Universidad Autónoma de San Luis Potosí.

Facultad de Ciencias

Generating music by fractals and grammars

TESIS

Que para obtener el Grado de

Maestro en Ciencias Aplicadas

P R E S E N T A :

Ing. Francisco Alfonso Alba Cadena

ASESORES:

Dr. Gelasio Salazar Anaya,
Dr. Jesús Urías Hermosillo.

San Luis Potosí, S.L.P. Julio de 2001.

To K, who makes things happen.

Acknowledgments

I would like to thank the following people, whose support and inspiration made this possible:

- My parents, for everything.
- My friends and family, for believing.
- Dalia, who understands like no one else.
- Gelasio Salazar and Jesús Urías, for their guidance.
- Dr. Ricardo Femat, for being a tough one.
- Everyone at the IICO, for their support.

The following people helped me a lot with the development of this project. Thank you.

- Omar Montaño Rivas
- Alexis Glass
- Kevin Breidenbach
- Kristopher W. Rhen
- John Close
- Phil Jackson

Contents

Chapter 1

Introduction

Music, as many other phenomena, shows a combination of order, disorder, and recurrence. A musical piece is usually made of similar patterns or phrases in order to give the composition some structure, but it also contains a certain degree of unpredictability to keep the listener interested. These properties can be modeled by mathematical tools such as chaotic dynamical systems, fractal sets, and grammars. Some restrictions can be applied to hold the most basic principles of music theory, but those restrictions can also be bended in order to produce experimental musical pieces.

This work focuses on the melodic and rhythmic structures of music and presents algorithms that can be used to develop new music generation tools. However, the intention of this project is not to replace a human composer, even though full musical compositions could be generated by the methods we present.

1.1 A mathematical approach of music

There are many well-known relations between mathematics and music: the exponential proportion of note intervals, the strong quantization of rhythm, and the modern sound synthesis methods are just a few examples of these relations. And still, vague concepts such as "inspiration" and "artistic performance" are what separates music from a random sucession of notes and chords. It is very hard, if not impossible, to model inspiration and creativity with mathematics. And it becomes even more difficult as new musical styles and genres are created, blurring the line between what's musical and what's not.

It is necessary to forget such concepts in order to generate music from a mathematical approach. In this work, we redefine "music" as sequences of notes that hold certain restrictions in order to resemble common musical structures.

The imposed restrictions will derive from the western music theory. We will use deterministic and probabilistic methods to generate one or more sequences of notes according to certain parameters. Those parameters will determine many aspects of the resulting sequences and will also be used to obtain compatible sequences; that is, sequences which can be part of the same composition.

1.2 Goals

The main goal of this project is to develop a set of algorithms that can be used to generate music. To achieve this goal we first have to show the relations between music and mathematics. We do this by defining the most common musical terms in a mathematical language. Those definitions will be our basis to construct algorithms that separately model different aspects of a musical sequence.

A computer program has been written to implement those algorithms. As a secondary goal, we will determine some of the potential applications of this project according to the responses of a few users.

1.3 Sound and music

In 1941, Joseph Schillinger published *The Schillinger System of Musical Composition*, an attempt to scientifically analyze music through mathematics. In this work, Schillinger mentioned an issue about the possible ways to analyze music:

> *“There are two sides to the problem of melody: one deals with the sound wave itself and its physical components and with physiological reactions to it. The other deals with the structure of melody as a whole, and esthetic reactions to it.*
>
> *Further analysis will show this dualism is an illusion and is due to considerable quantitative differences.”* [3]

This means that one can generate music by two distinct approaches. One consists in generating an audio signal, which usually involves digital signal processing methods. This approach is commonly used in sound synthesis and it can lead to very strange results. A representation of music as a digital signal is called a *low-level representation.*

The other approach consists in generating a stream of notes with quantized frequencies and durations. Instead of generating the actual sounds, we generate the score of a musical piece. This is known as a *high-level representation* of a composition.

Each approach has its advantages and disadvantages. The low-level approach allows the generation of complex sounds and noises. Music is produced by quantizing the frequencies of the generated waves and one has complete control over the actual output signal. The downside is that low-level representations require a huge amount of information and processing power, and it could be very hard to model certain types of sounds (for example, an acoustic piano sound). A high-level approach is usually (although not necessarily) more restricted in terms of frequency and time; that is, the allowed frequencies and note durations are heavily quantized, making it hard to produce unusual sounds. However, it is easier to model a musical piece as a sequence of notes, instead of a sequence of digital samples. The resulting sequence of notes is feeded through a sound synthesizer which produces the actual sound. The main advantage is the separation of melody and sound, which gives more flexibility to the composer.

According to Schillinger, both approaches are equivalent. Because of the advantages mentioned above, we will choose the high-level representation methods. Thus we need to construct mappings from mathematical objects to musical notes of the chromatic scale. To obtain an audio signal we must pass the note data to a synthesizer. The MIDI (Musical Instrument Digital Interface) standard is the best way to accomplish this since it is supported by most of today's electronic synthesizers. MIDI is described with further detail in Appendix A.

Chapter 2

Definitions

In this chapter we will define the necessary elements which will be a basis for our algorithms. As we have already mentioned, we will consider a musical composition as a sequence of notes which holds certain restrictions. Some of those restrictions derive from the definitions in this chapter. Other restrictions will be imposed by the algorithms themselves.

2.1 Scales

The basic element from which musical sequences will be constructed is the note. However, the frequency of a note is taken from a discrete, ordered set that has some important properties. Hence we start by defining these sets and their properties.

2.1.1 Basic scales

Suppose we have a piano and draw an integer number on each key. We draw a zero on the middle C key, a one on the following key to the right (C-sharp) and so on. Similarly, The B key on the left of the middle C will have a -1 on it, so the numbering will decrease to the left. If the keyboard was infinite on both sides, we would obtain a bijective mapping from Z to the discrete set of available note frequencies. Due to the limitations of the human ear, we will later restrict this range to about ten octaves, where one octave contains twelve adjacent note frequencies, but for now and until such restriction becomes necessary, we will assume an infinite range of frequencies.

A scale is a subset of the frequency range such that if a frequency f belongs to the scale, then the corresponding frequency one octave up $(2f)$ also belongs to the scale, and viceversa. Since it's much easier to work with integers instead of frequency values, we will use the mapping mentioned above to formally define a scale.

Definition 2.1 A *basic scale* $\mathcal{E}$ is a subset of Z with the following properties:

a) $\mathcal{E} \neq \emptyset$,
b) $0 \in \mathcal{E}$, and
c) $x \in \mathcal{E} \iff (x + 12) \in \mathcal{E}$.

Here are some examples of common scales:

a)	$\mathcal{E}_C$	$=$	Z	(chromatic scale)
b)	$\mathcal{E}_M$	$=$	$\{\ldots, -1, 0, 2, 4, 5, 7, 9, 11, 12, \ldots\}$	(major scale)
c)	$\mathcal{E}_m$	$=$	$\{\ldots, -2, 0, 2, 3, 5, 7, 8, 10, 12, \ldots\}$	(minor scale)
d)	$\mathcal{E}_P$	$=$	$\{\ldots, -5, -3, 0, 2, 5, 7, 9, 12, 14, \ldots\}$	(pentatonic scale)

2.1.2 Generating Octave

Definition 2.2 The set $O_{\mathcal{E}} = \{x \in S \mid 0 \leq x < 12\}$ is called the *generating octave* of $\mathcal{E}$.

It is easy to see that a basic scale can be obtained from its generating octave:

$$\mathcal{E} = \{x \in Z \mid (x \bmod 12) \in O_{\mathcal{E}}\}.$$

This is useful because we only need to store up to twelve numbers to reconstruct a full scale.

It will be also useful to know the number of note frequencies $n_{\mathcal{E}}$ in the generating octave of a scale; that is $n_{\mathcal{E}} = |O_{\mathcal{E}}|$. For simplicity purposes we will be referring to this $n_{\mathcal{E}}$ as the *number of notes* of $\mathcal{E}$. For example, major and minor scales are 7-note scales while the pentatonic scale has 5 notes.

2.1.3 Tone scales

In music, scales can be transposed; this is equivalent to adding a constant number to all the elements of a basic scale. The constant number to be added is called the *root* or *tone*. It is clear that if two roots τ_1 and τ_2 are modulo-12 congruent, transposing a scale by τ_1 will produce the same results as transposing by τ_2. Thus we can restrict the root values to be in the range $[0, 11]$ without any loss of generality.

Definition 2.3 A *tone scale* is a pair $\mathcal{E}(\tau) = (\mathcal{E}, \tau)$ where $\mathcal{E}$ is a basic scale and τ (the root) is an integer such that $0 \leq \tau < 12$.

When constructing a melody, we will take note frequencies from a scale $\mathcal{E}$, but those frequencies will be transposed by a root τ . A full musical composition will be constructed from different melodies, but all the melodies will be constructed from the same scale and the same root. This does not always happen in music but it usually leads to better results according to the laws of harmony in music theory.

We can assign symbols to the different values of τ in order to express more clearly the relation between tone scales and music theory:

Value of τ	Musical Symbol
0	C
1	$C\#$
2	D
3	$D\#$
4	E
5	F
6	$F\#$
7	G
8	$G\#$
9	A
10	$A\#$
11	B

Thus we can refer to scales such as $(\mathcal{E}_M, D)$, the scale of D Major, or $(\mathcal{E}_m, A)$, the scale of A Minor.

2.1.4 Indexation

Let $\mathcal{E}$ be a scale. There exists a natural bijective mapping $\phi_\mathcal{E} : Z \longrightarrow \mathcal{E}$ such that

$$\begin{array}{ll} \text{a)} & \phi_\mathcal{E}(0) = 0, \text{ and} \\ \text{b)} & a < b \Longleftrightarrow \phi_\mathcal{E}(a) < \phi_\mathcal{E}(b). \end{array}$$

Suppose $\mathcal{O}_\mathcal{E} = \{o_0, o_1, o_2, \ldots, o_{n_\mathcal{E}-1}\}$ is the generative octave of $\mathcal{E}$, where $o_i < o_j$ if $i < j$ for $0 \leq i < n_\mathcal{E},\ 0 \leq j < n_\mathcal{E}$. It is easy to see that

$$\phi_\mathcal{E}(x) = 12\lfloor x/n_\mathcal{E} \rfloor + o_{x \bmod n_\mathcal{E}}$$

holds both conditions exposed above since $0 \leq o_i < 12$ for $0 \leq i < n_\mathcal{E}$.

The mapping $\phi_\mathcal{E}$ is called the *index mapping* of $\mathcal{E}$. This mapping will allow us to transform a sequence of integers into a sequence of note frequencies within a scale. Finally, we will denote the i-th note of a scale $\mathcal{E}$ as $\mathcal{E}[i] = \phi_\mathcal{E}(i)$.

2.2 Notes

In music theory, the word "note" usually refers to two things: the main frequency of a sound and the duration of that sound. The latter is represented with different symbols placed in a staff. Those symbols specify distinct multiples of a determined unit of time (the beat). The frequency, on the other hand, is not measured in hertz. It is taken from a discreet set (usually the chromatic scale) of frequencies. In a staff, the frequency of a note depends on the vertical position of the note symbol.

There are other properties which can be written in a musical staff, but not for single notes. Two of those properties have the purpose to add expression to a musical piece. One of them, the *legato*, specifies how tied consecutive notes are. Legato means that the sound of a note will continue until the next note is heard. Stacatto, the opposite, means that each note is quickly hit and released. The sound stops but the next note will not be heard until the duration of the first note has passed. The other property has more to do with the volume dynamics of the musical sequence. These dynamics determine how loud or soft a musical part is played and are specified in a staff by using some defined terms, such as "forte", "mezzo forte", "pianissimo", and others. Since these terms are somewhat vague, we will add another property to a note (besides frequency and duartion) to indicate how hard that note should be played.

2.2.1 Definition of note

We can proceed now with our formal definition of a note.

Definition 2.4 A *note* is a triplet $\eta = (f, d, v)$ where

a) $f \in Z$ is called the *frequency or tone* of the note
b) $d \in Q^+$ is the *duration*
c) $v \in [0, 1]$ is the *velocity* of the note

Let $(\mathcal{E}, \tau)$ be a tone scale and $\eta = (f, d, v)$ a note. If $(f - \tau) \in \mathcal{E}$, the note is said to belong to the scale $(\mathcal{E}, \tau)$. All notes belong to the chromatic scale, since $\mathcal{E}_C = Z$. When constructing a melodic sequence, we will restrict the notes to belong to the same scale.

The duration $d \in Q^+$ of a note is measured in *beats*, where the exact length of a beat will be specified by the tempo of a song (measured in beats per minute). The values for d are usually restricted to powers of two, sometimes multiplied by three and sometimes divided by three. We will later define a set which contains all the possible values of note durations for a particular melody.

The velocity of the note determines how hard that note should be played. The term “hard” applies in a different way for each musical instrument and in most cases the velocity directly affects the volume of the produced sound when playing that note. For some instruments, like organs, the volume is independant of how hard the keys are hit; however, the velocity could alter the sound in other ways, allowing a certain degree of expression. Many music synthesizers can be programmed to map the velocity of a note to other things than volume, such as the cutoff frequency of a filter, the depth of a vibrato effect, or the relative volumes between two layered sounds.

If $\eta = (f_0, d_0, v_0)$ is a note, we will refer to each of its properties by defining the following functions:

$$\begin{aligned} f(\eta) &= f_0, \\ d(\eta) &= d_0, \\ v(\eta) &= v_0. \end{aligned}$$

There are some other note properties that could be considered. Musicians often add expression to their performance in different ways, for example, adding vibrato (small oscillations of the frequency) or tremolo (small oscillations of volume) to a note, sliding continuosly from one note to another, changing the speed of a melody, and many other things. While these things can be modelled, they also need some special considerations. The first one is that these means of expression are very specific to each instrument. For example, one cannot add tremolo to stringed instruments, or slide from one note to another in a piano. Thus we would have to either include instrument-specific parameters, or focus on a limited range of instruments. The other consideration is that these expression properties consist of smooth variations in the resulting sound. That means that we would have to include parameters that continuosly change in time. Due to these issues, we will not add other expression parameters than velocity, but we will propose an easy way to do it at the end of this work.

2.2.2 Rests

In music, silence can be as important as sound. A rest is a note that is not to be heard. We could add another property to our definition of note to specify if it is a rest or not but that is not needed. We can use the velocity property for that purpose; after all, the velocity determines how hard a note is hit. If a note is not actually hit, its velocity should be zero.

It could also happen that some notes have a velocity value so small that the note cannot be actually triggered, or that the amount of volume given by the note velocity is too soft for the note to be heard. Thus we can also consider those notes as rests. More specifically, we can say that a rest is a note whose velocity is below a certain threshold.

Definition 2.5 Given a constant threshold $u \in (0, 1]$, a note η is a *rest* if $v(\eta) < u$. The frequency of a rest has no meaning at all.

2.2.3 Sequences of notes

The simplest melodies are constructed by playing notes one after another in time (some of those notes could be rests). The sound of a note (or silence of a rest) lasts until its duration time has passed, then the next note starts immediately, until the last note is played. This can be expressed as a finite sequence of notes $\mathcal{N} = (\eta_0, \eta_1, \dots, \eta_{k-1})$, which has some useful properties. Assuming the first note starts at a time $t = 0$, one can easily calculate at what time each note should start playing or how long does it take to play the whole sequence. We proceed to define those characteristics.

Definition 2.6 The *starting time* of a note in a sequence $\mathcal{N}$ is defined as

$$t(\eta_i) = \begin{cases} 0 & \text{if } i = 0, \\ \sum_{j=0}^{i-1} d(\eta_j) & \text{if } 0 < i < k. \end{cases}$$

where k is the number of notes in the sequence.

Definition 2.7 The *length or duration* of the sequence is

$$d(\mathcal{N}) = \sum_{j=0}^{k-1} d(\eta_j).$$

It is also useful to know if the starting time of a note is a multiple of a given number of beats. We call this property the *alignment* of a note and it is used to emphasize notes which fall on a specific beat and to construct the rhythm of a melody.

Definition 2.8 A note η_i of a sequence $\mathcal{N}$ is said to be *aligned to* q beats, where $q \in Q^+$, if there exists a non-negative integer p such that

$$t(\eta_i) = pq.$$

2.3 Melodies

A melody is basically a sequence of notes which holds two restrictions. The first one is that the notes must belong to a given scale. The other one specifies that the length of the sequence must be a multiple of a given number of beats.

Definition 2.9 A *melody* is a quartet $\mathcal{M} = (\mathcal{E}, \tau, s, \mathcal{N})$ where

a) $(\mathcal{E}, \tau)$ is a tone scale,
b) $\mathcal{N} = (\eta_0, \eta_1, \dots, \eta_{k-1})$ is a sequence of k notes,
c) s is a positive integer and it is called the *signature* of the melody,
d) $(f(\eta_i) - \tau) \in \mathcal{E}$ for all $0 \leq i < k$,
e) There exists a positive integer p such that $d(\mathcal{N}) = ps$.

The length of a melody is usually specified in *measures*, where one measure is equal to s beats. Thus if $d(\mathcal{N}) = ps$, the melody is p measures long. In music, this is equivalent to having a signature of $s/4$, so typical values for s would be 2, 3, and 4. Two or more melodies that share the same tone scale and signature are said to be *harmonically and rhythmically compatible* and can be used to form more complex musical compositions.

Most of this work will be dedicated to construct melodies; however, our definition of melody is very loose and practically allows any sequence of notes to be considered a melody since all notes belong to the chromatic scale and a rest can be added to the sequence to complete a whole measure, if needed. The restrictions required for our melodies to resemble musical phrases will be developed as part of the algorithms that construct those melodies, which will be presented in the following chapters. This allows new algorithms to be created, based on the same definitions we present here.

It is important to note that melodies are monophonic; that is, two notes of a melody never overlap each other in time. Because of this, a single melody could hardly be considered as a musical composition. A full composition would be made of various melodies with similar properties, specifically, the same scale, tone, and signature. Another algorithm will be developed to arrange compatible melodies in order to form a complete musical piece.

2.4 Rhythm

Rhythm is a fundamental part of any musical piece. Basically, rhythm is obtained by

a) Quantizing the durations of notes in a melody.
b) Creating repetitive note duration patterns within the melody.
c) Emphasizing or accentuating notes which start at regular intervals.

The first part is obtained by restricting the set of eligible note durations. Instead of Q^+, we will choose note durations from a much more restricted (and finite) set. Such a set must contain a certain range of powers of two (according to music theory) and two types of variations, which consist of triplets (a duration value divided by three), and dotted notes (a duration value multiplied by 3/2). Such a set is easy to build:

Definition 2.10 Given $a, b \in Z$ such that $a \leq b$, we define the *duration set* $\mathcal{T}_{a,b}$ as

$$\mathcal{T}_{a,b} = \{2^i \mid i \in \{a, \dots, b\}\} \cup \{\frac{2^i}{3} \mid i \in \{a, \dots, b\}\} \cup \{\frac{3}{2}2^i \mid i \in \{a, \dots, b\}\},$$

where typical values for a and b are between -3 and 3.

$\mathcal{T}_{a,b}$ contains all the duration values that could be used not only in a single melody, but in a full musical composition (made with several melodies). A single melody would use durations from a subset of $\mathcal{T}_{a,b}$ and some durations may appear more frequently than others. This will be implemented in the rhythm generating algorithm by using a probability value for each elegible duration, for each melody.

2.4.1 Rhythmic patterns

Definition 2.11 Given a signature $s \in Z^+$, we define a *rhythmic pattern* as a succession $(d_0, d_1, \dots, d_{k-1})$ where $d_i \in \mathcal{T}_{a,b}$ for $0 \leq i < k$ and some fixed $a, b \in Z$, such that for some positive integer p the succession holds that

$$\sum_{i=0}^{k-1} d_i = ps,$$

and we say that the *length* of the rhythmic pattern is p. A rhythmic pattern is meant to become the succession of the note durations of a melody of length p and signature s.

A human composer does not usually think about how many notes should be in a melody. Instead, he decides *how long* a melody should be in measures. He starts adding notes until a whole number of measures is filled. Thus the rhythmic pattern is the very first thing we need to generate when constructing a melody. Once we have a rhythmic pattern, we will know how many note frequencies and velocities need to be generated. Because of this, it is a good idea to study the rhythmic patterns of fixed length p.

Let us consider the set $\mathcal{T}_{a,b}$ as an alphabet. A rhythmic pattern can be seen as a word of $\mathcal{T}_{a,b}^*$. Let $\mathcal{P}_p \subset \mathcal{T}_{a,b}^*$ be the language which contains all the patterns of length p. Let d_{min} be the shortest duration in $\mathcal{T}_{a,b}$. It is clear that for any $w \in \mathcal{P}_p$ we have that $|w| \leq ps/d_{\text{min}}$, where s is the signature of the melody. Thus the language $\mathcal{P}_p$ is finite and can be generated by a phrase structure grammar.

2.4.2 Dotted notes and triplets

In music theory, a *dotted note* is a note whose duration is 1.5 times the duration specified by its note symbol. To allow dotted notes in our melodies, we have included the corresponding durations in the set $\mathcal{T}_{a,b}$.

A *triplet* is a group of three notes whose duration is a third of the duration corresponing to their note symbol. The three notes have the same symbol in a staff, with a *tie* symbol (an arc) above them. In our melodies we can also include triplets since $\mathcal{T}_{a,b}$ contains the corresponding one-third durations. However, since triplets are not single notes, they must be handled in a slightly different way when constructing our grammar. That is, if our grammar generates the first note of a triplet, we must ensure that other two notes with the same duration will be generated. Another restriction we must consider is that triplets are always aligned to a beat; that is, the starting time of the first note of a triplet is always a whole number.

2.4.3 The rhythmic pattern grammar

Now we will construct a grammar that generates rhythmic patterns of length p, with durations taken from the set $\mathcal{T}_{a,b}$, where $a, b \in Z$ and $a \leq b$. Let $d' = 2^{a-1}$ and $d'' = 2^a/3$. Let $(d_0, d_1, \ldots, d_{k-1})$ be any rhythmic pattern of length p. Considering that triplets are grouped and aligned as explained in the last paragraph, we can see that the sums

$$t_j = \sum_{i=0}^{j} d_i, \qquad \text{for } j = 0, 1, \ldots, k-1$$

are either a multiple of d' or a multiple of d''; that is, for some integer $m \geq 0$, either $t_j = md'$, or $t_j = md''$. This means that the notes of any melody constructed from a rhythmic pattern with these properties will have starting times which are either multiples of d', or multiples of d''. This allow us to easily construct a set T which contains all the possible starting times:

$$\begin{aligned} T_1 &= \{md' \mid m \in Z,\ 0 \leq md' < ps\} \\ T_2 &= \{md'' \mid m \in Z,\ 0 \leq md'' < ps\} \\ T &= T_1 \cup T_2 \cup \{ps\} \end{aligned}$$

Even though a note cannot start at time ps, it will be useful to have ps in T as an end-mark.

We will also consider the following three subsets of $\mathcal{T}_{a,b}$:

$$\begin{aligned} \mathcal{T}_1 &= \{2^i \mid i \in \{a, \ldots, b\}\} \\ \mathcal{T}_2 &= \{\frac{2^i}{3} \mid i \in \{a, \ldots, b\}\} \\ \mathcal{T}_3 &= \{\frac{3}{2}2^i \mid i \in \{a, \ldots, b\}\} \end{aligned}$$

corresponding to normal notes, dotted notes, and triplets, respectively.

In order to construct our grammar, we define first the set of non-terminal symbols $\Sigma = \{\sigma_q \mid q \in T\}$. The starting symbol will be σ_0 and our alphabet will be the set of durations $\mathcal{T}_{a,b}$. Thus we define the rhythmic pattern generating grammar as $\Gamma = (\Sigma \cup \mathcal{T}_{a,b} \cup \{\eta\}, \mathcal{T}_{a,b}, \pi, \sigma_0)$, where η is the identity (terminal) symbol and π consists on the following productions:

$$\begin{aligned} \sigma_q &\longrightarrow \alpha\sigma_{q+\alpha} && \text{for all } q \in T_1,\ \alpha \in \mathcal{T}_1, \\ \sigma_q &\longrightarrow \alpha\sigma_{q+\alpha} && \text{for all } q \in T_1,\ \alpha \in \mathcal{T}_2, \\ \sigma_q &\longrightarrow \alpha\sigma_{q+\alpha} && \text{for all } q \in T_2,\ \alpha \in \mathcal{T}_3, \\ \sigma_{ps} &\longrightarrow \eta. \end{aligned}$$

It is easy to see how this grammar works. A production of the form $\sigma_q \longrightarrow \alpha\sigma_{q+\alpha}$ means that a note of duration α will be added at starting time q. When we reach the length of the pattern, the production $\sigma_{ps} \longrightarrow \eta$ will mark the end of the generated word.

This is the most generic and unrestrictive grammar for generating rhythmic patterns, considering the triplet restrictions. However, we want to be able to specifically control which durations have more chance of appearing in the pattern. This is done by assigning a constant probability $p_{q,w} \in [0,1]$ to each production $\sigma_q \longrightarrow w$ in π.

For a given starting time $q < ps$, the sum of the probabilities of the possible symbols that can be generated must be 1; that is

$$\sum_{(\sigma_q,w)\in\pi} p_{q,w} = 1.$$

And clearly, $p_{ps,\eta} = 1$.

For simplicity, we restrict the probabilities to be constant over time. This means that the probability of a duration α to be generated will be the same for every starting time q. We define a new probability p_α for every $\alpha \in \mathcal{T}_{a,b}$ and say that

$$p_{q,\alpha\sigma_{q+\alpha}} = p_\alpha \qquad \text{for every } \alpha \in \mathcal{T}_{a,b},$$

and therefore

$$\sum_{\alpha\in\mathcal{T}_{a,b}} p_\alpha = 1.$$

There are a few more restrictions that could be considered; especially on triplets. For example, one could restrict the three notes of a triplet to be the exact same length with productions like

$$\sigma_q \longrightarrow \alpha^3\sigma_{q+3\alpha} \qquad \text{for all } q \in T_2,\ \alpha \in \mathcal{T}_3.$$

However, those restrictions can be also achieved by proper modification of the probabilities of each production. The inclusion of a probability distribution of note durations adds much flexibility to the generating grammar and gives us a high degree of control over the rhythmic properties of our melodies.

2.5 Accents

In order to emphasize the rhythm of a melody, one can *accentuate* the notes which start at specific beats on a measure. An accentuated note is supposed to be played with more strength than a non-accentuated note. This means that, although accentuation is a rhythmic property, it affects directly the note velocities. In order to apply accentuation to a melody, first we need to specify which beats of a measure will be accented.

2.5.1 Accentuation patterns

Definition 2.12 Let $\mathcal{M} = (\mathcal{E}, \tau, s, \mathcal{N})$ be a melody. An *accentuation pattern* for $\mathcal{M}$ is a word $w = w_1 w_2 \ldots w_s \in \{0,1\}^s$, where $w_i = 0$ means that the i-th beat of every measure is a non-accented or weak beat, and $w_i = 1$ means that the i-th beat is accented or strong.

Only notes with an integer starting time can be accentuated. Any other notes are considered to be weak (non-accentuated). Therefore, a note η_i in $\mathcal{N}$ is accentuated if

$$t(\eta_i) \in Z, \text{ and } w_{(t(\eta_i) \bmod s)+1} = 1$$

for a given accentuation pattern w.

2.5.2 Applying accentuation to a melody

Given a melody $\mathcal{M} = (\mathcal{E}, \tau, s, \mathcal{N})$, where $\mathcal{N} = (\eta_0, \eta_1, \ldots, \eta_{k-1})$, and an accentuation pattern $w \in \{0,1\}^s$, we construct a new melody $\mathcal{M}'$ which is the result of *applying* w to $\mathcal{M}$. The only difference between $\mathcal{M}$ and $\mathcal{M}'$ will be the note velocities. We will not increase the velocity of accentuated notes; instead, we will reduce the velocity of the weak notes. This way, we do not have to worry about crossing the velocity upper limit. Thus we only need to obtain a new sequence of notes $\mathcal{N}' = (\eta'_0, \eta'_1, \ldots, \eta'_{k-1})$ where

$$\begin{aligned}
f(\eta'_i) &= f(\eta_i) \\
d(\eta'_i) &= d(\eta_i) \\
v(\eta'_i) &= \begin{cases} v(\eta_i) & \text{if } t(\eta_i) \in Z \text{ and } w_{(t(\eta_i) \bmod s)+1} = 1, \\ (1-\gamma)v(\eta_i) & \text{if } t(\eta_i) \in Z \text{ and } w_{(t(\eta_i) \bmod s)+1} = 0, \\ (1-\gamma)v(\eta_i) & \text{if } t(\eta_i) \notin Z, \end{cases}
\end{aligned}$$

for $0 \leq i < k$. The parameter $\gamma \in [0,1]$ is called the *accentuation degree* and it determines the amount of attenuation of the weak notes. Finally, the result of applying w to $\mathcal{M}$ is $\mathcal{M}' = (\mathcal{E}, \tau, s, \mathcal{N}')$.

Chapter 3

Generating melodies

A melody is meant to resemble a monophonic musical phrase. A melody is not supposed to be a full musical composition; instead, a composition would be made of various superimposed melodies which appear (and reappear) during the course of the composition. Arranging melodies this way is actually easier to do than constructing the melodies themselves; because of that we will focus mainly on how melodies can be generated.

Once we have specified the scale, signature and length of a melody, we have seen that the first thing we need to generate is a rhythmic pattern, from which we will extract the number of notes in the melody and their duration values. Then we can generate the note frequencies and velocities, and finally, apply an accentuation pattern.

3.1 Data structures

Our intention is to develop algorithms which are meant to be implemented in a computer program. From now on, our definitions will be a little less formal in the mathematical sense and will use some computer related terms. We consider all the new terms to be intuitive enough so formal definitions are not required. The only exception might be the concept of a data structure, which we define as follows:

Definition 3.1 A *data structure* is a finite set of ordered triads $\{(p_1, V_1, v_1), (p_2, V_2, v_2), \ldots, (p_n, V_n, v_n)\}$, where $p_1, p_2, \ldots, p_n$ are called the *fields or prop-*

erties of the data structure and $v_1, v_2, \ldots, v_n$ are the *corresponding values* for each field. Each set V_i contains all the values that v_i can take for $1 \leq i \leq n$.

An *instance* of a data structure $D = \{(p_1, V_1, v_1), (p_2, V_2, v_2), \ldots, (p_n, V_n, v_n)\}$ is another data structure D' with exactly the same fields and value sets, but not necessarily the same values. That is,

$$(p_1, V_1, v_1) \in D \Longleftrightarrow (p_1, V_1, v_1') \in D',$$

where $v_1, v_1' \in V_1$.

One usually defines a data structure which will serve only as a *template* for instances of that data structure. The values of the template data structure are not important at all and may be even skipped in the template definition. For example, let D be a template data structure with the fields {item, color, size}. Let the value set for the item field be V_i ={car, house, ball}, the color set be V_c ={red, blue, white, green}, and the size set be V_s ={big, small}. Then the following data structures are instances of D:

$$\begin{aligned} D_1 &= \{(\text{item}, V_i, \text{house}), (\text{color}, V_c, \text{white}), (\text{size}, V_s, \text{big})\} \\ D_2 &= \{(\text{item}, V_i, \text{car}), (\text{color}, V_c, \text{red}), (\text{size}, V_s, \text{small})\} \end{aligned}$$

Even if we had not defined D, it would be obvious that D_1 and D_2 are both instances of the same template. Moreover, if the value sets are implicit or already known, then it is not necessary to specify them for every instance. In fact, the value sets for each field are usually only specified when defining a template. The following example defines a template D and two instances D_1 and D_2 of D:

$$D = \{(\text{frequency}, Z\ ), (\text{duration}, \mathcal{T}_{a,b}), (\text{velocity}, [0,1])\}$$

$$\begin{aligned} D_1 &= \{(\text{frequency}, 3), (\text{duration}, 1), (\text{velocity}, 0.8)\} \\ D_2 &= \{(\text{frequency}, \text{-}7), (\text{duration}, 1/4), (\text{velocity}, 0.5)\} \end{aligned}$$

In this case, D serves as a template structure for notes, while D_1 and D_2 contain the information for two actual notes. In fact, this is exactly how notes would be implemented in a computer program.

3.2 How melodies are constructed

In order to construct a melody, we have to give a number of parameters (besides the scale, signature and length of the melody). Some of those parameters are necessary; for example, a range of the scale from which we will choose our notes, a threshold value for rests, a pair of integers a, b to construct the duration set $\mathcal{T}_{a,b}$, and the probabilities for the rhythmic pattern grammar. Other parameters will be included to add more flexibility, such as limiting the velocity range of the notes, or transposing the melody up or down within the scale. All those parameters will be defined in a data structure which we will call a *block*. In other words, a block will contain the necessary information to generate a melody, but not the melody itself. However, when the algorithms are implemented in a computer program, the difference between a block and its corresponding melody becomes less important and both terms may be used to refer to a sequence of notes, or the parameters used to produce it.

So far we have constructed a grammar that will produce rhythmic patterns from which we will obtain the note durations of a melody. Now we only need to determine the frequency and velocity of each note and we will have a new melody. These values will simply be obtained by mapping sequences of integer numbers into note frequencies and velocities. We are interested in a special kind of integer sequences:

Definition 3.2 A *microsequence* $M = (m_0,\ m_1, \ldots,\ m_{n-1},\ m_0,\ m_1, \ldots)$, where $m_i \in Z^+$ for $0 \leq i < n$, is a n-periodic sequence of non-negative integers. Microsequences can be indexed as follows:

$$M[i] = m_{i \bmod n}.$$

We want to construct a melody $\mathcal{M} = (\mathcal{E}, \tau, s, \mathcal{N})$, where $\mathcal{N} = (\eta_0, \eta_1, \ldots, \eta_{k-1})$. By generating a rhythmic pattern for $\mathcal{M}$, we will obtain the number of notes k and the note durations $d(\eta_i)$, for $0 \leq i < k$. We obtain the note frequencies from a microsequence M_f of period n_f and the following mapping:

$$f(\eta_i) \quad = \quad \mathcal{E}[M_f[i]] + \tau, \tag{3.1}$$

for $0 \leq i < k$. So basically, the frequency of the i-th note will be the $M_f[i]$-th frequency in the scale $\mathcal{E}$, transposed by τ.

In a real application, we would need to restrict the range of the numbers in M_f and add an extra transposition parameter to cover any desired range of frequencies. This can be accomplished in several ways, such as scaling the numbers in M_f or using a modulo operation.

Similarly, given a positive threshold value $u \leq 1$ and some suitable positive $g < 1$, we obtain the note velocities by applying the following mapping to a microsequence M_v:

$$v(\eta_i) = \begin{cases} 0 & \text{if } gM_v[i] < u, \\ 1 & \text{if } gM_v[i] > 1, \\ gM_v[i] & \text{otherwise.} \end{cases} \tag{3.2}$$

If a more restricted velocity range is required, one can multiply by an extra amplification/attenuation factor and add a velocity offset. In this case, the threshold may need to be corrected (by adding the same velocity offset to u) if one does not want rests to be affected. Once the velocities have been generated, an accentuation pattern can be applied as described in section 2.5.2 and our melody will be finished.

We have not said anything about how the microsequences are generated. For now, we will assume there is some black-box function Ω which takes a data structure D_m and returns a specific microsequence $M = \Omega(D_m)$. We will describe how to construct microsequences in the next chapter.

3.3 The main algorithm

Given a tone scale $(\mathcal{E}, \tau)$, a signature value s, and a desired length p (in measures), we construct a melody $\mathcal{M}$ as follows:

1. Given $a, b \in Z$ and a probability p_α for each $\alpha \in \mathcal{T}_{a,b}$, a rhythmic pattern of length p and signature s is generated. Let $k = |w|$ be the *number of notes.*

2. Let $\mathcal{M}' = (\mathcal{E}, \tau, s, \mathcal{N}')$ be a temporary melody where $\mathcal{N}' = (\eta_0, \eta_1, \ldots, \eta_{k-1})$ and $d(\eta_i) = w_i$.

3. Given two suitable data structures D_f and D_v, calculate the microsequences $M_f = \Omega(D_f)$ and $M_v = \Omega(D_v)$.

4. Calculate $f(\eta_i)$ and $v(\eta_i)$ for $0 \leq i < k$ from M_f and M_v, respectively, by using Equations 3.1 and 3.2.

5. Given an accentuation pattern $w_a \in \{0, 1\}^s$, obtain $\mathcal{M}$ by applying w_a to $\mathcal{M}'$.

Every step of this algorithm will be described with more detail in the following two chapters. Chapter 4 deals with the third step; that is, the production of microsequences and the parameters involved. Chapter 5 describes the block data structure and the actual mappings used to independently generate the note durations, frequencies and velocities.

We said in the Section 2.3 that two or more melodies $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ are harmonically and rhythmically compatible if they share the same tone scale $(\mathcal{E}, \tau)$ and signature s. This means that if we want to generate a set of melodies which are to become part of a more complex composition, we must keep some of the parameters (such as $\mathcal{E}$, τ, and s) constant. In Chapter 6, we will develop another algorithm to arrange compatible melodies into a full musical piece.

Chapter 4

Microsequences

In Section 3.2 we defined a microsequence as a periodic sequence of non-negative integers. These sequences are meant to be mapped to the frequency and velocity values of a sequence of notes. Equations 3.1 and 3.2 are a simplified version of the mappings that will be used. Such mappings already do a good job making our note sequences resemble some properties found in music; for example, the microsequence-to-frequencies mapping keeps the note frequencies within a limited range of a scale, while the microsequence-to-velocities mapping restrains the dynamic range of a melody and uses the rests threshold to produce zero-velocity notes. These mappings will be described with more detail in the next chapter.

This means that we could use random natural numbers to construct our microsequences and still get musical-sounding results. That is, in fact, one of the methods that will be described later in this chapter. However, we would like to obtain other properties commonly found in music. Most music consists in patterns which combine elements of structure and irregularity. It is neither completely random nor highly correlated. A complete musical piece could be constructed from sequences which are too repetitive and boring by themselves, sequences which are almost periodic, but actually change constantly, and sequences that seem to have chaotic behavior, all at the same time. Because of this, a random number generator is not enough to suit our needs. In fact, one single method of generating microsequences may not be enough, either. We want to develop one or more algorithms to generate microsequences and we would like each of these methods to be capable of at least one of the following:

a) Controlling the degree of randomness of the sequences.
b) Controlling the periodicity of the sequences
c) Produce sequences with a certain degree of self-similarity

There are many mathemathical objects and models from which we can extract sequences of numbers and present one or more of the characteristics stated above, these include cellular automata, subatomic particle systems, chaotic systems, and others. We will focus particularly in three models: random noise, the Mandelbrot fractal, and Morse-Thue sequences. For each mathematical object, we will develop one (or more) mappings to extract a microsequence from such object.

The algorithm presented in Section 3.3 does not depend on which method is used to generate the microsequences (although the results do). Because of this, the algorithm is a good framework to experiment with sequences produced by different types of mathematical objects and different mappings. The four methods presented in this chapter are meant to be only examples of how we can produce interesting sequences.

4.1 Random noise

The most simple way to obtain a non-trivial sequence of numbers is by using a random sequence. Computers use different algorithms to generate pseudo-random sequences of numbers. The algorithm starts with a seed value and performs some operations to obtain a "random" number. That number is stored and feeded back into the operations that will produce a new random number, and so on. One big advantage is that by using the same seed, we can reproduce the whole sequence whenever we want; unless the computer internal clock is involved in the algorithm, which is rather unusual.

Sequences generated with a good random number algorithm will show a behavior similar to *white noise*. This only means that a value in the sequence does not depend on the previous values at all. Figure 4.1 shows a sequence of this kind.

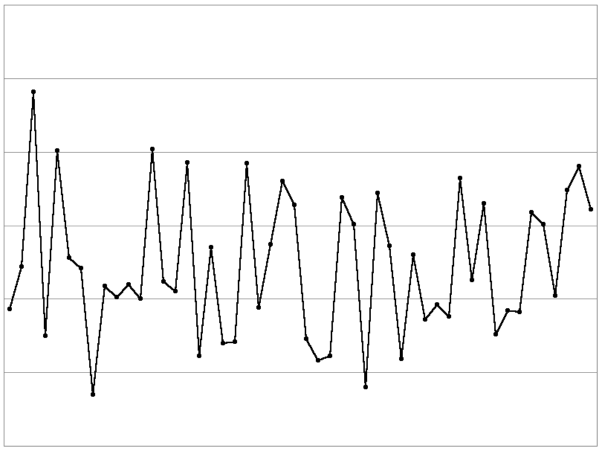

Figure 4.1: Example of white noise.

When analyzing random signals, one of the useful characterizations of the behavior of the signal is its *spectral density*. The spectral density of a signal $V(t)$ is a function $S_V(f)$ which measures the energy of V which is spread in a bandwidth centered on the frequency f. White noise has a spectral density $S(f) = 1/f^0$, which means that all frequencies are equally present.

Another kind of noise is *brown noise*, which simulates the behavior of the random movement of small particles suspended in a liquid and set into motion by thermal agitation of the molecules. This motion is called *Brownian motion* and it resembles a walk in three dimensions; a particle can move just one step from its current position in any direction. While still random, Brownian motion is highly correlated; which means that the position of a particle at a time t_1 cannot be too far from its position at a time t_0 if the time interval $|t_1 - t_0|$ is short enough. Figure 4.2 shows a one-dimensional example of Brownian motion, where the horizontal axis represents time. Brown noise is obtained by measuring the position of a particle set in Brownian motion in one dimension and it shows a spectral density $S(f) = 1/f^2$.

In the mid-1970's, a mathematical study of music was performed by Richard F. Voss and John Clark [4] at the University of California. They analyzed the spectral density of several recordings of music and found it had $1/f$ behavior, which is between white noise $(1/f^0)$ and brown noise $(1/f^2)$. They also found this behavior to hold for completely different kinds of mu-

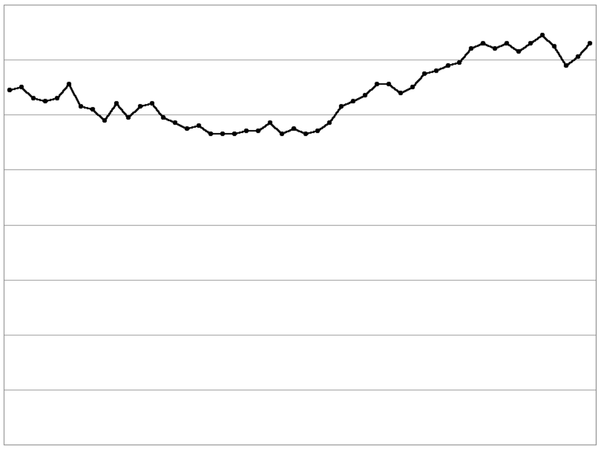

Figure 4.2: One-dimensional Brownian motion (brown noise).

sic; for example, they measured the spectral density of recordings from three different radio stations: a rock station, a classical station, and a news station. All of them displayed a spectral density of $1/f$. Interestingly, this behavior has been found in many other phenomena, such as sunspot activity and traffic flow on freeways.

$1/f$ noise is also called *pink noise*, since it is the middle point between white and brown noises. Figure 4.3 shows an example of pink noise.

Voss and Clarke attempted to compose music using white, brown, and pink noises and compared the results. The "composition" process was made in the following manner. A physical noise source was used to generate a fluctuating voltage with the desired spectrum. The voltages were digitally sampled and stored in a computer as sequences of numbers with the same spectral density as the noise source. These numbers were then rounded, scaled and mapped to notes over two octaves of a musical scale (something very similar to the frequency mappings used in our algorithm). The process was repeated to produce the durations of the notes. The resulting melodies were then played for several listeners. Most of them remarked that the $1/f$ music sounded the most like regular music. The white music seemed too random and the brown music too correlated.

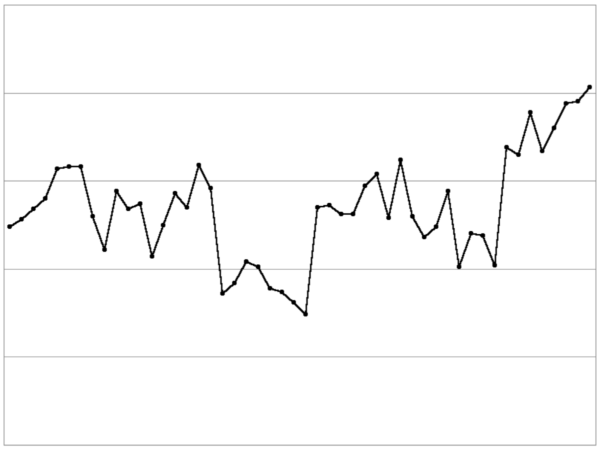

Figure 4.3: Example of pink noise.

4.1.1 Generating noise

Our conclussion is that random numbers can be used to produce microsequences, but we would like those sequences to show $1/f$ behavior, or even better, we would like to have control over the degree of correlation. Generating pink noise is not as easy as white or brown noise. A few algorithms were reviewed but only one of them allowed to produce all three kinds of noise. This algorithm was found in the *Fantastic Fractals* homepage [10] and it is described as a pseudo-pink noise generator.

The basic algorithm works in the following manner. One starts with a $n \times 3$ grid, where n is the number of elements in the resulting sequence (16 in this example). The first row is filled with random numbers (from a random number generator) between 0 and 9, as shown:

4	0	8	9	3	6	4	0	1	9	9	0	9	0	5	6

Then we place a random number on *every second box* of the second row:

4	0	8	9	3	6	4	0	1	9	9	0	9	0	5	6
	6		0		7		8		1		8		3		8

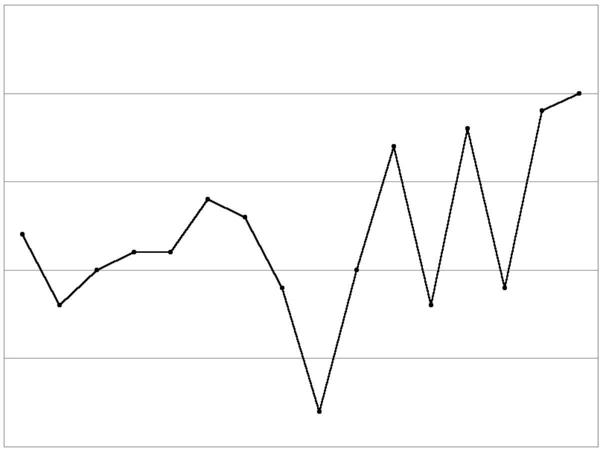

Figure 4.4: Graph of the generated pseudo-pink noise.

And do the same for *every fourth* box of the third row:

4	0	8	9	3	6	4	0	1	9	9	0	9	0	5	6
	6		0		7		8		1		8		3		8
			2				1				0				6

Now, every empty box must be filled with the value of the first non-empty box to its right. This results in the following table:

4	0	8	9	3	6	4	0	1	9	9	0	9	0	5	6
6	6	0	0	7	7	8	8	1	1	8	8	3	3	8	8
2	2	2	2	1	1	1	1	0	0	0	0	6	6	6	6

Finally, we add the values on each column. The result is placed in a fourth row.

4	0	8	9	3	6	4	0	1	9	9	0	9	0	5	6
6	6	0	0	7	7	8	8	1	1	8	8	3	3	8	8
2	2	2	2	1	1	1	1	0	0	0	0	6	6	6	6
12	8	10	11	11	14	13	9	2	10	17	8	18	9	19	20

The bottom row contains a sequence whose spectral density is near $1/f$. Figure 4.4 shows a graph of the sequence.

It is easy to see that some little modifications to this algorithm can make it much more flexible. The sequence which forms the first row is simply white noise. Each subsequent row adds correlation to the resulting sequence. Using three rows is the best approach to pink noise, but it is clear that using more rows will make the sequences more "brownish". On the other hand, the range of the values in the resulting sequence is limited from 0 to 27, which translates in a very low resolution when mapping to note velocities. Thus, instead of choosing random numbers from 0 to 9 for each box in the grid, we will choose numbers from 0 to a given positive integer N. Clearly, the actual range of the elements in the resulting sequence will be N multiplied by the number of rows used to create the sequence.

When generating a microsequence M of period p, we only generate and store the first p elements of the sequence. Thus we can see M as a finite sequence $M = (a_0, a_1, \ldots, a_{p-1})$. We use the following implementation (by Omar Montaño Rivas) to generate a microsequence by the pseudo-pink noise algorithm. Given the number of rows R, the value range N, and the period p of the microsequence,

1. Let $i = 0$.
2. Let $a_i = 0$.
3. For every $j = 0, 1, \ldots, R-1$ do the following:
   (a) If $i \bmod 2^j = 0$ then let r_j be a random integer between 0 and N, inclusive.
   (b) Increase a_i by r_j.
4. Increase i by one.
5. If $i < p$, go back to step 2.
6. The resulting microsequence is $M = (a_0, a_1, \ldots, a_{p-1})$.

The variables r_j hold the current value for the box at row i and column j, for $0 \leq j < R-1$. Those values are only updated when i is divisible by 2^j. When $i = 0$, every r_j is assigned its first value. This is a memory efficient implementation since it only stores one column and the resulting sequence instead of the whole grid. The range of the elements a_i in the sequence will be between 0 and RN.

4.2 The Mandelbrot fractal

Fractals are one of the preferred objects for algorithmic music composers. Since they have infinite detail, they are able to produce a practically infinite number of different sequences. Unlike noise, fractal-based sequences can be completely or partially repetitive, showing a mixture of order and unexpectedness, or they could be completely chaotic. For people who make algorithmic music, fractals provide a wider range of sequences than most mathematical objects and algorithms.

4.2.1 Fractal geometry

Benoit Mandelbrot was the first one to use the word *fractal* to denominate a kind of curves which had some strange properties. According to him, a fractal is a rough or fragmented geometric shape that can be subdivided in parts, each of which is (at least approximately) a reduced-size copy of the whole. Since each part is similar to the whole shape, it can also be subdivided in smaller parts, and so on. This infinite subdivision process is what gives fractals their most useful properties. It also means that fractals can be constructed by recursive equations, which are usually very simple.

The most characteristic properties of fractals are *self-similarity* and *self-reference*. Self-reference means that a reference of an object is contained in its own definition; thus we need a recursive algorithm to generate a fractal. Self-similarity means that the objects are *scale-invariant*: no matter how much we "zoom into" a fractal, we will always find shapes which resemble the original shape. This does not happen with classic euclidean geometry where, for example, a magnified section of a circle will resemble a straight line.

Fractals are useful to model certain shapes in nature. For example, a branch of a tree that resembles the whole tree. However, things in nature do not have an infinite degree of self-similarity and there is always a point where the subdivisions do not look like the original shape anymore. When we say that something in nature is fractal, we are only saying that it can, at some extent, be modeled by a fractal. When we want to construct a model of a shape from a fractal, all we can do is iterate the corresponding recursive equations a finite number of times, which must be enough to recreate the desired shape. Therefore, shapes in nature are not fractal, but they can be modeled by the same recursive algorithms used to construct fractals. The same statement can be said about music.

4.2.2 Fractal music

Music which has been generated by mapping sequences of numbers extracted from a fractal into sequences of notes is often called *fractal music*. This does not necessarily means that the resulting musical piece will show the same characteristics of self-similarity that were present in the initial sequences of numbers. The results greatly depend on the mappings used to transform points of a fractal into notes. However, fractal music does reflect a certain degree of similarity and chaos which is appealing to many composers, such as Phil Thompson, Dave Sthrobeen, José Oscar Marquez, and Phil Jackson, to name a few. Some of them have even released full albums of fractal music.

There has also been some research about the analogies between fractals and classical music. For example, the ending part of "Kunst der Fuge" by Johann Sebastian Bach shows a high degree of self-similarity, where the same patterns appear repeatedly with slight variations and some voices play the same melody as the main voice at the same time, but twice as fast. Other analogies have been found between the Cantor set (a fractal set) and Beethoven's first *Ecossaisen*. Although this pieces were not generated by any algorithm, they provide another reason for using fractals to compose music.

4.2.3 The Mandelbrot set

The Mandelbrot set is created by iterating the complex function

$$f_c(z) = z^2 + c$$

for each point c in a region R_C of the complex plane. By *iterating* $f_c(z)$ we mean that for each $c \in R_C$, we calculate a series of complex numbers $z_0, z_1, \dots$, where $z_0 = 0$ and $z_{n+1} = f_c(z_n)$. The *orbit* of the point c is the series $(f_c^0(0), f_c^1(0), f_c^2(0), \dots)$ and its behavior can act in one of the following

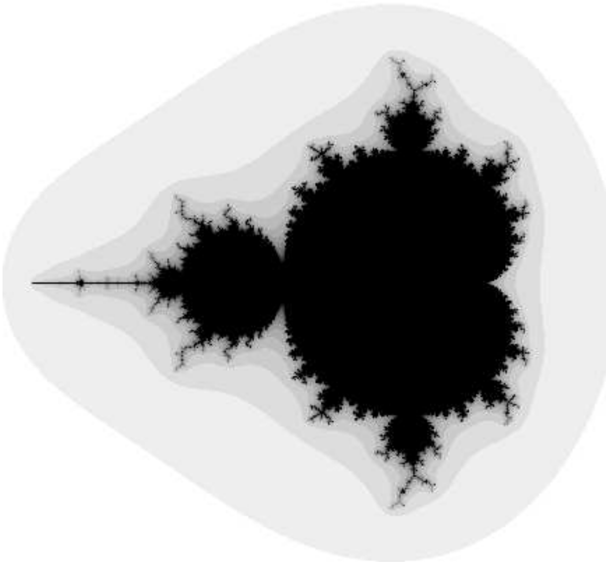

Figure 4.5: Graphic image of the Mandelbrot fractal.

ways:

a) The orbit tends to zero: $\lim_{n\to\infty} |f_c^n(0)| = 0$.
b) The orbit diverges: $\lim_{n\to\infty} |f_c^n(0)| = \infty$.
c) The orbit is periodic.
d) The orbit has no discernible pattern (it is chaotic).

The Mandelbrot set can be defined as the boundary of the set of points whose orbit diverges to infinity. This means that the orbit of a point c in the Mandelbrot set *does not* diverge, but there are points in the complex plane that are arbitrarily close to c and have diverging orbits. A graphic image of the set is usually drawn by assigning a color to each point on the plane depending on the behavior of its orbit. Figure 4.5 shows an image where the points drawn in black have either periodic or chaotic orbits, or their orbits tend to zero. Points drawn in different shades of gray have diverging orbits. The gray tone of the point depends on how fast its orbit diverges. The selected region is the rectangle from $-2-2i$ to $2+2i$. Outside this rectangle, all points have diverging orbits.

The Mandelbrot set shows infinite detail and variation. If one zooms into the boundary of the image (Figure 4.5), one would see that the boundary will never appear smooth, regardless of the scaling factor. This, of course, is limited in practice by the restricted resolution of the displaying media and the limited precision of the computers used to draw the image.

We want to extract microsequences from the Mandelbrot fractal. As we said in Section 4.2.1, it is enough to apply the recursive equation a finite number of times. Moreover, we will take points not only in the Mandelbrot set, but in a region of the complex plane. It would be impossible to follow the boundary which forms the Mandelbrot set because there is an infinite amount of detail between any two points of the boundary.

We propose two algorithms to extract microsequences from the equation $f_c(z) = z^2 + c$. One consists in following the orbit of a point c, and the other consists in following a path on the complex plane and calculating the "color" of each point in the path with the same method used to draw an image of the Mandelbrot fractal.

4.2.4 Mandelbrot orbits

An orbit of the Mandelbrot set is already a sequence of numbers; thus we only need to convert those numbers to integers. There are many ways to do this. One of them involves multiplying either the real or the imaginary part of each point in the orbit by a constant factor and taking only the integer part of the result. Other method consists on calculating the distance from each point in the orbit of c to c itself, then multiply the distance by a suitable constant and take the integer part. We prefer the latter method for two reasons: first, it is easy to find a constant factor that works for most cases, and second, it is visually more intuitive for the composer in the sense that points which are closer to c will be mapped to a note closer to the root note of the scale (the one with index zero) when the microsequence is mapped to note frequencies. Figure 4.6 shows two orbits corresponding to two points near the Mandelbrot set. The elements of the orbits are drawn as black dots. Intuitively, we can see that the first orbit (the one on the left picture) shows a periodic behavior while the other one seems to be chaotic.

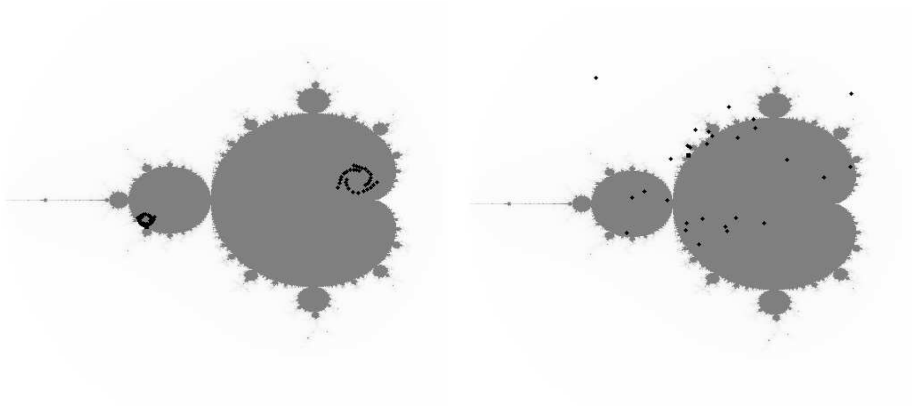

Figure 4.6: Orbits of two points near the Mandelbrot set. The orbit on the left picture is periodic while the other shows chaotic behavior.

In short, given a complex point c, we can easily construct a microsequence $M = (a_0, a_1, \dots, a_{p-1})$ of period p by letting

$$a_i = \lfloor k|f_c^{i+1}(0) - c|\rfloor, \text{ for } 0 \leq i < p,$$

where f_c is defined in the previous section and k is a suitable constant. Note that we are mapping the $(i+1)$-th point in the orbit to the i-th element of M. This will ensure that $a_0 = 0$, which means that, if M is mapped to note frequencies, the first note will be the root of the scale.

There is another issue to consider. If we select a point c whose orbit diverges, we would get notes far outside our desired range. It has been proven that if the modulo of a point in an orbit goes beyond 2, the orbit will diverge. Therefore, if we find a point z_i in an orbit, such that $|z_i| > 2$, then we will make $z_i = 0$, resetting the orbit and forcing the sequence to be periodic. The complete algorithm to construct a microsequence M of period p from the Mandelbrot orbit of a complex point c is as follows.

1. Let $z_0 = 0$ and $i = 0$.
2. If $z_i > 2$ then force z_i to be zero.
3. Calculate $z_{i+1} = z_i^2 + c$.
4. Calculate $a_i = \lfloor k|z_{i+1} - c|\rfloor$.

5. Increase i by one.

6. If $i < p$, go back to step 2.

Microsequences generated by Mandelbrot orbits are very well suited to be mapped to note frequencies. This is because the first note will always be the root note, unless the whole sequence is transposed, as we will see in the next chapter. Also, a graphic display of the orbit can give us much information about the behavior of the resulting melody. This does not mean that Mandelbrot orbits should not be mapped to note velocities; however, one must consider that the first note will always be a rest.

4.2.5 Mandelbrot image paths

The second method we propose to obtain microsequences from the Mandelbrot fractal consists in following a continuous path on the complex plane. We take p points equally distributed along the path and for each point c we calculate how many iterations of f_c are needed for the orbit to escape the circle of radius 2, centered on zero. As we have said, if any point in the orbit falls outside this circle, then the orbit will diverge. Thus we are somehow calculating how fast the orbit diverges for each point. If the orbit of a point does not diverge, the corresponding element of the microsequence will be zero.

In order to develop an algorithm to implement the idea in the last paragraph, we need to give a formal definition of a path.

Definition 4.1 A *path on the complex plane* is a continuous function $s : [0, 1] \longrightarrow C$.

Given a path s, we would like to choose p points $c_0, c_1, \ldots, c_{p-1}$ equally spaced along the path. We would also like c_0 to be the starting point of the path, that is $c_0 = s(0)$, and c_{p-1} the last point of the path, which would be $s(1)$. This can be achieved by letting

$$c_i = s(\frac{i}{p-1}),$$

for $0 \leq i < p$ and $p \geq 2$.

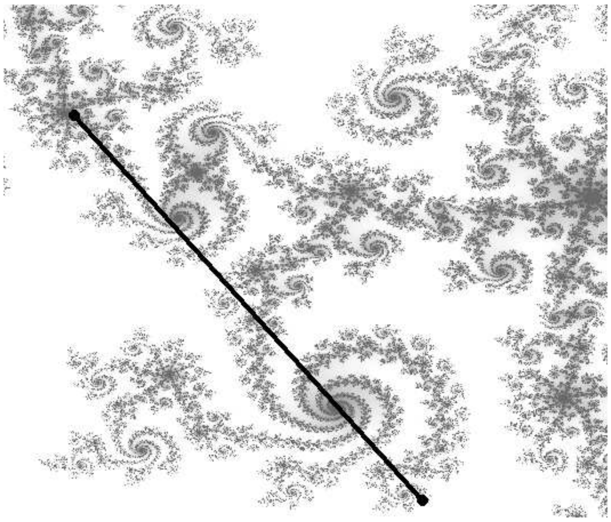

Figure 4.7: A straight-line path on the complex plane. This path crosses smooth and detailed sections of the Mandelbrot set.

We also define the *escape velocity* $v_e : C \longrightarrow Z^+ \cup \{0\}$ as follows:

$$v_e(c) = \begin{cases} \min\{k \in Z^+ \mid |f_c^k(0)| > 2\} & \text{if the orbit of } c \text{ diverges.} \\ 0 & \text{otherwise.} \end{cases}$$

In practice, we do not have the means to decide if the orbit of any point diverges or not. We overcome this by iterating f_c until we are sure that the orbit diverges (when $|f_c^k(0)| > 2$) or a maximum number of iterations is reached. If the latter happens, we will assume that the orbit of c does not diverge and let $v_e(c) = 0$.

Since $v_e(c)$ is a non-negative integer for every c, we can use it directly as an element of a microsequence. In short, we can construct a microsequence $M = (a_0, a_1, \ldots, a_{p-1})$ of period p from a path s by letting

$$a_i = v_e(s(\frac{i}{p-1}))$$

for $0 \leq i < p$.

A graphic image of the Mandelbrot fractal is calculated using $v_e(c)$; thus one can intuitively predict the behavior of a Mandelbrot-path microsequence by drawing the path over the image. If the path goes through a smooth area of the image, the resulting sequence will show smooth changes. If the path goes through a complex area with many details, the resulting sequence will be probably chaotic. It is a good idea to construct microsequences with a large period when using this method in order to resemble the degree of detail shown in the graphic image. Additionally, interesting paths can be modelled by using splines or Bezier curves in order to cross specific areas of the fractal. The example program (described in Chapter 7) only uses straight lines as paths, such as the one shown in Figure 4.7.

While Mandelbrot orbits are best suited to be mapped to note frequencies, microsequences generated by Mandelbrot paths can be used to model smooth or random-like velocity transitions, or any combination of the two. They can also generate interesting frequency sequences, although it is difficult to obtain the same degree of periodicity that can be achieved with Mandelbrot orbits, unless of course, the microsequence has a small period itself. We could say that both methods complement each other and provide a incredible range of possibilities for the creation of algorithmic music.

4.3 Morse-Thue sequences

Another useful method to produce microsequences is based on the *Morse-Thue* sequence. This sequence is constructed by taking the parity (the sum of binary digits modulo 2) of the natural numbers. First we write the natural numbers in binary form:

$$0, 1, 10, 11, 100, 101, 110, 111, \dots,$$

and then we take the sum of the digits of each binary number and apply a modulo 2 operation. The resulting sequence is

$$0, 1, 1, 0, 1, 0, 0, 1, \dots,$$

which is the Morse-Thue sequence.

Another way to construct the sequence is as follows. We start with the sequence $I_0 = (0)$. In each iteration we append the complement of I_k to I_k

itself to obtain I_{k+1}:

I_0	0
I_1	0 **1**
I_2	0 1 **1 0**
I_3	0 1 1 0 **1 0 0 1**

and the Morse-Thue sequence would be $\lim_{k\to\infty} I_k$.

This sequence has two important properties. One of them is that the sequence is not periodic, which is clear by looking at the construction method explained above. Because of this, we can extract an infinite number of different microsequences from the Morse-Thue sequence. The other property is that the sequence is highly self-similar. In the Mandelbrot set one can find zones which resemble the whole set, but do not look quite the same. On the other hand, the Morse-Thue sequence contains subsequences which are exactly the same as the original. For example, if we remove every second element of the sequence, as shown below, we obtain the original sequence.

Original sequence	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	...
Subsequence	0		1		1		0		1		0		0		1		...

Removing every second couple of elements also keeps the sequence unchanged.

Original sequence	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	...
Subsequence	0	1			1	0			1	0			0	1			...

The fact that each element of the Morse-Thue sequence is either 0 or 1 makes it unadequate to be directly mapped to elements in a microsequence. However, a small change in the construction algorithm will provide us with a much more interesting sequence. Instead of calculating the parity of each natural number, we only calculate the sum of the digits of each number in binary form, skipping the modulo 2 operation. This leaves us with the sequence shown in the last column of the table shown below. A graph of the resulting sequence is shown in Figure 4.8 (left picture).

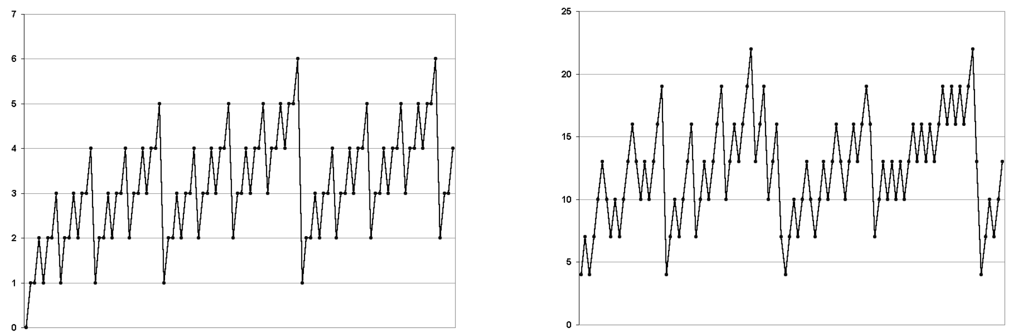

Figure 4.8: Graphs of two Morse-Thue sequences. The one on the left is the original sequence with base 2, step 1, and initial value 0. The one on the right has base 7, step 51, and initial value 10.

Decimal form	Binary form	Sum of binary digits
0	0000	0
1	0001	1
2	0010	1
3	0011	2
4	0100	1
5	0101	2
6	0110	2
7	0111	3
8	1000	1
9	1001	2
10	1010	2
11	1011	3
12	1100	2
13	1101	3
14	1110	3
15	1111	4

Further changes in the construction algorithm lead us to other interesting sequences. The first change consists on converting the natural numbers to a different base than 2 (binary), and taking the sum of the digits of each number expressed in a given base b. The second change allows using different increment values (other than one) for the sequence corresponding to the first column of the table shown above. This parameter will be called the *step* s of the sequence. Finally, we will let the sequence start with any natural number, not only zero. This will be the *initial value* of the sequence, although it will not necessarily be the first value of the resulting microsequence. Some combinations of base, step, and initial value will give sequences with many variations while others will result in sequences that will seem almost static. The right picture on Figure 4.8 shows an interesting sequence with base 7, step 51, and initial value 10.

The algorithm to generate a microsequence M of period p from a Morse-Thue sequence, given a base b, step s, and initial value v_0 is very simple.

1. Let $i = 0$.

2. Find the digits of v_i expressed in base b. The digits can be found as follows:
$$d_k = \lfloor \frac{v_i \bmod b^{k+1}}{b^k} \rfloor,$$
where
$$v_i = \sum_{k=0}^{\infty} d_k b^k.$$

3. Let the i-th element of M be the sum of the digits of v_i in base b:
$$a_i = \sum_{k=0}^{\infty} d_k.$$

4. Let $v_{i+1} = v_i + s$.

5. Increment i by one.

6. If $i < p$, go back to step 2.

7. The resulting microsequence is $M = (a_0, a_1, \ldots, a_{p-1})$.

In practice, we will not calculate d_k for $0 \leq k < \infty$. It is enough to calculate digits until $b^{k+1} > v_i$, since the following digits will be clearly zero.

4.4 Other possible methods

The four microsequence construction methods explained in this chapter cover a wide range of the desired properties. We have a method to generate random sequences with different degrees of correlation. A method that uses the Mandelbrot fractal to generate sequences which can be periodic or chaotic, and show smooth or random changes. And a method that produces sequences with a high degree of self-similarity. However, there are many other ways to generate microsequences, either using other types of fractals or numeric sequences, or even using data obtained from real-world phenomena, such as the number of red cars that pass through a highway section each minute. The only way to discover an algorithm that produces interesting sequences is by experimenting with different objects and mappings.

Also, microsequences could be used instead of a random number generator when constructing a rhythmic pattern by the grammar algorithm. A sequence of numbers is needed when trying to decide which production should be used, according to their probabilities. White noise is commonly used for this purpose; however, white noise does not show periodicity nor self-similarity, and therefore, these properties are not reflected in the rhythmic patterns. By substituting the white noise by a fractal-generated microsequence, more interesting results could be obtained. Such experiments are beyond the scope of this work but they should be very easy to implement in conjunction with the algorithms we have presented.

Chapter 5

Blocks

In most cases, a musical piece can be divided in *musical phrases*, which are polyphonic melodies with a short length (usually between 1 and 8 measures long). Our definition of melody allows only one note to be played at a time, but every polyphonic melody can be separated in two or more monophonic melodies. A *polyphonic melody* is obtained by playing two or more melodies together at the same time. The melodies must have the same tone scale, signature and length.

So basically, we can see a musical phrase as a short (monophonic) melody that repetitively appears in a musical composition. In other words, a composition is made of short monophonic melodies (musical phrases), and these melodies must be harmonically and rhythmically compatible.

We described in Chapter 3 an algorithm to construct the melodies we need. In this chapter we will analyze such algorithm and the parameters involved with more detail.

5.1 Definition of a block

There are many parameters involved in the melody generation algorithm. This makes it a good idea to put all those parameters in a data structure template.

Definition 5.1 A *block* is a data structure with the following fields:

Field	Description
$(\mathcal{E}, \tau)$	Tone scale
$s \in Z^+$	Signature
$a, b \in Z$	Range of the duration set $\mathcal{T}_{a,b}$
$p \in \{1, 2, \ldots, 8\}$	Length in measures
$\lambda \in (0, 1]$	Legato degree
M_f	Frequency microsequence
$r_f \in Z^+$	Frequency range
$t_f \in Z$	Frequency transposition
M_v	Velocity microsequence
$r_v \in [0, 1]$	Velocity range
$a_v \in R^+$	Velocity amplification/attenuation
$u \in [0, 1]$	Rest threshold
$p_\alpha \in [0, 1]$	Probability for each $\alpha \in \mathcal{T}_{a,b}$
$w_a \in \{0, 1\}^s$	Accentuation pattern
$\gamma \in [0, 1]$	Accentuation degree

The fields of a block are divided in various groups according to their purpose.

5.1.1 Common fields

In order to produce compatible melodies from various blocks, some fields must have the same values for all the blocks. Those fields are the tone scale $(\mathcal{E}, \tau)$, the signature s, and the duration range parameters a and b. If desired, one could use a more restricted scale (with less frequencies) than $\mathcal{E}$, but that would be the only allowable difference in the common fields.

5.1.2 Rhythm related fields

The fields needed to generate the rhythmic pattern are the probabilities p_α for each $\alpha \in \mathcal{T}_{a,b}$ and the length p of the melody. The *legato degree* λ is also considered a rhythm related field, since it directly affects the note durations.

All the note durations are multiplied by λ *after* their starting times are calculated. This doesn't affect the rhythmic properties of the melody at all, but it allows the notes to have a certain degree of separation in time.

One can apply legato by taking a rhythmic pattern $(d_0, d_1, \ldots, d_{k-1})$ and constructing a new rhythmic pattern

$$(\lambda d_0, (1-\lambda)d_0, \lambda d_1, (1-\lambda)d_1, \ldots, \lambda d_{k-1}, (1-\lambda)d_{k-1}),$$

and then, converting each second note (the ones whose duration is multiplied by $1-\lambda$) into a rest by assigning them a velocity of zero. However, this is not necessary in practice since we can calculate first the starting time of each note and then multiply its duration by λ. Of course, if $\lambda = 1$, there will be no need to apply legato.

5.1.3 Frequency related fields

The note frequencies are obtained by taking values from the microsequence M_f, applying a modulo operation and a transposition to keep them in the range specified by r_f and t_f, and mapping the resulting values into frequencies in the scale $(\mathcal{E}, \tau)$.

5.1.4 Velocity related fields

The velocity fields M_v, r_v, a_v, and u are used in a similar way than the frequency fields. The values from M_v are restricted to a certain range given by r_v and then, multiplied by an amplification/attenuation factor a_v. Then we use Equation 3.2 to obtain the note velocities.

5.1.5 Accentuation related fields

In order to apply accentuation to a melody we only need an accentuation pattern w_a and the accentuation degree γ, as described in Section 2.5.2.

5.2 A more detailed main algorithm

Given a block B, we generate a melody $\mathcal{M}$ as follows:

1. Generate a rhythmic pattern $(d_0, d_1, \ldots, d_{k-1})$ as follows:

   (a) Start with time $t = 0$ and $i = 0$.

   (b) Select some $\alpha \in \mathcal{T}_{a,b}$ according to the probabilities p_α.

   (c) If α is a triplet duration; that is, $\alpha = 2^q/3$ for some $a \leq q \leq b$, then

      i. If $t + 3\alpha > ps$, go back to step (b),

      ii. otherwise let $d_i = d_{i+1} = d_{i+2} = \alpha$, increase t by 3α, and increase i by 3.

   (d) If α is not a triplet duration then

      i. If $t + \alpha > ps$, go back to step (b),

      ii. otherwise, let $d_i = \alpha$, increase t by α, and increase i by 1.

   (e) If $t < ps$, go back to step (b),

   (f) otherwise, let $k = i$.

2. Construct a melody $\mathcal{M}' = (\mathcal{E}, \tau, s, \mathcal{N}')$ where $\mathcal{N}' = (\eta'_0, \eta'_1, \ldots, \eta'_{k-1})$ where $d(\eta'_i) = d_i,\ i = 0, 1, \ldots, k-1$.

3. Calculate the frecuencies of the notes in $\mathcal{N}'$ as follows:

$$f(\eta'_i) = \mathcal{E}[(M_f[i] \bmod r_f) + t_f] + \tau,$$

   for $i = 0, 1, \ldots, k-1$.

4. Calculate the velocity of η'_i for $0 \leq i < k$ as follows:

   (a) Let $v_i = a_v(r_v g M_v[i] + (1 - r_v))$.

   (b) Calculate

$$v(\eta'_i) = \begin{cases} 0 & \text{if } v_i < u, \\ 1 & \text{if } v_i > 1, \\ v_i & \text{otherwise.} \end{cases}$$

5. To apply the accentuation pattern w_a to $\mathcal{M}'$, create a new sequence of notes $\mathcal{N} = (\eta_0, \eta_1, \ldots, \eta_{k-1})$. For $i = 0, 1, \ldots, k-1$

(a) Let $f(\eta_i) = f(\eta'_i)$ and $d(\eta_i) = d(\eta'_i)$.

(b) Calculate $t_i = t(\eta_i)$; that is, the starting time of η_i.

(c) Calculate

$$v(\eta_i) = \begin{cases} v(\eta'_i) & \text{if } t_i \in Z \text{ and } w_{(t_i \bmod s)+1} = 1, \\ (1-\gamma)v(\eta'_i) & \text{otherwise,} \end{cases}$$

where w_q is the q-th symbol of w_a.

6. Let $\mathcal{M} = (\mathcal{E}, \tau, s, \mathcal{N})$.

5.3 Some remarks about the main algorithm

The algorithm assumes that at least one symbol can be generated by the rhythmic pattern grammar. This means that ps must be greater or equal than the smallest duration in $\mathcal{T}_{a,b}$ with positive probability; that is $ps \geq 2^\alpha$ for some $\alpha \in \mathcal{T}_{a,b}$ such that $p_\alpha > 0$.

The values on the frequency microsequence M_f are first mapped to the range $[t_f, t_f + r_f)$ before using them as indexes to obtain frequencies from a scale. This means that the true range of allowed frequencies depends not only on r_f, but on the scale itself.

The velocity microsequence must contain at least one non-zero value within its first k elements. This ensures that at least one note which is not a rest can be generated.

Note velocities are mapped from the velocity microsequence to the range $[1 - r_v, 1]$, and then multiplied by a_v. This means that decreasing r_v causes the notes to be played harder instead of softer. If needed, the factor a_v can be used to attenuate the velocities. The constant g is only used to scale the microsequence values to a suitable velocity range. A good choice for g would be $g = (\max_{0 \leq i < k}\{M_v[i]\})^{-1}$, which always produce a note with full velocity (before multiplying by a_v).

The legato degree is not considered in this algorithm. Legato can be applied more easily after calculating the start time for each note, or when creating MIDI data from melodies, although it could be included in this algorithm by creating a new sequence of notes as described in Section 5.1.2.

Chapter 6

Macrosequences

A musical phrase may appear many times in a song. Basically, the phrase starts at a certain time t_0 and it is played for a few measures, repeating itself if necessary, and then stops. Eventually, it may start once more at time t_1, be played for a while and stop. This behaviour is repeated until the song ends. When we say that a musical phrase (a short-length melody) starts at a given time t, we mean that all of its notes will be displaced in time by adding t to their starting times. The first note of the melody will effectively start at time t. The phrase will be played completely and after it stops, it can start once more. This means that the next time the melody may be heard, it will start at a time no less than $t + ps$, where p is the length of the melody and s its signature.

The restriction stated above avoids any *overlapping* of a phrase with itself, which is important from a technical point of view. Suppose we have a melody $\mathcal{M}$ of length p with the following sequence of notes $\mathcal{N} = (\eta_0, \eta_1, \dots, \eta_{k-1})$ where $f(\eta) = f$ and $v(\eta) = 1$ for $0 \leq i < k$, so all the notes have the same frequency and none of them is a rest. If we play the melody at $t_0 = 0$ and play it a second time at $t_1 < p$, a note of frequency f will be being played when another note of frequency f is supposed to start. It also could happen that two notes of frequency f should start at the same time. This is impossible to do with most instruments (except stringed instruments, where two different strings can play the same frequency), and it is also impossible to implement with the current MIDI specification (see Appendix A). Also, if no overlapping happens, then the result of this arrangement of a phrase in time is a monophonic melody itself; thus we can see the whole process as creating a larger melody from a musical phrase.

Musical phrases are usually aligned to a measure; that is, they start at a time which is a multiple of their signature. This keeps the sense of a measure as the time unit of a musical composition. They can also be modified in some ways; for example, a musical phrase is sometimes transposed up or down the scale, or the note durations could be multiplied in order to *stretch* the melody in time. In this case, the multiplier should be a positive integer to ensure that the modified phrase has integer length. There are other modifications that can be applied to a musical phrase but they mostly involve slight changes or rearrangements of the notes, which can be obtained by slight changes of some of the block fields, such as the probablities p_α or the velocity and frequency ranges. We will only focus on the ability to stretch our phrases and arrange them in time.

6.1 Definition of macrosequence

Basically, we want to take a musical phrase of length p and signature s, and place it at starting times $t_0, t_1, \ldots, t_{l-1}$. Each copy or instance of the original phrase will be stretched by a factor $e_0, e_1, \ldots, e_{l-1}$, respectively. By *stretching* we mean that the note durations of the i-th copy of the phrase will be multiplied by e_i; thus the length of the i-th copy will be $e_i p$. Also, the following restrictions must be considered:

- The phrases must be aligned to a measure, which means that their starting times t_i (which are expressed in beats) must be a multiple of the signature s.
- The scaling factors e_i must be positive integers.
- The instances of the original phrase must not overlap. This means that $t_{i+1} \geq t_i + e_i ps$ for $0 \leq i < l-1$, where p is the length of the original phrase and s its signature.
- One must not exceed a given *song length* P (in measures). Thus $t_{l-1} + e_i ps \leq Ps$.

What we obtain is a sequence $((t_0, e_0), (t_1, e_1), \ldots, (t_{l-1}, e_{l-1}))$ with the properties stated above.

Definition 6.1 Given a musical phrase $\mathcal{M} = (\mathcal{E}, \tau, s, \mathcal{N})$ of length p and a song length $P \in Z^+$ expressed in measures, a *macrosequence* for $\mathcal{M}$ is a finite sequence of ordered pairs $M = ((t_0, e_0), (t_1, e_1), \ldots, (t_{l-1}, e_{l-1}))$ such that

a) $t_i = k_i s$ for some non-negative integer k_i, for $0 \leq i < l$,
b) The *scaling factors* $e_1, e_2, \ldots, e_{l-1}$ must be positive integers,
c) $t_{i+1} \geq t_i + e_i ps$ for $0 \leq i < l - 1$,
d) $t_{l-1} + e_i ps \leq Ps$.

Macrosequences are very similar to rhythmic patterns. The only difference is that it is not necessary to specify the starting times on a rhythmic pattern, since they can be easily calculated. However, the important issue is that macrosequences can be generated in a similar way than rhythmic patterns, by means of a phrase structure grammar with a probability distribution. We will skip the grammar and present the algorithm to generate macrosequences, along with the parameters involved.

6.2 Creating macrosequences

Let $\mathcal{M}$ be a musical phrase of length p and signature s. In order to construct a macrosequence for $\mathcal{M}$ where the song length is P, we start with a time $t = 0$. We must decide if there will be an instance of $\mathcal{M}$ starting at time t (according to a certain probability). If not, then we increase t to the next allowable starting time, which would be $t' = s$. In fact, it is desirable to have further control over the alignment of the phrases, so we let $t' = rs$, where $r \in Z^+$ is the *alignment parameter*. This means that the instances of $\mathcal{M}$ can only start at times which are multiples of rs. On the other hand, if we decide to place a copy of $\mathcal{M}$ at time t, we need to select a scaling factor e for that copy. This can be done by restricting the range of values e can take and assigning a probability to each one of them. The range can be restricted to the set $\{1, 2, \ldots, q\}$ for some $q \in Z^+$, and the probabilities $p_i \in [0, 1]$ for $1 \leq i \leq q$ should be such that $\sum_{i=1}^{q} p_i = 1$. After choosing a scaling factor, we let $t' = t + eps$ be the next allowable starting time. This process is repeated until $t + eps$ reaches the length of the song.

The range $\{1, 2, \ldots, q\}$ can be the same for all the macrosequences which form a musical composition, or even for a large set of musical compositions.

In fact, it does not make much sense letting q be greater than the song length P.

Now let us define a data structure which groups all the parameters required to construct a macrosequence.

Definition 6.2 The *macrosequence data structure* contains the following fields:

Field			Description
p_a	$\in$	$[0,1]$	Instance probability
r	$\in$	Z^+	Alignment parameter
p_i	$\in$	$[0,1]$	Stretching probability for $i = 1, 2, \ldots, q$

Given a musical phrase $\mathcal{M} = (\mathcal{E}, \tau, s, \mathcal{N})$ of length p, a song length P, a stretching range $\{1, 2, \ldots, q\}$, and a macrosequence data structure D_M, the following algorithm constructs a macrosequence M for $\mathcal{M}$:

1. Start with time $t = 0$ and $i = 0$.
2. Choose some random number $g \in [0, 1)$.
3. If $g < p_a$, a copy of $\mathcal{M}$ must be placed at time t as follows:
   (a) Choose e_i from $\{1, 2, \ldots, q\}$, according to the respective stretching probabilities $p_1, p_2, \ldots, p_q$.
   (b) Let the i-th element of the macrosequence M be (t_i, e_i) where $t_i = t$.
   (c) Increase i by one.
   (d) Let $t' = t + e_i ps$ be the next allowable starting time.
4. If $g \geq p_a$, let t' be the minimum multiple of rs which is greater than t. That is,
$$t' = \min_{k \in Z^+} \{krs \mid krs > t\}.$$
5. Replace t by t'.
6. If $t < Ps$, go back to step 2.

7. Let $l = i$ be the number of elements in the macrosequence M.

One can optionally include a *time displacement* parameter d whose only purpose is to traslate all the instances of $\mathcal{M}$ forward in time. This is done by letting the i-th element of the macrosequence be $(t_i + d, e_i)$ in step 3.b, and comparing if $t + d < Ps$ in step 6. This modification of the algorithm only makes sense when constructing a song from more than one macrosequences.

6.3 Applying a macrosequence to a melody

A macrosequence is intended to arrange a number of copies or instances of a short melody in time, to obtain a longer, repetitive melody. Given a musical phrase $\mathcal{M} = (\mathcal{E}, \tau, s, \mathcal{N})$ of length p where $\mathcal{N} = (\eta_0, \eta_1, \ldots .\eta_{k-1})$, and a macrosequence $M = ((t_0, e_0), (t_1, e_1), \ldots, (t_{l-1}, e_{l-1}))$ for $\mathcal{M}$, and a suitable song length P, the result of applying M to $\mathcal{M}$ is a melody $\mathcal{M}' = (\mathcal{E}, \tau, s, \mathcal{N}')$ where $\mathcal{N}' = (\eta'_0, \eta'_1, \ldots, \eta'_{n-1})$ can be easily constructed by the following algorithm:

1. Let $t = 0$ and $i = 0$.
2. Do the following for $m = 0, 1, \ldots, l - 1$:
   (a) If $t < t_m$ then
      i. Let $f(\eta'_i) = \tau$, $v(\eta'_i) = 0$, and $d(\eta'_i) = t_m - t$.
      ii. Increase i by one.
      iii. Let $t = t_m$.
   (b) Do the following for $j = 0, 1, \ldots, k - 1$:
      i. Let $f(\eta'_i) = f(\eta_j)$, $v(\eta'_i) = v(\eta_j)$, and $d(\eta'_i) = e_m d(\eta_j)$.
      ii. Increase i by one.
   (c) Increase t by $e_m ps$.
3. If $t < Ps$ then
   (a) Let $f(\eta'_i) = \tau$, $v(\eta'_i) = 0$, and $d(\eta'_i) = Ps - t$.
   (b) Increase i by one.

4. Let $n = i$ be the number of notes in $\mathcal{N}'$.

The algorithm inserts the necessary rests (in step 2.a) for each copy of $\mathcal{M}$ to start at the correct time t_m. If necessary, a last rest is added to fill the song length P.

6.4 Compositions

We will no longer abuse of the intuitiveness of the terms "song" and "musical composition". At this point, we have the means to formally define the final product of our work.

Definition 6.3 Given musical phrases $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ with scale $(\mathcal{E}, \tau)$ and signature s, and macrosequences $M_1, M_2, \ldots, M_n$ for $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$, respectively, we define a *composition* or *song* of length P as the set of melodies $\{\mathcal{M}'_1, \mathcal{M}'_2, \ldots, \mathcal{M}'_n\}$ where $\mathcal{M}'_i$ is the result of applying M_i to $\mathcal{M}_i$ for $1 \leq i \leq n$. Each melody $\mathcal{M}'_i$ is called a *part* of the composition.

All parts of a composition are intended to be played together, starting at the same time. If we assume all musical instruments to be monophonic (which of course is not true), we could assign each part to a different instrument. Any polyphonic instrument can be handled as a certain number of separated monophonic instruments of the same type; for example, a six-string guitar can play up to six notes at the same time and it can be simulated by six guitars playing up to one note at a time.

Our definition of a song is obviously not meant to describe music in any way. Many considerations with great influence in the character of a musical piece have been left out or are simply undescribable. However, the algorithms exposed can produce a wide range of compositions, from completely random gibberish to very musical-sounding pieces, while maintaining a considerable degree of malleability over the results.

Chapter 7

Application example and results

In order to fully apply the theory and algorithms described in the previous chapters, a computer application has been built. This application can be found in the included CD-ROM, along with some examples in MIDI and MP3 format. The CD-ROM contains the following files and directories:

/gmf.exe	The example application
/documentation	Documentation for gmf.exe in HTML format
/gmf_examples	Example files for gmf.exe
/midi_examples	MIDI files generated with gmf.exe
/mp3_examples	Fully produced MP3 files
/reports	Reports sent by some users of the program
/thesis	The thesis in Adobe PDF and PostScript format

Some of the examples were generously donated by Alexis Glass and Kevin Breidenbach. Each one of the example directories contains a text file describing the credits for each example. All the files can also be found at the following World Wide Web address:

`http://cactus.iico.uaslp.mx/~fac/`

7.1 The example application

The example program was compiled using Visual C++ 5.0 and it is meant to be run under 32-bit Windows operating systems. We will refer to the example application as GMF from now on. The program was originally written in Spanish and the name derives from “Generador de Música Fractal” (Fractal Music Generator). Also, some dialog boxes may appear in Spanish.

In order to run the GMF program, a computer with the following minimum specifications is required:

- IBM PC compatible computer with Pentium processor or equivallent.
- Microsoft Windows 95/98/ME operating system.
- 32 Mb. of RAM.
- Video card and monitor capable of displaying a resolution of 800x600 pixels at HiColor or TrueColor (16, 24 or 32 bits per pixel).
- Soundcard with integrated General MIDI synthesizer or external MIDI interface.

We will only review the parameters handled by the program, as all the algorithms have already been explained. The CD-ROM includes full documentation and a step-by-step tutorial for those who would like to make their own melodies with GMF.

The program consists of four sections: Global, Microsequences, Blocks, and Macrosequences. The Global section (Figure 7.1) is where the parameters which are common for a song are specified. There is only one exception: the song length, which is specified in the Macrosequences section.

The parameters on the Global section include the scale and signature used for all the blocks, the duration range parameters (a and b for $\mathcal{T}_{a,b}$), and the tempo of the song in beats per minute. The tempo determines how fast or slow the melodies will be played.

The other parameters included in the Global section are the seed for the random number generator (although it is not really used in the program),

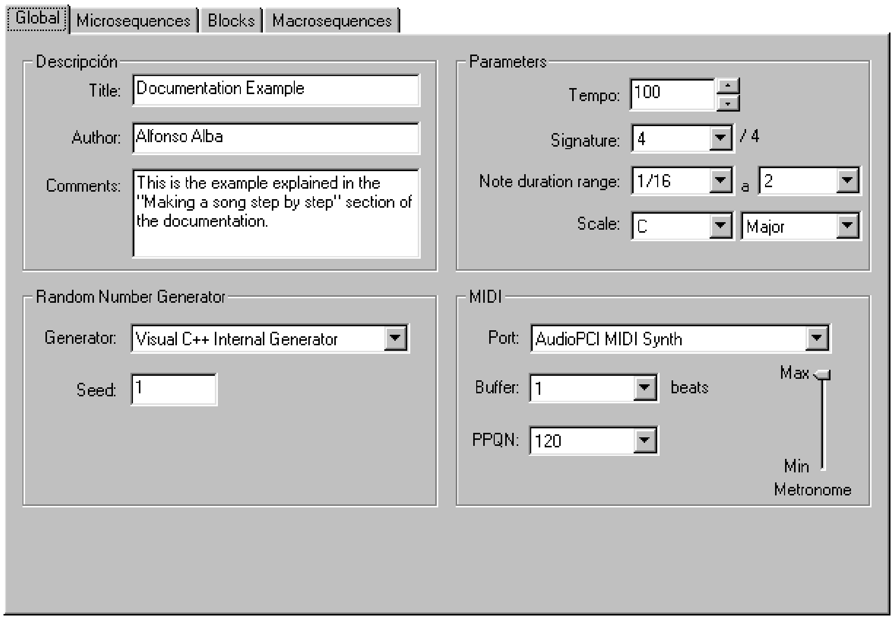

Figure 7.1: Global section.

and some MIDI related parameters. Please refer to Appendix A for more information about MIDI.

In the Microsequences section (Figure 7.2) one can define up to 32 microsequences and construct them using any of the four algorithms described in Chapter 4. These algorithms include Mandelbrot orbits, Mandelbrot straight-line paths, Morse-Thue sequences, and Noise. The documentation describes how to set the parameters for each algorithm.

The Block section (Figure 7.3) is by far the most complex section of the program. It allows the user to create the block data structures which contain the necessary information to generate melodies, as described in Chapter 5. These parameters include the probabilities for the rhythmic pattern grammar, the legato degree (described in Section 5.1.2), the accentuation pattern, the microsequences and parameters corresponding to note frequencies and velocities, and of course, the length of the melody. Up to 16 blocks may be created.

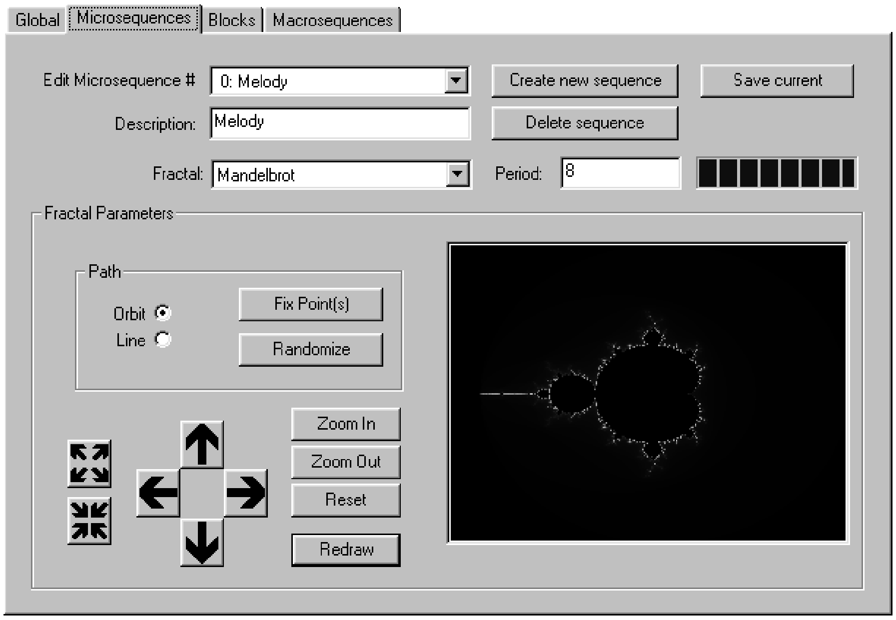

Figure 7.2: Microsequences section.

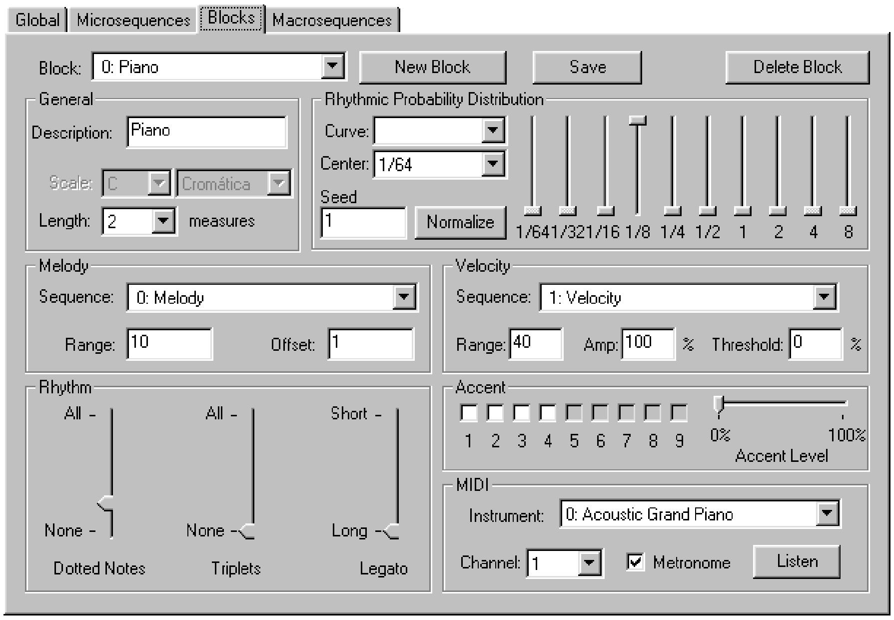

Figure 7.3: Blocks section.

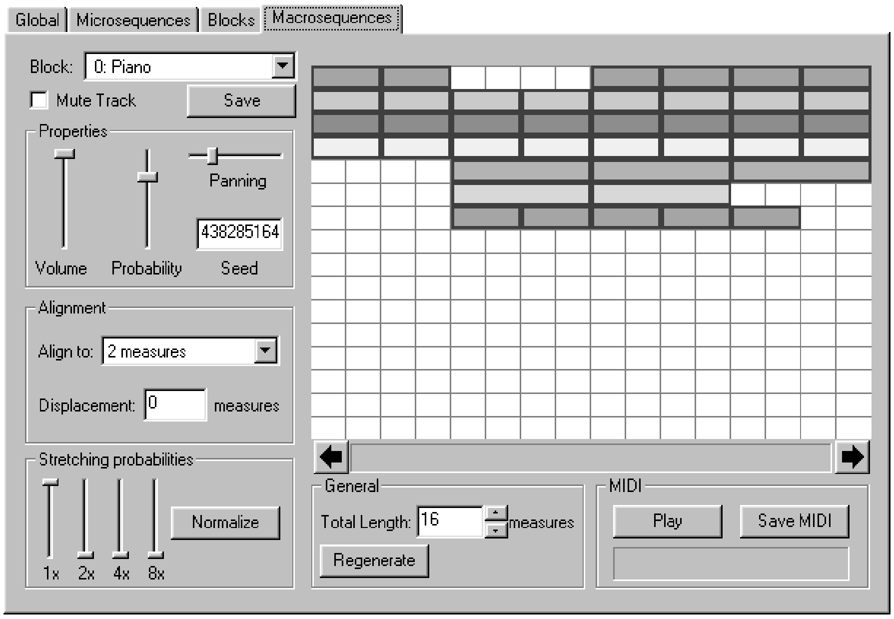

Figure 7.4: Macrosequences section.

Every melody will be assigned a General MIDI instrument and a MIDI channel in order to play a full composition through a device compatible with General MIDI, such as most computer soundcards, or to generate a Standard MIDI File which can be reproduced by most multimedia players.

Finally, the Macrosequences section (Figure 7.4) contains the parameters described in Chapter 6 that are used to construct a full composition from the melodies corresponding to each defined block. The macrosequence parameters are the instance probability, the stretching probabilities, the alignment value, and an additional time displacement parameter. One can also modify the volume and panning (stereo position) of each melody. This is done by using certain MIDI instructions (see Appendix A).

In this section, one can also specify the song length (in measures) and view a diagram of the arrangement of the melodies. The final composition can be played through the selected MIDI port (in the Global section), or can be saved as a Standard MIDI File for further processing (using a MIDI sequencer, for example).

The GMF program is intended to be used by anyone with little (or even none) music knowledge. We have tried to make the interface as friendly and less technical as possible without sacrifying flexibility. Still, creating a full composition involves a large number of parameters. With some practice and a little luck, very good results can be obtained, making this program an interesting tool for experimental musicians and multimedia developers.

7.2 Results

The GMF program was distributed among a few musicians in order to determine the potential use of the algorithms. Many interesting sequences were produced, including a percussion sequence, and also some instrumental compositions were produced by adding drum and percussion lines, effects, and special processing.

The following MIDI files are included in the /midi_examples directory of the CD-ROM:

- drumms.mid
- musicbox.mid
- pinkmister.mid
- ambiental.mid

These are unmodified MIDI files generated by the GMF program [1] and can be played in any General MIDI compatible device or soundcard. The corresponding GMF files (which contain all the parameters used to generate those sequences) are included in the /gmf_examples directory. These files can be opened and edited in the GMF program.

The program was also used to generate sequences that were used in fully produced compositions. Instead of using General MIDI sounds, the melodies were played with different types of synthesizers, adding drums, percussions, effects, and signal processing. The compositions were recorded as standard WAV files and converted to the MP3 format. The resulting files can be found in the /mp3_examples directory and are the following:

[1] The files drumms.mid and musicbox.mid were composed by Kevin Breidenbach.

- mutagene_gmf.mp3 [2]
- fractal_one.mp3
- pinkmister.mp3

The /gmf_examples directory contains the corresponding GMF files for these productions, except fractal_one.mp3. The original sequence for fractal_one.mp3 was generated by an early version of the GMF program whose files are not compatible with the current version.

Some users sent back a report with their opinions about the GMF program. Although some of the opinions differ, most of them agreed about the following issues.

- **Music genres to which the program could be applied.-** According to the users, the GMF program is potentially applicable in almost any type of electronic music, ambiental music, orchestral music, and soundtracks. All the users think that they could use sequences generated by the program, at least occasionally, in their own compositions.
- **Complexity.-** The program is a little difficult to use, but with some practice and with the aid of the included documentation, one can understand how the program works and what does each parameter do.
- **Malleability of the resulting sequences.-** All the users agreed that it takes some time and practice to be able to model interesting sequences which could be used in their compositions. Some of them have suggested some possible improvements in order to obtain more control over the results, although these improvements involve the use of different techniques than the ones developed in this work.
- **Richness of the results.-** The program can generate many original sequences by changing only some of the parameters. Of course, some parameters affect the sequences more than others. This is important because the composer does not have to specify *all* the parameters for every melody. It is enough to make a copy of the microsequence or block parameters and modify some of them.

[2] The file mutagene_gmf.mp3 was composed and produced by Alexis Glass.

Other opinions and comments which are less relevant for this work can be found in the reports sent by the users. These reports are located in the /reports directory of the CD-ROM.

In conclussion, we can say that the GMF program and the algorithms behind it are a promising tool for musicians who like to experiment with new forms of composing, and although it takes some practice and understanding, one can produce interesting results.

7.3 Further development

The algorithms discussed in the previous chapters are flexible enough to allow the inclussion of new methods and techniques that can improve the resulting sequences in many ways. As described in Section 4.4, one can obtain microsequences from other types of mathematical objects, and those same microsequences could be used instead of a random number generator when constructing a rhythmic pattern. This way, the self-similarity and periodicity properties of the microsequences would be also reflected in the rhythm of a composition.

Another useful addition would be to somehow map microsequences to different MIDI controllers. These controllers allow to change certain properties of the sound which is used to play a melody, and are usually related to some parameter of the synthesis method used by the MIDI device. For example, one could generate certain effects by mapping a microsequence to the cutoff frequency of a low pass filter. Other MIDI controllers can be used to modify the volume, panning (stereo position), vibrato (fast variations in pitch), and tremolo (fast variations in volume). Microsequences could be used to automatically modify these parameters and add much more expression to the resulting sequences. The only thing one must have in mind, is that most of these parameters are meant to be modified in a continuous way; thus some interpolation between the values of the microsequence may be required. Even this interpolation could be optional, in order to allow forms of expression which are impossible for a person to play.

There is still much more to discover in the field of algorithmic music, and as we have already said, experimentation is the key.

Appendix A

MIDI basics

The MIDI standard was created in 1983 by a few musical instrument manufacturers in order to provide a way to remotely or automatically control any electronic instrument. The standard was so well-planned that no significant changes have been made since it was first implemented.

Technically, MIDI is an asynchronous serial interface. The baud rate is 31.25 Kbaud, although some instruments have special interfaces for IBM PC compatible computers which run at 38.4 Kbaud. There is one start bit, eight data bits (a byte), and one stop bit. The standard connectors used for MIDI are 5-pin DIN connectors (female on the devices, male on the cables). There are separate jacks and pin connections for incoming and outcoming MIDI data and they must be clearly marked in each MIDI-compatible instrument as MIDI IN and MIDI OUT, respectively. Not all instruments present both connections; some instruments can only send data while some others can only receive it. Some devices present a third connector named MIDI THRU, which is used to pass the incoming signal from MIDI IN to another instrument. This is known as *daisy-chaining* two or more instruments. Due to a small, but accumulative delay in the transmissions, the number of instruments that can be daisy-chained together is limited. More than three devices in a daisy chain can cause serious synchronization problems.

Each physical MIDI connection allows 16 logical channels of transmission. When MIDI data is sent through the connection, the data contains its corresponding channel number, and only the devices programmed to receive data in that channel will process the incoming data. That is how daisy-chained instruments can automatically play different melodies at the same time.

A.1 MIDI messages

The MIDI protocol consists in *messages* which are sent from one device to another. A message is a series of *bytes* (8 bits) that may or may not be recognized by the receiving instrument. If some device receives an unrecognized message, it will do nothing. This allows future additions to the protocol while maintaining backwards compatibility.

Common messages are two or three bytes long. There is also a special type of message, called *system exclusive messages* that can have any length, but as their name implies, those messages are specific to each device model. One thing that all messages have in common is that the first byte of the message is the only one whose seventh bit is set; thus its value ranges from 128 to 255. This is called the *status byte*. The remaining bytes of a message will range from 0 to 127.

There are many messages already defined. We are particularly interested in a category named *Voice messages*. These messages are sent to one of the 16 channels on a MIDI connection and usually instruct the receiving device to play notes. There are other message categories with different purposes, such as sending timing information or controlling some parameters of a specific device.

In order to describe the Voice messages, we will use a binary representation for the status byte, and decimal representation for the remaining bytes. This is because all voice messages contain their corresponding channel number in the least-significative nibble (4 bits) of the status byte. The most-significative nibble contains the message type, which can be one of the following (remember that the most-significative bit is always set on the status byte):

High nibble	**Message type**
1000	Note Off
1001	Note On
1010	Aftertouch or key pressure
1011	Control change
1100	Program change
1101	Channel pressure
1110	Pitch wheel

Note Off - 1000

This message indicates that a particular note should be released. It does not necessarily mean that the note will stop sounding. For example, a key of a piano still sounds for a short time after it has been released. This is a 3-byte message, thus two data bytes follow the status byte. The first data byte contains the note number. There are 128 possible notes on a MIDI device, numbered from 0 to 127, where middle C corresponds to note number 60. The second data byte is the release velocity, from 0 to 127. This indicates how quickly the note was released. Most MIDI instruments do not consider this value and use the defaul value of 64 when sending Note Off messages.

Example: The message 10000010, 64, 100, where the first byte is in binary, indicates that the E note above middle C (with note number 64) has been released and the release velocity is 100. This message is sent in channel 2, since the lower nibble of the status byte is 0010.

Note On - 1001

Indicates that a particular note should be played. It does not necessarily mean that the corresponding note will start sounding since some sounds may have a very long attack (the *attack* is the time it takes for a sound to reach its full volume). This is also a 3-byte message. The first data byte (after the status byte) contains the note number and the second contains the velocity of the note. If the velocity is zero, this message acts as a Note Off message. It is very important that every Note On message is eventually followed by a corresponding Note Off (or Note On with zero velocity) message.

Example: The message 10011000, 60, 100 plays the middle C note with velocity 100 in channel 8. Eventually, a corresponding Note Off message 10001000, 60, 64 (or 10011000, 60, 0) should be sent in order to release the note.

Key pressure - 1010

Some electronic keyboards have pressure sensing circuitry that can detect with how much force a person is holding down a key. While the key is being held down, the musician can vary the pressure (also called *aftertouch*) as an additional way of expression. It is up to a MIDI device to map this pressure value to some parameter which can modify the sound in realtime. There are two types of aftertouch, one which is applied independently to each key, and one that is applied equally to all the notes being played in a certain channel. The former is handled by Key pressure messages, which have two data bytes after the status byte. The first data byte contains the note number to which aftertouch should be applied, and the second byte specifies the pressure amount from 0 to 127, where 127 is the most pressure.

Control change - 1011

A controller can be any variable parameter that implements some function in a MIDI device. Control change messages are used to set the value of a specific controller. There can be up to 128 controllers in any MIDI device, numbered from 0 to 127. Some controller numbers are already assigned to specific controls in every MIDI instrument. For example, the modulation wheel commonly found in electronic keyboards corresponds to controller number 1. When a person moves this wheel, Control change messages are sent through its MIDI OUT connection. The Control change messages contain two bytes after the status byte. The first byte specifies the controller number (from 0 to 127) and the second byte contains the value to which the control should be set, also from 0 to 127.

The following table lists some of the predefined controller numbers:

Controller	Function
0	Bank Select
1	Modulation Wheel
2	Breath Controller
4	Foot Pedal
5	Portamento Time
7	Channel Volume
10	Channel Panning

Example: The message 10110000, 7, 127 sets the volume of channel 0 to its fullest.

Program change - 1100

In a MIDI instrument, a *program* (also called a *patch, instrument* or *preset*) is one of the (possibly many) sounds which can be played. A common synthesizer can make a wide variety of sounds, such as piano, violin, guitar, trumpet, etc., and in many cases, those sounds can be modified or *programmed* and stored as new sounds. The programs in a MIDI device are numbered from 0 to 127; however, many instruments have much more sounds and the user must be able to select any of them. This is accomplished by grouping the programs in *banks* of up to 128 programs each. To select a different bank, one must use a Control change message.

Many MIDI instruments can play more than one program at the same time, but they have to receive MIDI data on a different channel for each program. This feature is called *multitimbrality* and lets a musician compose entire arrangements with only one instrument.

A Program change message consists of two bytes, being the first one the status byte. The second byte is the program which will be selected to play on the channel specified in the status byte.

Example: The message 11000100, 20 selects program number 20 to be played in channel 4. Every note sent to channel 4 will be played with the sound corresponding to program 20.

Channel pressure - 1101

The Channel pressure message specifies the amount of overall pressure which is applied on the keys at a given point. Most synthesizers can be programmed to respond to channel pressure (or channel aftertouch) in some way. This is a 2-byte message where the first data byte (after the status byte) specifies the pressure amount from 0 to 127.

Pitch wheel - 1110

The pitch wheel (commonly found in electronic keyboards) is used to continuously slide or bend the pitch of the notes being played up or down. When the wheel is moved, the device outputs Pitch wheel messages, which consist on three bytes. The first byte is the status byte. The other two bytes should be combined to form a 14-bit value. The first data byte contains the seven less-significative bits of the 14-bit number and the second byte contains the seven most-significative bits. They can be easily combined with the following formula:

$$\text{14-bit value} = \text{second data byte} \times 128 + \text{first data byte}.$$

A combined value of 8192 means that the pitch wheel is centered, thus the notes are not being transposed up or down. Higher values slide the pitch up, and lower values slide the pitch down. The transposing range is usually adjustable by programming the device.

Example: The message 11100000, 16, 78 indicates that the notes being played in channel 0 must be transposed up (since $78 \times 128 + 16 = 10000$ is above the center position). The transposing amount is usually determined by the selected program for channel 0.

A.2 Converting note sequences to MIDI data

Eventually, we need to somehow reproduce the melodies we create. We do this by converting the note sequence of a melody into a sequence of MIDI messages. Each one of these messages must be output through a MIDI connection at a specific time. Thus we can define a MIDI message as a quartet $m = (t, s, d_1, d_2)$ where t is the time at which the message should be sent, s is the status byte, and d_1, d_2 are the data bytes. Some messages will only use d_1. The time t is measured in *points per quarter note* or *ppqn*. A quarter note has a length equivallent to a beat. MIDI devices use a subdivision of a quarter note as their time unit. Common subdivisions are 48, 96 and 192 points per quarter note. We will denote the properties of a message m as $t(m)$, $s(m)$, $d_1(m)$, and $d_2(m)$.

For each note in $\mathcal{N}$ we must generate two MIDI messages: a Note On message and a corresponding Note Off message. Also, we will assume that a note frequency of zero corresponds to middle C, which in turn corresponds to MIDI note number 60. Thus we want to generate a sequence of messages $m_0, m_1, \ldots, m_{2k-1}$ where

$$\begin{aligned}
t(m_{2i}) &= \text{ppqn} \times t(\eta_i), \\
s(m_{2i}) &= 144 + \text{channel}, \\
d_1(m_{2i}) &= f(\eta_i) + 60, \\
d_2(m_{2i}) &= \lfloor 127 \times v(\eta_i) \rfloor
\end{aligned}$$

$$\begin{aligned}
t(m_{2i+1}) &= \text{ppqn} \times (t(\eta_i) + \lambda d(\eta_i)), \\
s(m_{2i+1}) &= 144 + \text{channel}, \\
d_1(m_{2i+1}) &= f(\eta_i) + 60, \\
d_2(m_{2i+1}) &= 0,
\end{aligned}$$

for $0 \leq i < k$.

The channel parameter specifies one of the sixteen channels available on a MIDI port. In the example application, a different channel and program number is assigned to each melody. A Program change message is sent at time $t = 0$ in order to play the melody with the desired sound. The ppqn parameter specifies the number of subdivisions of a quarter note used for timing. This value can be adjusted depending on the receiving MIDI device, although it is usually not necessary. Finally, the parameter λ is the legato degree described in Section 5.1.2. As we had previously said, the easiest way to apply legato is after calculating the starting time $t(\eta_i)$ for each note.

Usually, the messages should be sorted by their time property; however, this is not necessary since $t(\eta_i) + d(\eta_i) \leq t(\eta_{i+1})$ and $\lambda \leq 1$. The sorting must be done when playing more than one melody at the same time since their rhythmic patterns are not related.

A.3 General MIDI

If a MIDI sequence is played on different devices, the results may be completely different because of the different sounds assigned to the same program number in each device. A melody which is played with a piano program in one device will dramatically change if played on another device where the same program number corresponds to a drum kit. To overcome this problem, the General MIDI Specification was created. This specification defines a set of 128 programs corresponding to generic instrument sounds, such as piano, guitar, bass, drums, and others. It also defines a set of controllers to vary certain aspects of the sound. Many sample-based synthesizers and computer soundcards include a bank of programs which is compatible with General MIDI. Other devices, such as drum machines and effects processors, are unable to reproduce the sounds specified by General MIDI and have their own program specifications. The example application assumes General MIDI (GM) compatibility and allows to select any of the 128 predefined programs. If the sequences are sent to a device which is not GM compatible, the results will be completely device-dependant. A GM compatible device must also be multitimbral and able to reproduce 16 channels of MIDI data at the same time, with any GM program selected for each channel.

References

[1] Barnsley, Michael. *Fractals Everywhere* Academic Press, 1993.

[2] Howie, John M. *Automata and Languages* Clarendon Press, Oxford, 1991.

[3] Schillinger, Joseph. *The Schillinger System of Musical Composition Vol. 1.* Carl Fischer, Inc., New York, 1941.

[4] Voss, Richard F. and Clarke, John. $1/f$ *noise in Music: Music from 1/f noise.* J. Acoust. Soc. Am.

[5] Mandelbrot, Benoit. *The Fractal Geometry of Nature.* W.H.Freeman & Co., NY, 1982.

[6] A. Danhauser. *Teoría de la Música*

[7] Pérez Ortiz, Juan Antonio. *Música Fractal: El Sonido del Caos.*
`http://www.dlsi.ua.es/~japerez/`

[8] Schulz, Claus-Dieter. *The Fractal Music Project.*
`http://www-ks.rus.uni-stuttgart.de/people/schulz/fmusic`

[9] Jackson, Phil (moderator). *Fractal Music e-mail group.*
`http://groups.yahoo.com/group/cnfractal_music`

[10] Techlar Technologies. *Fantastic Fractals.*
`http://www.techlar.com/fractals`

[11] *The Fractal Music Lab.*
`http://www.fractalmusiclab.com`

[12] *MIDI Technical Fanatic's Brainwashing Center.*
`http://www.borg.com/~jglatt/`