Improving the Outcome of a Probabilistic Logic Music System Generator by Using Perlin Noise

Colin J. Nicholson[1], Danny De Schreye[1], and Jon Sneyers[1]

1 **Katholieke Universiteit Leuven**
Celestijnenlaan 200A
B-3001 Heverlee
Belgium
`<FirstName.LastName>@cs.kuleuven.be`

Abstract

APOPCALEAPS is a logic-based music generation program that uses high level probabilistic rules. The music produced by APOPCALEAPS is controlled by parameters that can be customized by a user to create personalized songs. Perlin noise is a type of gradient noise algorithm which generates smooth and controllable variations of random numbers. This paper introduces the idea of using a Perlin noise algorithm on songs produced by APOPCALEAPS to alter their melody. The noise system modifies the song's melody with noise values that fluctuate as measures change in a song. Songs with more notes and more elaborate differences between the notes are modified by the system more than simpler songs. The output of the system is a different but similar song. This research can be used for generation of music with structure where one would need to generate variants on a theme.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Probabilistic logic, Music generation, Perlin noise

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.231

1 Introduction

Automatic music generation is a broad area in the field of artificial intelligence, even the subfield of declarative logic programming contains work on the subject [1] [2]. Probabilistic methodologies for generating music are less explored. Research in the field is often specific to certain domains, such as certain types of piano music [5]. The focus of this paper is on music generation in the area of probabilistic logic programming; that is, logic programming with probabilistic features added on. APOPCALEAPS (Automatic Pop Composer And Learner of Parameters) [10] is a music generation program written in the CHRiSM language [11] which builds on a music classification/generation program [12] written in PRISM [9]. CHRiSM is an extension of Constraint Handling Rules [3] that contain multi-headed rules which are triggered only if certain probabilistic conditions are met. Rules can lead to different outcomes for the same input based on set probabilities.

In this paper, the APOPCALEAPS system will be extended by adding a noise component that takes the music files generated by the system and alters the melody to produce new music files that maintain some features of the original piece while showing diversity. The noise feature is based on a variation of Perlin Noise [7] (commonly used in the field of computer graphics to generate natural-looking patterns) which was introduced in [14]. Perlin noise has benefits over random noise when altering a pattern, such as the melody of a song. Perlin noise is controllable: we can decide what sort of a design the noise should take.

This paper will first introduce the CHRiSM language (Section 2), then go into detail about how APOPCALEAPS generates music (Section 3). Noise generation will be explained in Section 4. In Section 5 we will describe how noise can interact with the current APOP-CALEAPS system to produce interesting output. We end with conclusions and future work ideas.

2 CHRiSM

CHRiSM [11] is a programming language based on a combination of concepts of two logic programming languages: CHR [3] and PRISM [11]. The rules of CHRiSM act on constraints, which are similar to Prolog atoms. During execution, all constraints are stored as a multiset. This multiset is referred as the constraint store. An initial store (query) is given to the program, which applies all possible rules before ending with a final store (the result for the query). CHRiSM combines the benefits of CHR with those of PRISM, while maintaining a more user-friendly syntax than a CHR(PRISM) system.

2.1 Constraint Handling Rules

Constraint Handling Rules (CHR) is a programming language introduced by Frühwirth [3]. The name comes from the original intention of adding constraint solvers to a host language. CHR extends a host language. For example, CHR(Prolog) allows one to use Prolog as a host language but with the added benefit of CHR's multiheaded rules. A CHR program is a sequence of CHR rules which can be of the following types:

- Simplification: $h_1, ..., h_n \;<=>\; g_1, ..., g_m \mid b_1, ..., b_k$
- Propagation: $h_1, ..., h_n \;==>\; g_1, ..., g_m \mid b_1, ..., b_k$
- Simpagation: $h_1, ..., h_l \setminus h_1, ..., h_n \;<=>\; g_1, ..., g_m \mid b_1, ..., b_k$

When the head of a simplification rule matches with corresponding constraints in a constraint store and the (optional) guard conditions are met, then the rule adds constraints corresponding to the body to the constraint store. A propagation rule is the same only the constraints matching with its head are kept in the store. To illustrate the difference between simplification and propagation rules, consider the following example:

Listing 1 CHR example code

```
rain ==> wet.

rain ==> umbrella.
```

The query "rain" will give the result "rain, wet, umbrella". If the two propagation rules were replaced with simplification rules, however, the same query would give the result "wet" or "umbrella," non-deterministically. A simpagation rule removes the constraints matching with the part of the head after the backslash, but keeps those corresponding to the part before it. Consider the following example:

Listing 2 CHR example code

```
male(X) \ female(Y) <=> pair(X,Y).

pair(X,Y) :- write(X), write(' dances with '), write(Y), nl.
```

Here, the simpagation rule at the top leads to a PROLOG rule that outputs "X dances with Y" for variables X and Y. If a male and several females are provided as queries to the program, the output will be the male dancing with each female.

2.2 PRISM

PRISM is an extension of Prolog developed by Sato which includes probabilistic rules [9]. PRISM extends Prolog by adding the msw/2 (multiarity random switch) predicate. The probabilistic switch msw(id,v) enables one to choose a value *v* from a set labeled with *id*. For example, *id* could be blood type and *v* could be a, b, or o depending on which value is probabilistically chosen. Below is a sample PRISM program.

Listing 3 PRISM Coin Flip program

```
values(coin,[head,tail]).

direction(D):-
msw(coin,Face),
( Face == head -> D=left ; D=right).
```

In this program, the switch labeled "coin" has two possible values (head or tail) each with a 50% probability of occurrence. The query "direction" will return a value of left if head is chosen and right if tail is chosen. The parameters of the values can be learned from examples or set manually. Learning from examples entails providing a list of observed atoms to an expectation maximization algorithm to find the parameters with the greatest likelihood.

2.3 CHRiSM syntax

The following example from [10] shows a game of "rock, paper, scissors" represented by CHRiSM code:

Listing 4 CHRiSM Rock, Paper, Scissors program

```
player(P) <=> choice(P) ?? rock(P) ; paper(P);  scissors(P).

rock(P1), scissors(P2) ==> winner(P1).
scissors(P1), paper(P2) ==> winner(P1).
paper(P1), rock(P2) ==> winner(P1).
```

Here each player leads to a choice. The "??" introduces three choices (rock, paper, and scissors) of equal probability. Apart from just simulating games for groups of players and returning lists of winners and losers, the constraints can be used to learn playing styles of individual players. [10] has a more detailed analysis of this program and the statistical experiments CHRiSM can perform on it.

2.4 CHRiSM to CHR(PRISM)

A CHRiSM program maintains the ability to use the multiheaded rules of CHR while including the ability to use the probabilities, statistical sampling, and expectation maximization learning of PRISM. A CHRiSM program is translated to a CHR(PRISM) program. As shown in [10], a simplification rule such as

Listing 5 CHRiSM rule

```
player(P) <=> choice(P) ?? rock(P) ; scissors(P); paper(P)
```

is translated to

Listing 6 CHR(PRISM) translation

```
values(choice(_), [1,2,3]).

player(P) <=> msw(choice(P),X),
(X = 1 -> rock(P); X = 2 -> scissors(P);  X = 3 -> paper(P)).
```

in CHR(PRISM).

3 APOPCALEAPS

APOPCALEAPS is a music generation program written in CHRiSM. APOPCALEAPS generates a song in the form of a text file, and uses the program LilyPond [4] to make a MIDI music file and a PDF file of the sheet music. The program is unique in that it is the first music generation system that is both probabilistic and constraint-based. APOPCALEAPS can use the voices: melody, bass, chords, and drums, or any combination of the four. The user can specify properties of the song to be produced, such as number of measures or the shortest possible duration for the notes of a particular voice.

In its current state, APOPCALEAPS can generate all notes, where X flat is written as (X - 1) sharp. Only the chords C, F, G, A minor, E minor, and D minor (and their corresponding four-note seventh chords) are possible (this is an arbitrary limitation but covers the most common chords in pop music). APOPCALEAPS outputs one chord (or three and four-note chords of the same root) per measure. For example: one measure may have C and C7 as its chords (pauses, called rests, are always possible as well).

Listing 7 Probabilitistic choices in APOPCALEAPS

```
values(chord_choice(C), [c,g,f,am,em,dm]).
values(note_choice(V,C,B), [c,d,e,f,g,a,b,r]).
values(octave_choice(mid), [-2,-1,0,+1,+2]).
values(octave_choice(low), [0,+1,+2]).
values(octave_choice(high), [-2,-1,0]).
values(drum_choice(B), [bd,sn,hh,cymc,r]).
values(chord_type(_,B), [0,7,r]).
values(split_beat(V), [no,yes]).
values(join_notes(V,_,_), [no,yes]).
```

3.1 Chord Generation

APOPCALEAPS uses the chord C major for the first and last measure of a piece if the piece is in major key, and the chord A minor for the first and last measure if the piece is in minor key. The remaining chords are generated by looking at the current chord and choosing a chord to follow it based on preset probabilities for chord transitions (the probabilities were empirically chosen).

Listing 8 CHRiSM rules for Chord Generation

```
key(major), measure(1) ==> mchord(1,c).
key(major), measures(N) ==> mchord(N,c).
key(minor), measure(1) ==> mchord(1,am).
key(minor), measures(N) ==> mchord(N,am).

measures(N) ==> make measures(N).
make_measures(N) <=> N>0 | measure(N), N1 is N-1, next measure(N1,N),
                        make measures(N1).

mchord(X,C), next_measure(X,Y), measures(M) ==> Y < M |
                        msw(chord choice(C),NextC), mchord(Y,NextC).
```

3.2 Rhythm Generation

To generate rhythm, first APOPCALEAPS generates beats in each measure. Each voice has "beat(V,M,N,X,D)" constraints (V = voice, M = measure, N = beat number, X = sub-beat position, D = duration). A chance rule (with probabilities specific to each voice) determines if a beat will be split or not. Beat splitting entails taking a note and turning it into two notes with half the duration of the original note (beat splitting can only occur if the duration of the original beat is longer than the shortest duration the user has specified). Also, beats are sometimes joined together. This occurs with different frequencies depending on whether the notes are in the same measure and belong to the same beat. Like beat splitting, beat joining depends on probabilities unique to each voice.

Listing 9 CHRiSM rules for Splitting and Joining Beats

```
split_beat(V) ??
meter(_,OD), shortest_duration(V,SD) \  beat(V,M,N,X,D),
next_beat(V,M,N,X,Mn,Nn,Z)
<=>  D<SD | D2 is D*2,  Y is X+1/(D2/OD), next_beat(V,M,N,X,M,N,Y),
next_beat(V,M,N,Y,Mn,Nn,Z), beat(V,M,N,X,D2), beat(V,M,N,Y,D2).

join_notes(V,cond Ma=Mb,cond Na=Nb) ??
next_beat(V,Ma,Na,Xa,Mb,Nb,Xb), note(V,Mb,Nb,Xb,SameNote)
\ note(V,Ma,Na,Xa,SameNote) <=> note(V,Ma,Na,Xa,SameNote + '~').
```

4 Perlin Noise

Perlin noise was introduced in 1984 by Ken Perlin [7] as a way of bringing the controlled randomness of nature to computer graphics simulations. For example: one might want to have several patterns in a large cloud of smoke that are similar in shape and movement but not exactly the same. The algorithm behind Perlin noise has been revised slightly over time [8] but remains largely the same. Perlin noise can add variety to many domains besides computer graphics. Perlin noise is bounded, band-limited, non-periodic, stationary and isotropic.

Using a traditional random number generator has a few disadvantages over a Perlin noise algorithm. True randomness (also called white noise) lacks the smoothness of a Perlin noise algorithm, that is more continuous in its output. Also, a Perlin noise algorithm can be controllable. Figure 1 shows a 1-dimensional example of noise. The input must be an integer and the outputs will be random numbers that show no pattern. Figure 2 shows the noise in Figure 1 after it has been smoothed with an interpolation function. Now we can give real input values and the change in them will be much more gradual.

The noise function takes in a coordinate and returns a real number between -1 and 1. The coordinate can be of any dimension. In this paper we will be using 1-dimensional coordinates.

In 2005 Yoon et al. introduced the idea of using Perlin noise to modify melody of an existing song to produce a new song [14] using a Perlin noise variation they had developed earlier [13]. This paper will use the same noise algorithm used in [14]. The steps for the algorithm (with x as a real input value) are as follows:

- 1: Generate M pseudorandom numbers (PRNs) between -1 and 1 and store them in a list G. Make another list (P) which is a random permutation of the set of integers from 0 to M - 1.
- 2: Calculate the integer interval $[q_0, q_1]$, where $q_0 = \lfloor x \rfloor \bmod M$, and $q_1 = (q_0 + 1) \bmod M$.

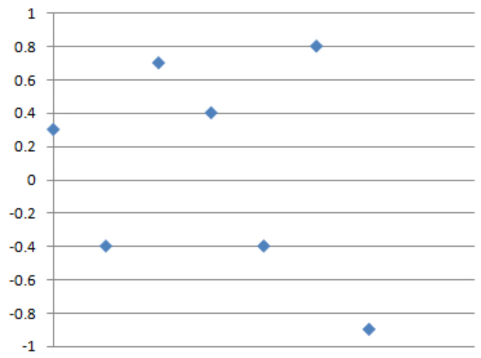

Figure 1 Scattered graph points, representing white noise.

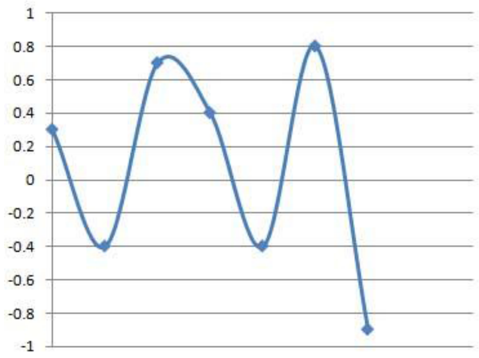

Figure 2 Smoothed graph points, representing Perlin noise.

- 3: Obtain two PRNs $g_j = G[P[q_j]]$, where j = 0, 1.
- 4: The noise value is computed by: $\text{Noise}(x) = (1 - s(d_0))g_0d_0 + s(d_0)g_1d_1$, where $d_j = q_j - x$, j = 0, 1, and $s(t) = 6t^5 - 15t^4 + 10t^3$.

Here, a pseudorandom number is a number that is returned by a function that appears to be random, but will always return the same number for the same input. The M value can vary; M values of 200 are usual. Also, s(t) is an ease curve. The ease curve in the Perlin noise algorithm takes in a number and returns a number that exaggerates the numbers proximity to zero or one; an input number close to zero would return a number much closer to zero while an input number close to one would return a number much closer to one. This has a smoothing effect on the numbers and will produce waves similar to the ones in Figure 2. The integer interval corresponds to the two integer grid points surrounding our real input value. Each integer point has a PRN (obtained in Step 3 from a list of PRNs generated in Step 1). The d value represents the distance between our input value and the two surrounding grid points. Each grid point's PRN value has an influence on the value of the input point in between. The closer grid point will have greater influence. So, if our grid points were 1 and 2 and our input 1.99, the algorithm would output a PRN that is much closer to 2's PRN than 1's PRN. If the input were 1.5, the output value would be in the middle of 1 and 2's PRN values, and so on.

5 Noise Component in APOPCALEAPS

Until this point, APOPCALEAPS generated one sound file. In this paper, we take files generated by APOPCALEAPS and modify the melodies using the Perlin noise algorithm presented earlier to produce new but similar songs. Figure 3 shows an example of how noise values can modify a melody by altering notes. In this example, a note will stay the same if the noise value is between -0.1 and 0.1; a value between 0.1 and 0.2 will increase the note's position on the scale by one place, above 0.2 will increase the position by two places. Similarly, a noise value between -0.1 and -0.2 will decrease the note's position by one place while a value below -0.2 will decrease the note's position by two places.

5.1 Noise on Notes

As can be seen in Figure 3, each measure contains noise values that are similar. This has been proven to work well for songs with a small amount of notes in each measure. It enables consistent changes to notes that are grouped together, while enabling diversity for the entire song. Generating similar noise values for notes within a measure is done simply by providing close input values to the Perlin noise algorithm. So real input values are fed to the noise algorithm which are close, but change drastically, every time the measure changes.

However, we must also consider cases where there are several notes within a certain measure. In fact, it is possible for APOPCALEAPS to produce music that has many notes all in one measure; we would not want to use very similar noise values for all of these notes just because they are in the same measure, the output would not show much diversity. So, if a measure contains too many notes, we split the measure into two measures (each with distinct noise values) and distribute the notes equally among the new measures. Determination of whether or not to split a measure depends on the ratio of the shortest note duration in the measure and the duration of the measure itself (the maximum number of notes we can fit in a measure if it was exclusively filled with notes of the shortest duration). That raio must have an upper bound of six, or we split measures until that ratio is below six. It is not necessary to enforce a lower bound since we do not need to group notes in terms of noise that were not in the same measure in the original song.

If a note is a rest in the original melody, it will remain a rest in the new melody. Apart from just changing notes, we can also change the duration (the length of time a note is held). The main aspects of the piece (meter, tempo, length) will remain the same.

5.2 Duration Modification

The duration of notes must be changed in a precise manner. The sum of the duration for all the notes in each measure of the melody must be the same in a changed song as it was in the original song or the melody will lose connection with the rest of the song. Experiments with changes in duration seem to indicate that large changes can cause a melody to lose smoothness and become unlistenable.

One method that works well for duration changes is to produce an altered melody that uses a random permutation of the duration values for the notes in each measure. This ensures each measure maintains the same duration values (though in a different order) as the original song contained. Also, some simple songs (such as "Twinkle Twinkle Little Star") just have a few notes with the same duration values in each measure. These values are kept the same as altering them can be too dramatic a change for a song with a just few notes in each measure. So, while more simple songs are more protected from atonality, they are also less diverse when changed by the system, in terms of note duration.

It is also possible to use Perlin noise to change the duration values. Noise can alter the

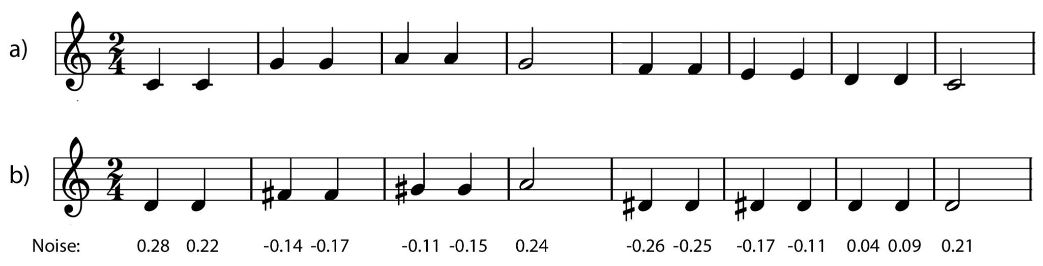

Figure 3 Example of Noise Effect on Notes: a) original melody b) melody modified with noise on each note

duration of a few notes while the durations of the rest of the notes can be altered to balance out the difference from the change brought by the noise. At this point, using noise to alter duration produces limited results that are not better than using permutations. However, as the noise feature is extended in later work to include possibilities such beat splitting or joining, using noise to alter durations as well as note values could become the better option.

5.3 Conclusions From Noise Experiments

Songs generally sound better when the melody rounds the new (noise altered) notes to the nearest chord note (dictated by the chord in the corresponding measure) as opposed to just changing to the note returned by the noise function. However, using chord rounding with every melody can produce output which is too similar. It is more desirable for the purpose of machine learning to have distinct melodies created each time the noise function is run on the same input melody. Chord rounding should be done for some of our output melodies, but not all. Songs with more notes can generally handle more variation in the frequency of chord rounding. Also, chord rounding is more important for long notes on strong beat positions than for short notes on intermediate positions.

It is possible for APOPCALEAPS to produce a rest instead of a chord for a given measure. In cases like these, it is best to use the chord from the previous measure when performing chord rounding on the notes (if the chords are all rests, no rounding will be done). This can be seen especially in more simple songs with few notes and not much variation, where the difference in perception between using chord rounding and not can be dramatic.

6 Future Work

Future work for this system involves two main areas: using the information that can be obtained from the new melodies for new purposes and extending the current noise system. For the first area it is clear that the various outputs produced by the noise algorithm pave the way for future work on a machine learning component for the APOPCALEAPS system. The user can hear different melodies produced by APOPCALEAPS and indicate to the system which sound preferable and which do not. A forthcoming machine learning algorithm can use this data sample to improve its rules for melody generation. First, a distance function is needed to show differences between different LilyPond files output by the noise system. At this point, the differences would be related to note and duration changes. The user can then classify the noise variations of a song based on taste. The parameters used by APOPCALEAPS to produce melody can be updated by the new data. Research already exists where previous notes are used to predict ideal new notes for a music system [6].

The second area of research shows much promise as well. The noise component is currently specialized for the melody. It could also be extended to the other voices of the APOPCALEAPS system. Furthermore, instead of just changing notes, the noise function could split or join beats in a similar way that APOPCALEAPS does in rhythm generation. As the APOPCALEAPS system develops, the noise function can be used to modify features not currently present in the system. In addition, the noise patterns could be used to change the volume of a song (to fade in and out).[1]

[1] Examples of APOPCALEAPS songs that were altered by the noise function in this paper can be found online: `http://dooz.myweb.uga.edu/noisecolin/music.html`

Acknowledgements We wish to thank the anonymous reviewers for their helpful comments and suggestions.

References

1 Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. ANTON: Composing Logic and Logic Composing. LPNMR 2009: 542-547.
2 Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. Automatic Composition of Melodic and Harmonic Music by Answer Set Programming. ICLP 2008: 160-174.
3 Thom Frühwirth. Constraint simplification rules. Tech. Rep. ECRC-92-18, European Computer-Industry Research Centre, Munich, Germany, July 1992.
4 Han-Wen Nienhuys and Jan Nieuwenhuizen. LilyPond, a system for automated music engraving. Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003), Firenze, Italy, May 2003.
5 Tae Hun Kim, Satoru Fukayama, Tanuya Nishimoto, and Shigeki Sagayama. Performance rendering for polyphonic piano music with a combination of probabilistic models for melody and harmony. Proceedings of the 7th International Conference of Sound and Music Computing (SMC2010), 23-30, 2010.
6 Tomasz Oliwa and Markus Wagner. Composing Music with Neural Networks and Probabilistic Finite-State Machines. Proceedings of the Sixth European Workshop on Evolutionary and Biologically Inspired Music, Sound, Art and Design (EvoMUSART 2008), Springer Berlin / Heidelberg, Springer, 503-508, 2008.
7 Ken Perlin. ACM SIGGRAPH 84 conference, course in Advanced Image Synthesis, 1984.
8 Ken Perlin. Improving noise. SIGGRAPH 02, Proceedings 729735, 2002.
9 Taisuke Sato. A glimpse of symbolic-statistical modeling by PRISM. Journal of Intelligent Information Systems, 31(2):161176, 2008.
10 Jon Sneyers and Danny De Schreye. APOPCALEAPS: Automatic Music Generation with CHRiSM. 22nd Benelux Conference on Artificial Intelligence (BNAIC'10), Luxembourg, October 2010.
11 Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. CHR(PRISM)-based probabilistic logic learning. 26th International Conference on Logic Programming, Edinburgh, UK, July 2010.
12 Jon Sneyers, Joost Vennekens, and Danny De Schreye. Probabilistic-logical modeling of music. Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL06), 6072, Charleston, SC, USA, January 2006.
13 Jong-Chul Yoon, In-Kwon Lee, and Jung-Ju Choi. Editing Noise. Journal of Computer Animation and Virtual Worlds, 15:3, 277287, 2004.
14 Yong-Woo Jeon, In-Kwon Lee, and Jong-Chul Yoon. Generating and modifying melody using editable noise function. In CMMR, volume 3902 of Lecture Notes in Computer Science, 164168. Springer, 2005.