AN INTRODUCTION TO *SLIPPERY CHICKEN*

Michael Edwards

Music
University of Edinburgh

ABSTRACT

This article introduces a new open-source algorithmic composition system, *slippery chicken*, which enables a top-down approach to musical composition. Specific techniques in *slippery chicken* are introduced along with examples of their usage in the author's compositions. The software was originally tailor-made to encapsulate the author's personal composition techniques, however many general-purpose algorithmic composition tools have been programmed that should be useful to a range of composers. The main goal of the project is to facilitate a melding of electronic and instrumental sound worlds, not just at the sonic but at the structural level. Techniques for the innovative combination of rhythmic and pitch data—arguably one of the most difficult aspects of making convincing musical algorithms—are also offered. The software was developed by the author in the *Common Lisp Object System* and released as open-source software in May 2012; see http://www.michael-edwards.org/sc.[1]

1. INTRODUCTION

"Formerly, when one worked alone, at a given point a decision was made, and one went in one direction rather than another; whereas, in the case of working with another person and with computer facilities, the need to work as though decisions were scarce—as though you had to limit yourself to one idea—is no longer pressing. It's a change from the influences of scarcity or economy to the influences of abundance and—I'd be willing to say—waste." (John Cage, quoted in [1])

The potential for software algorithms to enrich our musical culture has been established, in the 50+ years since such techniques were first introduced, by personalities as diverse as Hiller, Xenakis, Cage, and Eno. Algorithmic composition usually involves the use of a finite set of step-by-step procedures, most often encapsulated in software routines, to create music. The power of such systems is, arguably, still not fully understood or deeply investigated by the majority of musicians and composers, whether highly trained or not. Indeed, in the author's experience, a lot of the prejudice algorithmic composition pioneer Hiller suffered under [3] is still with us today. But there are clearly many riches to be mined in algorithmic composition, as the expression of compositional ideas in software often leads to unexpected and surprisingly new, exciting results, and these can seldom be achieved via traditional means.[2] Algorithmic composition techniques can thus play a vital and energising role in the development of modern music across all genres and styles.

slippery chicken is an open-source, specialised algorithmic composition program written in the general programming language *Common Lisp* and its object-oriented extension, the *Common Lisp Object System* (CLOS). Work on *slippery chicken* has been ongoing since 2000. By specialised as opposed to generalised, it is meant that the software was originally tailor-made to encapsulate the author's personal composition techniques and to suit his own compositional needs and goals. As the software has developed however, many general-purpose algorithmic composition tools have been programmed that should be useful to a range of composers. The system does not produce music of any particular aesthetic strain—for example, although not programmed to generate tonal music the system is quite capable of producing it. But if it is to be used to generate complete pieces it does prescribe a certain specialised approach; this will be described below.

slippery chicken has no graphical user interface and there are no plans to make one. Whilst it is clear that this will be off-putting to some, there are many benefits to interacting with such a system through the programming language it was created in, not least of which is the infinite-extensibility that such an approach infers. As the computer science adage goes, "When using WYSIWYG [What You See Is What You Get] systems, What You See Is All You'll Ever Get."[3]

The algorithmic system in *slippery chicken* is mainly deterministic but also includes stochastic elements if desired[4]. It has been used to create musical structure for pieces since its inception and for several years now has been at the stage where it can generate, in one pass, complete musical scores for traditional instruments. It can also, with the same data used to generate those scores, write sound files using samples, or MIDI file realisations of the instrumental

[1] Preparation of this software for open-source public release was supported by the UK's Arts & Humanities Research Council [grant number AH/J004529/1].

[2] Though the composer Clarence Barlow would perhaps disagree with this, as he states that in his algorithmic works "he would obtain the same results without the help of a computer" [16, 49].

[3] Despite much internet attribution, Donald E. Knuth has confirmed to the author that this quotation did not stem from him.

[4] Including, but not limited to, permutations in both a deterministic and random order. *slippery chicken* offers the use of fixed-seed randomness, so that repeatable results may be generated. For a discussion of the usefulness of such see [5, 64].

score. The project's main aim is to facilitate a melding of electronic and instrumental sound worlds, not just at the sonic but also at the structural level.[1] Hence certain processes common in one medium (for instance sound file slicing and looping) are transferred to another (the slicing up of notated musical phrases and the instigation of sub-phrase loops, for example). Techniques for the innovative combination of rhythmic and pitch data—arguably one of the most difficult aspects of making convincing musical algorithms—are also offered.

The system includes but is not mainly concerned with the automation of some of the more laborious aspects of instrumental composition—transposition, harmonic and rhythmic manipulation for example—and thus facilitates and encourages experimentation with musical data before committing to final forms. By generating music data algorithmically, independent of output format, structures become available for use in the preparation of digital and notated music—in the digital case, particularly for the generation of parameters for the digital synthesis and signal processing language Common Lisp Music (CLM [13]). The programme in nascent form was first used for the generation of the tape part of a piece by the author for solo violin, ensemble, and stereo tape: *slippery when wet*[2]. Its effectiveness in sonically and structurally integrating the instrumental and digital resources in that piece provided the impetus to pursue the idea further.

What *slippery chicken* is focused upon then is harnessing the rich data structure management of Common Lisp and CLOS to achieve a top-down approach to musical composition: defining, ordering, combining, and manipulating rhythmic, pitch, sound file, instrumental, and dynamic information, etc., into complete pieces of music or structures ready for further processing within or outwith the system. The output of the program is in the form of:

- MIDI files, generated with the help of Common Music's MIDI routines (CM 2.6.0 [16]), and containing all the tempo and meter information that facilitates reading into music notation software such as *Sibelius* or *Finale*
- music scores:
  - postscript files generated by interfacing with Common Music Notation (CMN [14]), and thus allowing the algorithmic use of arbitrary symbols, note heads, etc., for the encapsulation of extended instrumental techniques that are difficult or impossible to encode in MIDI
  - LilyPond input text files [10], with similar advantages to CMN, but more scope for post-generation intervention
- sound files, using samples driven by a custom, multi-channel CLM instrument.

The approach to algorithmic composition here is sequence or phrase-based (though this should not be confused with MIDI sequencing). In its most basic form, we define a certain number—a palette, in *slippery chicken* terms—of rhythmic phrases and pitch sets, then map these onto instruments through *rhythm* and *set map* objects which, when combined, select notes to form a complete piece.

One of the more challenging aspects of algorithmic composition—at least in pieces where there should be a semblance of phrases formed of horizontally connected notes—is the satisfactory combination of rhythms with pitches.[3] For instance, if we were to place a rhythmic phrase without pitch information in front of a trained composer, s/he could no doubt sing or play back a number of pitch contours that would subjectively work with these rhythms. The corollary of this is that not all pitch contours would work with the given rhythms, and that the contours would be influenced by the given rhythms, even if several solutions were available and the general shape of a line were more important than the exact pitches chosen. The reverse is also true: if the composer were offered a pitch contour without rhythms, then the selected rhythms would be influenced by the shape of the line. The process of matching one to the other is complex and idiosyncratic, dependent on culture, musical experience, taste, etc. Thus formalisation of this process is difficult.

slippery chicken's solution is to allow for the provision of an arbitrary number of *pitch sequences* (perhaps even algorithmically) to each *rhythm sequence*. The *pitch sequences* consist of a list of simple integers, one for each attacked rhythm (i.e. not for tied notes), over a user-defined range but where, for example, 2 would indicate a higher pitch than 1.

When *rhythm sequences* have been mapped to ensemble players, and *pitch sets* (harmonic material) to *rhythm sequences*, it is then a matter of *slippery chicken* selecting pitches from the current *pitch set* and *pitch sequence*. The algorithm will of course only choose notes that are within each instrument's range. A hierarchy to specify which instrument is given priority when the algorithm is assigning notes to instruments can also be defined, as an algorithmic attempt will be made to use as many notes of the set as possible, spreading them out amongst the instruments in the ensemble.[4]

Figure 1 and **Figure 2** show how this pitch selection algorithm can work for two instruments,

[1] Though it can be used purely for instrumental or computer music also.

[2] http://www.sumtone.com/work.php?workid=5.

[3] There are of course musical systems which decouple the organisation of pitch and rhythm material: medieval isorhythmic motets, integral serialism, and Cage's music composed with the *I Ching*, for example. But the interdependence of these two parameters continues to exist in a wide variety of musical contexts.

[4] Though there is a preference for selecting unused pitches from the set, if this is not possible then previously used pitches will be added until the number of notes available are as close as possible to the *pitch sequence's* ideal number.

flute and bassoon, with the *pitch sequence* 9 9 9 (3) 9 9 (3) 5. Numbers in parentheses indicate that a chord may be selected from the *pitch set*, if appropriate for the instrument. There is provision for adding chord selection hook functions (a default is provided) so that custom chords can be created for each instrument, or piece, as desired.

The *pitch set* in **Figure 1** has intentionally few low notes so that we can see how the range of the algorithmically generated bassoon line is comparatively narrow in comparison with that of the flute. The number of pitches available for an instrument might be considerably more or less than would ideally be demanded by the numbers in the *pitch sequence*, so the integer range of the pitch-sequence is either shifted or scaled by the number of available notes. The actual notes chosen are a function of a lookup routine, using the scaled/shifted and rounded *pitch sequence* numbers as indices. It is clear then that *pitch sequences* marry pitch contour to rhythm sequences but that they cannot be coerced into always and in any context specifying exact pitches. This is not the aim, especially as any *pitch sequence* can be applied to any *pitch set* and any instrument.

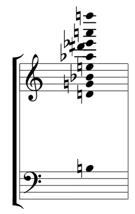


Figure 1. pitch set example used in **Figure 2**.

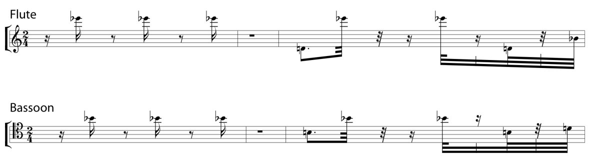


Figure 2. Example pitch selections for the pitch sequence 9 9 9 (3) 9 9 (3) 5.

Simple in concept at least then, the basic procedure for using *slippery chicken*—any part of which may be algorithmically or manually delivered—can be summed up as defining:

1. the instruments':
    a. ranges
    b. transpositions
    c. chord selection functions (if applicable)
    d. microtonal potential
    e. unplayable notes (e.g. microtones)
2. the instrument changes for individual players (e.g. flute to piccolo)
3. the *set palette* (harmonic fields) that the piece will use
4. the *rhythm sequence palette*
5. the *set map*: sets onto sequences
6. the *rhythm sequence map*: sequences onto instruments
7. the *tempo maps*
8. the *set limits*: for the whole piece and/or instruments.

2. GENERAL FEATURES

Although *slippery chicken* can of course perform many labour-intensive tasks (such as score writing, transposition, and, through its fundamental algorithms, pitch selection and sequence compiling), its main attraction is not, as the general computer myth would have it, as a labour saving system. For composers, arguably the primary benefit of this project and of algorithmic composition in general is that the encapsulation and expression of compositional structure in software often involves a form of practical experimentation which can lead to surprisingly new, rewarding, and exciting results. Randomness is not the issue here: many deterministic and, upon initial examination, seemingly predictable algorithms lead through the combination of a few steps to unimaginable music.

slippery chicken straddles the two poles of compositional process formalisation and what some would consider a relinquishing of compositional autonomy. With one of its main (but not unique) features being a bridging solution—leading composers with little algorithmic experience into the world of music computing, and bringing computer generation techniques to the world of instrumental music—*slippery chicken* offers a structured method as opposed to a composition software library. Clearly, any algorithmic composition system demands a certain, often idiosyncratic approach: it is a question of to what degree. Some systems are more open than others, and are therefore more akin to a software library: *SuperCollider* [8], *Pure Data* [11], *Max/MSP* [12], and systems made with the latter such as the *Real Time Composition Library* [6], for example. Others are more specialised: *FractMus* [7], David Cope's *Experiments in Musical Intelligence* [4], and Bernard Bel's *Bol Processor* [2].

slippery chicken is more akin to the specialised group. This is clearly its greatest advantage: complete pieces of music can be generated with relatively little input from the user. But, individualistic as they most often are, many composers won't find the method to their taste. These may still consider some of the *slippery chicken* classes, algorithms, and methods attractive. Many of the techniques can be applied without being coerced into the map/palette approach to generating complete pieces, so the package could be employed more as a software tool library also.

Because of its delivery format as an open-source, object-oriented Lisp package, *slippery chicken* is infinitely extensible. It can be used in its simplest form by entering the necessary musical data in lists and allowing the system to generate a complete piece of music. Or it can generate pieces, sections, phrases, etc., by making more sophisticated use of its internal generative classes and/or user-programmed extensions and subclasses. The generated data structures can also be altered through a host of included editing functions and methods. Here is where the tension between idealism and pragmatism found at the very beginnings of computer-based algorithmic composition and discussed in [5, 62-63] arises. To summarize briefly: Lejaren Hiller believed that if the output of the algorithm is deemed deficient, then the programme should be modified and the output regenerated; whereas Koenig and Xenakis took a more practical approach, treating the output of their algorithms to transcription, modification, and elaboration. Despite its perhaps idealistic goal of generating complete and coherent musical works, with its collection of internal data editing functions, *slippery chicken* remains solidly pragmatic. In its pre-2006 form, before the introduction of the pitch-selection algorithm, *slippery chicken* was arguably more in the camp of computer-aided composition than that of algorithmic composition, to use Monro's distinction [9]. It is now more firmly in the algorithmic composition camp, with the potential to act merely as a digital composition assistant if so desired.

3. SLIPPERY CHICKEN TECHNIQUES AND ALGORITHMS

3.1. With CLM

When writing sound files with a custom, 4-channel CLM instrument, *slippery chicken* uses essentially the same data as used for generating MIDI and score files. Notation details such as ties, dots, clefs, etc., have no bearing upon sound file generation of course, and are thus ignored. In order to generate sound files, a *sound file palette* slot in the *slippery chicken* object is used. This stores data associated with groups of sound files; they will be cycled through during the algorithmic processing. There can be any number of sound file groups, allowing the musical data to be applied to several different categories of sound. This can create interesting variations of recognizably similar musical material. Moreover, the use of the same musical data for the generation of scores and sound files in this fashion creates, when they are combined in hybrid musical works, exactly the kind of melding of electronic and instrumental sound worlds mentioned in the overview: striking pitch and rhythm structures will be audibly related whether presented by acoustic instruments or in sound files.

The chronological placement and mixing in of sound files with CLM is triggered according to the start times of notes in the score. This may be scaled up or down for a faster or slower rendition of the score. Upward and downward transpositions, from a user-defined zero-transposition point, will also be carried out if the user so wishes, in accordance with the pitches' deviation from the zero-transposition point. This may also be scaled, as desired. *slippery chicken* takes advantage here of the high-quality transposition algorithm of CLM[1]; this convolves its input with a sinc function. Duration may also be scaled so as to create a thicker texture through overlaps (if input sound file lengths allow). Start time within the input sound file can be automatically incremented upon reuse, in order to avoid repetitions of opening details: if the algorithm is long enough, it will slowly increment its way through the complete duration of each of the sound files as they are processed in turn.

This engagement with *slippery chicken* in combination with CLM has become a self-sufficient project, one that has, apart from the author's main instrument-with-computer works, generated a collection of short pieces in the form of downloadable sound files. Contrary to traditional electroacoustic studio work (which can often be thought of as sound sculpting) the approach here is to generate perhaps hundreds of sound files automatically from a given *sound file palette* and set of compositional data. Sound file selection then becomes the main activity when using the output of *slippery chicken*, in keeping with the Cage quotation at the beginning of the overview. The best results of the algorithms (with no post-output editing) are to be found on the internet as short but complete pieces.[2]

3.2. Fibonacci Transitions

Transitions between different musical sections or states have been an important characteristic of Western Classical music for centuries. In the eighteenth and nineteenth centuries there were transitions between the first and second subject groups in sonata form; in the twentieth century, there is the textural morphing of Ligeti's micro-polyphonic structures in, for example, *Atmosphères* (1961). Transition strategies are built into *slippery chicken* in two main forms: the *procession* algorithm[3] and *Fibonacci Transitions*.

Transitions in *slippery chicken* are aimed at the development and variation of musical material at the macrostructural level. For example, this could be used to intersperse a new audio segment into a sample loop; to gradually transform the repetition of one *rhythm sequence* into another; or to transition between different harmonic fields; etc.

In *Fibonacci Transitions*, new elements are 'folded into' existing repeating elements according to a number of repetitions determined by Fibonacci numbers[4]. Such transitions are available in simple

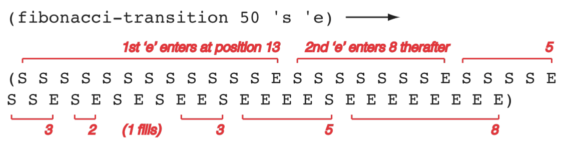
[1] Developed by Bill Schottstaedt in collaboration with Perry Cook and Julius Smith.

[2] http://www.sumtone.com/search.php?title=scei

[3] This moves through a list by alternating adjacent elements, progressing through to the higher-order elements by interspersion, but with an algorithmic eye on more or less equal statistical distribution (e.g. 1 2 1 2 3 1 3 1 1 4 2 3 2 4 3 4 3 4 5 2 4 2 2 5 1 3 1 5 4 5 4 5 6 3 5 3 3 6 1).

[4] Fibonacci was the Italian mathematician (c.1170-c.1250) after whom the famous number series is named. This is a simple

two-datum (**Figure 3**) or multi-datum forms (**Figure 4**).



```
(fibonacci-transition 50 's 'e) ——→
(S S S S S S S S S S S S E S S S S S S S E S S S S E
S S E S E S E S E E S E E E E S E E E E E E E E)
```

Figure 3. Simple two-datum *Fibonacci Transition*.[1]

```
(fibonacci-transitions 100 '(s e q h)) →
(S S S S S S S S S E S S S S E S S S E S S E S
 E S E E S E E S E E E E E E E E Q E E Q E E
Q E Q E Q Q E Q Q E Q Q Q Q Q Q Q Q H Q Q H
Q Q H Q H Q H H Q H H Q H H H H H Q H H H H H
H H H H H H H H H)
```

Figure 4. Multi-datum *Fibonacci Transition*.

slippery chicken algorithms such as *rhythm chains*[2] use *Fibonacci Transitions* transparently. They were first used directly in the author's *breathing Charlie*,[3] for alto saxophone and computer, to control the interspersion and development of audio loop segments. A good example is from the author's *in limine*,[4] for two soprano saxophones and computer. The reader can listen to a sound file (*sweet-sax-loops.mp3*) from this piece online.[5] The segment interspersions here are combined with microtonal transpositions for greater pitch variety and interest. The process is of course automated: loop points can be entered in the sound file editing software Audacity or WaveLab, and marker files created by these programmes can be read by *slippery chicken*. An audio slice, from one marker to the next, will be repeated a number of times determined by *Fibonacci Transitions*, before the next audio slice is cut in. Once the second slice dominates, according to the *Fibonacci Transition*, a third begins to be interspersed and the process is repeated as many times as determined by the user, with as many slices as desired, and with or without transpositions, shuffled permutations, etc.

3.3. Intra-Phrasal Loop Chopping

As the intention of *slippery chicken* was always the structural marrying of computer-generated and instrumental resources, it became compelling to transfer the idea of *Fibonacci Transitioning* audio loops into the score domain. Chopping and looping of notated rhythms was first applied in the author's *cheat sheet*,[6] for solo electric guitar and eight-piece ensemble. The five bars of four-part counterpoint shown in **Figure 5** represent all the rhythmic and contrapuntal material available for this 1167 bar piece.

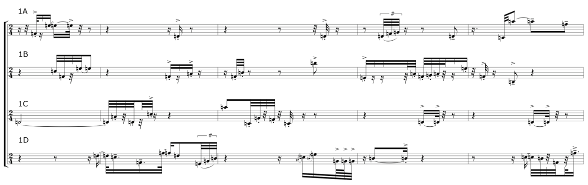


Figure 5. Original *rhythm sequence palette* for the author's *cheat sheet*.

The essentially DSP-inspired looping technique is applied to conventionally notated musical material by dividing the five bars of four-part counterpoint into 400 segments: 100 per voice, ten per crotchet (quarter note), with ten crotchets total in five 2/4 bars. The ten crotchet loop points have the semiquaver as the shortest unit. The start and stop points were defined as (1 4) (1 3) (1 2) (2 4) (2 3) (3 4) (1 1) (2 2) (3 3) (4 4); see **Figure 6**.

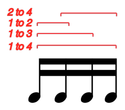


Figure 6. Semiquaver (16th note) loop points within a single crotchet (quarter note).

The progression through the 100 loop points per voice is controlled by a modified *Fibonacci Transitions* call. The modification is a subroutine called *remix-in*. This inserts earlier segments between adjacent segments so as to avoid a purely binary opposition of rhythmic materials and thus enrich the musical development and associations. It also provides more structural cohesion, investigating previous segments' rhythmic and metrical effects in the new context.

The *Fibonacci Transition* in *cheat sheet* creates 2177 segments per voice, moving from the beginning of the first bar to the end of the fifth of the original material shown in **Figure 5**. It returns numbered references ranging from 1 to 100: 1-10 refer to the loop points in the first crotchet (quarter note); similarly 11-20 refer to the loop points in the second crotchet, etc. The beginning and end of the transition is shown in **Figure 7**.

progression where successive numbers are the sum of the previous two: (0), 1, 1, 2, 3, 5, 8, 13, 21.... As we ascend the sequence, the ratio of two adjacent numbers becomes closer to the so-called Golden Ratio (approx. 1:1.618).

[1] Note that the number of items returned will always correspond to the first argument. Varying repetitions of the lower order alternations will fill any shortfall; these are labelled *(1 fills)* in **Figure 3**.

[2] http://sites.ace.ed.ac.uk/algocomp/2011/07/01/you-are-coming-into-us-who-cannot-withstand-you/

[3] http://www.sumtone.com/work.php?workid=132

[4] http://www.sumtone.com/work.php?workid=131

[5] http://www.michael-edwards.org/sc/paper/

[6] http://www.sumtone.com/work.php?workid=182

```
(1 1 1 1 2 1 1 2 1 2 1 2 2 1 2 1 2 1 2 2 2 1 2
 2 3 2 3 2 3 1 3 3 2 1 3 3 1 3 2 1 3 2 3 4 1 3
 4 1 3 4 2 4 1 3 2 4 4 4 4 5 1 4 4 5 2 4 5 2 4
 ...
  98 98 46 98 97 47 98 98 46 99 47 98 99 46 98
 99 47 99 99 47 98 46 99 47 99 99 98 99 99 46
 100 99 100 47 99 100 46 100)
```

Figure 7. Fibonacci Transition with *remix-in* modification

Taking the opening flute and clarinet parts, and comparing the score with the original four-part counterpoint (line 1A in **Figure 5**), we can see how this develops in **Figure 8**. The rhythms and meter were doubled for ease of reading.

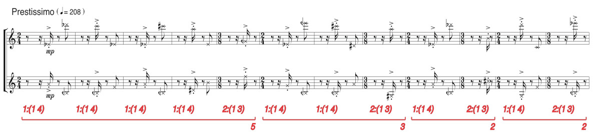


Figure 8. Rhythmic loop slice mapping in the flute and clarinet parts at the beginning of the author's *cheat sheet*.

3.4. Transitioning Lindenmayer Systems

Lindenmayer Systems (or L-Systems) are in their simplest form deterministic. For musical composition, this class of algorithms is often preferable to its stochastic counterpart due to the repeatability of results when regenerating material (regeneration typically being necessary in the generate-modify-regenerate iterative process of algorithmic composition). See [5, 64] for further discussion of this issue and for examples of a basic L-System.

One of the attractions of L-Systems is self-similarity; see **Figure 10** for an illustration of this. The generated numbers (or any data type) can of course be applied to any musical parameter or material.

A *slippery chicken* development, *Transitioning L-Systems* use data returned by an L-System as lookup indices into a substitution table. This table may contain any data, including further references to other data structures (e.g. *rhythm sequence palettes*). The result of the substitution depends on transitions between an arbitrary number of related but perhaps developing material—such relationships are envisaged though they are not of course enforced. The transitions are created by *Fibonacci Transitions*. Each of the transitions may also contain an arbitrary number of data points in a list; these will be cycled through each time a particular transition is returned.

Returning to *cheat sheet, slippery chicken*'s L-System implementation is used in three main ways: as a straightforward cycling mechanism; as a simple L-System without transitions; and as a *Transitioning L-System*. The latter two uses in this musical context will now be discussed to illustrate their properties.

As mentioned, in *cheat sheet* there are 2177 *rhythm sequences* created as loop segments, each of which has an associated *pitch set*. All *pitch sets* are based on six basic guitar fingerings but include pitch extensions above and below the guitar's range to be used by the ensemble. This illustrates a basic principle that the author tends to follow in all algorithmic works: that working from the instrument outwards—in particular, a software representation of how it is or can be played—tends to create more instrumentally idiomatic works than trying to fit the results of an algorithm, however interesting it may be, onto an instrument for which the algorithm wasn't designed.

The guitar fingerings are superimposed onto the scordatura guitar tuning shown in **Figure 9**, where strings two and six are detuned by a quartertone.

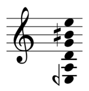
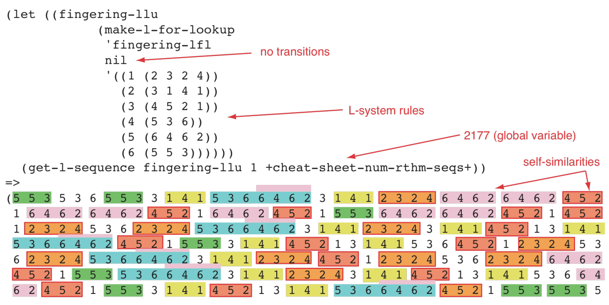


Figure 9. Guitar tuning (scordatura) for the author's *cheat sheet*.

A threefold process created the pitch sets:

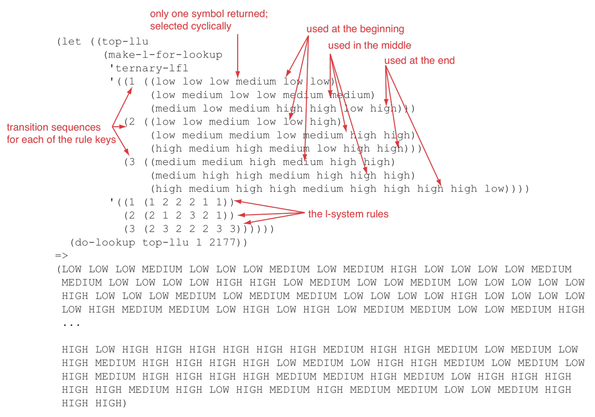
1) 6 guitar chords were chosen by ear. These had fingerings: `(4 2 1 3) (4 1 2 3) (4 1 3 2) (3 2 1 4) (3 4 1 2) (4 3 1 2)`. The sequencing of these was organised by a simple L-sequence: there are no transitions here but self-similar patterns do emerge (**Figure 10**).



```
(let ((fingering-llu
       (make-l-for-lookup
        'fingering-lfl
        nil
        '((1 (2 3 2 4))
          (2 (3 1 4 1))
          (3 (4 5 2 1))
          (4 (5 3 6))
          (5 (6 4 6 2))
          (6 (5 5 3))))))
  (get-l-sequence fingering-llu 1 +cheat-sheet-num-rthm-seqs+))
=>
(5 5 3 5 3 6 5 5 3 3 1 4 1 5 3 6 6 4 6 2 3 1 4 1 2 3 2 4 6 4 6 2 6 4 6 2 4 5 2
 1 6 4 6 2 6 4 6 2 4 5 2 1 6 4 6 2 4 5 2 1 5 5 3 6 4 6 2 6 4 6 2 4 5 2 1 4 5 2
 1 2 3 2 4 5 3 6 2 3 2 4 5 3 6 6 4 6 2 3 1 4 1 2 3 2 4 3 1 4 1 4 5 2 1 3 1 4 1
 5 3 6 6 4 6 2 4 5 2 1 5 5 3 3 1 4 1 4 5 2 1 3 1 4 1 5 3 6 4 5 2 1 2 3 2 4 5 3
 6 2 3 2 4 5 3 6 6 4 6 2 3 1 4 1 2 3 2 4 4 5 2 1 2 3 2 4 5 3 6 2 3 2 4 6 4 6 2
 4 5 2 1 5 5 3 5 3 6 6 4 6 2 3 1 4 1 2 3 2 4 3 1 4 1 4 5 2 1 3 1 4 1 5 3 6 6 4
 6 2 4 5 2 1 5 5 3 3 1 4 1 4 5 2 1 3 1 4 1 5 3 6 6 4 6 2 4 5 2 1 5 5 3 5 5 3 5
```

Figure 10. Self-similarity in L-System results.

2) Whether to play these chords on the four lowest, four middle, or four highest strings (using the first finger as a *barre* on the remaining strings in each case) is determined by a *Transitioning L-System* (**Figure 11**).



```
(let ((top-llu
       (make-l-for-lookup
        'ternary-lfl
        '((1 ((low low low medium low low)
              (low medium low low medium medium)
              (medium low medium high high low high)))
          (2 ((low low medium low low high)
              (low medium medium low medium high high)
              (high medium high medium low high high)))
          (3 ((medium medium high medium high high)
              (medium high high medium high high high)
              (high medium high high medium high high high high low))))
        '((1 (1 2 2 2 1 1))
          (2 (2 1 2 3 2 1))
          (3 (2 3 2 2 2 3 3))))))
  (do-lookup top-llu 1 2177))
=>
(LOW LOW LOW MEDIUM LOW LOW LOW MEDIUM LOW MEDIUM HIGH LOW LOW LOW LOW MEDIUM
 MEDIUM LOW LOW LOW LOW HIGH HIGH LOW MEDIUM LOW LOW MEDIUM LOW LOW LOW LOW LOW
 HIGH LOW LOW LOW MEDIUM LOW MEDIUM MEDIUM LOW LOW LOW LOW HIGH LOW LOW LOW LOW
 LOW HIGH MEDIUM MEDIUM LOW HIGH LOW HIGH LOW MEDIUM MEDIUM LOW LOW MEDIUM HIGH
 ...

 HIGH LOW HIGH HIGH HIGH HIGH HIGH HIGH MEDIUM HIGH HIGH MEDIUM LOW MEDIUM LOW
 HIGH MEDIUM HIGH HIGH HIGH HIGH LOW MEDIUM LOW HIGH HIGH MEDIUM LOW MEDIUM LOW
 HIGH MEDIUM HIGH HIGH HIGH HIGH MEDIUM MEDIUM HIGH MEDIUM LOW HIGH HIGH HIGH
 HIGH HIGH MEDIUM HIGH LOW HIGH MEDIUM HIGH MEDIUM MEDIUM LOW LOW MEDIUM HIGH
 HIGH HIGH)
```

Figure 11. *cheat sheet's* guitar chord ternary Transitioning L-System.

Figure 11 has three transition sequences, each with three cyclic lists. The latter correspond to which cyclic list will be returned at which stage of the *Fibonacci Transition* as it is spread over the 2177 chords, i.e. beginning, middle, and end. Each of these cyclic lists, in all three transition sequences, tend to become higher by the end, with transition sequence 1 also generally being lower than 2, which is lower than 3. What this creates, when the algorithm is run, is a tendency to move from lower to higher chords as viewed across the whole piece. However, by creating the transition in this manner, the development does not take place in an obvious, linear way—which our sophisticated auditory-cognitive system could all too quickly pick up on and perhaps judge as being too predictable—rather, it takes place in an unpredictable manner when viewed locally, but in a clearly rising manner when perceived globally.

3) We now have fingerings and strings, but no fret position. This is determined in an altogether different manner, by a 'fret curve' (or breakpoint function: see **Figure 12**.). The guitar generally has 19 frets, so the chords can be created with the first finger placed on frets 1-15.

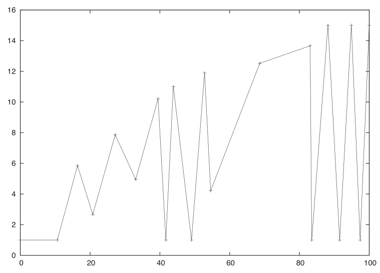


Figure 12. First finger fret selection curve for the author's *cheat sheet*. The x-axis is scaled automatically to fit the 2177 chords.

When this threefold process is combined, the result is a list of chord references each consisting of three elements:

1. which four of the six strings to finger (lowest four, middle four, or highest four)
2. the first finger fret (1-15)
3. the fingering, as an index (1-6) into the fingering list ((4 2 1 3) (4 1 2 3) (4 1 3 2) (3 2 1 4) (3 4 1 2) (4 3 1 2)))

The beginning and end of the results of this process are shown in **Figure 13**.

```
(LOW 1 5) (LOW 1 3) (LOW 1 5) (MEDIUM 1 3)
(LOW 1 5) (LOW 1 6)     (LOW 1 5) (MEDIUM 1 3)
(LOW 1 5) (MEDIUM 1 3) (HIGH 1 1) (LOW 1 4)
(LOW 1 1) (LOW 1 5) (LOW 1 3) (MEDIUM 1 6)
(MEDIUM 1 4) (LOW 1 6) (LOW 1 2) (LOW 1 6)
(LOW 1 3) (HIGH 1 1) (HIGH 1 4) (LOW 1 1)
…
(HIGH 11 6) (HIGH 11 4) (HIGH 11 6) (HIGH 12 2)
(MEDIUM 12 6) (HIGH 12 3) (LOW 12 1)
(HIGH 13 4) (MEDIUM 13 1) (HIGH 13 2)
(MEDIUM 13 3) (MEDIUM 14 2) (LOW 14 4)
(LOW 14 3) (MEDIUM 14 1) (HIGH 15 4)
(HIGH 15 1) (HIGH 15 4)
```

Figure 13. Results of the Transitioning L-System combined with the simple L-System and the fret curve for the generation of the chord sequence in the author's *cheat sheet.*

See **Figure 14** for some results of this process using the six guitar fingering patterns discussed.

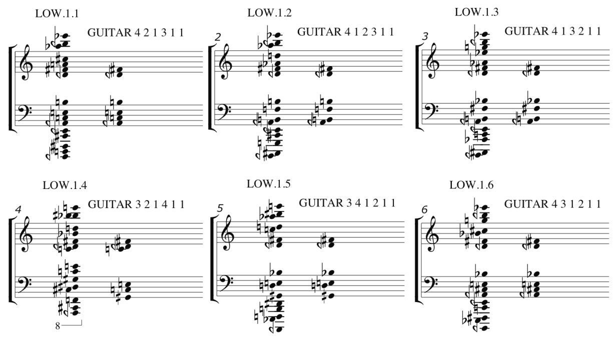


Figure 14. Guitar chords and ensemble extensions for the author's *cheat sheet*.

4. CONCLUSION

Though focussed mainly on the algorithmic production of complete pieces for instruments and computer, the *slippery chicken* package includes several unique approaches to generating musical structure that may be used in other contexts. Its top-down approach to compositional organisation offers considerable potential for explorative, iterative development, freeing the composer from the commitment to a single labour-intensive path. This can lead, if so desired, to unimagined aesthetic realms with relative ease. Its integration of score and MIDI file writing, along with the use of the same musical data for the generation of sample-driven sound files, strengthens the audible structural links between the often-disparate worlds of acoustic and electronic composition. Its approach to various transition strategies can be employed towards evolving musical structures out of relatively little, and therefore coherent, musical material. Its release as open-source, object-oriented Common Lisp code encourages further development and extensions on the part of the user.

5. REFERENCES

[1] Austin, L., J. Cage, and L. Hiller. 1992. "An Interview with John Cage and Lejaren Hiller". *Computer Music Journal*, 16 (4), 15–29.

[2] Bel, B. 1998. "Migrating Musical Concepts: An Overview of the Bol Processor." *Computer Music Journal,* 22 (2), 56–64.

[3] Bewley, J. 2004. "Lejaren A. Hiller: Computer Music Pioneer." Music Library Exhibit, University of Buffalo. http://library.buffalo.edu/libraries/units/music/ exhibits/hillerexhibitsummary.pdf (accessed August 12th 2009).

[4] Cope, D. 1996. "Experiments in Musical Intelligence." Madison, WI: A-R Editions.

[5] Edwards, M. 2011a. "Algorithmic Composition: Computational Thinking in Music." *Communications of the Association for Computing Machinery,* 54 (7), 58–67.

[6] Essl, K. 2010. *Real Time Composition Library*. http://www.essl.at/works/rtc.html (accessed 22/1/2012).

[7] Diaz-Jerez, G. 2000. *FractMus*. http://www.gustavodiazjerez.com/fractmus_overview.html (accessed 22/1/2012).

[8] McCartney, J. 2011. *SuperCollider*. http://supercollider.sourceforge.net (accessed 22/1/2012).

[9] Monro, G. 1997. "This is art, not science." *Leonardo Music Journal,* 7 (1), 77.

[10] Nienhuys, H.-W. 2011. *Lilypond*. http://www.lilypond.org/ (accessed 22/1/2012).

[11] Puckette, M. 2011. *Pure Data*. http://crca.ucsd.edu/~msp/software.html (accessed 22/1/2012).

[12] Puckette, M. and D. Zicarelli 2011. *Max/MSP*. http://cycling74.com/products/max/ (accessed 22/1/2012).

[13] Schottstaedt, B. 2011a. *Common Lisp Music*. https://ccrma.stanford.edu/software/clm/ (accessed 22/1/2012).

[14] Schottstaedt, B. 2011b. *Common Music Notation*. Open-source music software. https://ccrma.stanford.edu/software/cmn/ (accessed 22/1/2012).

[15] Supper, M. (2001). "A Few Remarks on Algorithmic Composition." *Computer Music Journal,* 25 (1), 48–53.

[16] Taube, H. 2005. *Common Music 2.6.0*. http://commonmusic.sourceforge.net/ (accessed 22/1/2012).