turns out that evaluating n up to 15 will give results significant to about 3 decimal places.

Table F.1 gives a demonstration of harmonics generated using f_a = 100Hz and f_b = 100Hz (ie. n=1) using ranges [-2,2] for i and [-2,2] for k

If only the 1st harmonic was wanted the algorithm adds together all harmonic partials ($J_i(I_a).J_k(I_b)$) when h=1 and subtracts from this all harmonic partials when h=-1. So the first harmonic will have a value = -(0.165) -(0.088) +(-0.165) -(0.088) +(-0.088) -(0.165) +(-0.088) +(-0.165) = -0.812

This same process is done for all the harmonics in the DFM wave using ranges of i=[-15,15] and k=[-15,15] and the C code can be seen in the FM-wave code in Appendix B.

i	k	h = i+nk	$J_i(I_a).J_k(I_b)$
-2	-2	-4	0.236
-2	-1	-3	-0.165
-2	-0	-2	-0.126
-2	1	-1	0.165
-2	2	0	0.236
-1	-2	-3	-0.165
-1	-1	-2	0.115
-1	0	-1	0.088
-1	1	0	-0.115
-1	2	1	-0.165
0	-2	-2	-0.126
0	-1	-1	0.088
0	0	0	0.068
0	1	1	-0.088
0	2	2	-0.126
1	-2	-1	0.165
1	-1	0	-0.115
1	0	1	-0.088
1	1	2	0.115
1	2	3	0.165
2	-2	0	0.236
2	-1	1	-0.165
2	0	2	-0.126
2	1	3	0.165
2	2	4	0.236

Table F.1 DFM harmonic partials with fa=fb=100 and Ia=Ib=3

APPENDIX G PARAMETER FILES

The Genetic Algorthm parameter file

```
Experiments = 1            // Total number of experiments
Total Trials = 10000       // Number of gene evaluations     before termination
Population Size = 50       //
Structure Length = 52      // Length of gene bit string
Crossover Rate = 0.6       // Crossover rate = 60%
Mutation Rate = 0.01       // Mutation rate = 1%
Generation Gap = 1.0       // Fraction of population replaced in each generation
Scaling Window = 5         // (Grefenstette 1990)
Report Interval = 50       // Print status every 50 trials
Structures Saved = 10      // Number of best structures saved
Max Gens w/o Eval = 2      // Maximum generations allowed without evaluation
Dump Interval = 0          //
Dumps Saved = 0            //
Options = abcDLot          // (Grefenstette 1990)
Random Seed = 123456789    // Seed for random number generator
Rank Min = 0.75            // Used for rank selection
```

FM synthesis parameter file

```
Sampling rate = 28160      // Create wave at sample rate of 28160 Hz
Sampling time = 2          // Create a 2 second wave
Base frequency = 220       // Create the wave at 220Hz
FM type = 0                // 0=CFM 1=DFM
Envelope type = 0          // 0=momentary excitation envelope
                           // 1=continuous excitation envelope
Carriers = 2               // Total number of carriers
a bits = 5                 // Use 5 bits to encode amplitude
envp bits = 5              // Use 5 bits to encode peak envelope position
f bits = 4                 // Use 4 bits to encode frequency
I bits = 7                 // Use 7 bits to encode Modulation index
Maximum I value = 5        // I value range = [0,5]
Minimum env value = 0.0    // Minimum envelope position 0s
Maximum env value = 0.5    // Maximum envelope position is 0.5*2s=1s
```