

Algorithmic Composition through Probabilistic Recurrence Models

Roger B. Dannenberg

Carnegie Mellon University

USA

rbd@cs.cmu.edu

Abstract

Innumerable approaches to algorithmic composition have been described, often based on attempts to formalize music or formalize AI techniques. A few simple ideas distilled from experience are presented in the form of a schema for organizing algorithmic composition programs. The schema uses probabilistic rules or tendencies, which can be written and combined modularly, and plans that can be created on-the-fly to guide future choices. Plans are just a form of state or context information that can be generated or renewed along with music creation, making this a recurrence model for sequence generation. Multiple passes can be used to introduce top-down hierarchical composition strategies.

Keywords

Algorithmic Composition, Music, Probabilistic, Rules, Tendencies, Planning, Context, State, Hierarchy

Introduction

Algorithmic composition has been explored since the earliest days of computing. [1] There is probably at least one piece of music to exemplify every programming language and programming framework, not to mention every approach to creating simulations, computational models, and artificially intelligent systems. In fact, composers often invent their own formalisms and techniques, inspired by their musical interests and intuition.

Since my participation in Callejón del Ruido Festival resulted in performances of pieces with very different approaches to algorithmic composition and interaction, it seems fitting to offer some current ideas about music creation with computers for this report. This contribution is inspired by recent work on generating songs in a popular music style, but here, I will present a refined version of the approach that I believe could be suitable for a wide variety of musical explorations.

In algorithmic composition, we usually want some randomness in the computation. Creating just one great composition is wonderful, but in practice, randomness has at least three nice properties. First, it is hard to specify output so completely that only one output is possible. Finding the “optimal” music according to some objective function might produce a single result, but music is complex, so optimization is rarely feasible, and often the “best” music according to some simple ideas is less interesting than music with elements of randomness. Second, randomness lets us generate many outputs, allowing us to focus more on a particular style or “logic” as opposed to a specific piece. Finally, in live performance, introducing variation and the unexpected is esthetically interesting. It is a unique capability that computers can bring to composition and performance.

Another important practical matter is the ability to refine ideas and models. Typically, some initial ideas are turned into code, executed, and the output is evaluated. If the result is great, you are done, but more often, shortcomings are immediately apparent. We need a flexible approach where results are easy to refine. Often, we find undesirable output that could be avoided by the addition of constraints, rules or tendencies. An approach that supports incremental refinement is generally better than a monolithic algorithm that is hard to change.

Search is an important aspect of many AI systems, but with music, search is often not very productive. One reason is that many search problems are exponential. A sequence of 20 pitches selected from a scale of 12 pitches has 12^{20} possibilities. Evaluating 1,000,000 sequences per second, we could not explore all possibilities in a million centuries! A second problem is that search assumes that you know a good solution when you see one. Without a good evaluation function, even an exhaustive search may find a poor solution. Therefore, it is usually more practical to create music incrementally and minimize search, iteration or backtracking.

The next section presents a simple probabilistic approach to algorithmic composition that has been used effectively. Music often has a hierarchical structure, so we extend our approach to support multi-pass, top-down music construction. Another source of structure in music is the existence of plans that influence a sequence of future music events. (An example is a decision to start a crescendo and rising pitch contour whenever a sufficiently low pitch is reached.) Once again, we extend the approach to facilitate plan-based music generation. Finally, we will consider some variations and optimizations that may be useful in practice.

Rules and Probabilities

Consider a probabilistic melody generation task. For simplicity, we will assume a diatonic scale of 15 pitches (two octaves), no rests, durations quantized to 16th notes, and a maximum duration of the whole note. Each note is thus selected from a space of $15 \times 16 = 240$ possibilities. We will compute a weight for each choice and generate notes sequentially by making choices according to weights.

Initially, consider setting all weights to 1. The output will be a completely random sequence in terms of both pitch and duration. Not very interesting. Perhaps we want our melodies to avoid extremes of range. We could multiply each weight by a Gaussian (bell) curve centered around 7 (assuming pitches are numbered 0 through 14) with a variance of 5, i.e. the weight for pitch p becomes $N(7-p, 5)$. This rule says nothing about duration.

While the pitch “rule” is continuous and probabilistic, we can also incorporate logic rules into this framework. Suppose we decide that odd durations longer than 3 (sixteenths) should not be used. We can write a function returning 0 or 1: $f(d) = (1 \text{ if } (d < 5 \text{ or } \text{iseven}(d)) \text{ else } 0)$ to express this. Again, we can multiply each weight by this function to eliminate some of the durations. Multiplying a weight by zero eliminates the possibility of making that choice entirely.

We could go on, for example, by adding a third “rule” that prefers shorter durations to longer ones. Putting this together, we can state the framework more formally as follows: Our music composition system creates sequences of tokens (we call them “tokens” because they could be pitch/duration pairs, chords, sound types, articulations, etc.). The system consists of a set of tokens $A = \{a_i\}$ and a set of rules $R = \{r_j\}$. Each rule is a function from a token to a weight: $r_j: A \rightarrow \mathbb{R}$, where weights are real numbers (preferably expressed as probabilities in the range 0 to 1). To select the next element of the output sequence, we first obtain an overall weight for each token (a_i) by multiplying the weights given by each rule:

$$w_i = \prod_k r_k(a_i)$$

Then select and output a_i with probability

$$P(i) = w_i / \sum_k w_k$$

```
function compute_next_note():
  for i in [0:240] w[i] = 1 –construct initial weights
  for p in [0:15]:
    for d in [1:17]:
      i = p + (d-1) × 16 –combine to form index
      w[i] = w[i] × favor_middle_register(p)
      w[i] = w[i] × restrict_odd_durations(d)
      w[i] = w[i] × favor_shorter_durations(d)
  output(weighted_selection(w))
```

Figure 1. Psuedo-code for probabilistic composition.

Computationally, this looks like a loop (or nested loop) to enumerate all tokens and the application of all rules. For the pitch/duration tokens and three rules, the code looks like Figure 1.

Note that we can easily extend the computation with new rules or temporarily remove individual rules by turning lines of code into comments. It may seem a bit cumbersome to always consider all tokens and all rules. Below, we will consider that pitch and duration could be computed separately, saving a great deal of computation. However, a more realistic melody generator would have rules concerning both pitch and duration, e.g., “smaller intervals are favored when durations are shorter.”

Hierarchical Composition

Sometimes, it is useful to create a high-level structure before filling in details. Before constructing a melody, we might wish to compose a chord progression. Melody could then be guided by the harmony in place at each beat. A top-down music composition system like this can be created by separating the composition process into stages. Starting at the top level, each stage creates the next level of the hierarchy. In my popular song writer, based on work by Elowsson and Friberg [2], the first stage designs a phrase structure, the second stage designs a rhythmic accent structure, the third stage composes a chord progression, and the last stage writes a melody as shown in the previous section (however, many more rules are used). We have experimented with yet another level of hierarchy, generating a “basic melody” [3] consisting of half notes before elaborating this to form the final melody. [4]

In a multi-pass hierarchical scheme, the output of each pass must be saved in a data structure that can be easily accessed by the next passes. I have no general solution for music representation. We limit our pop songs to units of 16th notes and use arrays containing data for every 16th note (i.e., every possible time point) of the song, but even conventional music that admits triplets and finer subdivisions creates representational challenges. In general, the simplest representation that can express your music is the best choice: Other schemes can be more expressive, but encoding and accessing music using a very general representation system can be very tedious.

In most cases, rules need to consult previously generated tokens. For example, a rule that favors small melodic in-

tervals over large ones must know the previous pitch to compute the interval from that pitch to each possible next pitch. Thus, rules are not simply functions of one token. It would be more complete to describe rules as dependent upon both the token a_i and previously output tokens (a sequence of type A^n):

$$r_j: A \times A^n \rightarrow \mathbb{R}$$

If we denote higher levels of the hierarchy by types B, C , etc., then the full form of the rule is:

$$r_j: A \times A^n \times B \times C \times \dots \rightarrow \mathbb{R}$$

In practice, these additional parameters for r_j might be implemented as global variables that all rules can access.

In addition to referencing data at higher levels of a hierarchical construction, rules can reference real-time sensor data or human input in live performance situations, creating a path for interactive algorithmic composition and improvisation.

Planning

Sometimes, we want to say something about the future rather than create output one-token-at-a-time. Essentially, we decide to adopt and carry out a plan over the next several tokens that will influence or even override the probabilistic algorithm described so far. An example from pop music is choosing a cadential chord progression that will end a phrase on the tonic (I) chord. Once decided, we do not want to deviate from the plan.

To express plans, we add yet additional information that rules can access. What I call *plans* could also be viewed simply as *state*. The key idea is that, in addition to computing the next token in a music sequence, we also compute a new *plan* or *state*, which becomes *context* for the next computation. In very abstract terms, we can write:

$$(m_{k+1}, state_{k+1}) = f(m_{1\dots k}, b, c, \dots, state_k),$$

where m is our output sequence of music tokens, f is our probabilistic rule system (combining all of the r_j), b, c, \dots are sequences computed earlier at higher levels of the music hierarchy, and each rule r_j has *state* as an additional parameter. This is a recurrence relation because each $state_{k+1}$ depends on the previous $state_k$.

In practice, states are mainly used to represent plans and conditions to be considered in the rules, and we implement *state* as a simple data structure that can be modified at each iteration of the music sequence computation. The computa-

```
function compute_next_note():
  for i in [0:240] w[i] = 1 –construct initial weights
  for p in [0:15]
    for d in [1:17]
      i = p + d × 16 –“flatten” p and d to index
      w[i] = w[i] × rule_1(p)
      w[i] = w[i] × rule_2(p)
      ... –apply all rules to w
  output(weighted_selection(w))
  update_plan()
```

Figure 2. Incorporating plans as recurrent state.

tion now looks like Figure 2.

The main change is the last line in **bold**. Now, each time we compute a new output token, we can make or modify any existing plans, as represented in some program variables. We must also modify or extend rules to consider plans. Three approaches are: (1) in each rule, add tests as necessary to see how the rule should be applied in the context of any plans. This sacrifices some modularity, since adding new types of plans may require changes to every rule; (2) extend the set of rules to include plan-specific rules. This approach is good when the plan is a general tendency that simply biases the existing probabilities; (3) override computed probabilities when a plan is in place. In the extreme case, after the general rules are applied, a new rule could reset w_i and apply a completely new set of rules, but only when a certain plan is in place.

Plans are likely to have a finite duration. When a plan is added to the state, the plan should include a duration or timeout, and the *update_plan()* function should check for and remove expired plans.

Search and Optimization

Earlier, I presented some drawbacks of search-based algorithms. However, there is an interesting place for search in the approach described here. Mainly, we would like to avoid some problems that inevitably occur with this probabilistic approach. First, even if we select tokens according to weights or probabilities, it is very likely that at least one token in a long sequence will have very low probability. Secondly, it is possible for earlier token choices to leave no good options later. These problems can be reduced by composing multiple pieces and selecting the one with the highest overall probability.¹

For example, in my work on popular song creation, I found that occasionally, songs had several strange intervals or rhythms that simply sounded like mistakes. I modified the program to generate 10 songs and pick the most probable of 10, and this “filtered out” the weak ones.

Whether it is better to use probabilities, which reflect the amount of surprise introduced at each step, or original weights, which can be considered a form of absolute quality assessment, is an interesting question which I have not pursued. The best approach is likely to depend on the rules. Also, when output lengths vary, one might want to use the average probability or weight per token.²

¹ The simplest approach to estimating overall probability of a sequence is to simply assume that all tokens are independent and multiply their probabilities. Since the product of hundreds of small probabilities can be exceedingly small, one typically forms the sum of logarithms to avoid numerical problems, taking advantage of the rule $\log(a \times b) = \log(a) + \log(b)$. If we want to pick the result with the highest probability, it is equivalent to picking the result with the highest sum of logs of token probabilities.

² To do this properly, divide the sum of the logs of token probabilities by the length of the token sequence.

We should be careful though, because rules are unlikely to form a sophisticated music evaluation system. Extensive search, such as finding the most likely of 1 million songs will not necessarily produce the best outcome. Consider the similar approach of outputting the most likely token *at every step* rather than making a weighted choice. In most cases, this will create a “high probability” output, but it is likely to get stuck repeating high probability tokens, and the result will lack variety and interest.

One more application of search has been found to be useful. Recall that rules are allowed to forbid certain choices by returning zero weights. Situations can arise where rules manage to zero the weights of *all* tokens. This means that, according to the rules, no acceptable choice can be made. In my system, I detect this and simply start over. A more sophisticated AI approach might be to back up one or more tokens or even try to find the cause of the impasse and correct it. However, if these problems are common, one can study their cause and design a solution (it might involve spotting problematic situations and forming plans to guide the system past them). Alternatively, if problems are rare, then restarting is very likely to succeed. Either way, sophisticated search seems to be unnecessary.

The Size of A

One potential problem with this approach is that the number of different tokens can be large. For example, in Xenakis’ *Stochastic Music Programme* (SMP) [5], sounds are described by onset time, duration, pitch, instrument, glissando rate and “intensity form” (44 forms of dynamic variation such as crescendo, diminuendo, louder then softer, etc.). Considering all combinations, there could be many millions of sound types.

When the token space becomes too large, we have two basic methods to simplify the computation. First, we can separate the dimensions of the tokens if they are independent. Imagine in our previous example if *pitch* and *duration* were independent. Then, our nested loop to consider all 240 combinations:

```
for p in [0:15] –iterate over all pitches
  for d in [1:17] –iterate over all durations
    ...
```

could be separated into two separate loops with a total of only $15 + 16 = 31$ iterations:

```
for p in [0:15] –iterate over all pitches
  pw[p] = pw[p] × pitch_rule_1(p)
  ... – more rules for pitch weightings
for d in [1:17] –iterate over all durations
  dw[p] = dw[p] × duration_rule_1(p)
  ... – more rules for duration weightings
```

Then the output token would be determined by making two independent weighted choices using pitch weights *pw* and duration weights *dw*.

The second possibility is one-way dependencies.. Suppose that pitch weights do not consider the proposed duration, but duration weights vary with the proposed pitch. If so, we can use only the pitch-weighting rules to compute *pw* (as shown above) and make a weighted choice of *pitch*. Now, *pitch* is known, so we can use it within duration rules to compute duration weights.

It should also be mentioned that if an attribute such as duration can be computed independently of other attributes (but possibly depending on them), then it can be *continuous* rather than discrete. For example, one can compute duration in seconds rather than a choice of *n* sixteenths. Continuous values can be efficiently computed from a single random distribution such as the Gaussian, but it is not so simple to combine multiple rules that relate to different factors or influences. The choice is yours.

In Xenakis’ SMP, parameters appear to be highly independent. E.g., the instrument choice is selected using weights that depend only on the density, which is fixed for the duration of each section. However, pitch and duration are both based on the instrument. We can compute them independently *after* the instrument has been selected.

It is the composer’s choice as to how musical tokens are represented and whether their attributes are independent. Strong dependencies arise from concepts such as “short note durations require smaller pitch intervals,” but many other attributes are weakly connected, allowing us to compute them separately and more efficiently.

Evaluation

I have presented a general approach to algorithmic composition that I believe has many good properties:

1. It is simple to implement, which means one can spend more time adjusting rules and thinking about musical concepts.
2. It is modular, allowing concepts, tendencies and “hints” to be added in the form of independent rules.
3. It is probabilistic and non-deterministic. Multiple runs can be used to obtain a large pallet of compositional materials, and live performance systems can offer variety and surprise.
4. One can express tendencies as well as absolute constraints.
5. Rules can be based on simple statistics of existing music, so in that sense, rules can be “learned” from data.
6. The approach is efficient because it avoids large amounts of search. To make up for the lack of search and backtracking, *plans* can be generated to offer a degree of look-ahead and guidance.

There are also some shortcomings:

1. Search and optimization are weakly supported, but we argue this is usually sufficient.
2. The approach works best for computing discrete tokens, although any computation that computes a continuous attribute is easily incorporated.

3. This is only a conceptual framework: There is no language, library, or ready-made software. (Perhaps this is also a “feature.”)

Conclusions

Innumerable approaches to algorithmic music have been explored. Often, software is developed for just one composition. Still, many composers develop a way of working. My work for the Callejón del Ruido Festival was formative in my thinking about composition and particularly algorithmic composition and interaction. There is nothing like a public forum and high expectations to put one to the test, and I am deeply grateful for the experience and learning that resulted from this great opportunity.

The framework described here distills and generalizes many ideas and approaches that I have found useful over the years. If readers find it overly simplistic, consider that that may be its greatest advantage! If we find the design of software to be trivial and obvious, then we stand a much better chance of a correct and efficient implementation, not to mention modifying and improving the first version. It is much better to focus on music than complex AI techniques, provided we can still express our musical intentions.

Ultimately, composers (and perhaps listeners) must be the judge of any approach or methodology. Programming languages, systems and methodologies are notoriously difficult to measure, and it does not make sense to say one approach is better than another. However, I hope that thinking about the *structure* of an algorithmic composition system as opposed to the actual *algorithms* will inspire others to do the same and perhaps take away some useful ideas for their own creative work.

Acknowledgements

Thanks to Roberto Morales for inviting this paper and for so much work on the Festival over the years. My work would not have been possible without the support of Carnegie Mellon University.

References

- [1] Mary Simoni and Roger B. Dannenberg, *Algorithmic Composition: A Guide to Composing Music with Nyquist* (Ann Arbor: The University of Michigan Press, 2013).
- [2] Anders Elowsson and Anders Friberg, “Algorithmic Composition of Popular Music,” *Proceedings of the 12th International Conference on Music Cognition and Perception and the 8th Triennial Conference of the European Society for the Cognitive Sciences of Music*, 2012, 276-285.
- [3] Lejaren Hiller, Charles Ames and Robert Franki, “Automated Composition: An Installation at the 1985 International Exposition in Tsukuba, Japan,” *Perspectives of New Music*, Vol. 23, No. 2, Spring – Summer, 1985, 196-215.
- [4] Shuqi Dai, Zeyu Jin, Celso Gomes, and Roger B. Dannenberg, “Controllable Deep Melody Generation via Hierarchical Music

Structure Representation,” *Proceedings of the 22nd International Society for Music Information Retrieval Conference*, 2021, 143-150.

[5] Iannis Xenakis, *Formalized Music: Thought and Mathematics in Music* (Hillsdale, NY) Pendragon Press, 1992.