Algorithmic Composition of Directional Self-Similar Music

Abstract

This project attempts to produce music in which every single note's purpose for existence can be explained on some musical level, and it does so by using a recursive algorithm written in Haskell using the Euteropia library to output MIDI. There are three musical elements in particular used to describe why each note is there: self-similarity, meaning that the musical material is repeated in some way, allowing any module, even potentially one of random noise, to be described by its repetitions; directionality, meaning that each note can be explained as either coming from a previous note or leading to a following note at some distance away; and lastly, pre-defined Classical and Galant structures of musical balance, such as periods (antecedents and consequents) and sentences (basic ideas and continuation). The construction of this music is rhythm-centric, starting with broad strokes that lay out the foundational structure of a piece, then focusing closer and closer to apply the three elements at various sizes. For example, directionality is approached by placing large pick-up notes and after-beats surrounding some of the notes, then doing the same again but at half the reach, and so on. This allows the listener to perceive a higher “direction” or “thread” that the music is following. The main musical example in this project is a piano piece generator entitled “Sturm und Drang”, allowing the user to specify the length of the piece and receive a randomly-designed dramatic keyboard work inspired by an 18th-century movement of the same name. The methodology used to construct the piece can be extended to match a wide variety of musical styles.
Background: Giving Purpose to Each Note

This project is ultimately inspired by rhythm in classical music, and the earliest experimental code had only one pitch being struck at various durations and volumes. One common approach to automate the generation of rhythm is to assign a hierarchy of importance to all of the possible note positions within a measure, then treat this hierarchy as a probability distribution. For example, in a duple meter like 4/4, if we consider the downbeat as the most important position, then halfway between two downbeats (beat 3) is second-most important, and halfway between beat 3 and the downbeats (beats 2 and 4) are third-most important, and so on. The final result of these divisions looks something like the distribution in Figure 1. If these divisions are treated as a probability distribution, it is more likely for a note to land on the beat rather than off the beat, which makes intuitive sense.

Certain musical styles lend themselves better to computer-generated improvisations of this type. For example, in “Six Techniques for Algorithmic Music Composition”, one of the techniques is called “stochastic binary subdivision” in which a drum set rhythm is produced by recursively splitting notes into two smaller notes, therefore maintaining the emphasis on the more important locations (Langston). However, many musical styles have aesthetic requirements that are more structured. For example, look at the following violin excerpt from Mozart's Symphony No. 31:

Figure 2: An excerpt from the first and second violin part from the first movement of Mozart's Symphony No. 31, K.297/300a
Although the half-note and quarter notes in Figure 2 all happen on strong beats, if the distribution in Figure 1 was treated as a probability distribution, it would be very unlikely to form the rhythm in Mozart's symphony due to the many short sixteenth note pick-ups and its repetitive (or self-referential) nature.

Each of the notes in Figure 2's passage can be explained in some way by their relation to the surrounding notes. For example, the group of sixteenth notes in the first bar form a scale up to the high D, which is carried over from the lower D's that came before; the last two notes of the second bar are a pickup to the downbeat of the third bar; the groupings that follow (dotted eighth + sixteenth + quarter) are repetitions of that original pickup; each 3-note repetition forms an arpeggio of the next chord; the quarter notes in the third and fourth bars are part of a descending scale; and so forth. Unfortunately it would be very difficult to reproduce such intricately constructed music using probability distributions and other statistical or AI models (such as Markov chains and neural networks), because capturing each note's explanatory value or significance within the greater work is no trivial task. Additionally, from this analysis, it should be evident that rhythm plays just as important a role in producing a style of music as pitch does, and the two concepts are highly interdependent.

This project attempts to produce music in which each note's purpose can be explained on some level. While there are many analytical methods one could use to explain why a note exists, this project will focus on three aspects in particular:

1. **Directional relationship to notes reaching out on either side** – in other words, does a note serve as a pick-up to the next note, or as coming away from the previous note, or both?

2. **Self-similarity between modules** – One interesting observation is that even if the sounds within a module are completely arbitrary, repeating the module forms a recognizable pattern.

3. **Pre-defined Galant and Classical structure of balance** – these include antecedent phrases, consequent phrases, sentences (idea + continuation), key aims, and so forth
Terms

This paper will use several terms that have meanings specific to this project's code. The first is what is meant by *embellishment*. Each piece starts as a given outline of some kind, which in everyday language might be thought of as “play a phrase in C for 8 bars, then a phrase in G for 4 bars, then a final phrase in C for 4 bars.” The process of turning this outline into a fully-written piece is accomplished by starting with large strokes to make sweeping adjustments to the piece, such as splitting the 8-bar phrase into two 4-bar phrases, then doing the same sweeping process again but at a smaller stroke size, such as splitting each 4-bar phrase into 2-bar modules. The size of the strokes gets smaller and smaller, until the program is, for example, fine-tuning on the thirty-second-note level. Each time the program makes a change like this, it is called an *embellishment*, which is to say that a note would be *embellished* if it was changed or emphasized in some fashion, such as having an after-beat placed shortly after it or a pick-up note shortly before it.

The next term is a *gesture*. Here we define a *gesture* as the onset of a new musical event as well as the information describing it. The need for this term stems from the fact that each “note” contains much more information than one might expect. When a piece has not been fully formed yet, a single note might represent an entire 64-bar section of a piece, and therefore it should potentially have a say on aspects like key goals, dynamics, overall energy, tendencies, and any other properties that might differentiate one section from another section. Thus instead of calling this a note, we call it a *gesture*, which contains much more than just the note's information, such as a representation of which other *gestures* this one should be similar to. This weight of information becomes bulky when the brush stroke becomes small, as each thirty-second-note contains just as much detailed info as the larger notes, but nonetheless this is an important way to specify any degree of variety to a piece.

The last terms have already made their appearance: classical and galant *sentences* and *periods*. These structures take many forms in the music of this time period in history, but the general idea that
taught in music theory classes is that *periods* have an antecedent phrase followed by a consequent phrase, and a *sentence* starts with some repeating basic ideas that lead into a continuation of smaller fragments before landing on a cadence. In this project, we choose only one simple representation of each of these forms, and they are used to break sections into smaller parts at many levels. For example, an entire piece might be made of two similar halves (a *period*), or an 8-bar phrase might sound like a question followed by an answer (also a *period*).

**Code Layout**

There are four code files included with this project: WaveGen.lhs, AtonalPieces.lhs, GalantKeys.lhs, and GalantPieces.lhs. At the most general level is WaveGen.lhs, whose name stems from the idea that if a piece is constructed with broad strokes first, there is a smooth “thread” or series of “waves” that the listener can perceive through the discrete positioning of gestures. The name itself is just a metaphor, because in reality this file only stores the core foundation of the algorithm: give it a series of decreasing “brush stroke” sizes (durations), a function for embellishing the piece at those levels, and a set of initial gestures. It runs the recursive algorithm and keeps track of fundamental systems, such as the self-similarity between gestures. The two most crucial functions are shown here:

```haskell
> embellishAtLevels :: [Dur] -> [Gesture a] -> EmbellishFunc a -> StdGen -> [Gesture a]
> embellishAtLevels (d:ds) m f s = let (s1,s2) = split s
>                                       in embellishAtLevels ds (embellish d m f s2) f s1
> embellishAtLevels [] m _ _ = m

> embellish :: Dur -> [Gesture a] -> EmbellishFunc a -> StdGen -> [Gesture a]
> embellish d (g1:(g2:m)) f s = let (s1,s2) = split s
>                                  in concat [f d g1 g2 s1, embellish d (g2:m) f s2]
> embellish d m _ _ = m
```

The demands of generalized self-similarity can be quite complicated. If you want two musical passages to be similar, in what way should they be similar? Pitch-wise contour is one thing, keys are another. If a chain of notes, such as a scale, is told to treat each note similarly, what do you do if you want this chain to be similar to another chain a few bars later? A self-similarity system should be able
to accomplish a wide variety of demands depending on the task at hand – perhaps one section will have self-similarity of pitch contour, while another section will have self-similarity of rhythm. The writer of the code that makes these logical connections should be able to choose which aspects are similar, which ones depend on the surrounding environment, and which aspects are not similar at all. In this project, this is accomplished by attaching a random number seed to each gesture. If you want two gestures to be similar, you give them the same seed, and pass new seeds onto their embellishments, and suddenly all of their future embellishments will be carried out in a “similar” fashion. Any elements that the composer does not wish to have similar between the two can be stemmed from the random seed passed down through WaveGen rather than by the individual gestures.

The remaining files each depend on WaveGen: AtonalPieces has a set of simple pieces whose pitches do not fall within the diatonic system, yet are still constructed in self-similar ways such as a classical period or a set of short phrases that gradually transform from one end of the piece to the other. GalantKeys stores a ring-like representation for musical keys and chords, and GalantPieces stores the logic for a two-step embellishment process: first by dividing larger structures into smaller ones, then by embellishing individual voices independently within each section.

**Key Representation**

Although it is not one of the fundamental pillars of this project's goals, key representation is a critical feature when attempting to construct many types of music. Because of the classical and galant tendencies of this project, I chose a ring of intervals to represent “chord space”, shown in Figure 3, because it lends itself well to classical harmonic progressions. The numbers in the diagram represent intervals in units of half-steps (so a 3 indicates a minor third and a 4 indicates a major third). The ball right now is rooted at the T for Tonic position, with its tail extending through a 4 then a 3, which makes a major triad when stacked above the tonic note. The idea is that this ball travels clockwise around the
ring with its tail representing the chord's stacked thirds. This ordering falls roughly in line with common classical progressions: starting with tonic, we have \( I \rightarrow vi \rightarrow IV \rightarrow ii \rightarrow vii^\circ \rightarrow V \) (or \( V^7 \) if the tail is extended slightly), then jumping back to \( I \). We commonly only make important stops along this chain as needed, such as by going \( I \rightarrow IV \rightarrow V \) (which in Figure 3 would be \( T \) for Tonic \( \rightarrow \) S for Subdominant \( \rightarrow \) D for Dominant), or most often just \( I \rightarrow V \). In this project's Haskell code (the GalantKeys module), a chord is represented as the ball and tail within a certain ring, and the ring's structure exists as a result of the functions that operate upon the ball/chord, such as “thirdUp” and “thirdDown” which move to a neighboring position on the ring, or “nextChords” and “prevChords” which reveal the nearby acceptable positions up and down the ring. The chord representation is shown below:

```haskell
> type TonicAbsPitch = AbsPitch -- Should be in the range 0 to 11
> type ScaleDegree = Int -- Should be in the range 1 to 7
> type ChordSize = Int -- Should be either 3 or 4
> type Chord = (TonicAbsPitch, ScaleDegree, ChordSize, Mode)
```

There are additional implications made possible by the ring, such as the removal of the thin 3-line at the bottom which then produces a circle of fifths (because \( 3 + 4 = 7 \), which is a perfect fifth), an arbitrary repetition of the thin 3-line at the bottom which would then produce a fully diminished tetrachord, and the idea that the ball can modulate at any time because its tail is enharmonically equivalent to similar positions on this ring or rings of different keys. However, these more advanced concepts were not needed for the “Sturm und Drang” piece, so they are left as potential future possibilities in exploring how well this ring diagram might actually line up with common practice.
Galant Pieces

The GalantPieces module contains the necessary logic for generating the “Sturm und Drang” piano pieces. While there is a lot of code packed into this module, it can be divided into two main categories. The first sweep embellishes gestures of “sections” – at this initial stage, the piece is represented only by its larger-scale musical blocks (sections), and each block contains information such as its primary chord or “key area” or what the target chord is (such as a block that might start on I but wish to land on V), as well as the rough positioning of the four voices. Thus, embellishment at this stage is a matter of breaking each section into smaller and smaller self-similar chunks by imposing period and sentence structures. The crux of this first sweep is the following function:

```haskell
> -- Break a section into smaller chunks of size d. This can be a period, sentence, module string, or ignored.
> breakSection :: Dur -> Gesture Section -> Gesture Section -> StdGen -> [Gesture Section]
> breakSection d g1 g2 s = if (d >= gDur g1)
> then [g1]
> else let r1 :: Float
> r2 :: Float
> (r1,s2) = random (gSeed g1 s)
> (r2,s3) = random s2
> in if (r1 < 0.5 && d == (gDur g1)/2)
> then makePeriod d g1 g2 s3
> else if (r2 < 0.8 && d == (gDur g1)/8)
> then makeSentence d g1 g2 s3
> else if (2 <= d && d <= (gDur g1)/2) then makeModules d g1 g2 s3 else [g1]
```

The above function is called by WaveGen and takes a current embellishment “brush size” as a duration, along with two gestures. Like all embellishment functions (EmbvellishFuncs, so to speak) it applies its actions to the first gesture, using the second one only as a reference for what immediately follows. The decision-making is based on probability, and the seed for this comes from the gesture if it has one, otherwise coming from the seed passed in by WaveGen (the (gSeed g1 s) expression takes care of this distinction). Then it decides whether to produce a period, a sentence, a chain of unrelated modules (for variety), or to just do nothing with this gesture at the current level.

The second pass is slightly different. The original series of “Section” gestures is converted into four series of “VoiceNote” gestures, one listing for each voice. The voice notes store slightly different information from the sections (most notably is that they know nothing about the current chord), and
these gestures are organized horizontally as individual voice lines, rather than as vertical stacks like they were within each section. This allows another embellishment sweep to take place, one for each voice. The function that handles this is called \texttt{renderVoice} and looks very similar to \texttt{breakSection} shown earlier, except that instead of deciding whether or not to break into predefined structures like sentences, it decides whether or not to add pick-up notes and after-beat notes (the mirror of a pick-up) surrounding each \texttt{VoiceNote}. These extra notes are added as gestures for the next (smaller) level of embellishment, and the volume of each note decays with each further step away from the structural pillars. For “Sturm und Drang”, only voices marked as “Melody” will get these extra embellishments.

Lastly, the fully-embellished chains of \texttt{VoiceNote} gestures are then converted to \texttt{Music1} data types and attached in parallel. Part of this conversion involves post-processing techniques – only one was explored for “Sturm und Drang”, which is repeating harmony voices for an added layer of dramatic expression. Another feature might be the Alberti bass figure, which alternates between three harmonic voices (bottom, top, middle, top, repeat).

### The Progression of the Sample MIDIs

Included with the project are 13 MIDIs generated over the course of the project. The following list is a brief overview of what they represent:

- \textit{examplePiece94.mid} (“Learning to Play Piano with One Finger”) and \textit{examplePiece76872.mid} (“Walking Up the White Keys”) are the earliest demonstrations of “directional” embellishment. Only four starting notes were given, and the rest is filled in by the Haskell code, with the embellishments connecting and circling around in a logical direction.

- \textit{examplePiece7.mid} (“Somewhat Self-Similar”) and \textit{examplePiece12.mid} (“Classical Question and Answer”) are the earliest demonstrations of self-similarity at play.
• *examplePiece12_atonal.mid* (“The Same Question and Answer?”) is in fact the same question and answer as *examplePiece12.mid*, but with wildly exaggerated intervals. I wanted to expose how little the code knew about harmonic functions at the time and isolate the interesting rhythmic elements within the structure. *examplePiece488.mid* (“Exotic Dance”) is the same piece with a different random seed.

• *examplePiece1287935.mid* (“Banging out the Chords”) is yet another version of the same question and answer, but this time the basics of chord progressions and multiple voices are beginning to emerge.

• Lastly, some “Sturm und Drang” selections demonstrate the main musical example of this project. I once generated a version that was 1024 measures long, but I stopped listening after 16 minutes when I realized I was less than halfway through.

**Results, Further Work, and Conclusion**

The end result of this project is a style that sounds inspired by the “Sturm und Drang” movement from the years surrounding 1770, with a mix of other styles thrown in. Nearly every subsystem within the Haskell code could be adjusted to have a major impact on the resulting style and form. For example, right now there is a lack of punctuated cadences, and an easy way to fix this would be to include the common galant “triple hammer blow” structure as something to tack onto endings of large sections. Mentioned earlier was that an Alberti bass line would be feasible and would drastically alter the vibe of a pressing drive given off by the repeated harmonies.

Most noticeably there are many elements missing that are normally found in computer-generated music, such as four-part counterpoint and voice leading rules (of which there are none followed here), or a proper treatment of the minor scale degrees 6 and 7 which can fluctuate depending on their usage (this program just uses a harmonic minor scale, which leads to a somewhat unorthodox
sound at times). However, these concerns are outside the scope of the project, as the interesting aspects are the treatment of self-similarity, as well as the overall recursive approach that carries a continuous thread through discrete time points. Matters of style and musical sense were only briefly touched upon within this project, and they could fluctuate to different genres entirely with a few tweaks to the code. This project's approach to generating music is full of opportunity and possibilities – while there are certainly many probabilistic decisions taking place in the composition of the final piece, there are real musical patterns transcribed into code that take priority, based on human observations of common musical patterns and reducible hierarchies. Every note can be explained in some way, but like all programs of this type, the code is simply a rule-following compositional tool meant to aid or inspire rather than produce finished products on its own. For those who acquire a deep understanding of such compositional code and are able to control the results to their desires, though, it can be a useful tool indeed.
References


Mozart, Wolfgang A. *Symphony No.31 in D major, K.297/300a*. Score on IMSLP:

<http://imslp.org/wiki/Symphony_No.31_in_D_major,_K.297/300a_(Mozart,_Wolfgang_Amadeus)>