Musical Rendering of Mathematical Objects

Erik A. Schafer and Marcus H. Pendergrass
Department of Mathematics and Computer Science, Hampden-Sydney College, Hampden-Sydney, VA 23943

This project explored methods of applying mathematics to the generation of music, and the methods for analyzing music with mathematics. Music was generated randomly using Markov Chains, with specific start nodes, end nodes, and lengths. We applied specific transition probabilities and were able to generate random melodies with consistent and similar structures. Music was also generated deterministically using a binary operation across pitch patterns called composition. We proved that the composition operation is associative. Green's relations, a set of equivalence relations that describe how a binary operation acts in a semigroup, were scrutinized. Finally, the project has added to a software suite which produces and plays melodies utilizing these methods.

Musical Rendering of Mathematical Objects

Erik Schafer

March 16, 2012

Abstract

This project explored methods of applying mathematics to the generation of music, and the methods for analysing music with mathematics. Music was generated randomly using Markov Chains, with specific start nodes, end nodes, and lengths. We applied specific transition probabilities and were able to generate random melodies with consistent and similar structures. Music was also generated deterministically using a binary operation across pitch patterns called *composition*. We proved that the composition operation is associative. Green's relations, a set of equivalence relations that describe how a binary operation acts in a semigroup, were scrutinized. Finally, the project has added to a software suite which produces and plays melodies utilizing these methods.

1 Introduction

This project explored methods to represent mathematical objects in a musical way. Two primary approaches were used, a random method was developed using finite state graphs, and a deterministic method was developed using a binary associative operation defined across the set of all melodies.

The project developed a system to use Markov Chains to generate melodies with specific constraints, and a method to randomly select from those melodies. In the development of this process, the concept of Sparse Matrix Multiplication was created and used to improve the efficiency of the algorithms. Furthermore, an algorithm was created which could find all paths of a given start, end, and length through a graph. Finally, algorithms were constructed which could calculate the relative weight of a single path among all paths of similar qualities.

In the exploration of deterministic generation the operation of *composition* was examined. It was proven that the binary operation across the set of all melodies was associative. This allowed the set of all melodies to be seen as a *Semigroup*. Finally, this invited the analysis of Green's Relations across the semigroup. The result was that all of Green's Relations were equivalent, and could be summarized using the musical relation of *Transposition*.[2]

2 Random Methods of Generating Music

One part of this project is to generate music randomly. The simplest conceivable way to do this would be to generate entirely random numbers to represent pitches and durations. An approach this simple, however, would produce uninteresting and insignificant results. Instead, a method was sought which could randomly generate a sequence of notes, albeit according to a number of rules. These dependencies imbue the result with a structure and order, necessary components to prevent disorganized cacophony from resulting. The rules determine the general qualities of the music, while the individual notes are actually generated in a random way.

2.1 Graph-Based Methods

Graphs were the primary method used to generate melodies randomly. For the purposes of the project, graphs were seen as finite state chains, where numbered *nodes* linked to others along *edges*. A graph can be either sparse or dense. A sparse connected graph has many nodes with fewer edges, resulting in a limited number of *paths* from point a to point b. Conversely, a dense connected graph would have many edges, and a great number of path options from one node to another. This concept of nodes with edges became the foundation of graph- based music generation.

A melody taken from this graph would simply be a sequence of numbers corresponding to 'visited' nodes. To produce a melody, one would trace a path of any length through the graph. Additionally, one could trace all paths of a given length from a given start node, or even from a given start node to an end node. Each path, or list of integer node numbers, could then be indexed into a scale, and produce a melody.[1] This project focused on calculating all paths between two given points, and then selecting between them.

Markov Chains allow the ordered-yet-random selection of paths. A Markov Chain is just like a graph, with numbered nodes and edges, but with one distinct difference: edges are given 'weight' values. These weight values are used to calculate 'transition probabilities' by dividing all of the weights leaving a node by their sum. For example, if a given node has three edges leading away from it with weights $2, 7, 1$ respectively, the probability of each transition is the weight divided by the sum of the weights at

that node : $.2, .7, .1$. These transition probabilities are in turn used to calculate the weight of each path. An algorithm first finds all possible paths of a given length from one node to another. Then, the probabilities of each transition of a path are multiplied by one another, presenting the 'path weight'. Finally, each path weight is divided by the sum of path weights, and the result becomes the path's probability.[4]

The chain that this graph generates is represented conceptually simply as a finite sequence of positive integers. In order to render the chain musically, it first must be converted to notes. To do this, a sequence of notes is generated (often according to a musical scale) and each integer becomes an index into that sequence. For instance the sequence $0, 1, 3, 2$ applied to a C-major pentatonic scale (C, D, E, G, A) would result in the melody C, D, G, E. If the chain sequence attempts to index a note not included in the scale, it can either *reflect* or *wrap*. For example $0, 2, 8, 6, 3$ indexed into C, D, E, G, A can become C, E, C, E, G if 'reflecting' or C, E, G, D, G if 'wrapping'. The difference in musical quality is not immediately evident. Reflecting better emulates the actual composition of music because it prevents large intervals between notes from arising. When using large scales (i.e. a major pentatonic scale that contains several octaves) and wrapping around, a very high note could be found adjacent to a series of lower notes, which would not be musically pleasing.

Using the scale indexing Markov Chain graphs can be much easier to understand conceptually. Consider a random piano player Mr.Markov. Mr.Markov first observes his piano keyboard and selects a subset of the keys to be played. This is analogous to the selection of a sequence of notes. Mr.Markov then selects a first key to play, and begins to transition randomly through his notes. According to his choice of transition probabilities, he may only rarely transition from an A to a C (for instance), while almost always following an F with a G. Every key he has selected may be subject to similar rules. Finally, Mr.Markov decides to finish his piece on a specific note. This is a simplification of what the program does to select a chain.

2.1.1 Sparse Matrices

All Markov Chain graphs can be represented using a matrix 'A'. $A_{i,j}$ represents the transition weight from node i to j, with 0 representing no edge from i to j. Nodes are indexed from 0 throughout. For example, the matrix

$$\begin{matrix} 1 & 9 & 0 & 0 \\ 0 & 0 & 7 & 3 \\ 0 & 0 & 0 & 2 \\ 6 & 4 & 0 & 0 \end{matrix}$$

represents the following graph:

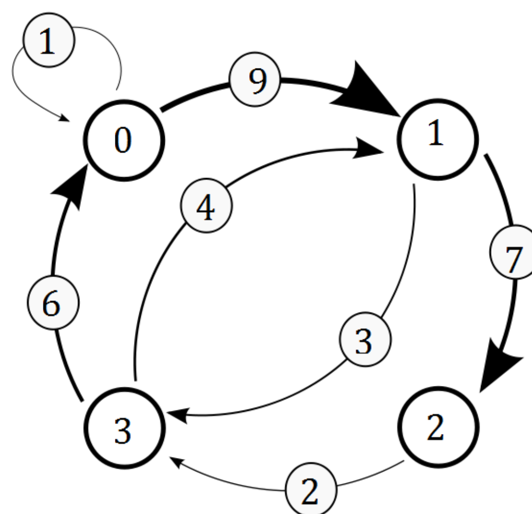

Figure 1. A Markov Chain, with labelled nodes and edge weights. Additionally, the width of the arrows are scaled with respect to the weights.

Notice that most of the entries of the matrix are 0. In order to prevent chaos in composition, and in order to provide some structure to the melody, the graph is sparse. This means that each node in the graph has far fewer connections than possible. In order to more efficiently handle these situations, graphs are represented as two "sparse matrices" in the program. Thus the above example is reduced to the ragged index array (left) and weight array (right). The value of the index array represent the places the values of the weight array would hold in a full matrix.

$$\begin{matrix} 0 & 1 \\ 2 & 3 \\ 3 & \\ 0 & 1 \end{matrix} \qquad\qquad \begin{matrix} 1 & 9 \\ 7 & 3 \\ 2 & \\ 6 & 4 \end{matrix}$$

This saves memory space, but makes function handling difficult, requiring a new version of matrix multiplication to be used. It is important to note that every entry of the resulting matrix C is the sum of the product of a unique row of A and column of B, in other words $c_{i,j} = a_{i,*} * b_{*,j}$.

The algorithm goes through all non-zero entries in the first matrix, then finds any non-zero entries in the second matrix that it would be multiplied by, and adds the result to the corresponding entry of the resulting matrix. Finally, the matrix is cropped and put into sparse matrix form, if applicable.

Here is pseudo-code for the Sparse Matrix Multiply Algorithm in the form $A * B = C$:

Sparse Matrix Multiply: A * B = C
FOR every row of A, (A_i)
  FOR every nonzero entry of the row (A_{ij}),
    FOR every nonzero entry (B_{jk})
      MULTIPLY A_{ij} and B_{jk}
      STORE the result by adding it to C_{ik}
    END FOR
  END FOR
END FOR
CONVERT C into a sparse matrix.

2.1.2 A Path Generation Algorithm

In order to randomly select a single path from a graph, every path of a specified length, start, and end node had to be calculated. One important fact in the development of the algorithm which did this was to note that the i, j entry of the *transition matrix* to the nth represents the number of paths of length n from node i to node j. Additionally, the i, j entry of the weight matrix to the nth power is non-zero if there is a path of length n from node i to node j. The algorithm hinges upon this fact, using it to check whether or not a given path is capable of reaching the end node from a given position and length.

The getPaths algorithm uses a Java *vector* to process the paths. Vectors allow elements to be removed, processed and reinserted in a determined order. The algorithm begins by initializing the vector to contain paths of length two, starting at the start node, and ending at any neighbouring nodes which have paths linking them to the end node. Then each element in the vector stack is processed, where each step adds another vertex to the path. At each step as these paths grow, all neighbours of the current last vertex area analysed and checked. If it is found that they may lead to the final vertex the extended path is added back into the vector for an additional processing pass. Finally, to save processing power, the final vertex is blindly appended to every path.

Here is pseudo- code for the getPaths algorithm.
Let n represent the desired path length. Let i be the desired start node. Let j be the desired end node. Let A be the transition matrix.
To begin, initialize an array of all paths of length one (two nodes) starting at i.

FOR each Path, starting at i, length 1, ending at neighbouring node k
  IF $A^{n-1}_{k,b} \neq 0$
    INSERT the path into the vector
  END IF
END FOR

FOR r starting at 0, as long as $r < n - 2$, add one to r for each iteration. FOR each element of the vector
    REMOVE the path from the vector
    CONSTRUCT an array of Paths extending the current Path by 1
      FOR each Path in the array, ending at node k
        IF $A^{n-r-3}_{k,b} \neq 0$
          INSERT the path into the vector
        END IF
      END FOR
  END FOR
END FOR

FOR each element of the vector
  REMOVE the path from the vector
  ADD b to the path
  INSERT the path into the vector
END FOR

RETURN the vector converted to a Path array

3 Deterministic Methods of Generating Music

One of the things that makes music pleasing is the presence of symmetry, patterns, and self reference. In order to capture these qualities a *Composition* operation was developed. It allows one or two sequences of numbers (which as discussed earlier represent notes) to produce a new sequence that contains self similarity, producing a euphonic quality when indexed into a scale. Furthermore this function when shown to be associative allows sequences of notes to be viewed through the lens of Semigroup Theory, a branch of Abstract Algebra.[2]

3.1 Definition of Composition

We begin by defining the concept of a *pitch pattern.* Intuitively, a pitch pattern is simply an ordered sequence of pitches, without reference to any rhythmic relationships they might have. We focus on the case in which the pitches come from a fixed scale, in which case they can be represented by an integer index into the sequence of notes making up the scale. Thus, mathematically, a pitch pattern is simply a finite sequence of integers. Let S denote the set of all pitch patterns:

$$S = \{(a_0, a_1, a_2, ..., a_{n-1}) : a_i \in \mathbb{Z}, n \in \mathbb{N}\}, \quad (1)$$

Composition of pitch patterns is a binary operation on S, such that the result consists of the concatenated results of the addition of each element of the first sequence a in turn to every element of the second series b. Letting $*$ denote the composition operator, we have, for any a and b in S,

$$*(a, b) \equiv ab = (a_0 + b, a_1 + b, ..., a_{n-1} + b) \quad (2)$$

Here, the addition of a scalar to a vector (e.g. $a_0 + b$) is component-wise, and the comma denotes the concatenation operation.

When two pitch patterns are composed together, they produce a result that contains structures from both operands. There exists a *Macro* structure to the

result, which mimics the intervals of the first operand. The second structure is termed a *Micro* structure, where the intervals of the second operand are copied across the *Macrostructure*. As an examples, consider ab, where $a = (0, 1, 3, 2)$ and $b = (3, 1, 5, 2)$.

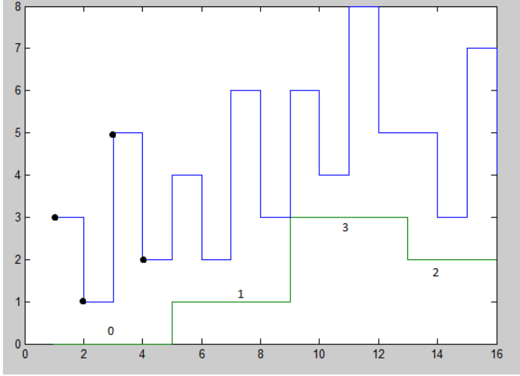

Figure 2. A graph demonstrating composition.

As you can see the micro structure (black dots) are copied and shifted according the the macro structure.

Each entry, ab_i, in the resulting series is a sum of two specific entries of each operand. The entry of b which is added "wraps" around (much like a clock)[5], every $|b|$th entry, so we can say that the index of the entry of b is r, where r is the integer remainder of the division of the index of the result i and the length of b. r is necessarily less than the length of b while being greater than or equal to 0. Additionally, the index of the entry of a which is added increments after every $|b|$ entries, being the integer quotient of i and the length of b. The relationships are summarized as:**??**

$$ab_i = a_r + b_q$$

$$i = |b|q + r \qquad 0 \leq r < |b|$$

The quotient q of the division of i and the length of the second operand $|b|$ is the entry of a , while the remainder is the entry of b. This equation represents the nature of the operation, where every b is copied $|a|$ times, and each segment of the series of length $|b|$ has a common a_n added to it.

3.2 Associativity

Theorem 1 (Associativity of Composition)**.** *The operation of* composition *is associative on the set of all pitch patterns*

Proof. Let α, β, and γ be pitch patterns of length m, n, and p respectively. Calculating the ith entry of $\alpha(\beta\gamma)$, one finds that

$$\left(\alpha(\beta\gamma)\right)_i = \alpha_{q_1} + \beta_{q_2} + \gamma_{r_2} \tag{3}$$

where the indices are nonnegative integers satisfying

$$i = q_1 np + r_1, \quad 0 \leq r_1 < np \tag{4}$$
$$r_1 = q_2 p + r_2, \quad 0 \leq r_2 < p \tag{5}$$

Similarly the ith entry of $(\alpha\beta)\gamma$ is

$$\left((\alpha\beta)\gamma\right)_i = \alpha_{q_4} + \beta_{r_4} + \gamma_{r_3} \tag{6}$$

where

$$i = q_3 p + r_3, \quad 0 \leq r_3 < p \tag{7}$$
$$q_3 = q_4 n + r_4, \quad 0 \leq r_4 < n \tag{8}$$

Note first that

$$r_2 = (i \bmod np) \bmod p = i \bmod p = r_3$$

so that the indices on γ are equal in equations (3) and (6). Also, q_3 is the number of p's in i, which by (4) equals the $q_1 n$ plus the number of p's in r_1, i.e.

$$q_3 = q_1 n + q_2 \tag{9}$$

We claim that $q_2 = r_4$. From (8) and (9) it suffices to show that $q_2 < n$. But this follows directly from (4) and (5). Now $q_1 = q_4$ follows from (8) and (9), and we have shown that the indices on α, β, and γ are the same in equations (3) and (6). Thus $\alpha(\beta\gamma) = (\alpha\beta)\gamma$, and composition is associative. □

3.3 Green's Relations

Green's relations are a series of fundamental equivalence relations defined in the field of semigroup theory. There are five relations defined as Green's relations, named $\mathcal{R}, \mathcal{L}, \mathcal{J}, \mathcal{H}, \mathcal{D}$. For the definitions of the relations, S^1 is used to represent any monoid semigroup, or in other words, any semigroup with an identity element. Green's relations utilize the concept of an *ideal*. An ideal is a subsemigroup whose elements map back into the subsemigroup under the binary associative operation of the semigroup.[2]

Let Green's $\mathcal{R}$ relation be defined as: $a\mathcal{R}b \iff aS^1 = bS^1$

Let Green's $\mathcal{L}$ relation be defined as: $a\mathcal{L}b \iff S^1a = S^1b$

Let Green's $\mathcal{J}$ relation be defined as: $a\mathcal{J}b \iff S^1aS^1 = S^1bS^1$

Let Green's $\mathcal{H}$ relation be defined as: $a\mathcal{H}b = \mathcal{R} \cap \mathcal{L}$ in other words if $a\mathcal{R}b$ and $a\mathcal{L}b$ then $a\mathcal{H}b$

Let Green's $\mathcal{D}$ relation be defined as: $a\mathcal{D}b \iff \exists\gamma \in S^1 \quad : \quad a\mathcal{L}\gamma \wedge \gamma\mathcal{R}b$

Before the relations are examined, it shall be proven that $\mathcal{R}$ is an equivalence relation. The other three relations all have trivially similar proofs for equivalence, and shall be omitted from this paper. A relation $(\sim)$ is considered an equivalence relation if it satisfies the following properties:

1) Reflexive

$$a \sim a \quad \forall a$$

2) Symmetric

$$a \sim b \iff b \sim a \quad \forall a, b$$

3) Transitive

$$a \sim b, b \sim c \iff a \sim c \quad \forall a, b, c$$

Proof. Let $a, b, c \in S^1$

Suppose $a\mathcal{R}a$. By the definition of $\mathcal{R}$

$$a\mathcal{R}a \iff aS^1 = aS^1$$

Therefore, $\mathcal{R}$ satisfies the reflexive property.

Suppose $a\mathcal{R}b$.

$\iff aS^1 = bS^1$
$\iff bS^1 = aS^1$
$\iff b\mathcal{R}a$
Therefore, $\mathcal{R}$ satisfies the symmetric property.

Suppose $a\mathcal{R}b$, $b\mathcal{R}c$.
$\iff aS^1 = bS^1$, $bS^1 = cS^1$
$\iff aS^1 = cS^1$
$\iff a\mathcal{R}c$
Therefore, $\mathcal{R}$ satisfies the transitive property.

Since $\mathcal{R}$ satisfies the reflexive, symmetric, and transitive properties, $\mathcal{R}$ is an equivalence relation. □

3.3.1 Examining Green's Relations

On the semigroup of pitch patterns, it turns out that all of Green's relation's can be reduced to the idea of *transposition*. This is exciting because it reflects an already defined relationship in music. When two pieces are said to be transpositionally related we mean to say that they are identical if shifted linearly up or down a scale. More formally, the transposition relation $\mathcal{T}$ is defined by:

$a\mathcal{T}b$ if and only if $b = k + a$ for some $k \in \mathbb{Z}$.

(Recall that $k + a$ means add k to each entry in a.) Clearly $\mathcal{T}$ is an equivalence relation.

Theorem 2 (Green's Relations on the Semigroup of Pitch Patterns)**.** *All of Green's Relations are equivalent across pitch patterns, being summarized by the relation of transposition, $\mathcal{T}$.*

$$\mathcal{R} = \mathcal{L} = \mathcal{H} = \mathcal{J} = \mathcal{D} = \mathcal{T}$$

Proof. Recall:

$$a\mathcal{R}b \iff \forall s_1 \in S \quad \exists s_2 \in S : \quad as_1 = bs_2$$

Given $(0) = e \in S$,
if $s_1 = e$, or $s_2 = e$ then
$a = bs_2$ or $b = as_1$
Let $|a|$ denote "the length of a"
then
$|a| = |b||s_2|$ or $|b| = |a||s_1|$
and by substitution $|a| = |a||s_1||s_2|$ which simplifies to

$$1 = |s_1||s_2|$$

Since 1 is its only factor, $|s_1|, |s_2| = 1$. Substituted into $|a| = |b||s_2|$ we find that

$$|a| = |b|$$

Finally, noting that $a = bs_2$ (if $s_1 = e$) it can be concluded that

$$a = (b_0 + k, b_1 + k, b_2 + k, ..., b_{n-1} + k) k \in \mathbb{Z}, n = |b|$$

and therefore $a\mathcal{R}b \iff a\mathcal{T}b$

The proof that $\mathcal{L} = \mathcal{T}$ is entirely analagous.

Recall:

$$a\mathcal{H}b \iff a\mathcal{R}b \wedge a\mathcal{L}b$$

It has been shown that $a\mathcal{R}b \iff a\mathcal{T}b \iff a\mathcal{L}b$ it follows that $a\mathcal{R}b \iff a\mathcal{L}b \iff a\mathcal{H}b$ or simply

$$a\mathcal{H}b \iff a\mathcal{T}b$$

Recall:

$$a\mathcal{D}b \iff \exists\gamma \in S \quad : \quad a\mathcal{L}\gamma \wedge \gamma\mathcal{R}b$$

Given $a\mathcal{R}b \iff a\mathcal{T}b$ and $a\mathcal{L}b \iff a\mathcal{T}b$ the relation $\mathcal{D}$ can be redefined as:

$$a\mathcal{D}b \iff \exists\gamma \in S \quad : \quad a\mathcal{T}\gamma \wedge \gamma\mathcal{T}b$$

which is true by the transitive property of equivalence relations, so it can be concluded that

$$a\mathcal{D}b \iff a\mathcal{T}b$$

Recall:

$$a\mathcal{J}b \iff \forall s_1, s_2 \in S^1 \quad \exists s_3, s_4 \in S^1 : \quad s_1as_2 = s_3bs_4$$

Given $(0) = e \in S$,
if $s_1 = e$, and $s_2 = e$ then
$a = s_3bs_4$
and by the symmetric property, $b = s_1as_2$ Let $|a|$ denote "the length of a"
then
$|a| = |s_3||b||s_4|$
and $|b| = |s_1||a||s_2|$
and by substitution $|a| = |s_3||s_1||a||s_2||s_4|$ which simplifies to

$$1 = |s_3||s_1||s_2||s_4|$$

Since 1 has no factors, $|s_1|, |s_2|, |s_3|, s_4| = 1$. Substituted into $|a| = |s_3||b||s_4|$ we find that

$$|a| = |b|$$

Finally, noting that $a = s_3bs_4$ (if $s_1, s_2 = e$) it can be concluded that

$$a = (b_0 + k, b_1 + k, b_2 + k, ..., b_{n-1} + k) k \in \mathbb{Z}, n = |b|$$

and therefore $a\mathcal{J}b \iff a\mathcal{T}b$ □

4 Conclusion

This project succeeded in exploring methods to generate music by both deterministic and random processes. It developed specific methods for the processing the multiplication of large, sparse matrices. This was used to expand the field of algorithmic composition by allowing Markov Chains to generate melodies of specific start and end states, as well as specific length. Additionally, an operation was defined which allowed the composition of melodies to be seen as a purely mathematical process. Reducing melodies to semigroups, the composition function allowed Green's relations to be observed. A theorem was created which generalizes the meaning of Green's relations.

Future work can continue to expand the breadth of the software suite, was well as the application of the process developed in this project. Rhythms have yet to be introduced to the algorithmic composition, and could be generated and assigned by reapplying Markov Chains or an operation similar to composition to rhythms. Also, different combinations of the composition operation could be analysed, as well as different configurations of Markov Chains. For instance, it has been observed that pitch patterns starting in 0 all have a similar musical quality, and appear to have rhythms. There are surely many more generalizations, both qualitative and quantitative, that could be made from continued study and experimentation with the structures that have been developed.

References

[1] Harkleroad, Leon. The math behind the music . Cambridge: Cambridge University Press, 2006. Print.

[2] Hollings, Christopher D. "Some First Tantalizing Steps into Semigroup Theory." Mathematics Magazine 80.5 (Dec., 2007): 331-44. Print.

[3] "Division Algorithm." Wikipedia, the Free Encyclopedia. Web. 04 Sept. 2011.

`<http://en.wikipedia.org/wiki/Division_algorithm>`

[4] "Markov Chain." Wikipedia, the Free Encyclopedia. Web. 04 Sept. 2011.

`<http://en.wikipedia.org/wiki/Markov_chains>`

[5] "Mod – from Wolfram MathWorld." Wolfram MathWorld: The Web's Most Extensive Mathematics Resource. Web. 04 Sept. 2011.

`<http://mathworld.wolfram.com/Mod.html>`