ALGORITHMIC COMPUTER MUSIC,
3rd Revised Edition
by
DR. JOHN R. FRANCIS, Ph.D.

Computer Music Algorithms explains algorithmic music and contains 50 programs, 19 styles, & 23 chapters that will generate music of different styles, new each time executed, as a midi file. One chapter teaches how to write wav music directly from numbers and formulas, without recording equipment. The 'styles' produce music from the algorithm of Kircher (1650) to all styles including the author's own unique fractal music. The algorithms are explained in technical terms of music theory, including the special data structures constructed. Styles range from medieval music to modern and the genre of game music. There are folders of c files for each chapter with code, that may be compiled and when executed, generate the midi files.

May 18, 2016
e-mail: Bluewolf3.14@gmail.com

Musical Structure, if not evident in the music itself, will be imposed by the mind of the listener.

paraphrased from author's dissertation,
"Structure in the Solo Piano Music of John Cage"

WARNING – WARNING – WARNING

I once owned a free-ware game of SpaceCastleII, a PC game. When the game started, it stated, "if you paid for this game, you paid too much."

This is the first copyright for the original work, Computer Algorithmic Music in 2011.

Certificate of Registration

This Certificate issued under the seal of the Copyright Office in accordance with title 17, *United States Code*, attests that registration has been made for the work identified below. The information on this certificate has been made a part of the Copyright Office records.

Maria A. Pallante

Acting Register of Copyrights, United States of America

Registration Number
TXu 1-737-423

Effective date of registration:
March 14, 2011

After 12 revised editions of the above work, the book and source code, as suggested by the copyright office, has been completely overhauled into a new work and new copyright, now named Algorithmic Computer Music.

Certificate of Registration

This Certificate issued under the seal of the Copyright Office in accordance with title 17, *United States Code*, attests that registration has been made for the work identified below. The information on this certificate has been made a part of the Copyright Office records.

Maria A. Pallante

Register of Copyrights, United States of America

Registration Number
TXu 1-953-144
Effective Date of Registration:
January 27, 2015

Title

Title of Work: Algorithmic Computer Music

Completion/Publication

Year of Completion: 2015

Author

- **Author:** John Richard Francis
  Author Created: text
  Citizen of: United States

DEDICATION

I dedicate this book to my son, Ethan, who is truly brilliant in all ways, and an inspiration to me, and also has made suggestions to the programs as well as helped me find those nasty little and big bugs that are always present when composing a C program. It is truly amazing to me how fast he can find a mistake in the code, and give an instant suggestion. He has graduated from VT with a degree in computer software science, and I truly hopes he finds whatever he is seeking.

I have also written two computer games for the purpose of showing that there can be an adequate 'fog of war' for solo board games.

Also, since most solo board games have tons of rules, often confusing, writing a solo board game first as computer program guarantees the rules will be completely, and hopefully clear and organized. These two games will play the games, as well as generate new game boards for extra 'fog or war'. Also, 32 board games are already included in each game.

I am putting this plug in because these games can simply not be found on SourceForge by searching the Internet or SourceForge. This is an odd-ball category, so the link is here:

http://twocomputerdungeoncrawls.sourceforge.net

This link is a mistake. I assumed when I changed the title that the link would be changed also with sourceforge. So the above link should be:

https://sourceforge.net/projects/fivecomputerdungeoncrawls/

(Since this update, a third program/game/boardgame has been added.)

About the Author

Dr. John R Francis graduated from the New England Conservatory with a Bachelors in Music in Piano Performance, a Masters in Music in Music Theory with Honors from the same Conservatory, and a Ph. D. In Music Theory from Florida State University. His dissertation was "Structure in the Solo Piano Music of John Cage". He taught all of the music theory courses to the music majors for more than twenty six years at a small college in Charleston, S.C. These courses include Elementary Music Theory (two semesters), Elementary Keyboard Harmony (two semesters), Elementary Ear Training and Sight singing, (two semesters), Advanced Music Theory (two semesters), Advanced Keyboard Harmony (two semesters), and Advanced Keyboard Harmony (two semesters), as well as the upper level course Arranging/Composition. In addition, he has taught private composition and piano to college level students, as well as private students. He has given lectures, demonstrations, and performances of his own algorithmic music at the University of Alabama, University of South Carolina, the College of Charleston, and Charleston Southern 'University'. He also has given piano concerts of his own works, as well as a concert on ten 'prepared' pianos of the piano music of John Cage.

Although self-taught in the language of 'c', he took computer courses in Pascal, Data Structures, and Assembly from Dr. Kennelly, who started the computer system at the Berkeley Power Plant, including hardware and software.

This book was written using LibreOffice Writer in Ubuntu 12.04 Linux.

TABLE OF CONTENTS

Notes

The c files and the important header file convert3.h (that is necessary to compile all the programs), are in separate folders with chapters labeling them.

All of the files are in standard 'c'. There are no special graphical libraries that are need, although, as I said above, each program will need the convert3.h which I have repeatedly included in the folders.

There is no user input. All of the algorithmic programs that compose music, once compiled and executed, will produce a new and different midi file (or a wav file in one chapter). The music composing programs are idiot proof for the user simply because there are no decisions to be made.

CHAPTER ONE - INTRODUCTION TO ALGORITHMIC MUSIC

Algorithmic music composition is a method of composing music, usually by computers, that is not dependent on anyone, but can create a new composition each time that the program is executed. Most people simply do not believe it is possible. If they ask what type of music is produced, but it is really the type of algorithmic computer program that is executed; that is, the programs may create many different types of music.

I will start historically at the beginning, and recreate the algorithms before computers; algorithms by 'hand' and quickly move to algorithms I created that will compose certain styles of music. Each style of music is unique, and demands a new and different type of algorithm to produce that style. All of the programs in this book are written in the language C, and I used the standard C so that it may be compiled in Linux with GCC, or any Windows program or Macintosh program. The compilers for building the programs in the book can be found free anywhere on the Internet. My suggestion is to use the free Linux programs, especially Debian or the Debian based platforms such as Ubuntu. The output of the algorithmic programs in this book are midi files, except in one chapter, which produces wav files. The user may use any sequencer program that plays midi files, which again, may be found on the Internet for free. You may then wish to convert your saved midi files you produce from these programs into some sort of wav or ogg or mp3 file, again all free programs.

The question frequently is asked if computers are capable of human emotions. It is clear that people feel threatened by algorithmic music, just as the chess masters felt very threatened by the very first chess programs.

What people do not realize is that we can cut the Gordian knot by getting rid of the meaningless question “can computers think or have emotions” and replace it by the **verifiable question** “can computers produce compositions that humans like to hear”. And of course the answer already is YES. Does a computer need to have emotions? I state now that the emotions and interesting things in music occur in the mind of the listener, and people of all cultures are 'trained' to listen to certain things in their system of music. The first program in the book is a wind chime program, and though wind chimes produce fairly random sounds, they are pleasing to the ear. But then the human brain also likes structure, and there is plenty of structure in the following programs. The criteria for music is broad, and it it not just pleasing sounds, either, so that some programs may sound dissonant to the listener, though not necessarily so. Most theories of music believe that music can cause emotions by interrupting our expectations temporarily, and then resolving them. Two kinds of interruptions come easily to mind: non-harmonic tones and the retrogression of a normal harmonic progression. The second program in the book is based on a Kircher algorithm before computers, which uses no non-harmonic tones at all, and his music does not use harmonic progressions at all, but rather harmonic chord successions. Still, the music is quite pleasing. The third program in the book is the musical dice game by Mozart, another algorithm before computers, and uses no interruption in the normal harmonic progression, but uses some incidental non-harmonic tones. Yet it is

also definitely music, yet vastly different in sound of the earlier Kircher algorithm of the previous chapter. The later programs in the book use all kinds of devices, and some are intellectually 'interesting', such as the fractal music I invented, or even the serial and atonal music. The beauty in a this type of music is intellectual as in viewing a beautiful symmetrical crystal from different angles. And sometimes the beauty is in the sounds and pure randomness, as in a Pollock painting or composition from John Cage.

In understanding this book, it is important that we should not call the subject taught in college as 'Music Theory', but rather, 'Systems of Music'. For there are many different systems, all different from one another, which are sometimes referred to as 'styles'. However, even music as diverse as serial and fractal music from the standard 'tonal' music of 1750-1900, if understood, has value and interests. Nothing really makes one system of music more important that others; they are all important.

The programs presented here use different algorithms for different styles of music. The programs are presented in their entirety, and are guaranteed to run perfectly without any bugs in the code or runtime errors. Although many will not understand the C language, the outline of the code will be explained in each chapter, and also the music theory for the code.

I am a music theorist/composer and a programmer. The language will be technical; as I mentioned above, everything is explained. However, the reader is not babied and every term that is well known in a programming language or in music theory can be investigated from outside sources. This book does not pretend to teach music theory or a programming language, although it should present new ideas, most musical styles, including historically western music as well as ethnic music, and new genres of music including game music and some types of music based on popular mathematical ideas.

Although some of these programs may seem long, such as the counterpoint program, they are really short compared to most programs today. For the brevity of the programs, they produce an amazing output of different styles, and speaking as a theory teacher, the programs compose music better than most college music students or amateur composers. Hence, if the programs were expanded, just as the early chess programs were expanded, it would get to a point where the music is simply 'better' than current human composition. This is not depressing. It is simply a new tool in future of music composition in the hands of the programmer/musician.

CHAPTER TWO - INTRODUCTION TO MIDI FILES

In every one of the algorithms, there are certain functions in C that will be used. Of course, every compiler changes the program you write in whatever language into a string of binary numbers. The certain functions I mentioned above are special functions that will change the output of the program into a midi file, which is still a string of binary numbers, but in a certain format. That is, it is identified by the beginning string of binary numbers and the computer recognizes it as a midi file, just as some files are jpeg files.

So to clear this up a little, each program's source code (the code given in the book in the language of standard C) will be transformed into machine language (binary numbers) by your compiler. That program, once executed (run by the user), will in turn produce a midi file, which are also binary numbers. That midi file can then be played by a midi sequencer, which turns the binary numbers into pitch, volume, timbre, duration, and other parameters of music. I have written all the functions below, which produce the midi files. There are a number of books on midi and midi files, but the best can be found on the web by looking for the midi file standard. There are also helpful articles. I will not try to explain in detail the below functions, as it is unnecessary for the understanding of the music algorithms, but I will give a brief explanation of these functions I have written. They are a simplified set of functions, so that one may write mainly duration, pitch, instrument, and volume. Much more elaborate functions could be written, but this is all that is necessary to produce music. I have found it adequate for my needs.

The functions produce two files; one is a temporary file, the other is

the midi file. The temp file is the music the program produces. But a midi file needs to know the length of the file. Thus the beginning of the file, which is the header that identifies the type of file and some characteristics about the file, must be placed before the temp file and at the end of the temp file. So after the temp file is produced, the length is calculated with the beginning and the end above mentioned, and thus the midi file is constructed. Thus the FILE *fp2 and FILE *fp1. **To play the composition produced by these algorithmic programs, it is necessary to find the "music.mid" file produced by the program, and use a midi sequencer such as timidity to play it. There are plenty of free midi players on the web.**

The play() file is the most important for you. It first gives the duration, then the channel (or instrument), then the pitch to be played, and lastly the volume.

Now this is very important. I use the play() file also for the beats, for lack of a better term. Let me explain.

```
Function1    play(0,144,60,100);
Function2    play(48,152-16,100,0);
Function3    play(0,144-16,60,0);
```

Function1 plays immediately (the duration is 0), on the channel 144 (which will have an instrument assigned to it), with the pitch 60 (which is middle C), at the volume 100.
Function2 is a wait function. The duration that it waits is 48, which is a quarter of a second. I NEVER use channel 152 for any notes except as a wait function. The 152 minus 16 means turn off the note playing on channel 152. The note is 100, which is meaningless; it could have been any note. It was never played in the first place. The

purpose is merely to wait for the 48 (quarter of a second) before the next event. The last event in the function is 0, which is the volume, to ensure nothing is played.
Function1 has now been held for a unit of 48 in length, and Function3 is going to turn off the note C in Function1. The duration is 0, as we want it turned off immediately, the channel is 144, but we subtract 16 from it to indicate that the note on channel 144 is turned off, and the pitch 60 is given to tell which note is tuned off. THE NEXT PART IS VERY IMPORTANT. In some poor sequencers, the 144-16 is not enough to turn it off. It is important to turn the volume to 0 as well in function 3 to ensure the note is no longer playing. In other sequencers, the 0 may not be enough and the 144-16 must be present. You should have both 144-16 and also 0 for the volume at the end to turn off that note C.

This is what you really need to know to read some of the code in C to understand what is going on, although I will give the pseudo code explained in English.

The other parts of the functions given I will explain briefly. The two convert functions are because of something called delta time. Briefly, instead of having 4 bytes of information for each length of time, which would make the midi file very lengthy, if it is possible to express the time as less, then the 8^{th} bit of the first byte tells the sequencers to stop there, otherwise, the 8^{th} bit will tell the sequencer to look at the second byte as well. Four bytes of length would be very long indeed, but the makers of midi wanted it to express any type of piece, so there is adequate information in midi to perform almost any type of composition. Delta time is confusing. It is explained, to some extent, in the Midi convention on the web. I spent a great deal of time before I wrote the code to convert time to delta

time.

The general midi function merely tells the sequencer to use the standard midi instruments. It was not easy for me, years ago when there was little written on midi to write these functions. I was delighted when I actually got a sound to come out of the computer. So much was not written that I had to do much experimentation, and to look at simple midi files already written and try to figure out what was wrong with my code. This is the nasty part. I have no wish to go back and redo this code. Once I produced one note, it did not take long at all to write chords, and then the problem was to write algorithms that would write a new midi file of music each time.

This part is optional to read but interesting. It is often said that there is no true randomness in computers. Yet all of the programs that will be given depend on randomness. A random function is often written by using a very long division, or some similar process. Thus, eventually, it will start to repeat. It is possible to 'seed' it by using the clock computer, but even so, it is starting in somewhere in the same sequence of long numbers. Thus many believe there is no true randomness in computer programming. But this is not true. In Linux, it is possible to use the hash or the garbage file (information the computer used previously and discarded) as a source of random numbers, and thus the random events of the day determine the random function in the program.

Just because you are using random functions does not mean the algorithm is random. A random function selects numbers from a set of numbers, and if your algorithm does not allow those numbers, it can try again till it finds something that works with your criteria. A random function only guarantees that the pieces will differ.

The header file <u>needed</u> for all the programs is called <u>convert3.h</u> and is in every folder for the programs. You must have that file in the same folder where you are compiling your program.

CHAPTER THREE - WIND CHIME MUSIC

Supposedly, wind chimes date back to 30,000 years ago, and definitely 3000 years ago, and were very common in the East and traveling to the West 2000 years ago.
Yes, wind chime music is random, but not entirely so. There are the materials that produce the sounds, from glass to bone (prehistoric) to brass and bronze bells. The way they are constructed determines not only the pitch, but which bell is rung more frequently. Some chimes are bent in such a way that two notes are produced from one chime. Also, of course, the bell may be cut in such a way to make the frequency desired. Thus there are certain favorite 'scales' used for certain chimes. The below program uses one of the oldest scales, the pentatonic scale. Next it uses the whole-tone scale, a favorite of Debussy and the music of Bali, and then the locrian scale, a very bright scale (a major scale with the 4^{th} degree raised, thus two leading tones in the scale). Some music theorists believe that the lydian scale should have been used in Western music instead of the major scale.

This is a simple program and I will explain it after showing this code:

```
counter = 0;
do
{
for (i = 0; i < 7; ++i)
{
note[i] = rand()%6;
octave = 24;
note[i] = (pentatonic[note[i]])+ octave;
j[i] = rand()%8;
play(0,channel[j[i]],note[i],volume);
play(rand()%(1500) + 1,152-16,60,0);
```

```
}
play(200,152-16,60,0);
for (i = 0; i < 7; ++i)
play(0,channel[j[i]]-16,note[i],0);
++counter;
}while( counter < 7);
counter = 0;
do
{
for (i = 0; i < 7; ++i)
{
note[i] = rand()%7;
octave =  24;
note[i] = (whole_tone[note[i]]) + octave;
j[i] = rand()%8;
play(0,channel[j[i]],note[i],volume);
play(rand()%(200) + 1,152-16,60,0);
}
play(200,152-16,60,0);
for (i = 0; i < 7; ++i)
play(0,channel[j[i]]-16,note[i],0);
++counter;
}while( counter < 7);
do
{
for (i = 0; i < 7; ++i)
{
note[i] = rand()%6;
octave = 24;
note[i] = (pentatonic[note[i]])+ octave;
j[i] = rand()%8;
play(0,channel[j[i]],note[i],volume);
play(rand()%(1300) + 1,152-16,60,0);
}
play(200,152-16,60,0);
for (i = 0; i < 7; ++i)
play(0,channel[j[i]]-16,note[i],0);
++counter;
}while( counter < 7);
counter = 0;
do
{
for (i = 0; i < 7; ++i)
{
note[i] = rand()%8;
octave = 24;
```

```
note[i] = (lydian[note[i]]) + octave;
j[i] = rand()%8;
play(0,channel[j[i]],note[i],volume);
play(rand()%(2000) + 1,152-16,60,0);
}
play(200,152-16,60,0);
for (i = 0; i < 7; ++i)
play(0,channel[j[i]]-16,note[i],0);
++counter;
}while( counter < 7);
```

The first part of the program uses the pentatonic scale to play 7 random notes, before cutting them off. The tempo is slow. It uses a bell sounding instrument.

The second part of the program uses a whole tone scale to again play random notes. The tempo is much faster so I guess the wind has picked up.

The third part of the program slows down a little more and uses the pentatonic scale again.

The fourth part of the program goes back to the original scale and uses the 'modern' lydian mode (as opposed to the Greek lydian scale, which used different names for their modes).

Windchimes.c is in the folder. As always, do not forgot to include the header file, convert3.h, where-ever you place your c program.

CHAPTER FOUR - Music of the Trouvers (~1100-1300)

The music of the medieval ages is marked with controversy. However, some examples have survived, and some documents that try to explain the theory of the time, give an idea how the music probably sounded, though it is likely that the music varied greatly in performance from different groups of musicians.

I have used the music of the Trouvers, as it seemed to be likely that they used a form of the rhythmic modes. Though the examples that survive may not totally be in agreement with the theory of the rhythmic modes, it seems clear that at times the music had clear cut phrases of four measures, generally in the 'time signature' of 3 quarter notes to the measure. And though not all agree that the Trouvers used the form of the Virelai, which is ABCCABAB, there are examples that exist of the Trouvers that indeed use this form, as in the book “Masterpieces of Music Before 1750” by Carl Parrish.

Thus I based the next algorithmic composition on a Virelai that uses at least part of the ideas of the rhythmic modes, since the examples of the Trouvers seems to lend themselves to 3 quarter notes and not 6 eighth notes or two measures of 3 quarter notes with the same rhythm.

No doubt, the original church modes were much in use at this time, and also the range of the melodies were limited by the singers and the instruments. Harmony, if any can be said to exist, is now agreed to be open fifths at the beginning of phrases and cadences of phrases of monophonic music.

This then is the algorithm that is used to make a simple type of Virelai, using a limited range of notes based on the church modes, and also based on the first half of the rhythmic modes that were most often used (in contrast to the ones that were rarely used). It is understood that this was a time of great change, and the music that followed this became much more complex rhythmically and harmonically. However, the music of this time is delightful, with infinite variety.

In this algorithm, the melody was constructed first. It began on the tonic of the mode, but at the end of the A phrase it did not have to end on the tonic. It could end on any note. Melodic movement is mostly by step, as it was in this time period, although their were and could be skips of the third. It is necessary for the end of the B phrases to end on the tonic of the mode for the listener to be aware of the mode. The C section, being a contrasting section, need not begin or end on the tonic. It is repeated in the form of the Virelai and thus gives unity just as Debussy did by repetition of unfamiliar harmonic structures. After the C section, the A and B sections are repeated twice. When the open fifth as accompaniment is used at the beginning and end of phrases, the lowest note of the fifth is the same as the note of the melody. Many now believe that if harmony was used in this period, it was probably used in this way. The names of the variables and functions in the source code, as well as comments, provide the information to understand how the rest of the program was written.

Medieval.c is in a folder.

CHAPTER FIVE - THE ALGORITHMIC MUSIC OF A. KIRCHER (1650)

Athanasius Kircher was a Jesuit Scholar and was often called the “last Renaissance man”. He had interests in everything and wrote many books on obscure subjects, bringing back such interests as the ancient amphitheater, which was really a box full of mirrors that gave the illusion of perspective and great depth in a small box, which was derived from the 'ancients'. He also invented the aeolean harp, the megaphone, the magic lantern, and a pantometrum for solving geometric problems. He has been overlooked by musicologists and musicians, and although he was an amateur musician, his contribution to music was enormous.

Although there are a few other types of algorithms, most can be broken down to combinatoric algorithms or rule-based algorithms. Kircher was a master of combinatoric algorithms, but he also used a few ingenious ideas that vastly increased the amount of combinations of compositions that could be constructed.

First of all, the title of his eighth book translates to “The Universal Musical Art – the eighth book of the great art of consonance and dissonance that is the wonderful musical art, a new, recently invented musical arithmetical skill, by which everyone is able to acquire in a short time a profound knowledge of composing, even if he knows nothing about music”. The heart of the book is the Musurgia Mirisica, where this part is explained and tables of numbers for rhythm and pitch are given. Kircher even made a box of wooden slates that could be pulled out and manipulated in the manner of Napier's rods, which he called the Arca, and which contained the same numbers as in the Musurgia Mirisica.

The idea behind the Musurgia Mirisica is ingenious. Kircher made a number of short melodies, of different lengths for almost any number of syllables, so that 'hymns' or contrapuntal compositions could be constructed. This was the heyday of combinatorics, which fascinated the mathematician Mersenne before him, and the pseudo philosopher Lull before him.

Before explaining in detail how his combinatorics work, I should mention that they are still in use in many algorithmic works today as Markov chains. Markov chains are strings of notes, which from analyzing a composer statistically, discovers chains of probabilistic notes and chords. Then, chaining these strings together, we get a composition somewhat similar to compositions of the original composer.

Kircher made his very large number of melodies, and they were harmonized in four parts- Bass, Tenor, Alto, Soprano. There were four phrases to the hymn, and for each phrase, there were a certain amount of syllables. There were as many as ten different possibilities with a new melody and harmonization for each part of the hymn. Thus, any of the ten 4-part harmonized melodies could go to any of the ten 4-part harmonized melodies of the second phrase, which could go to any of the ten 4-part harmonized melodies of the third phrase, to any of the ten 4-part harmonized melodies of the final phrase. But that is not all. Not only did he have 2-note melodies (for 2 syllables), but also 3-note melodies, etc, on up to many syllables. This already gives a lot of combinations. However, since in the hymn part of this method, he gave a variety of rhythms since all notes of a chord sounded together, note against note. For each phrase he gave many possibilities of rhythms. And it did not stop there. You, as the 'composer', were free to pick the order of the upper

voices. Thus the Alto could be in the range of the highest voice (thus becoming the new Soprano), the Tenor could be the next to highest voice, and the Soprano could be in the range of the Tenor. Any of the three upper voices could be mixed in any order in a new range. Then, in addition, the melodies, which were notated as numbers of the scale, could be put to various modes, thus altering some of the pitches and giving a different character or mood to the composition.

I only used one page (page 85) from the voluminous pages of tables (pages 60 to 146 = 86 pages!! of tables) from the Musurgia Mirisica to construct the program of Kircher's algorithm. The one page of table I did use makes so many combinations that I doubt you will tire of it. Even so, the program itself is simple, even though it contains many tables of data on that one page (page 85).

These are some the rhythms allowed on page 85. I left out the very long notes at the bottom of the page, as it seems unnecessary for the purpose of this demonstration.

```
int rhythm_anakreontic[RHYTHMIC_PATTERNS][RHYTHMIC_UNITS] =
{{WHOLE,DOTTED_WHOLE,HALF,HALF,HALF,WHOLE, WHOLE},
{HALF,WHOLE,WHOLE,WHOLE,HALF,WHOLE,WHOLE},
{HALF,QUARTER,QUARTER,QUARTER,QUARTER,HALF,WHOLE},
{HALF,DOTTED_QUARTER,EIGHTH,DOTTED_QUARTER,EIGHTH,HALF,WHOLE},
{HALF,WHOLE,HALF,HALF,HALF,WHOLE,WHOLE},
{WHOLE,DOTTED_HALF,QUARTER,QUARTER, QUARTER,HALF,WHOLE},
{QUARTER,HALF,QUARTER,QUARTER,QUARTER,HALF, WHOLE},
{QUARTER,EIGHTH,EIGHTH,QUARTER,QUARTER,HALF, HALF},
{QUARTER,QUARTER,QUARTER,DOTTED_HALF.QUARTER,WHOLE,WHOLE}};
```

Now I give an example of some of the table that produces the Soprano Alto Tenor Bass (this page is based on 7 notes per phrase).

```
int pitch_anakreontic[PHRASES_OF_VOICES][ALTERNATES][NUMBER_VOICES]
```

```
[PITCH_LENGTH] =

                {{{{5,5,5,4,3,2,3},
                  {7,7,8,6,5,5,5}, /* 1 */
                  {2,3,3,8,8,7,8},
                  {5,3,8,4,8,5,8}},

                 {{3,3,2,8,8,7,8},
                  {8,8,7,6,6,5,5}, /* 2 */
                  {5,5,5,3,4,2,3},
                  {8,8,5,6,4,5,1}},

                 {{5,5,5,5,5,5,5},
                  {8,8,8,8,7,8,8}, /*3*/
                  {3,3,3,3,2,3,3},
                  {8,8,8,8,5,1,1}},
```

And etc. You can see that in the first group the soprano starts on the 5th degree of the scale or mode, plays it two more times, then moves to the 4th degree of the scale, and so on. The Alto would then begin on the 7th degree of the scale and repeat before moving on to the 8th degree of the scale, which is of course the tonic. And so forth. Thus, hypothetically, if we use a C Major scale where the tonic is 1, then the above first chord would be

G (soprano)
B (alto)
D (tenor)
G (bass)

Remember that later in the program there will be a random procedure that allows the entire first phrase to put the upper three voices in any order. This will be shown later in the program.

Next I show two of the modes:

```
int mode[MODETYPE][VOICES][PITCHES] =
{{{C5,D5,Eb5,F5,G5,A5,B5,C6},
{C4,D4,Eb4,F4,G4,A4,B4,C5}, /* dorian */
{C3,D3,Eb3,F3,G3,A3,B3,C4},
{C2,D2,Eb2,F2,G2,A2,B2,C3}},

{{C5,D5,Eb5,F5,G5,Ab5,Bb5,C6},
{C4,D4,Eb4,F4,G4,Ab4,Bb4,C5}, /* phrygian */
{C3,D3,Eb3,F3,G3,Ab3,Bb3,C4},
{C2,D2,Eb2,F2,G2,Ab2,Bb2,C3}},
```

You will notice that the Dorian mode has been altered. The B, which should be flat, is raised, for a leading tone, which was common in that era. Also In the Phrygian mode, the B was not raised, as it would cause the augmented second between Ab and B, which was not allowed. The D has been lowered as it would make a V chord (G, Bb, Db) contain a tritone (G to Db), which was not allowed. So in essence, the Phrygian has turned into the natural minor scale.
Now we are ready to put it all together, to make a composition of Kircher. Here is the simple and concise part of the main function that composes the music:

```
m = rand()%(MODETYPE);

for (p = 0; p < 4; ++p)
{
j = rand()%6; /* order of upper voices */
order_of_upper_voices[j][3];    /* access one at a time [j][0] [j][1] [j][2] */
n = rand()%(RHYTHMIC_PATTERNS);
k = rand()%ALTERNATES;
printf("we are using the mode %d for the entire piece\n",m+1);
printf("where 1 = C Scale with Eb, 2 = C Scale with Eb Ab Bb, 3 = C scale with F#, 4 = C scale with
Bb\n");
printf("the order of voices is %d \n ", j+1);
```

```
printf("where 1 is SAT, 2 is STA, 3 is AST, 4 is ATS, 5 is TSA, 5 is TAS\n");
printf("this is the %d  phrase, using %d  alternate\n",p+1, k+1);
printf("this is the rhythmic variations using %d\n", n+1);
printf("where 1 = WHOLE,DOTTED_WHOLE,HALF,HALF,HALF,WHOLE,WHOLE\n");
printf("and 2 = HALF,WHOLE,WHOLE,WHOLE,HALF,WHOLE,WHOLE\n");
printf("and 3 = HALF,QUARTER,QUARTER,QUARTER,QUARTER,HALF,WHOLE\n");
printf("and 4 =
HALF,DOTTED_QUARTER,EIGHTH,DOTTED_QUARTER,EIGHTH,HALF,WHOLE\n");
printf("and 5 = HALF,WHOLE,HALF,HALF,HALF,WHOLE,WHOLE\n");
printf("and 6 = WHOLE,DOTTED_HALF,QUARTER,QUARTER,QUARTER,HALF,WHOLE\n");
printf("and 7 = QUARTER,HALF,QUARTER,QUARTER,QUARTER,HALF,WHOLE\n");
printf("and 8 = QUARTER,EIGHTH,EIGHTH,QUARTER,QUARTER,HALF,HALF\n");
printf("and 9 =
QUARTER,QUARTER,QUARTER,DOTTED_HALF,QUARTER,WHOLE,WHOLE\n");
printf("NOW FOR THE NEXT PHRASE\n");
for (i = 0; i < PITCH_LENGTH; ++i)
{
play(0,144,mode[m][0][pitch_anakreontic [p]  [k] [order_of_upper_voices[j][0]]  [i] -1],100);
play(0,145,mode[m][1][pitch_anakreontic [p]  [k] [order_of_upper_voices[j][1]]  [i] -1],100);
play(0,146,mode[m][2][pitch_anakreontic [p]  [k] [order_of_upper_voices[j][2]]  [i] -1],100);
play(0,147,mode[m][3][pitch_anakreontic [p]  [k]                                [3]   [i] -1],100);
play((rhythm_anakreontic[n][i])*tempo,152 - 16, 100,0);
play(0,144 - 16,mode[m][0][pitch_anakreontic [p]  [k] [order_of_upper_voices[j][0]]  [i] -1],0);
play(0,145 - 16,mode[m][1][pitch_anakreontic [p]  [k] [order_of_upper_voices[j][1]]  [i] -1],0);
play(0,146 - 16,mode[m][2][pitch_anakreontic [p]  [k] [order_of_upper_voices[j][2]]  [i] -1],0);
play(0,147 - 16,mode[m][3][pitch_anakreontic [p]  [k]                                [3]   [i] -1],0);
}
}
```

As you can see, first the mode chosen is randomized, then the order of the upper voices, then a pattern of rhythm is chosen by random. Next the pitches from one of the tables is chosen, played, and there is a delay to hear the sounds, then everything is cut off, as it moves on to the next chord. After finishing the phrase it moves on to the next phrase in the same manner. This is a very simple program using combinatorics to produce music. We will see that in the next chapter that Mozart's solution to the musical dice game was not quite so elegant.

Kircher.c is in a folder.

CHAPTER SIX - MOZART'S MUSICAL DICE GAME

The Musical Dice Game of the Classical Period was very popular; Haydn and others tried their hand at them. Probably the most well known is Mozart's Musical Dice Game, because of piano teachers.

This algorithm is extremely simple. Mozart had a table of 12 different measures for each roll of the dice. The composition consisted of two phrases of 8 measures each, with a cadence at measures 8 and 16. Mozart made a table to consult which measure to use when throwing the dice, and it would have been simple to put the first twelve possibilities grouped together, then the second twelve possibilities grouped together, and etc. It would have been easier to use; however, it may have taken the mystery of the 'game' away, because each of the twelve possibilities of each measure have exactly the same chord function. I will analyze which functions he used shortly. However, it should be noted that some measures, for example measures 8 and 16, are exactly alike. If Mozart would have grouped them together as suggested above, the game player 'composing' the piece would have seen how Mozart constructed this little algorithm. It will be admitted that it is a well defined chord progression rather than a chord succession as in the Kircher algorithm, but functionally it is the same progression for every roll of the dice. Only the melody and some variations of chord inversion are different. One final observation – Mozart used 2 dice, and thus the probabilities for the sum are not equally weighted. Rolling a 2 is less likely than rolling a 7, for example. Either Mozart didn't realize that or did not care.

Here is the chord progression and the key of Mozart's algorithm:

The piece beings in C Major in 3 beats per measure.
The first measure is the Tonic Chord.
The second measure is the Tonic Chord in Root or First Inversion.
The third measure is the Dominant Chord in Root or First Inversion or even a Dominant Seventh.
The fourth measure is a Tonic Chord in Root or First Inversion.
The fifth measure is a Supertonic Seventh Chord in Third Inversion.
The sixth measure is a Dominant Chord in First Inversion.
The seventh measure has three chords in succession: the first chord is the pivot chord for a modulation. In the first key of C it is a Submediant Chord in first inversion, but in G Major it is a Supertonic Chord in first inversion – OR, for the first chord Mozart uses a Tonic Chord in the first key of C, but in the new key of G major it is simply a Subdominant Chord. This takes care of the first beat of measure seven. The second beat is not in G major and is a Tonic chord in Second Inversion which leads immediately to the third beat as a Dominant Chord.
The eighth measure is the end of the phrase and cadences on the Tonic Chord of G Major.

Mozart has used a very simple progression, but has cleverly modulated from C Major to G Major. Notice that Supertonic chord is a substitute chord for the Subdominant chord, and the Submediant Chord, especially in first inversion, is a substitute for a Tonic chord, so this is a very simple progression indeed.

Mozart must get back to the key of C Major in the next phrase, as all Classical compositions end in the same key that they began.

Measure nine begins in the key of G Major, and begins with a Dominant or Dominant Seventh in third inversion.

Measure ten is a Tonic Chord (still in G Major) in first inversion.
Measure eleven is the pivot chord to get back to C Major. It is a Subdominant chord in G Major, but a Tonic Chord in C Major. We are now back in the key of C Major.
Measure twelve is either a Tonic chord in second inversion progressing to a Dominant Chord, or simply a Dominant Chord the entire measure.
Measure thirteen is a Tonic Chord.
Measure fourteen is a Tonic Chord.
Measure fifteen is a Supertonic Chord in first inversion progressing to a Dominant Chord in the same measure.
And of course measure sixteen is the cadence measure of a Tonic Chord.

This is a very simple algorithm, and uses the cut and paste method to construct the measures. The composition is charming, but on repeated hearing it grows boring, since it is always the same progression of chords. Only the melody and rhythm gives variety. In the way of an algorithm, it does not contain the many possibilities that the Kircher algorithm did.

This is the last algorithm in this book that uses the combinatoric method of composition, which is very similar to the Markov method, as already discussed. The next group of algorithms will use the 'rule' method of composition, in which the rules are given for a certain style, and by using repeated random notes and many trials and errors, a composition is found that 'fits' the rules. This does not mean at all that this method of composition is simple. The algorithms for each style are unique, and much harder than the combinatoric method of music.

Mozart.c is is in a folder.

CHAPTER SEVEN – FOLK MUSIC

My first program after discovering how to convert a collection of sounds into a midi file was called folk.c as it was a program to compose simple folk music in a major key. To this day it is still one of my favorites, as it seems to come up with unique melodies coupled with an arpeggiated accompaniment, and a nice harmonic progression. Before discussing the algorithm, I would like to say that it took much thought and trial and error (on paper) before coming up with the "a-ha" idea that made the program a reality. I generally think what steps I do when solving a problem, and then writing those steps into a program. It did not work in this case. Most people come up with a clear melody, and then harmonize it. Although this may work, I am sure there are lots of subconscious procedures in the mind and volumes of ideas already in the brain that we are unaware. I know of the problems of Freshmen students coming up with a simple composition by the end of their first semester, and most if not all of the results are highly disappointing. There is no direction with the harmony, and the melody is incoherent and constantly changing in character and rhythm from measure to measure. It is literally thrown together and the students really have no idea how to compose at this early stage. So, then, how would I program a computer to come out with a nice and different melody each time, harmonized properly, and perhaps another accompaniment of a third instrument with a secondary supportive linear melody?

After a while, I had an "a-ha" idea, and though I was not sure it would work, I thought I would try it. It turned out to be the algorithm for this chapter.

The algorithm does not start with a melody, as a human would start

naturally. Rather, I thought, the music must have a good harmonic progression. This type of progression is defined in all theory books, and one of the best explanations was by Allen Irving McHose in the early twentieth century. The way I teach the harmonic progression to my college students is to start with the last chord, which is the Tonic chord. The chord that leads most often to the Tonic chord is the Dominant Chord, or the Leading Tone Chord in first inversion. The chord before that is Supertonic chord, or also the Subdominant Chord. Any of these chords may progress to to any of the following chords, in other words, the Subdominant chords may progress to the Dominant Chords or to the Leading Tone Chord, just as the Subdominant chord may progress to the same two chords. It was explained by Rameau in the early 1700's that because of the overtone series, the perfect fifth above the fundamental of any note is a generator of chord progressions a fifth down. Certain chords are 'substitute' chords, because of the similarity between two chords. For example, since the Leading Tone chord in first inversion is very similar to the Dominant chord, the Leading Tone chord *functions* as a Dominant chord. A Fifth above the Dominant chord is the Supertonic chord, and again, the Subdominant chord is a chord that is similar and *functions* as the Supertonic Chord. Using this line of logic, the chord a fifth above the Supertonic Chord is the Submediant chord, and the chord a fifth above the Submediant chord is the Mediant chord. This accounts for all of the simple diatonic chords of a Major scale. According to Allen McHose, a Tonic chord may progress to any chord. Then, if we did start with a Mediant chord, the natural flow would be from Mediant to Submediant to Supertonic or Subdominant, to Dominant or Leading Tone in first inversion, to Tonic. This is a *good* progression, and if all of music used only these progressions, it would seem boring, therefore it is important to surprise the listener and interrupt the natural flow and

occasionally use different chords in the progression. I do not want to get into a debate that Rameau was right or wrong in his ideas that the overtone theories account for the natural harmonic progression, or whether it was a cultural aspect of our music at the time. The fact is, that progression **was** used most often, as McHose did a statistical analysis of Bach's chorales and came to the same conclusion. I am sure there are other factors involved, such as our use of the major scale (and the melodic minor scale, which is similar), and the importance of the leading tone being only a half step away from the Tonic. These factors as well as cultural factors, and perhaps some of the ideas of the overtones, determined our harmonic progression from about the 1600's to the 1900's, and still today in Popular music, Jazz, Folk, Country, and religious music of contemporary churches (they generally use three chords only).

I realized then that a good folk piece must have some good progressions in the music. I knew it would end with a Tonic chord. Why, then, should I not write the progression first, and then add the melody (and accompaniment) to it later. Instead of starting with a chord, and trying to end with a Tonic chord, it would be much easier to start with the last chord of the piece, and write the good progression (with variations) backward. I chose to keep this composition simple and use only one chord per measure. Also, I would use the form AA'BA of 8 measures each per letter. This is a ternary type of form. I would split the 8 measures into two parts, an antecedent phrase and a consequence phrase, so that at the fourth measure, the music would cadence on the Dominant chord, and on the eighth measure the music would cadence on the Tonic chord. By cadence, I mean that the melody would be a whole note for each cadence measure (since I chose the composition to be in 4 quarter notes for a time signature. The A' means the same progression would

be used, and minor variations to the melody and accompaniment already established. The B section, true to a ternary form, would be a brand new section, and in this case written in the Dominant Key, with a fresh melody and chord progression. Then it would return to the A section at the end.

So, the first problem was solved. Write a 'good' chord progression **backward** from the Tonic on the eighth measure to the fifth measure, one chord per measure, and for variety, it is equally possible for a chord to move to another chord by stepwise diatonic motion. This is a stepwise diatonic progression that is used often in popular music. Thus, a chord may move in the diatonic scale to another chord 'above' or 'below' (thus the Third chord to the Four chord or to the Two chord), or use the 'good' chord progressions, already discussed. However, Leading Tone chord was left out, as it is fairly dissonant even in first inversion. All were given equal weight of probabilities for variety, as noted above. Then start on the fourth measure with the Dominant chord and write backward to the first measure. This was the main idea that solved the folk music algorithm. However, not starting on the Tonic chord was sometimes confusing, so the first measure was adjusted to the Tonic chord, as the Tonic chord may lead to any chord. In other words, no matter what the original chord in the first measure was determined to be, it was replaced by the Tonic chord. It was very important to choose the right type of progression depending on the style of music, and even the make up of the chords (whether they are suspension chords or the ninth chords, eleventh chords, or even thirteenth chords used later in the blues, jazz or Irish program).

It is now easy to see that even if I did start with a melody rather than

the chord progression, I would have to search for a good harmonic progression for the melody, which might not exist. I just turned the tables and started with the chord progression first. Thus, the melody had to fit the chord progression. I kept the rules simple for the melody. I kept the melody within a certain range so that it would not get too high or low, and started every beat on some note of the chord that I had just created from the above paragraphs. Of course that could be three choices. I let the melody only have a possibility of four notes per measure, but no more, however, much fewer are generally used. This was only making a skeleton melody based on the chord, as all notes of the melody were notes of the first, third, or fifth of the chord used. Later in the program non-harmonic tones (diatonic to the scale) were added, as one note higher than the harmonic note, or one note lower than the harmonic note. This process was repeated again to the non-harmonic tone of the melody note, And then the melody was sustained until a new note was reached. This technique generated a pretty nice melody.

There was an accompaniment to the chords and melody, which consisted of some type of arpeggiation of the chord. For a kind of unity, the measure was divided into eight parts this time (for eighth notes), randomly picked for each group of a four measure phrase which beats of the eight notes would be used, and also for each of those beats whether it would the the first, third, or fifth of the chord. On the some phrases, the beats would be picked sparsely, that is, only a few notes used, where in another phrase the beats would be more densely picked by random. Each phrase had its unique quality. Then the same non-harmonic tone technique was used similarly as was done to the melody, twice, first on the accompaniment note, then to the non-harmonic tone preceding it. Some of the accompaniment phrases without the non-harmonic tones would be

saved, and could be used in similar phrases for unity, with new non-harmonic tones. This simple technique generated a simple but satisfying composition that would prove to be better than the Freshmen could compose at the end of their first semester. It also had more harmonic direction than the Kircher program, and more variety than the Mozart program. It is for this reason that I prefer 'rule-based' algorithms rather than 'pre-composed' chains of melodies and chords or sonic structures'.

As this was my first algorithmic program, and I do like the results of this program, I have included as of the sixth edition a folk music program for the minor. It is called folk_music_minor.c The natural minor (aeolian mode) was used, and since the Supertonic chord would be diminished, it was not allowed in the progressions. I did not want to go through the extra code to make it in first inversion, when there is very little difference between the first inversion diminished Supertonic chord and the minor Subdominant chord in root position, and indeed, that have the same function. However, what would have been the leading tone chord was now the Subtonic Major chord. Emotionally, I enjoy the folk music minor to the major folk music program.

Folk.c and folk_music_minor.c are in a folder.

An added section on folk music for keyboard

As the folk music program was the first every written and seems to create fairly pleasing results, I decided to write it for keyboard, as it is sometimes impossible to play the above two programs as it is now written, for instrumental ensemble.

The lower sustaining chords are no longer in root position, but, for ease of playing with the least motion in the left hand, some of the chords have been changed to inversion, so that the least motion is necessary. This involved that addition of two additional functions, that would change root chord to the necessary inversions.

In addition, the middle voice was too distance to move to, from the lower sustained chord, therefore, the middle voice was moved in the same range as the sustained chord. Also, it was changed so that the first eighth note value or last eighth note value would be written in the middle voice, so that time is given to prepare to play the long sustained chord on the bottom staff. There are still there tracks, or staves, and a keyboard player would need to use the pedal to capture the sustaining chord, and then, play the middle voice.

I was surprised to find it also sounds as good on the organ, by merely holding the parts of the chord that can be held while playing the inner voice. So, a synthesizer, piano or organ will play the next two programs. Only the piano will not be able to change instruments.

The two new music programs are PianoMajFolk.c and PianoMinFolk.c are in folders.

CHAPTER EIGHT - IRISH MUSIC

Little needs to be discussed about this music, as it used the same algorithm as above, with two main differences. The melody is pentatonic, (the fourth and seventh degree of the major scale is omitted), and chords using these scale tones were altered. Thus the Supertonic chord, instead of using the 2^{nd} degree, 4^{th} degree, and 6^{th} degree of the scale, used the 2^{nd} degree, 5^{th} degree, and 6^{th} degree of the scale. Thus there is instead a “suspension” chord that allows the chord to be built without the 4^{th} degree (or 7^{th} degree). Similarly other chords are altered to leave out these tones. These minor adjustment to the algorithm is enough to give it an Irish flavor. Little else was done.

Irish.c is in a folder.

CHAPTER NINE - JAZZ MUSIC

This is actually the second algorithmic program I wrote, based again on the folk.c program. There are four important changes. The chords are not triads but 7th chords, of all five types:

1. The M7th chord should be a Major traid with a Major 7th above the root. At one time, this program had a 9ths, 11ths, and 13ths in some of the chords. However, since the melody is based on the chords, these have been removed for melodic reasons.
2. The Mm7th chord (also known as the Dominant 7th, or secondary Dominant 7th) is the second of the five important 7th chords.
3. The m7th chord is of course a minor triad with a minor seventh above the root.
4. The half-diminished 7th chord is a diminished triad with a minor seventh above the root.
5. The full diminished 7th chord is a diminished triad with a diminished 7th above the root.

The non-harmonic tones are carefully chosen by the preparation note before the harmonic tone and the resolution note after, as in the textbook use of non-harmonic tones. Also, modes were used both for non-harmonic tones of melody and accompaniment.

The modes are especially important as it gives a different 'flavor' to the piece. By the time of Jerry Coker, jazz theory had stabilized and evolved to the point that it was taught that the Dorian mode should be used with any minor seventh chord, the Mixolydian mode should be used with any Dominant seventh chord (major-minor seventh

chord), the Locrian mode should be used with any half-diminished seventh chord, the Ionian or major scale (or even Lydian) used for any major seventh chord, and the Octatonic Scale used for any full diminished seventh chord. The modal theory has not changed much since then. Thus, when adding non-harmonic tones not in the chord, the modes must be consulted for each chord type rather than the diatonic scale.
The progressions used not only use the standard 'good' progressions and the stepwise diatonic progressions, but chromatic progressions, which were not used in folk.c. Chromatic progressions generally fall down a half step to the next chord (in the normal forward progression). Thus writing the progression backward, the chromatic progression must move up a half step to the chord in the measure before it.

Jazz.c is in a folder.

CHAPTER TEN - BLUES MUSIC

This is a simple program that used some unique tricks. The blues.c program only uses the I, IV, V chords as a 12 bar blues; that is, I, I, I, I, IV, IV, I, I, V, IV, I, I. The chord used for the I chord is, from the bass up, C, Bb, E, A, which is a C Dominant 13^{th} chord. The IV chord is F, A, Eb, G, which is a F Dominant 11^{th} chord. The V chord is G, B, F, A, which is a G Dominant 11^{th} chord. Thus the piece is only 12 measures long, subdivided into 16 units (sixteenth notes). The first beat of each measure always begins with the root of the chord, and the rest of the upper voices of the chord occur randomly chosen on certain sixteenth note beats, but once chosen, always occur on the same sixteenth note beats. The skeleton melody is chosen exactly the same way as in folk.c music, with the exception that only two different skeleton melodies are produced, and those two melodies are used throughout the piece on various measures, each time elaborated by non-harmonic tones a half step away from the harmonic tone, differently for each measure.

This produces a pretty interesting blues piece, of course sometimes better than other times, and for some, the Dominant 11^{th} and 13^{th} chords make it more complex than the simple blues most people are accustomed.

The programs folk.c, irish.c, jazz.c, blues.c all used similar algorithms with important differences, but the next three programs are very different in nature.

Blues.c is in a folder.

CHAPTER ELEVEN – THE CANON (PENTATONIC TO POLYTONAL)

This program produces a canon of four voices of a scale determined by the user in the source file, or a random scale can be made. Canon means rule, so whatever is produced in one voice is reproduced in another voice at a time delay. This canon is set up in three sections, so that each voices gets a little closer each time (stretto). Each section is in a different key, and each section has a rhythmic accompaniment of three drums that increase in the density of a rhythm that is set up for each drum, repeated each time as an ostinato, however one drum uses a six beat pattern, another a seven beat pattern, and the final drum uses an eight beat pattern, so that it overlaps and comes together at times.

The scale used must be explained first. A short pattern of intervals are used, and the length of those patterns can be changed in the source file. Thus, for the pentatonic scale a Major second is used, then a Major third, then a Major second. This pattern is used repeatedly to build a scale throughout the entire range.

The melody is produced by using random notes from the scale in the following manner: silences, notes moving up or down by step, or even repeated notes.

As mentioned above, it is possible for the user to change the source code so that one of the notes of the pattern of intervals may be chosen by random. However, if the two patterns in a row do not end on the same note that it began, each voice will be in a different 'key', as they are exact duplicates but starting on a different note. The canon then becomes a polytonal canon.

After an introduction of the canon, the composition starts to become more interesting by mixing short fragments of earlier parts of the canon. This builds up to a climax and also introduces a certain element of chaos to the canon.

The canon does not use the conventional rules of counterpoint, so any interval between voices may be used, even if they are dissonant. This especially occurs in the polytonal canons. However, if the pentatonic scale is used, there is little that is dissonant and it is a 'correct' canon.

Canon.c is in a folder.

CHAPTER TWELVE - COUNTERPOINT
(16th Century Strict Counterpoint)

The algorithm for this program is quite complex. I thought about approaching it the traditional way - that is, using the rules of counterpoint and using a trial and error method of composing a counterpoint in three voices. However, as anyone knows about counterpoint, the rules are endless. There are fat textbooks full of rules for counterpoint. I taught counterpoint in the second semester of the second year to college majors, and since I had only four weeks to teach a crash course in it, I had to simplify it as much as possible yet make it a complete and correct counterpoint. I chose the R.O. Morris book "Harmony and Counterpoint", which used a few rules to teach 16th century in a small portion of that thin book. Even though I wrote out all the rules for the students, few students could really master it in four weeks. As for computer programs, the first electronic computer program was the Illiac Suite in 1956, which was a disaster as a work of counterpoint. I wanted something that would work as a student model – that is, it only had to have a Cantus Firmus of 12 measures long.

I spent many months trying various approaches to this problem, writing the pseudocode on paper, but it seemed the rules were endless. Finally, I found a solution. R. O. Morris said that there need only be two harmonies per measure, in the meter of 2 half notes. In the case of suspensions, really only one harmony would suffice. Without doing this harmonically, I did find a new approach that made the teaching of counterpoint much easier. First I used some simple rules for writing a Cantus Firmus, which was in the bass. I observed that the Cantus must start and end on the Tonic, and that the next to last next must be the Supertonic. The problem of too

many leaps in the same direction or moving stepwise after many leaps was solved by keeping the Cantus in a limited range. The tritone was of course avoided. Large leaps were also avoided. At this time the Cantus was only allowed to move in half notes.

After the Cantus was formed, the Alto part was added. Again only half notes were allowed, and all notes had to be consonant with the bass, and the 4^{th} and tritone not allowed from the Bass to the Alto. The first chord must be the Tonic chord, so only the 3^{rd} or 5^{th} above the bass was allowed. As always, consecutive parallel fifths and octaves were not allowed. The next to the last chord must be a leading tone chord in first inversion, since the Cantus is on the Supertonic. The last chord must be a Tonic chord, with the leading tone moving to the tonic. To simplify matters, the last two chords were not randomized.

After the Alto part was added, the Soprano part was added. Now the interval of the 4^{th} or tritone was allowed between the Alto and Soprano, but of course not between the Bass and Soprano.

At this point, there were two chords per measure, except the last chord, which was the Tonic chord.

To get to these two chords per measure, many different notes were tried in the Alto part for each interval, and repeated tries of different notes were tried for each chord of the Soprano. After so many tries there was a 'bail-out', in which case the Cantus was abandoned and a new Cantus was tried. This is a brute force method. It might have been better to back up a measure or so, and try different notes for the Alto and Soprano, but then that might not work, and again the program would have to back up again. Although this is what students

do when they get to a point that just will not work, that is, back up, I felt on obligation to keep the Cantus and within milliseconds the computer will find a Cantus with an Alto and Soprano that works.

Now that we have a series of chords, two per measure except the last measure, it is important to find some non-harmonic tones. The three types allowed in Strict Counterpoint is the neighbor note, the passing tone, and the suspension. The suspension is generally regarded as the favorite type of dissonance. Thus, in each voice, if a note was lower on the second beat than on the first beat, a suspension could be formed by simply tying the second beat over the measure to the same note, and then dropping it on the second beat. For now, the other notes would remain stationary on that second measure so that the suspension would resolve correctly.

After going through the entire piece making suspensions, the program started at the beginning and looked for neighbor notes and passing tones to add. Of course, any voice that leaped a third might have a passing tone added between the two notes, as long as certain conditions such as “quarter quarter half note” rhythm, or perfect fifths or octaves, did not occur. Any voice that remained stationary from one beat to the other might have a neighbor tone added to it, again if the above conditions were met.

This is a simplification of the algorithm I used to make a correct strict counterpoint in 3 voices. In the program itself, it shows step by step how the notes are being formed, and the notes added for a suspension, neighbor note, or passing tone. If there are parallel fifths or octaves, it will give a warning that it was not allowed because of those conditions. Thus the program itself is a teaching tool and explains in great detail the algorithm used to make the 16th century

counterpoint. Some counterpoints are better than others. All are correct. And they are much better than the students can do in four weeks of intense counterpoint lessons.

Counterpoint.c is in a folder.

CHAPTER THIRTEEN– MOTIVIC MUSIC (20th CENTURY)

My son has always enjoyed game music (such as Doom or Descent) or good movie music (such as the wonderful score from the movie “Inception” by Hans Zimmer). There will be a later chapter on game music, but this chapter is devoted to the more motivic music as would occur in 'movie' music.

I wanted to try to make a program that would be different each time yet incorporate some of the ideas of the 20th century as well as ideas from the medieval ages, the isorhythm and isomelody, the Tihai rhythmic principle from India, and the phasing of minimalistic music. For harmonic and melodic material, I also wanted to borrow from the 20th century. From the program I wrote, motivic.c, I extract the comments at the top of the program:

```
/*
This uses 20th and 21st century devices, as well as ideas based on Indian (Hindustan) rhythms,
medieval isorhythm,isomelody, parallelism, polychord with triad and quartal chords, chromatic third
relationships and atonal operations. Motives are 4 different notes by random except for one case where
the motive is extended to a fifth note. Atonal operations are performed on the motives; the inversion,
the retrograde, and retrograde inversion and of course transposition. In addition the motive may be in
mixed order in secondary motives. Chords may appear as be major or minor using 1st, 3rd or 5th of
chord using a common note of motive, if in bass, in open wide position; if above motive, closed,
narrow position. Chords may be related by real parallelism in which every note of the motive
determines the chord. Or chords may be in a chromatic third relationship with each other, with no
relationship to the motive except first chord. Chords may also be polychords with notes of one of the
motive forms being 5th of chord in soprano and 1st of other motive form the chord in bass. The
Polychords may be major chords (for brightness) or quartal chords. Chords may also be constructed
from the motive or any form of the motive. Silence and Unision Harmonies, esp. at the beginning of a
'section' should not be underestimated. 'Sections' are varied by mixing the order of different types of
sections randomly. A vary important part is the use of rhythm. Single generated drum beats may be a
part of the measure, or overlapping patterns of 5 beats with 6 beats with 7 beats with 8 beats, until all
beats come together, as in the Indian Tihai. The rhythmic ostinatos  change frequently.
*/
```

Thus the main idea comes from the motive, but it is altered by atonal techniques (or for that matter techniques of Bach in his Fugues). To put all this together, I made a large number of functions which I

name in the program as 'procedures'. Here are the procedures used in the program and an explanation by each of them.

```
void procedure1(void);      /* beginning of motive */
void procedure2(void);      /* short silence */
void procedure3(void);      /* motive modified */
void procedure3a(void);     /* motive with sustaining low note and secondary modified motive */
void procedure3b(void);     /* motive in sixths with sustaining low note and secondary modified motive
and drum rhythm */
void procedure3c(void);     /* motive in first inversion chords with sustaining low note and secondary
modified motive transposed higher */
void procedure3d(void);     /* motive with secondary motive in sixths with sustaining low note and
drums transposed higher still */
void procedure3e(void);     /* retrograde inversion in sixths with low sustaining note with secondary
motive and drums transposed higher still */
void procedure3f(void);     /* simple motive with drums */
void procedure3g(void);     /* simple retrograde with drums */
void procedure3h(void);     /* wide variety of techniques from above */
void procedure3i(void);     /* retrograde and motive together, one high, one low */
void procedure4(void);      /* longer silence */
void procedure5(void);      /* inversion motive */
void procedure6(void);      /* mixed motive */
void procedure7(void);      /* different mixed motive */
void procedure9(void);      /* addes new note to motive */
void procedure10(void);     /* plays only the new note */
void procedure11(void);     /* play a broken harp accompaniment melody under slow flute melody */
void procedure12(void);     /* minor chord put to rhythmic pattern repeated */
void procedure13(void);     /* series of chormatic third relationships */
void procedure14a(void);    /* four note motive combined with five note motive repeated make two
ostinatos out of phase */
void procedure14b(void);    /* same as above but faster */
void procedure14c(void);    /* same as 14a but with inversion */
void procedure14d(void);    /* same as above but faster */
void procedure14e(void);    /* same as above but with motive */
void procedure15(void);     /* motive and retrograde inversion together*/
void procedure15a(void);    /* procedure 15 but in  polychords (triads) */
void procedure15b(void);    /* procedure 15 but in polyquartalchords  */
void procedure15c(void);    /* procedure 15 but but in polychords (triads) again  */
void procedure15z(void);    /* long silence */
void procedure15_1(void);   /* motive  and retrograde inversion together */
void procedure15a_1(void);  /* polychords with 4 ostinato drums of different lenghts */
void procedure15b_1(void);  /* quartalchords with 2 ostinato motives of 4 and 5 overlapping */
void procedure15c_1(void);  /* basically the same as procedure15a_1 */
void procedure15z_1(void);  /* silence */
void procedure16(void);     /* same but with new note and its retrograde */
```

```
void procedure16a(void);    /* similar to 15a */
void procedure16b(void);    /* similar to 15b */
void procedure16c(void);    /* similar to 15c */
void procedure16z(void);    /* very long silence */
void procedure16_1(void);   /* motive and its retrograde */
void procedure16a_1(void);  /* similar to 15a_1 */
void procedure16b_1(void);  /* similar to 15b_1 */
void procedure16c_1(void);  /* similar to 15c_1 */
void procedure16z_1(void);  /* very long silence */
void procedure17(void);     /* two single voices mixed */
void subprocedure18(void);  /* making the atonal form from the motive, plus mixed notes of motive,
and the 5 note motive with the new note */
void procedure18(void);     /* rising with motive forms as parellel first inversion chords (parallelism)
with low sustained note of each motive   */
void procedure18a(void);    /* rising motive forms as solo melody with minor chord sustained as the
first, or third, or fifth note of the chord being the first note of the motive */
void procedure19(void);     /* good close of piece - all instruments on retrogradeinversion  slow and
loud */
void procedure20(void);     /* rhythmic section  with all drums playing overlapping ostinato rhythms
plus piano playing motive notes when osinatos come together */
```

Now it is possible to pre-organize the procedures so that they build in a certain way, but that would lead to much similarity from one composition to another. Or, like John Cage in his “Piano Concerto”, it is possible to call the procedures at random. The fact that motive is the unifying principle still gives the music consistency.

This idea of using 'procedures' in random order (which in themselves a random motive and its derivations are picked), is somewhat unique in the programs so far. How would such a program be analyzed as a piece of music. The music itself could be analyzed, but there are so many possibilities. This is the same problem I had when doing my dissertation on the 'Structure in Piano Music of John Cage'. It was easy enough to make a statistical analysis of pieces that were randomly composed, such as 'Music for Piano 1'. However, when the music itself was determined by means of chance operations with respect to performance, the music was not stable. It could vary

greatly from performance to performance. Thus I came up with the idea of a potential universe of the parameters of music, in which I found the limits of each composition, from the null set to the maximum potential of the piece. Thus pieces of indeterminacy could be compared. Some pieces had limited potential in terms of pitch, timbre, rhythm, etc. and others had much freedom.

All of these program principles I taught in third year college composition. I do know that this program certainly composes these techniques better than the students.

Motivic.c is in a folder.

CHAPTER FOURTEEN - MIRROR MUSIC
(MULTIPLE REFLEXIVE MUSIC)

I have never been satisfied with the Fractal music I have heard. Too much of it is similar to a mathematician putting Pi to music. The self similarity is not there, and the idea of octave equivalence is ignored. Simply taking a drawing of a fractal work of art and graphing it in some way to a music score does not portray the idea of 'fractal' in music.

I never intended to write a fractal piece. Instead, I was fascinated by a series of notes that would generate a new type of music. I was trying to interlock Major, Minor, Augmented, and Diminished chords into a series. Starting with B, I continued down in pitch, thus:

B5 / B5 / B5
Ab5 / G5 / G5
F5 / E5 / Eb5
Db5 / C#5 / C5
A4 / A4 / A4
F#4 / F4 / F4
Eb4/ D5/ Db4

As you can see, the B5 Ab5 F5 is an enharmonic Diminished triad.
Starting on the next note down, the Ab5 F5 Dd5 is a Major triad.
The F5 Db5 A4 is an Augmented chord.
The Db5 A4 F#4 again begins a Diminished triad.

I was curious if I combined all the notes of this series into one gamut of tones what the result would be. Here it is:

B5
Ab5
G5
F5
E5
Eb5
Db5
C5
A4 ----> Node
F#4
F4
Eb4
D4
Db

Now, studying the above series, you can see that if you start at the note A4 and travel up this 'scale', it will be the same as traveling down the scale in terms of interval content. A4 up to C5 is a minor third, and A4 down to F# is a minor third. Now something interesting happens. Traveling up the scale, when we reach E5 it is also symmetrical, however E5 up to F5 is a half step, and E5 down to Eb5 is also a half step, but not the minor third that we started with. So the scale is reflexive in two ways: the scale starts at A4 and travels up to E5 with a certain order of intervals, and then at E5 travels up with the reverse intervals. Thus the 'scale' repeats, though not on the same note, but at a transposition of the Node A4 a whole step higher.

Since each 'node' is a whole step higher, there will be one each a whole step higher, thus making all 'nodes' the notes of a whole tone scale. The important part is that now we can generalize this procedure to make any reflexive scale. Use a limited type of intervals (small intervals are better as will be explained later), and a limited number of intervals, perhaps 2 or 3 or 4. Now we can make

reflexive scales in the following way.

Let us use three intervals, a m2, a m3, and a M2, in that order.

We start with C4 and build up.

F#4
E4
Db4
C4

Now what do we do? The previous gamut of notes built up to a note, and then reversed the order, which would now be M2, m3, m2.

So now we have

C5
B4
G#4
F#4
E4
Db4
C4

Now we continue the process, back up a m2, a m3, and a M2.

F#6
E6
Db6
C5
B4
G#4
F#4
E4
Db
C4

And continuing on up, reserving the intervals, which again is M2, m3, and m2.

So now we have

C6 -----> node
B5
G#5
F#5
E5
Db5
C5 -----> node
B4
G#4
F#4
E4
Db4
C4 -----> node

This is generally unusual for each of the reflexive points, or nodes, to be the same note. Generally it outlines an augmented chord, or diminished 7^{th} chord, or a whole tone scale, or even just a tritone interval: anything that divides the octave equally. However, the above scale is reflexive, which means going down from a node uses the same intervals that were used going up from the node. Also going up two nodes have the same intervals in parallel motion. This is true of all reflexive scales.

I found later that Vincent Persichetti's Twelfth Piano Sonata is a 'mirror' sonata, with D4 as the note that divides the mirror between the left hand and right hand. Any black note in the right hand will be a black note on the left hand, and any white note in the right hand will be a white note in the left hand. This kind of symmetry is also possible using Ab as a dividing point. Although Persichetti was

aware of dual mirror chords (two different chords in the treble mirrored in the bass clef), he did not seem to pursue a mirror scale the length of the keyboard, that used only those notes in the mirror scale and no other notes.

The program I wrote uses a limited number of intervals and small interval chosen by random. The source code may be altered to make more or less of the intervals and kind. The smaller the number of intervals and the smaller the intervals makes more 'nodes' and makes the piece very 'dense'. Too large intervals and too many of them could have the capacity of making perhaps only one node, in case the composition would be 'sparse' and too boring. However, these parameters can be adjusted in the beginning of the program under the flowing define lines of code:

```
#define PATTERNLENGTH 3
#define INTERVALS 4
#define LENGTH_OF_PIECE 1000
#define SPEED_OF_PIECE 24
#define DENSITY_OF_NOTES  3/* 2 is max, 3 is less, etc. */
```

However, please feel to change these if you find too many simultaneous notes, or want more of a variety in the intervals used in the scale, or the speed of the speed slower or faster. All can be adjusted, but remember that each execution of the the program as above picks INTERVALS at random, though the pieces are not only different from composition to composition, but also a new symmetrical scale is generally produced each time. I chose a main melody that wanders by random up a note of the scale or down a note of the scale. Each time the 'melody' hits a new node, several nodes are chosen at random and the motion of the other melodies is either parallel with the main melody, or contrary, using only the notes of the reflexive scale. To ensure that it does not become too

boring, there is a random transpose factor each time new nodes are selected. Also all the notes occur at the same time. However, for rhythmic variety, the notes are given one unit of time or two units of time, chosen by random. Thus the rhythm may sound in seven eight or five eight, or even four eight. The listener may hear it in any way his mind structures it.

There is a related program in this chapter that gathers a few of the notes together and makes a series of chords. Combining that with the above, with the series of chords in retrograde, inversion, and retrograde inversion makes the composition more interesting.

The mirror music presented here could be the beginning of many interesting programs using the same program and concepts using micro-tones instead of half steps. Thus, the program would be reflexive to an astronomical amount compared to the 12 tones to an octave system we have now.

I presented this program to the 50th Anniversary of the Society of Southeastern Composers, as a lecture/demonstration/recital. It was also present at the University of South Carolina for the first electronic music concert.

Mirror1.c is in the folder of 'c' files.
The next version is with the atonal chords as discussed above.
Mirror2.c is in the folder of 'c' files.

The revised additions vastly adds to the type of symmetrical scales.

Mirror1.c was explained above. Mirror1_alt.c is different in that, instead of having one note in the ascending intervals of the

“PATTERNLENGTH”, the last interval used, instead of immediately descending to the previous interval, is used again. An example will explain this:

C ---> Node
B
A
Ab ---Here is the difference - after the half step from G to Ab, instead of moving on to the whole step, there, is a the half step just used to descend down to the reverse intervals.
G
F
E ---> Node

In addition, mirror2_alt.c is the same as mirror1_alt.c except that it also uses the serial chords as in mirror2.c

Mirror1_alt.c is given in folder of 'c' files.
Also, Mirror2_alt.c is given in the folder of 'c' files.

However, there is a second revision that is considerable. So far, all the scales have been constructed with one node at the beginning of the reflexive scale. However, what if the actual node is between two notes, at the quarter step? This will give entirely different types of harmonies with the different type of reflexive scales. The code that changes this is:

```
scale[scalecounter] = 1;
nodes[scalecounter] = 1;
 ++scalecounter;
scale[scalecounter] = 1;
nodes[scalecounter] = 1;
```

The difference is that there are **two** nodes, a half step apart. Then there is the symmetrical scale (that in itself is symmetrical) below and above the two nodes. This music is intense, as there will never be a single melody in less dense parts, but always two parts at the two notes using the symmetrical scale. Personally, I like this frantic orgasmic music the most, and with the version that includes the prime, inversion, retrograde, and retrograde-inversion chords added the most.

Ab -->node
G -- >node
E
D
B
A
Ab --> new revised extra note explained above in first revision (first version possible instead)
G
F
D
C
A --->node
Ab -->node

Although there are two nodes following each other, it produces different types of reflexive scales, Really, the true node can be though of as between the two nodes.

Since there are a pair of nodes a half step apart now, the code had to be changed so the lower node would be in contrary to the higher node that is a half step higher, and if one is 'active', the other is also made 'active' as well.
Mirror1_alt_alt.c has the second revision of two nodes a half step apart. Mirror2_alt_alt.c also includes the second revision of two nodes a half step apart, along with the prime, retrograde, inversion, and retrograde-inversion. Of course, it is possible to mix the first and second reversions, with or without the serial chords, and remember

that the parameters of Patternlength, Intervals, Length_of_Piece, Speed_of_Speed, can be changed, however, the Density cannot be changed in this version, since it is used to control both nodes a half step apart. This is explained in the code in the programs.

Mirror1_alt_alt.c is in the folder of 'c' files.
Also, mirror2_alt_alt.c is in the folder of 'c' files.

Although I have used the instrument 'piano' through all these mirror programs, I have done so because it is clear and percussive and the pitches can be heard with the rapid speed of the compositions. However, for an example, other instruments could be used. For an example of this, the code at the beginning could be changed to

```
static int channel[8] = {144,145,146,146,146,146,147,148};
program_number[0] =    47;  /* Timpani */
program_number[1] =    0;
program_number[2] =    0;
program_number[3] =    0;
program_number[4] =    0;
program_number[5] =    0;
program_number[6] =    0;
program_number[7] =    13; /* Marimba */
program_setup();
```

I have not rewritten the program in c for this in this book, but I have added a midi example of it as mirror2_alt_alt2.mid. I urge you to listen to it to hear the added color that it gives. Of course, any of the programs in this book could have different instruments by changing

program_numbers with the assigned midi numbers. Although you may see the program numbers of General Midi on the Internet starting with 1 for the first instrument, it is more common now for it to list the first instrument (piano) as 0. You need to subtract 1 from any of the numbers of the instrument list if it claims to start with a 1 to get the true number accepted and played by the computer. Unfortunately, the timidity I am using does not use all the instruments, so generally I have chosen instruments in this book that it does use.

All of the programs from this chapter are in a folder.

CHAPTER FIFTEEN - A RETURN TO RANDOM MUSIC

The first algorithmic piece of this piece was on windchime music, and the final algorithm returns to 'random' music with a vengeance. I wanted a truly chaotic random composition. However, in studying the music of John Cage, which was the topic of my dissertation, I learned that parameters of choice must be made on music. Even that famous 4'33" of John Cage still has parameters of what can potentiality be heard, with respect to timbre, pitch, amplitude, and duration. This being said, I still tried to make a very chaotic composition, and I am sure that John Cage would have approved, as he often said that anyone could 'compose' as he did, and the composition I constructed, or rather algorithmic program that produces a different composition each time, is highly chaotic as is John Cage's Concerto for Orchestra.

This algorithm uses 7 tracks of sound simultaneously. In each track, all possibilities of midi pitch, midi timbre, midi amplitude are used. However, in each track, the pitches can last in duration different lengths of time, all short very short lengths of time, but different from each other so that the pitches may overlap others in the different simultaneous tracks. After approximately 1 min and 30 seconds, the second movement starts, where amplitude of each note is generally much softer, however still random within the soft range. Also, in that second movement, the durations of the note-events are longer, however still random. Thus there is much less 'motion' in the second movement. It is approximately 1 min and 30 seconds also. The last movement is full amplitude, so it is very loud, and the durations of each pitch-event are shorter, though still random, than the first movement, so that it sounds totally chaotic.

The lesson learned from this algorithm, is that structure must still be imposed to make any composition. Choices have to be made in the materials of the music used in the composition, including the element of time. And in fact, in listening to the music, it is possible to hear relationships that are probably imposed by the mind.

The program r_music.c is in a folder.

CHAPTER SIXTEEN – TOTAL SERIALIZED MUSIC

The serial music of Schoenberg, Webern, Babbitt, and others can be fascinating. I find that not only in the analysis of the music, but on close listening of the music, especially of Webern and Babbitt, the experience is similar to looking at a crystal and turning it various ways, while enjoying the intellectual pleasure of 'viewing' the similarities and complexities of the object. Although the music of Webern and Babbitt are not 'plot-driven', as in a Beethoven Sonata, there is an emotional response from the pure intellectual beauty of the composition.

An excellent example of this type of music is seen in the extreme beauty of Anton Webern's 'Concerto for Nine Instruments, Op 24', written in 1934. This is a case where the rows used in the composition is from one 'derived row', in which a motive of three notes, labeled the prime form, in turn produces the other nine notes of the chromatic scale by the process of retrograde, retrograde-inversion, and inversion. As usual, one row is formed, which in turn, leads to 48 forms of the row by transposition, retrograde, retrograde-inversion, and inversion. Thus there is one row that is divided in trichords, each little mini row derived from the first three notes, which in turn allows that one row itself to form the usual 48 forms of the row, generally expressed in a matrix. The interesting aspect about some of these rows is that they may be all-combinatorial, meaning that the P form of the row may be combined with some R form, some RI, form, and some I form of the row. If the row is all-combinatorial at the hexachordal level, then the first six notes of the P row will have nothing in common with the first six notes of the other forms. Similarly, there may be all-combinatorial rows at the

trichordal level, so that a P row may be combined with some R row, a RI, row, an I row, and in each of the three note segments of four forms of the row, all twelve notes of the chromatic scale will occur. And this is true of each of the remaining trichords of the rest of the four forms of the row.

Although Webern did use rows that had a combinatorial row in the matrix in his compositions, he actually chose rows that had one or more notes in common when combining rows. Looking at the matrix of row in the above cited Op 24 of Webern, there are many interesting patterns at the hexachordal level as well as the trichordal level.

Babbitt took it one more step in 1947 with the composition 'Three Compositions for Piano'. Not only did Babbit use an all combinatorial row, but he also serialized the rhythm and dynamics. This type of music is often described as 'pre-composed', since it is so carefully calculated.

How may one serialize a rhythm? The row used in the next algorithmic program was calculated from one of the 6 source sets of Babbitt that can form all of the all-combinatorial hexachordal rows. However, the notes were selected in an order such that the row's first three notes are a basic set, and the other trichords are formed by R, RI, and I. The row is also all-combinatorial at the trichordal level. Now, giving the rhythm to the P form of the trichord of {1, 3, 2}, it is easy to see that the retrograde would be {2,3,1}. Figuring out the 'inversion' of a rhythm was solved by Babbitt in his first movement of the 'Three compositions for Piano'. Using a technique similar to his, the constant 4 was chosen, and subtracting the numbers of the P form, the inversion rhythm would be {3,1,2}. Then it follows that

the retrograde inversion would simply be {2,1,3}. Thus each trichord form is associated with the proper rhythm.
The row was itself was taken from Babbitt's source set 0,2,4,5,7,9 and the 'basic' trichord was {0,7,4}, which has a tonal reference to a major chord. The P-0 row is 0,7,4,1,10,5,6,9,2,3,8,11. Thus, the inversion of the basic trichord has a tonal reference to a minor chord. Incidentally, P-0 form of the 12 note row is trichordally combinatorial with I-1, R-0, and RI-1. In addition, the P-0 form of the 12 note row is also trichordally combinatorial with I-9, R-0, and RI-9. However, if confined only to those rows, there would only be three 4 note chords, which would repeat 4 times during the course of the 4 rows played simultaneously. Therefore, although the all combinatorial aspect is interesting, it would be boring to repeat, even with transposition. The order of the trichords of the P row is P, RI, R, I. I chose the tonal implications on purpose, such as in the music of Dallapiccola. Because of these tonal implications, and the invariance of common tones used in rows of Webern, this program uses 4 rows simultaneously, each a P, an I, a RI, and a R. However, each row is transposed randomly, and interesting invariance patterns will occur. Since there is so much structure not only in the row itself, but also at the trichordal level, the music will be extremely interesting in the patterns that emerge. Since the music is so complex, more patterns emerge with repeated performances of the same composition. In this algorithmic program, only one row was used. However, the combinations of rows are random, and in the comments inside the program, there are several other rows that are combinatorial at the the trichordal level that could be substituted for the row used in the program, with interesting results. I did not serialize the dynamics, as it was hard to hear all four voices. There I wanted all four voices to be of equal volume. It would be very easy, as Babbitt did, to assign forte to the prime level, piano to the

inversion level, mezzo-forte to the retrograde level, and mezzo-piano to the retrograde-inversion level, or some arbitrary combination similar to this.

A second program was written that did NOT involve tonal implications. The row was itself was taken from Babbitt's source set 0,1,2,3,4,5 and the 'basic' generator trichord was {0,2,1}. The P-0 row is 0,2,1,4,5,3,10,9,11,8,6,7. The 12 note P row is made up of the trichords P, R, RI, and I. It is all-combinatorial at the trichordal level with P-0, I-5, R-0, RI-0. The trichords do not use pointillism, and each are within a space of a whole step. Surprisingly to me, it sounds more interesting than the first program.

The comments in the program not only give possibilities of other rows, but also explain how the notes sustain and cut off with the assigned rhythms.

I have added a new program called serial.c. This program makes a totally random row of 12 different notes, Associated with the note is a unique rhythm as above, but rather than 3 rhythms for each of the trichords, there are now 12, one for each note. In the same way as the other programs, a row is chosen at random from the Prime set of rows, the Retrograde set of rows, and the Inversion set of rows, and the Retrograde-inversion set of rows. Each of the rhythms are calculated in the same manner as the two above programs. The music thus is again in four parts, all parts moving at the same time, which the respective rhythm of each row type determining when the note will be played. All four rows thus end at the same time, and then a new set of rows from the same matrix start. This happens for a number of times, and can be adjusted in the program. Where is the organization? The row itself, using the three other forms other than

Prime is really a sort of canon even by Bach standards. Thus, although the row was chosen randomly, there is great organization 'built into' each composition. In fact, this method will produce all possible set of rows, including the all-combinatorial as well as any other special types of rows, but it is improbable that a row would be produced by random means. This, although this program has more 'variety' in the row, it has less 'structure' and repetitiveness in the row.

The programs serial.c, total_serial1.c and total_serial 2.c are in a folder.

EXTRA

I have added two programs for the theorist and the composer. One is simply called row.c, and requires input. Read the instructions carefully when entering the row. You will need to enter the row using lower case hexadecimal numbers of P-0 (the instructions will make this clear in the program. I do not have error checking on, so if you make a mistake, you will get garbage. Otherwise, it will print out the entire matrix of the row you entered, to examine for compositional purposes as well as analytical purposes of a serial composition.

The other program is simply called atonal.c. It gives the best normal form, in terms of Allen Forte, the inventor of that system, to help identify atonal structures. Again, there are little error checking devices for input, so be careful, Otherwise the programs run perfectly. It is an ideal tool for help in analyzing atonal compositions as well as serial compositions for structure.
They are called row.c and atonal.c, and are in the same folders as above.

Chapter Seventeen – DIRECTLY PRODUCING WAV FILES

I think it is important in a book such as this on algorithmic music to also be able to include any type of sound, rather than just the midi sounds. Therefore this chapter is devoted to showing how to create the wav file format, and how, using numbers and formulas alone, to produce any possible sounds that can exist to the ear. Thus, going way beyond the overtone series (though this may be used in your construction to make orchestral sounds), we are now capable of constructing any sound possible.

When I was a young child, I use to listen to a phonograph recording and be fascinated by fact that the little wavy circular lines that the needle traveled in could produce such a variety of sound. It was my dream to use a microscope to analyze the ripples on the recording, and thus, make new ones and produce new sounds.

Without going to this trouble, we now have the means to do that just with numbers and computers. The program that I give in the programs that are included with this book has one that is called wav1.c. That program contains a model for producing the wav file format. All that is needed is to add the data, and I added a function that would count the data to tell the program of the length. The program is self explanatory with comments, however, without going into another textbook, I will simply say that many specifications for writing a wav file program are on the Internet, and I spent many months analyzing simple wav programs and comparing them to the not so readable 'explanations' of the wav file format. However, I did, in less time than it took to write my first working midi file, write the

program that will generate wav files. As I said, I do not need to go into detail as it is already given by the Specifications of the Wav File Microsoft. There are also more readable explanations, some of them now gone from the Internet.

Therefore, I will simply say that the first program, wav1.c not only contains the File Format that you may use, but also a method of generating single frequencies, or multiple frequencies. The first program uses sine tones, and therefore the math library will need to be included in the header. The mathematics of sine tones and music is also a textbook in itself, so rather than deal with all of it, I will simply say that the first program demonstrates how, using one complete period of a wave, may generate any frequency of any amplitude of any length. Unfortunately, when you change the frequency, it may also change the length of the wave and also the amplitude. Much experimentation will be needed.

Musical instruments all have overtones. By adjusting the amplitude of each overtone, you may simulate the sound of an instrument. There are complications, of course. When the note of an instrument is first produced, it is called the 'attack', and the overtones and even the fundamental changes suddenly at the beginning. Then there is the decay in which the fundamental and overtones are also adjusted. Thus it can be a complex business to simulate musical instruments. However, there are many sounds that also can be made not from musical instruments, and they are all possible through formulas that you make yourself, or even directly make a large table of numbers. Any sound is now possible.

One sound that is popular in game music and even modern music is noise. Noise that has a certain limited pitch range is a 'colored' noise,

such as pink noise, where the entire bandwidth would be white noise. Program wav2.c shows how to make a fairly pure sawtooth wave or to increase the noise level. The comments in the program tell you how to adjust it.

Just for fun, I wanted to know what sounds prime numbers would make. I made a chart of prime numbers up to the largest signed integer possible in 'c'. The program is in wav3.c. However, finding this boring, I used the table of prime numbers, and took the prime numbers of the index of the table to produce a smaller set of prime numbers – the prime index of the prime numbers. Still not satisfied, I took the prime index of the prime index of the prime numbers. That is what is used in the program to generate the sound. To me, it is fairly boring and you probably could do something better by other means.

In the 1940's, Music Concrete was a fairly hot topic. The roots go back to 'The Art of Noises' by Luigi Russolo in 1913, but Music Concrete refers more specifically to Electro-acoustic music and Tape music. In fact, when I was receiving my doctorate in music theory, I took 'electronic music', and I elected to use some studio tape recorders with the very wide tapes. There was of course feedback and loops, or cutting the tapes at a diagonal and splicing them with another tape at the opposite diagonal. In my project, I used my gold wedding ring (this was probably the only good use for it), and recording it spinning on a table, till it finally came to rest noisily. I slowed down the recording by means of another tape player, and again, and again, until it lasted the length of the recording. It sounded as if there was a gigantic manhole cover spinning, and when it finally came to rest, it was with a great reverberation. Along, as it was playing that very low sound, I also

had the ring speed up many times, so it was just a very brief high pitched sound, really indescribable.

The reason I bring this up in this chapter it that it is also possible to make wav files from recorded sounds. You can record a sound from your camera, or even capture it from a source such as YouTube, and if you have a free program such as Audacity, you may slow down the wave as much as desired, even without changing the length, or you may increase the length as the pitch is lowered. And of course you may do the opposite and combine the wave forms. In the end, you are left with a wave form and you can easily convert it to any type that you wish, such as an Flac or Ogg file.

Now, using the programs first described in this chapter AND programs such as Audacity, you are limited only by your imagination to produce any type of sound you wish for games or modern music.

The programs wav1.c, wav2.c, and wav3.c are in folders.

CHAPTER EIGHTEEN – Game Music

One of the most exciting types of music to evolve has been game music, increasing in popularity since series games such as Doom. It must now be taken as a serious musical art form, and concerts have been dedicated to some of the music from some of the bigger games. In fact, as of this writing, there have been a few of the more elite colleges, universities or conservatories that give game music courses. I see it has an opening to an exciting world of possibilities.

After studying tons of the most popular types of games, some principles are noticed. The music must not obtrude on the listener, but rather aid in the character of the game. This means that lofty long and developed themes as used in classical music just will not work. Music that can be well developed into a complex composition that is recognized each time distract the listener and come in conflict with the action of the game. For example, playing the first movement of Beethoven's Fifth Symphony would be ludicrous for not only movie music but certainly game music. What is looked for is something much more basic and shorter.

Typical game music does not last very long. A look at the tracks at most game music will reveal that they last on the average of two minutes. It can be less than that. In the game itself, there must be 'trigger' points in the plot that will start the music, and for example, when a battle is over, another 'trigger' point that will suddenly have that section of music come to a halt rather quickly with the game.

There is a passage written on his own music of Freespace2 in 1999 by Dan Wentz. He says that in most uneventful parts of a mission general ambiance is used. However, he then adds when talking of the

battle music, “depending on how well the player is doing and how long it takes him, the music will gradually switch to the higher, more intense tracks.”

This is a real gem of information. I have noticed that in the 'ambient' sections of most game music, it can be as little as a single chord, with a very slow melody over it. It is also true that there may be a minimal chord progression there, generally of four chords at most.

In the 'action' music of the game, I have also noticed that it is also short, and generally has two variations on the original theme. The theme itself will lend itself to a small variation in the second measure of the theme, or even a sequence of the theme on different scale degrees. There may also be new material. Most often, the themes are in the minor, or even a minor type of mode, such as Dorian or Phrygian. Also, most often the music uses the notes of the minor chord and as passing tones or neighbor tones, the second and fourth degrees of the minor scale. There is a tendency to also just use the first or in some cases, the last tetrachord of the minor scale for motivic material. It cannot be emphasized enough that the motive just does not just take, for example, the first or second tetrachord or even the first five notes of the minor scale in order, but rather, mixes them using the simple variation techniques. It should be, however, simple. We are now talking about the first part of the music of 'action' scene before it builds.

By changing the first five notes of the minor scale, or simply the first tetrachord, I do not mean changing it to the extreme. Simply making a theme from it, and perhaps changing some of the notes to the upper or lower octave while still staying in a small range is sufficient. The beginning of “Fundamentals of Musical Composition” by Arnold

Schoenberg shows exceptional ways to take a simple idea and change it by variation techniques, however, in game music, it does not need to be changed to excess. Remember that we are now talking about the music of the first 'action' scene.

Recapping a little, we know that we can use a long static chord with aimless mostly stepwise melodies (usually in the minor), or a minimum of two, three chords or four chords with the same pattern repeated.

We also know that the theme of the action should be fairly simple. There may be a well defined melody, and repetition is used often in game music with contrasting material. Most often, but not always, it is in a minor key. Intensification occurs by transposition of the main melodies. Again, the melody is generally shaped by the minor chord itself with passing and neighbor notes, and varied by octave displacement and simple rhythms.

The rhythms of the main theme should not be too complex. Most music is in 4 4 in a minor key. Rhythms generally employ strong beats. Anything too complicated distracts from the game, though the music must be as exciting as the game play, so battles will have faster note values with many percussive instruments, even though the rhythms are not overwhelmingly complex.

Of course, any type of thematic rhythm is possible, but I cannot emphasize strongly enough that from the many themes I analyzed in game music, that I am always surprised how *simple* they are.

So far, I have only been discussing the music that is in 4 4 time, but there are many themes that are also in 3 4 time. This is possible, as is

a major key theme, but not as often.

What is being done is something that can be repeated, and as the action events in the game are triggers, it can be built using more intense accompaniments and more background rhythms. From what I have heard in Mass Effect, you should have a main slow theme, that is capable of being intensified by a higher level of accompaniments (more on the details later) and background rhythms, and then a third very intense variation that has very busy accompaniments and background rhythms. The details will be given shortly.

However, in a lot of game music, they have borrowed Wagner's idea of a 'lietmotif', which is nothing more than a short musical idea of melodic, harmonic, and/or rhythmic figure that is associated with a specific type of event, situation, or important person. This theme, which is certainly the ideal already used in game music then should be used again in certain parts of the game, when it is appropriate. In movie music, everyone knows the 'Darth Vader' theme and how it signaled something ominous about him in the movie. Using a musical idea again when it is appropriate in the game will tie the game together. Also, the theme can have what is called 'thematic metamorphosis', so that the same theme can be modified by meter, tempo, chord changes, key, etc. This can also tie the game music together, and it happens quite often in the Mass Effect games. An excellent example of this in classical music is the 'Faust' symphony by Liszt, where the themes in each movement have undergone thematic metamorphosis.

Now, in game music, when the main theme and harmony are being intensified the first time, if the main theme is in quarter notes and

half notes, then a secondary background of the main harmony triad may be used in something smaller (possibly with the inclusion of the sustaining harmony chord). Thus, say the harmony chord is C minor, then the first variation or intensification of the chord would be to add broken notes in eighth notes of something like:
C Eb C Eb etc.
C D C D etc.
C G C G etc.
C3 C4 C3 C4 etc.

The second section of intensification could involve faster notes of the above C minor chord, where, if it had been eighth notes before, now it is sixteenth notes, and so on. Simple patterns that occur often is:

C D Eb G etc.
C C Eb Eb etc.
C D Eb D etc.
C Eb F G etc.
C G Eb C etc.
any type of variation on this or something similar.

The same is true of the rhythm background now. It is simply more percussion instruments and faster, but again is not polyrhythmic. It may have many layers of rhythm, but rather than overlapping 8 with 7 with 6 etc. patterns, as in minimalistic or Indian rhythms, it would use the same time pattern of say, 8 beats or 16 beats to the measure, and perhaps five layers of percussion, with percussion instruments being assigned each a pattern of attacks for each layer.

Also, it should be mentioned that the main themes may not only be intensified by the above means, but also by transposition, primarily up. Also a very useful device not only transposition by step when using a single harmony chord, but by the famous chromatic third relationship. Thus if you have a phrase that has a harmony primarily in C minor, that is very intense, if you then transpose it to Eb Minor or even Ab minor down can be very effective. The chromatic third relationship was used in the chapter 'The Motivic'. It was very common in the late nineteenth century music and is often used as a device in game music, even a progression in the theme itself.
I have written 6 programs to exemplify the more common type of game music. The first program, called ambient.c, is simply a minor chord that moves slowly by half steps, whole steps, or a minor third up or down in a limited range. The chords change slowly, and it is a simple program just for background music when little is going on in the game.

The next program, called ambient_ostinato.c, is a common device used in game music. It is the same as above except with a simple melody, in this case the notes of a minor scale without repetition and in random order, repeated with transposition while the chords move under it as explained above each time the pattern of notes repeat.

The next program, called ambient_intensity.c, is again moving chords with a background of fast moving notes, as explained earlier in this chapter, of a repeated pattern of harmonic and non harmonic tones of the scale of the chord.

Before explaining the next three programs, it is necessary to explain part of the algorithm of the programs. Instead of having one harmony per measure, as in folkmusic.c, I wished to have a number

of possible chord changes in one measure. Thus I set up a two dimensional array of rhythms: int rhythm[22][8].
One example of the array would be {1,1,2,1,1,2,0,0}. What this means is that in the above array, the total is equal to 8, but that there would be a melody note of one eighth, followed by another eighth, followed by one quarter, followed by one eighth, followed by one eighth, followed by 2 eighths. The harmony would be the same for the entire 'measure' or array. However, it is possible to have a 'half' measure such as {1,1,2,0,0,0,0,0}. In this case there would be one eighth note melody followed by an one eighth note melody followed by a quarter note melody. This would be half the length of the previous array, and would have one harmony for that half measure. There could indeed be a measure of {1,1,0,0,0,0,0,0}, in which there is only a quarter measure consisting of two eighth note melodies notes with one harmony per quarter measure. In such a case, you might want to have two half measures together, and either four quarter measures together or something like one half measure combined with two quarter measures. The nice thing about this type of structure is that you could indeed have an array such as {1,1,2,1,0,0,0,} in which case the measure would be in 5 8 time. In order to implement this type of array, and to repeat certain measures once the rhythms were assigned to notes, there had to be a variable called an index_number (which is the first number in the two dimentional array index. Thus in the array rhythm[5][8], the array is {2,2,3,1,0,0,0,0}. The index_number is 5. A function is created that calculates the rhythm_events, that is, the number of 'attacks' in the array. There are 4 events in the above array – 2,2,3,1. And of course that refers to the rhythms of the notes of the melody. There are, as noted previously, 22 arrays of rhythms. The last two are {8,0,0,0,0,0,0,0} so that there will be two types of measures of one note for the duration of 8 eighth notes, and thus the next to the last

measure may be the Dominant chord, while the final measure may be the Tonic chord. Also, there are a number of arrays reserved for half measures and quarter measures. All of the full measures (except of the full beat cadence measures discussed above, are randomized by having the index_numbers put in a randomized not repeated unique order. Then harmonies are assigned to them by a function that determines harmonic progression.

Gamemusic4.c is useful for well defined repeating phrases and contrasting phrases in the 'lietmotif' type of music, where it is not ambient and you may wish to associate a melody to a character or situation in the game. It is has a well defined melody and is not simply ambient. The first section, which consists simply of sustained chords and melody, is followed by the first type of intensification discussed previously, where a piano part adds broken notes in the background. The tempo is faster and everything is transposed up one half step. A second intensification follows, in which more instruments are added, the notes in the background are now in sixteenth notes as described previously also, but with other instruments. Also, rhythm instruments are added for effect, but they are added at random in the final section with no pattern or regular beat. Personally, I find that exciting, but since most game music does indeed include a regular pattern even in the percussion section, I re-wrote it so that the many layered percussions section had the same pattern beat to each measure, though the beat pattern was set up randomly.

The final program, gamemusicrepeated.c, is very simple but is typical of much game music. The first three measures are repeated, where the fourth measure is varied. This happens again, with a new fourth measure. It then ends on the tonic chord.

I can in no way do justice to all game music with these few programs, but hopefully the programs exemplify some of the principles of game music, especially the main theme, the first intensification of it, and the second intensification of it. Remember that in an actual game, you will repeat the first theme as much as possible, and if the action increases, move into the first intensification and repeat as much as possible (with possible transposition for interest), and again if needed move on to the second intensification of the theme. Then when the action is over, at the end of the phrase, the music may quit or you may have a musical passage that will instantly halt this music and play some ambiance chords until the next action scene.

I do hope that you convert your file to wav or something similar so that you may add the special effects that were covered in the last chapter.

All of the programs are in a folder.

CHAPTER NINETEEN – Just for Fun – Cellular Automation Music

I have never been impressed by any attempts to see Conway's famous 'Game of Life', which is really a 2D cellular automation program made with very simple rules, translated into music form.

From what I have seen, people never get it right. Even in their so-called fractal music, they get the 2-dimensional graph and time mixed up in an incredible jumble of bad ideas. This is an attempt to represent sounds accurately using Conway's Game of Life. As I mentioned earlier, the use of multiple reflective scales, in my mirror music chapter, is totally an invention of mine, over many years, and more accurately represents real fractal music.

Conway's Life game is uses very simple rules, and probably every student of computer science has to construct the game as an exercise. That is not hard to do. However, the Game of Life has proven to be Turing complete, with the advantage of seeing a miniature universe grow from little cells to larger form, and then, generally die. Since it is Turing complete, it is also possible to use it as a computer, and in fact with some possible set-ups of the cells, it is impossible to tell if the program will ever terminate – thus the Halting problem exists for it as well.

Any one not familiar with the Conway's Game of Life should look it up on the Internet before proceeding.

As I said, however, attempts have been made to set it to music. Most are simple ambient music that have nothing to do with the program, however, a few use some elements of the program to make some music. The problem is that The Game of Life is a 2-D graph of

pixels or characters, that change with every turn by the rules set down by Conway. Thus, we have a 2-D representation, as well as the time element.

I have represented the game of life as sounds in time, or perhaps even music in time. First, let me say that I had by necessity restricted the size of the board of the game of life, though it is unbounded in the fact that characters that goes off one the board on any side will appear on the opposite side, including the corners The total count of the 'cells' of the game are not lost by moving off the board. It is an unbounded board in a finite space.

There are a few depressing facts with midi; there are only 128 notes, but in midi, the very high can simply be clicks, and the very low sound like rumbles. Thus there is a practical limit, lower than 128 notes that the ear can hear as actual musical notes.

In the following programs, I have found a way, I think, to accurately represent the game of life in sound and in time, but it generally necessitates a small board.

The first program is small, and is called tinylifemusic.c.
The 'playing' size of the board is only an 8 by 8 matrix, where the first line of midi numbers are from 24 to 31, the second 32-39, the third 40 – 47, the fourth 48 – 55, the fifth 56 – 63, the sixth 64 – 71, the seventh 72 – 79, and the eighth 80 – 87.

I realize this is limited, but I did want to test it, and I was pleased with the result. You can accurately 'hear' the types of objects produced in Life. The stable objects will sustain, and rather than have each new change of the board be equal in time, the number of

cells were counted; the more cells active, the faster the board changed, and the less, the slower it changed – within limits. I used bells as the instrument in this game as it sounded 'delicate'.

I have done something in this program that I have done in no other programs in this book. I displayed the game as it produced the music, and when the game 'dies', you can press a key and it will end the program and close the midi file. In fact, I added a feature so that you can actually place the active cells of life at the beginning, or, you can set it up by random. To do this, it involves curses in the c library. I understand that it is available in Windows, but have never used it. It may also be available in a Mac, as the operating systems of Linux and Mac are closer than that of Windows. So, if you are compiling a program in Linux using GCC (that includes curses), as this one does, then the compile command is:

```
gcc  program_name.c -o program_name -lcurses
```

and then to execute

```
./program_name
```

This is good because you may terminate the program if it looks uninteresting, or if it dies too quickly.

However, I know many do not use curses or want to try to find it for Windows. So I have also made parallel program that does not use it. The disadvantage is that you cannot 'see' the Game of Life run, and you cannot input any characters. There is just the function that puts a certain amount in by random. Then, the program will run a definite number of 'turns', around 50, depending on the program. (There are others below).

So, since I wanted to have other operating system be able to run this program, I put a 1 after the program name. The name of the program above was tinylifemusic.c and used the curses in Linux. So, the 'parallel' program that has less options that will run in any operating system is tinylifemusic1.c, and it can be complied as any of the other music programs in this book.

The next program is lifemusic.c and is larger.
It is based exactly on the same principles as above. The playing area of the board is a 10 by 10 matrix, and uses the midi numbers from 12 to 111. Since some of these sounds don't sound so great in general midi, and I also wanted a sustaining instrument, I used four instruments for each pitch. It is capable of representing more shapes. There is the one using curses in Linux and the parallel program for all platforms, though more limited in choices. The program that is parallel to it but does not use curses is lifemusic1.c

The next program is largelifemus.c and needs some explanation. The playing matrix is 18 by 18, and I show the matrix with the pitches below:

{12,13,14,15,16,17,18,19,20,21, 20,19,18,17,16,15,14,13},
{22,23,24,25,26,27,28,29,30,31, 30,29,28,27,26,25,24,23},
{32,33,34,35,36,37,38,39,40,41, 40,39,38,37,36,35,34,33},
{42,43,44,45,46,47,48,49,50,51, 50,49,48,47,46,45,44,43},
{52,53,54,55,56,57,58,59,60,61, 60,59,58,57,56,55,54,53},

```
{62,63,64,65,66,67,68,69,70,71,    70,69,68,67,66,65,64,63},
{72,73,74,75,76,77,78,79,80,81,    80,79,78,77,76,75,74,73},
{82,83,84,85,86,87,88,89,90,91,    90,89,88,87,86,85,84,83},
{92,93,94,95,96,97,98,99,100,101,   100,99,98,97,96,95,94},
{102,103,104,105,106,107,108,109,110,111,
110,109,108,107,106,105,104,103},


{92,93,94,95,96,97,98,99,100,101,   100,99,98,97,96,95,94,93},
{82,83,84,85,86,87,88,89,90,91,     90,89,88,87,86,85,84,83},
{72,73,74,75,76,77,78,79,80,81,     80,79,78,77,76,75,74,73},
{62,63,64,65,66,67,68,69,70,71,     70,69,68,67,66,65,64,63},
{52,53,54,55,56,57,58,59,60,61,     60,59,58,57,56,55,54,53},
{42,43,44,45,46,47,48,49,50,51,     50,49,48,47,46,45,44,43},
{32,33,34,35,36,37,38,39,40,41,     40,39,38,37,36,35,34,33},
{22,23,24,25,26,27,28,29,30,31,     30,29,28,27,26,25,24,23},
};
```

As you can see, the notes 'wrap around' even within the matrix. To keep the notes separate even though they have the same numbers in many parts of the matrix, I assigned different instruments to each section that is separated, so that they would be heard as different, and also, the same note would not be turned off by some other 'event' in the game of life during that turn. This allows a larger matrix of more objects in the game, and yes, I can still hear the 'objects' of life with this program. I have also made the parallel program without curses and it is called largelifemus1.c.

I wanted not to end here, but try some matrices with pitches that

were just not chromatic. Although the chromatic scale is a nice way to represent the events, it is not the only way.

In a previous chapter, I had a program based on a random 12 tone row. It was called simply serial.c

I used that program to generate rowlifemusic.c
The serial.c functions produce a row, where the P-0 first note is 0, and the rest of P-0 is random. Then the matrix is a serial matrix, as described in the chapter on serial music.

However, the matrix here only uses a 10 by 10 matrix for the notes, with the lowest note being midi note 12. To do this, the matrix is similar to the range of lifemusic.c. Thus, instead of using the entire 12 by 12 serial row, it uses a subset of the 12 by 12 row; that is, a 10 by 10 row taken from the 12 by 12 serial row. Notice that each time executed, the P-0 row will be different, hence a different pitch matrix as well.

There is a parallel program without curses called lifemusic1.c.

And finally, the last set of programs are generated by my own invention of what I thought fractal music to be in a Chapter 14.
I use the program mirror1.c that produces a multiple reflexive symmetrical scale, as described in that chapter. And then, I use it for the pitches of the matrix in the life program, in a 10 by 10 playing matrix, similar to the row program, above. Since the scale is symmetrical, and thus leaves many notes out, that 'objects' in the life program 'sounds' at times more tonal, are at least with structures the minds will recognize more immediately, and also, the scales

generally needs to repeat in the matrix to make 100 pitches. The curses version is called fractlifemus.c, and the parallel version for all platforms with less options is fratlifemus1.c

All of the programs are at in a folder.

Chapter Twenty: Even More Fun - Maze Music

Long ago, before I wrote algorithmic computer music, I wrote some game and also, wanted to write a maze program that would solve itself.

There were basically two types of mazes I work with. One is a maze that has only one solution to any spot on the board, but a lot of dead ends. The other would have loops, so there are many solutions, but for a computer, it may be easy with a bad algorithmic to wander around and around and not find the goal. So, I wrote programs for both, and use them as screen savers, watching it generate a maze and then solving it, and the to repeat the process making a new maze each time.

The algorithm in making a maze is simple and probably has been rediscovered by many a computer student. Start at some point, and start to trace a path a random. When the path hits itself, you put up a wall there, and starting anywhere on the given path you have made, make a exit from that and again, wander until you hit one of the paths again. Soon, the entire board will be filled with paths with a solution anywhere but with only one true path.

Then, if you want multiple paths, merely take some of the path walls you have made, and make then empty spaces so that they become part of a path. Soon, you have many way to get to any place. There are many way to solve both mazes.

Since people lately are so interested in diverse ways of producing

sound-music, such as the game of life, I wonder how I could write music from a maze that was solving itself. My maze, after running down a path and finding a dead end, blocked that part off, and started over, until it found a solution. So, I noticed that many times it would run down the same path if that were part of the solution, or, if not, at some point distractedly change to a new path. At any rate, there was a lot of repetition and variation, what we are first taught is the basis of music.

The problem, just as in the Game of Life, is how to represent the maze so that you could actually hear what it was doing. I tried many times- tonal rows, melodies, even the more tonal twelve tone rows as in Berg. None was adequate and it was all boring. Finally, the simplest solution came to me after seeing one of the dumbest movies ever made. The vertical lines of the board would be a melodic minor ascending scale of two octaves. The horizontal would be a different instrument, but the same range, also using a melodic minor ascending scale of two octaves. I was surprised how much structure there was when I finally programmed it and heard it. I decided to make it more interesting by using the same device for rhythm as I did in the mirror music, that is, each pair of notes would be either an eighth note or a quarter note, by random. And again, this seemed to be in a time signature but always syncopated. Then, for variety, after the maze had started back at the home position (which was the lowest pair of pitches on the left upper side of the board) a number of times, I would use a transpose factor and change keys. If I did this five times or so, generally the maze would be solved in that time. The goal, by the way, was on the lower right side of the board, which was the highest pair of pitches. Thus, you could actually hear it solve the maze, and hear the repetition and variety when it changed to a new path. It was much more interesting than I thought it would be.

It would be very interesting for you to see the maze in color, and of course, you would need curses for that, and I am not sure how easy that is to include in Microsoft or Mac. With Linux, if it curses is not with the gcc distribution, then you can download it using the software manager. For the people that do not have this option, one of the version using no color, and you can step through it step by step by hitting the enter key. It will take a long time but you will see how it works. And then, I also made versions of both mazes that use no display, so it will not be necessarily to bother with curses there. You just run the program and the midi files are generated.

So, there are a number of programs included here in the folder, and the requirements are at the top of the code (whether you need curses).

After I got the program running, it was much more interesting and melodic than I would have thought.

All of the programs are in a folder.

Chapter Twenty One: Reverse Music

If these links are still good, please watch them.

https://www.youtube.com/watch?v=l5PI9jvtu5E
(The second movement of Beethoven's Seventh Symphony transcribed to roller piano, with the roll being reversed!)

https://www.youtube.com/watch?v=WhfRs6NQovA
(The first movement of Beethoven's Fifth Symphony transcribed to roller piano, with the rolls be reversed!)

I do not know if these links will stay on YouTube. I found it quite by accident. But it answered a question for me, which I asked myself as an undergraduate student in music. In fact, I pondered the question, trying to make the question more exact, as I would have written an article or even had made it the study of a dissertation later. I never answered it fully, but now, it is quite clear.

In traditional music, frequently the melody is upper voice. The accompaniment of harmony is mostly on the bottom, and then, of course, the roots are determined in triads and seventh chords as the lowest even number above the bass (Prout in the 19th century came up with this mostly forgotten rule). At any rate, the harmonies are generally, and I say generally most liberally, as I know there are many exceptions, especially in polyphonic music, and other works of classical music where the themes are not in the upper voice, but any voice. I am just staying for the most part, we hear the melody and simple folk songs with the it being in the upper voice, and the

harmonies as heard in lower voices, and when the chords are in root position, the root is on the bottom.

My question is not only why would have be, be could there be another type of music where the melody is on the bottom, and the chords are melodically inverted (more on that below), and with this 'reverse' music, there would be a one to one equivalence to the original music.

The piano rolls above lead me back to that question, after so many years. And in so doing, I revised the very first algorithmic music program I wrote so many years ago, as to make reverse music. music.

Before I explain the detail of how to reverse it, it will say that the problem is now solved. What was the first real music problem that was unanswered for me is now answered, proved, and I have a program that now generate reverse folk music, in the same way that the piano rolls did.

It is so simple what they did in the YouTube video. Taking an ordinary piano roll, they simply flipped it over, and all the notes were reversed. The results were surprising, and now there is a new grammar, new progression, and the melody reversed from what it would have been. Also the melody is in the bass, and the harmonies are in the upper regions. Any special accompaniment of the harmonies as arpeggios are in the middle range.

So, before explaining the theory of what happens, I can now state that no, there is not a one to one equivalence, but yes, there is a one to one correspondence!

The first of the YouTube pieces was originally in a minor key, and in a very slow range. The chords were very low. In reversing the piano roll, it changed to a minor key, and of course the low chords were now in the upper regions. As the minor is sometimes associated with sad, what resulted when reversing it was a happy piece, very high at the beginning, almost heavenly sounding. The first is the second (very depressing in the original, minor key, and very low pitched), and here it is on YouTube, taking the reduction of the piano roll, reversing it, so that the intervals go truly up, no adjustment like I do with the keys to make it sound 'better'.

Listen to the second piece. Fifth Symphony inverted the same way, and just as exciting, and many more so. And certainly ends correctly. There are lots of sevenths chords, but they are changed. What I can say is that this piece is just as exciting and perhaps, with the different grammar of chords and function, a new type of tonal music. This is fascinating.

So, let's take the folk.c piece in detail and make a reverse piece of it.

The new program is called reversed_folk.c and only needs one extra function, on line 237 called make_reverse_layers(), and all it does is subtract the midi number of the original folk note from 127. (In the case of sustaining notes, and silences, extra code is added). This will effectively reverse the piece just as the piano rolls. So, what should we expect? In the original, the chords were used to make the melody, and the used the chords I, ii, iii, IV, V, vi. (vii0, because of the tritone in making the melody, was not used).

Reversing the chord is taking the melodic inversion of the chords,

(not the harmonic inversion). The result is iv, bIII, bII, i, bVII, bVI.

Had I used a vii0 in the folk music, the melodic inversion would have been v0.

Now, assuming that the first key of the folk music was in C, then the new chords, if we use the bottom root, would start on F, in fact, F natural minor. Thus the progression is changed to: i, bVII, bVI, v, iv, bIII, ii0, i. Thus, exactly the chords of F natural minor. So that is why the major reversed sounds in a natural minor key.

But hold on. We were asking if there could be a music where everything is reversed? That would assume that the root of the chord, in thirds, is the top note. If that were the case, the resultant scale scale would be C Phrygian. This makes a difference, not a lot, but some. If I used seventh chords of a major scale, the reverse scale would change from Lydian if the roots are on the bottom of the chord, to Phyrgian if the roots are considered on the tops of the chords.

However, we are conditioned to hear roots at the bottom of a root position triad, and this could be the result of overtones and our ear. The question still remains, though, if we could possible, if never having heard the tonal music, but rather the reverse tonal structures, if we would not hear the top of the root position chord as the true 'root'.

What ever the case, we can see that the reverse_folk.c music produces pleasant music, even if it is in the minor key or a minor type mode, and the progressions now are typically plagal sounding

harmonic progressions. What has been demonstrated is that this new system of music actually works, and is akin to the tonal music.

The program reverse_folk.c is in the folder.

CHAPTER TWENTY TWO – Future Directions

Except for the notes in Chapter 23 for compiling, playing the midi files, and other helpful programs, we have come to the end of the book.

I would like to discuss possible directions that algorithmic music and what the this chapter, if gets written, may be. Most of the programs in this book have been from a set of rules, not fragments pieced together of previous composers or a statistical analysis of a composer and reconstruction of his works from the analysis.
The only music in this book that as been from little bits and pieces and then stuck together are the historic algorithms of Kircher and Mozart. I am going to go out on a limb here, and give a strong objection to the current tendencies of the algorithmic music using Markov chains. Using fragments of a composer, and then putting them back together in different ways, is not creating new music, but stuck in a stylistic period of the past, and is creating anything new. Most of the examples I have heard of big pieces using this method have recognizable fragments, and it is simply a hodgepodge of the composers works. I do not see this as creation, but just stuck in the past.

I believe that it would be possible to generate unrealized styles of music based on rules, just as different branches of mathematics are based on different or additional axioms.

A very ambitious project would be to have a program that generates a set of well formed rules, for whatever parameters of music, and use those rules to algorithmically compose music never dreamed of

before. This would of course be all inclusive, and thus the programs in this book would be a possible subset of such a program. But it would extend for beyond, and indeed, if the enjoyment of music comes from understanding and listening to the structure of music, this is just the beginning of untold types of music. I may attempt part of this ambitious project in the future, but for now, it is an ideal to think of: a program that generates algorithmic programs. Why not.

Also, these small little programs of this book that produce music, in my opinion, very well, could be extended, just as the early chess programs were extended to beyond the capabilities of the grandmaster of chess. In fact, the folk music programs and the jazz music program, as well as the counterpoint programs far exceed the compositions of Freshmen and Sophomore music theory students, and the more advanced programs that use 20th century techniques, such as the serial chapter or the motivic chapter, exceeds third year college students in more advanced composition. These are small programs. With some improvement and enlargement, they could possibly far exceed many of today's movie or game composers. There is no reason to fear algorithmic music; it is a tool that may be used, and the composing of music has passed on from the composer to the composer/programmer.

CHAPTER TWENTY THREE – Notes on compiling and converting

If you wish to compile the programs yourself, it is important to include the header file “convert3.h” in the same directory as the programs, as each program depends on this header file. If you use a command line to compile the program, you should consult the compiler you are using. In Linux, the command for compiling the program “folk.c” is:

gcc folk.c -o folk

which in turn creates an executable file called folk.

To execute the file, use the command:

./folk (period forward slash folk)

and a temp.doc file and music.mid will be written to your directory.

In Linux, you can currently use timidity to convert a mid file to wav. Again, you will need to use the another command line command. At present, it is:

timidity input.mid -Ow -o output.wav

If you like the music the program has generated, you may rename it so that it will not be overwritten the next time you run another algorithmic program (or the same program).

Compiling the programs that produce wav format files are explained in each program. Also the chapter “Just for Fun”, contains the

directions to compile 'c' programs that require curses.

Audacity and Sound Converter are both programs capable of changing wav files to other types of files, such as mp3 or ogg. Also MuseScore is an excellent program for changing the folk music programs, and others, to score format. Of course, all of them are free.

The future of music IS algorithmic music. We can talk about it or do it, and I would rather do it. This being said, I think I will go play a game of NetHack.

Good luck with your programs.

Dr. John R. Francis
May 18, 2016