slippery chicken
a specialised algorithmic composition program

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 also at the structural level. Techniques for the innovative combination of rhythm 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.

Keywords

- Algorithmic Composition
- Automatic Composition
- Computer-aided Composition
- Sound file generation
- MIDI file generation
- Score generation
- Common Lisp
- Common Lisp Object System (CLOS)
- Common Lisp Music

  - Common Music
  - Common Music Notation
  - LilyPond
  - Algorithms:
    - Fibonacci Transitions
    - Lindenmayer Systems
    - Rhythm Chains
    - Set Manipulation
    - Permutations
    - Popcorn Dynamics
    - Intra-Phrasal Loop Chopping

Overview

> "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 Austin et al. 1992)

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 (Bewley 2004) 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.[1] Algorithmic composition techniques can thus play a vital and energising role in the development of modern music across all genres and styles.

This article won't debate the history of algorithmic composition as this is done elsewhere (e.g. Edwards 2011a; Essl 2007; Nierhaus 2009; Roads 1996, 856-909). Instead, it will introduce and illustrate techniques implemented in *slippery chicken*, with examples of their usage in the author's compositions.

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

[1] 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" (Supper 2001, 49).

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."[2]

The algorithmic system in *slippery chicken* is mainly deterministic but also includes stochastic elements if desired.[3] 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

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

[3] 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 (Edwards 2011a, 64).

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 score. The project's main aim is to facilitate a melding of electronic and instrumental sound worlds,[4] not just at the sonic but also at the structural level. 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: Schottstaedt 2011a). 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* (Edwards 2000). Its effectiveness in sonically and

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

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: Taube 2005), 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: Schottstaedt 2011b), 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 (LilyPond 2011), 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. For an example of the type of rhythm data used initially, see Figure 1 for the first eight *rhythm sequences* of the first piece composed by the author using this software*, slippery when wet* (Edwards 2000).

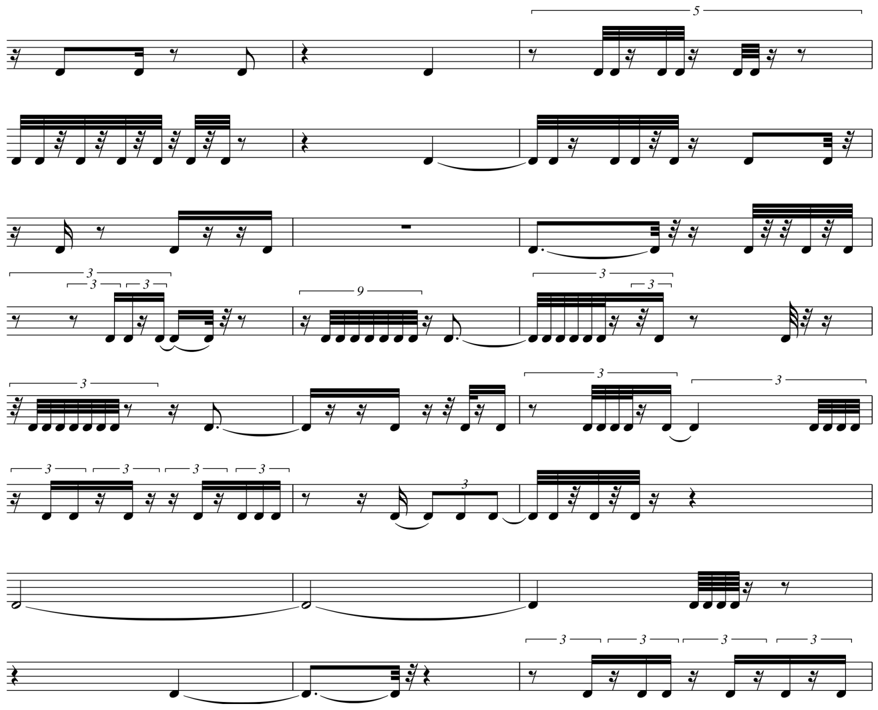


Figure 1: Part of the *rhythm sequence palette* of the author's *slippery when wet.*

These *rhythm sequences* were initially sketched by hand before being entered into Lisp-readable format.[5] (Nowadays they would more likely be generated by algorithmic techniques such as *rhythm chains*, as discussed below.) Of paramount importance at the time of sketching and selection was that each sequence should be both contrapuntally (vertically) and melodically (horizontally, one after the other) combinable with any other sequence, and that it should be "playable" on any of the instruments in the ensemble. Once a palette of 21 sequences was decided upon for *slippery when wet*, they were mapped across the ensemble for the twelve sections of the whole piece. This is illustrated in Figure 2. By planning compositional structure in this top-down manner, we immediately see the advantage of being able to design and control large-scale formal development. For example, the piece has two tendencies visible here: First, a movement from the whole ensemble playing together (*tutti*), but gradually reducing to section 10, where the solo violin is playing alone; and second, from each instrument playing different *rhythm sequences*, to more and more playing in rhythmic unison (these are indicated by yellow highlighting).

[5] Rhythm data can now include all articulations, ties, tuplets, etc. necessary for notation.

1	solo violin	1	10	19	13	12	5	21	**2**	2	11	20	15	10	9	8	**3**	3	12	21		14	13	10
	violin	2	11	20	15	10	9	8			12	21	17	14	13	10			13	1		21	17	14
	viola	3	12	21	17	14	13	10			13	1	16	19	15	14			14	3		21	17	16
	cell	4	13	1	16	19	15	14			14	3	18	21	11	16		6	15	7		21	17	19
	flute	5	14	3	18	21	11	16			15	7	2	20	17	19			16	5		1	2	12
	clarinet	6	15	7	2	20	17	19			16	5	4	1	2	12			17	9		3	4	18
	horn	7	16	5	4	1	2	12			17	9	6	3	4	18			18	8	11	7	6	20
	percussion	8	17	9	6	3	4	18			11	8	11	7	6	20			10	19		12	5	21

4	solo violin		13	1	16	19	15	14	**5**		14	3		21	11		**6**		15	7	2	20	17	19
	violin		14	3	18	21	11	16				7		20	17	19			16		13	1	2	12
	viola			7	2	20	17	19		8	10	5		1				8				3		18
	cell		16		4	1	2	12		8	10	9				18		9	18		13	1	6	
	flute	8	17	19		3	4	18			18	8			6	20				19	13	1	5	21
	clarinet	9	18	19	11		6	20			10			12	5	21					15	10	9	
	horn		10	19	13	12		21		2				10	9	8			12			1		
	percussion		11	20	15	10	9				12	21	17	14	13	10			13	1		19		14

7	solo violin			5	4	1	2	12	**8**	8	17	9	6		4	18	**9**	9	18	8	11		6	20
	violin			9	17	14				9	18					20		6				12		21
	viola	9		8	11	7	6	20		3		19			5	21		2	11	20		12		21
	cell		10		17	14				3		19		10	5			3		21		12		21
	flute		11	20				8		3		21	17					4		1	16			21
	clarinet		12		17	14	13	10		4		19	16					6		3		21		16
	horn				16					3	14	19						6				20	17	19
	percussion		14		18		11			6		7						6				1		21

10	solo violin	1	10	19	13	12	5	21	**11**	2	11	20	15	10	9	8	**12**	3	12	21	17	14	13	10
	violin									3	12	21	17	14	13	10		4	13	1	18	14		
	viola									3				19	15	14		5	14	3	18	14		
	cell									5								6	7	3	18	14		
	flute									3								7			18	14	2	12
	clarinet									3										3	18	14		
	horn									8								7				14		
	percussion									3								7				12	5	21

Figure 2: ***Rhythm sequence map*** **for the author's** ***slippery when wet*****. Larger bold numbers indicate section number; smaller numbers indicate the** ***rhythm sequence*** **number to be used by each particular instrument.**

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.[6] For

[6] There are of course musical systems which decouple the organisation of pitch and rhythm material: medieval isorhythmic motets, integral serialism,

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. Though showing at least some similarities to the integral serialists' approach to music-parametric organisation, the marrying of pitch to rhythm data in this manner—a relation of the qualities of a group of rhythms and their implications for the associated pitch contour—is

---

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.

arguably preferable to the basic serial method of a separate relation of pitch and rhythm to the series only, on a note-by-note basis.

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.[7]

Figure 3 and Figure 4 show how this pitch selection algorithm can work for two instruments, 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 3 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

[7] 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.

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.

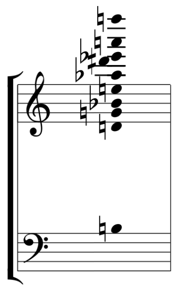


Figure 3: pitch set example used in Figure 4.

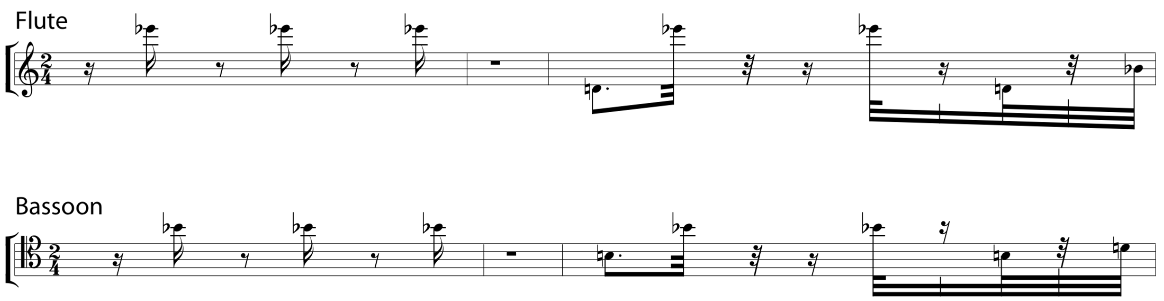


Figure 4: Example pitch selections for the pitch sequence `9 9 9 (3) 9 9 (3) 5.`

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.

Three limitations or requirements of the system should be mentioned here:

1. No doubt it will not have escaped the careful reader's notice that one and only one *pitch set* is mapped onto each *rhythm sequence.* In the aesthetic context of the author's works made with *slippery chicken* to date, this presents no problem. However, if we think of *rhythm sequences* as phrases, which they can indeed be thought of as at least conceptually relating to, then in the context of tonal and some other

musics, one chord per phrase is certainly out of the ordinary, perhaps even restrictive. *Rhythm sequences* may be of any arbitrary length however, so if a faster harmonic rhythm is desired, shorter sequences can be used—at the expense of then needing more *rhythm sequences* to make up the piece, and with the challenge of creating horizontally connected lines from smaller units, with the attendant difficulty of making the *pitch sequences* work when combined.

2. All *rhythm sequences* to be combined contrapuntally must be of the same length. As one *pitch set* is applied to exactly one *rhythm sequence* there can be no overlapping of *pitch sets*. There is provision however for shorter *rhythm sequences* to be combined to match (exactly) the length of a contrapuntally combined longer *rhythm sequence.*
3. The system is non-real time. There is no provision for real-time, infinite generation of material. There are two main reasons for this:
    a. Several processes, especially the DSP routines performed in CLM, are potentially so computationally intensive as to be impossible in real-time to date.
    b. As several non-DSP processes rely on breakpoint functions for the generation of data, the finite length of the piece must be known in advance in order to make the required calculations. This problem could be obviated by generating chunks of material in advance, using breakpoint curves of finite lengths not necessarily relating to the complete duration of a piece. This

would be an interesting extension to *slippery chicken* that may form part of future development work.

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 that 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.

Through the ability to generate pieces of an arbitrary length an arbitrary number of times, refining data as the musical results are evaluated and improved, the advantage of auditioning pieces in various guises or flavours, with different instrumentation, harmony, speeds of transition, etc., cannot be overestimated: even small changes to parameter values can have a significant and often unforeseeable impact on the musical output. The potential to expand the composer's vision and lead his or her ear to unimagined places in this manner becomes very powerful.

Roger Alsop writes, "an algorithmic system relieves the composer of many decisions" (Alsop 1999). This may be true if the user was not the designer of the system and makes only superficial use of it, but the opposite is more often the case. An algorithmic system forces the composer to confront decisions and, especially if the system is to be of the composer's own design, to formalise what might only loosely or even unconsciously be a system. In using algorithmic systems, composers are encouraged to stretch themselves and be explicit about their musical data and their organization. The serendipities that may arise from this process can spur the composer to break norms in a way that pure intuition inculcated by experience is perhaps less likely to do.

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* (McCartney 2011), *Pure Data* (Puckette 2011), *Max/MSP* (Puckette and Zicarelli 2011), and systems made with the latter such as the *Real Time Composition Library* (Essl 2010), for example. Others are more specialised: *FractMus* (Diaz-Jerez 2000), David Cope's *Experiments in Musical Intelligence* (Cope 1996), and Bernard Bel's *Bol Processor* (Bel 1998).

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 (hence in this way at least it fits Alsop's view). 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. Two of the author's recent works do this: *who says this, saying it's me?*, for tenor saxophone and quadraphonic sound files (Edwards 2009); and *don't flinch*, for acoustic-electric guitar and computer (Edwards 2011b). In both these cases, musical events generated with *slippery chicken* algorithms were written to MIDI files and further processed thanks to the incorporation in *slippery chicken* of Common Music's MIDI subsystem (Taube 2005).

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 (Edwards 2011a, 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 Monroe's distinction (Monroe 1997). It is now more firmly in the algorithmic composition camp, with the potential to act merely as a digital composition assistant if so desired.

slippery chicken techniques and algorithms

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;[8] 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

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

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.[9]

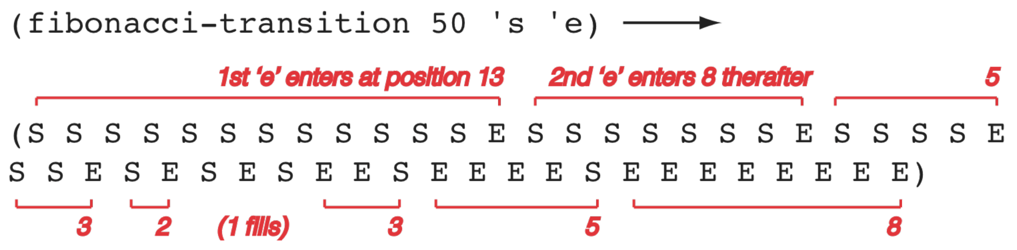
Fibonacci Transitions

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, which is described in the section on *rhythm chains* below, 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.

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

In *Fibonacci Transitions*, new elements are ‘folded into’ existing repeating elements according to a number of repetitions determined by Fibonacci numbers.[10] Such transitions are available in simple two-datum (Figure 5) or multi-datum forms (Figure 6).



```
(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 5: Simple two-datum *Fibonacci Transition*.[11]

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

[11] 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 5.

```
(fibonacci-transitions 100 '(s e q h)) →
(S S S S S S S S S E S S S S E S S E S S E S
E S E E S E E S E 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 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 6: Multi-datum *Fibonacci Transition*.

Algorithms such as *rhythm chains* (see below) use *Fibonacci Transitions* transparently. They were first used directly in the author's *breathing Charlie*, for alto saxophone and computer (Edwards 2004), to control the interspersion and development of audio loop segments. A good example is from the author's *in limine*, for two soprano saxophones and computer (Edwards 2005). The reader can listen to a sound file (*sweet-sax-loops.mp3*) from this piece online at http://www.michael-edwards.org/sc/paper/. 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.

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* (Edwards 2007a), for solo electric guitar and eight-piece ensemble. The five bars of four-part counterpoint shown in Figure 7 represent all the rhythmic and contrapuntal material available for this 1167 bar piece.

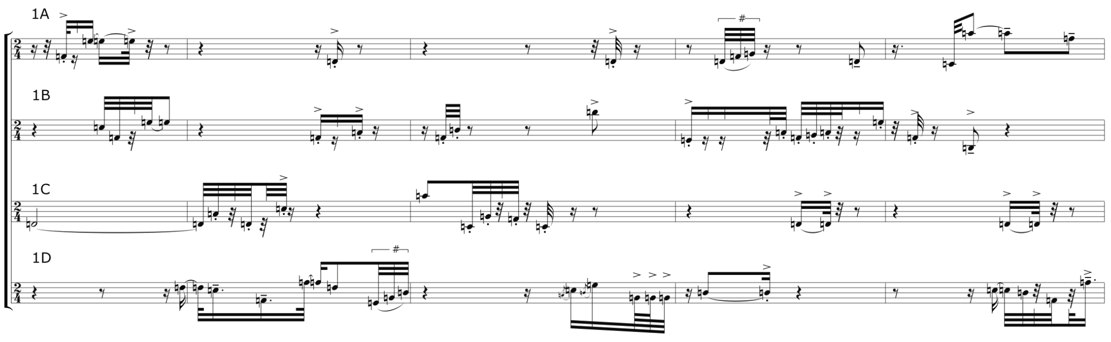


Figure 7: Original *rhythm sequence palette* for the author's *cheat sheet.*

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), as illustrated in Figure 8.

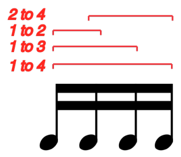


Figure 8: Semiquaver (16th note) loop points within a single crotchet (quarter note).

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 7. 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 9.

```
(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 5 1 5 2 4 5 1 4 5 2 4 5 1 5 2 5 2
4 5 5 6 5 2 6 5 6 1 6 6 2 5 6 2 6 3 6 5 2 6 6 3 7 6 2 7 3
6 3 7 7 6 7 7 3 7 7 8 2 7 7 3 8 7 3 8
…
96 96 45 96 46 95 45 96 96 97 96 97 46 96 97 97 45 96 97
46 97 97 46 97 45 98 97 46 97 98 46 97 98 46 97 46 98 47
98 97 98 97 98 46 97 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 9: 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 7), we can see how this develops in Figure 10. The rhythms and meter were doubled for ease of reading.

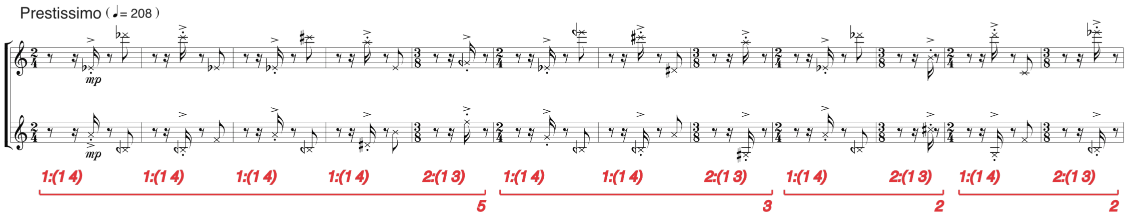


Figure 10: Rhythmic loop slice mapping in the flute and clarinet parts at the beginning of the author's *cheat sheet.*

Figure 11 is an example from later in the piece (bar 226). It shows how the process moves through the original material and what it yields rhythmically. Only the solo guitar and strings shown. At this point in the piece the guitar is at the beginning of bar 2 of 1D in the original *rhythm sequence palette* (Figure 7); the strings are at the beginning of bar 2 of 1C. Note that the pitches are automatically selected by *slippery chicken* according to the criteria discussed above. Particularly interesting here are the repeated six-note microtonal chords in the guitar. These are also algorithmically chosen and the fingerings are added automatically. See below for a discussion of their generation.

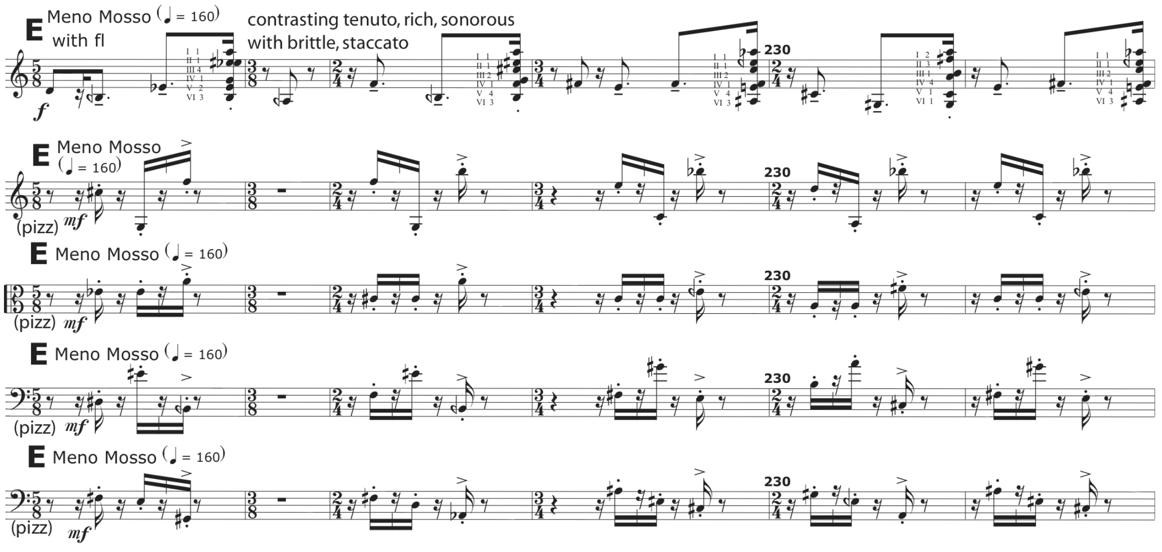


Figure 11: Solo guitar and strings from page 23 of the author's *cheat sheet.*

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 (Edwards 2011a, 64) for further discussion of this issue.

In order to illustrate L-Systems, take a very simple example where a set of rules is defined. These associate a key with two resulting keys, each of which in turn forms a lookup key for an arbitrary number of substitutions (Figure 12).

```
1 → 2 3
2 → 1 3
3 → 2 1
```

Figure 12: Simple L-System rules.

Given a starting seed for the substitution procedure (or rewriting, as it is more generally known), an infinite number of results can be generated. See Figure 13 for an example of four generations of rewriting.

```
                              seed: 2
                                1 3
                             2 3 | 2 1
                     1 3 | 2 1 | 1 3 | 2 3
         2 3 | 2 1 | 1 3 | 2 3 | 2 3 | 2 1 | 1 3 | 2 1
```

Figure 13: Step-by-step generation of results from simple L-System rules and a seed.

One of the properties and attractions of L-Systems is the self-similarity that results. This could, for example, be used to organise recognisable musical motifs.[12] Self-similarity becomes clear when larger result sets are produced; see

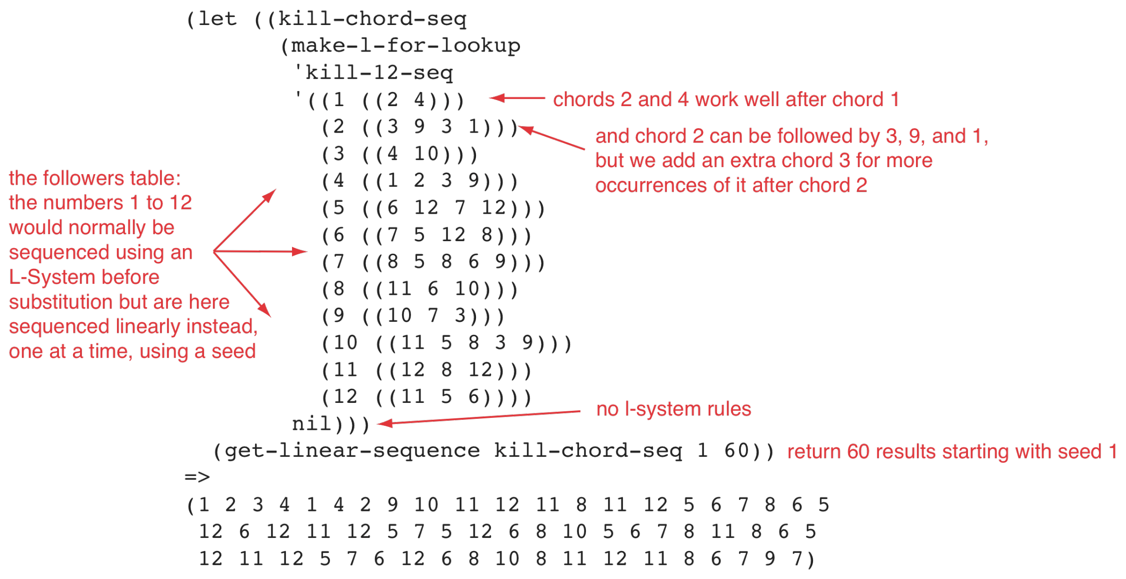
[12] See, for example, (Supper 2001, 50-52) for a discussion of the use of L-Systems in Hanspeter Kyburz's music.

Figure 16 for an illustration of this. The returned 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.

Before considering the *Transitioning L-Systems*, perhaps a simple example of cycling alone would clarify this. The *slippery chicken* implementation can be used for simple cycling sequences by using the substitution data only, i.e. with no L-System. This is not merely by way of illustration as such an approach has been used, for instance, in the generation of harmonic sequences in the author's *I Kill by Proxy*, for piano, percussion, and computer (Edwards 2007b). This hour-long triptych uses only 12 *pitch sets*, but in different transpositions for each of the three linked pieces. When designing these sets, each was tested by ear for potential subsequent sets. In some cases only two seemed to work, in others up to five. Once the subsequent sets for each of the twelve were entered into the followers (or more usually transitioning substitutions) list, a harmonic sequence was generated that continuously varied but strictly followed the rules of progression determined pragmatically, by ear. See Figure 14 for the structure used in *I Kill by*

Proxy, noting that no transitions are yet present, rather, just a seed and its first and subsequent followers.[13]



```
(let ((kill-chord-seq
       (make-l-for-lookup
        'kill-12-seq
        '((1 ((2 4)))
          (2 ((3 9 3 1)))
          (3 ((4 10)))
          (4 ((1 2 3 9)))
          (5 ((6 12 7 12)))
          (6 ((7 5 12 8)))
          (7 ((8 5 8 6 9)))
          (8 ((11 6 10)))
          (9 ((10 7 3)))
          (10 ((11 5 8 3 9)))
          (11 ((12 8 12)))
          (12 ((11 5 6))))
        nil)))
  (get-linear-sequence kill-chord-seq 1 60))
=>
(1 2 3 4 1 4 2 9 10 11 12 11 8 11 12 5 6 7 8 6 5
 12 6 12 11 12 5 7 5 12 6 8 10 5 6 7 8 11 8 6 5
 12 11 12 5 7 6 12 6 8 10 8 11 12 11 8 6 7 9 7)
```

chords 2 and 4 work well after chord 1

and chord 2 can be followed by 3, 9, and 1, but we add an extra chord 3 for more occurrences of it after chord 2

the followers table: the numbers 1 to 12 would normally be sequenced using an L-System before substitution but are here sequenced linearly instead, one at a time, using a seed

no l-system rules

return 60 results starting with seed 1

Figure 14: Simple sequences generated for the harmonic structure of the author's *I Kill by Proxy*.

Returning to *cheat sheet, slippery chicken*'s L-System implementation is used in two further ways: as a simple L-System without transitions, and as a

[13] To clarify, seed 1 returns as a first result 2, which is the first item in its follower list. The next time we encounter 1, it will return 4, the second item in its follower list. The next time again, 1 will return 2, etc. etc.

Transitioning L-System. Their use 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 15, where strings two and six are detuned by a quartertone.

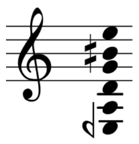
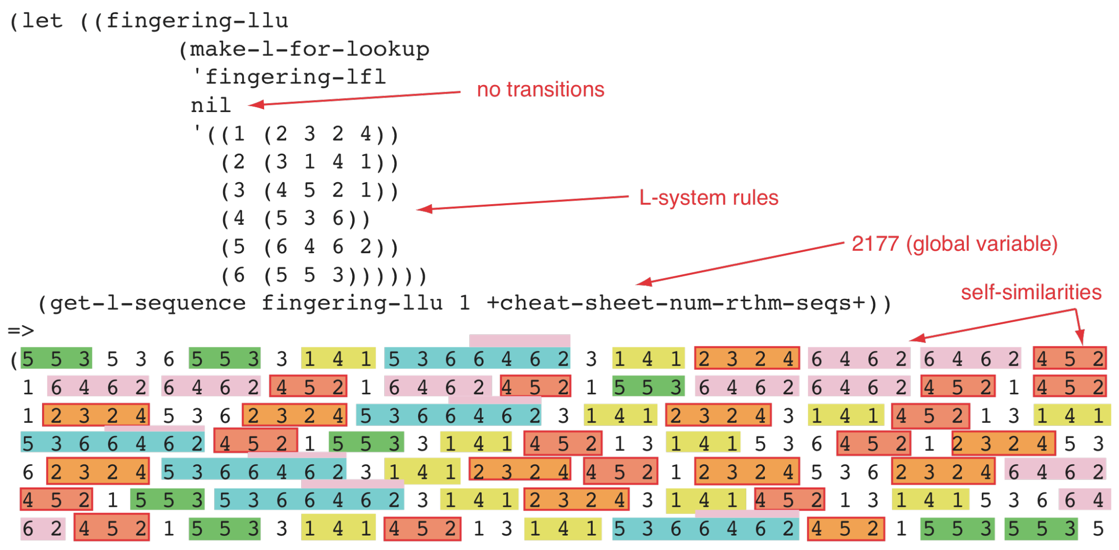


Figure 15: Guitar tuning (scordatura) for the author's *cheat sheet.*

The *pitch sets* were then created by a threefold process:

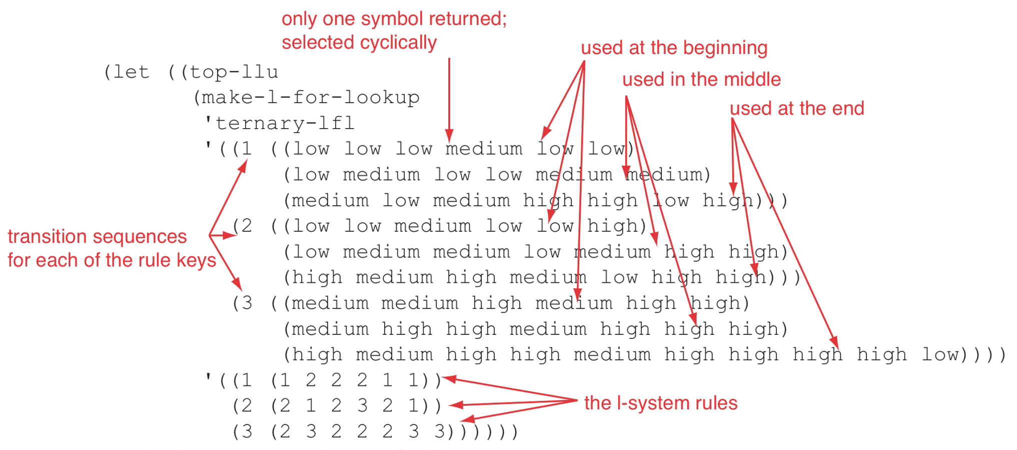
1) 6 guitar chords were chosen by ear. These had fingerings: (4 2 1 3) (4 1 2 3) (4 1 3 2) (3 2 1 4) (3 4 1 2) (4 3 1 2). The sequencing of these was organised by a simple L-sequence: there are no transitions here but self-similar patterns do emerge (Figure 16).



```
(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 16: 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 17).



```
(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 17: *cheat sheet's* guitar chord ternary Transitioning L-System.

Figure 17 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 18.). The guitar generally has 19 frets, so the chords can be created with the first finger placed on frets 1-15.

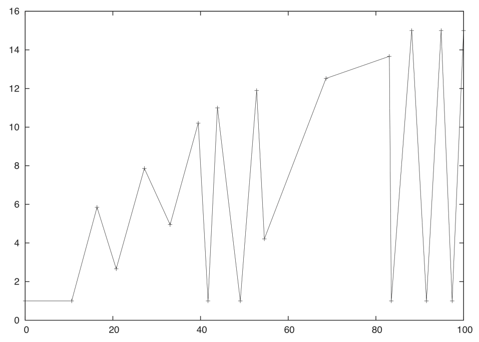


Figure 18: First finger fret selection curve for the author's *cheat sheet*. The x-axis is scaled automatically to fit the 2177 chords.

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 19.

```
(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)
(MEDIUM 1 2) (LOW 1 3) (LOW 1 2) (MEDIUM 1 4)
 …
(HIGH 5 3) (HIGH 5 6) (LOW 6 2) (MEDIUM 6 3) (LOW 6 2) (HIGH 6 4)
(HIGH 7 3) (MEDIUM 7 1) (LOW 7 4) (MEDIUM 7 1) (LOW 8 4) (HIGH 8 5)
(MEDIUM 8 2) (HIGH 8 1) (HIGH 9 3) (HIGH 9 1) (HIGH 9 4) (MEDIUM 9 1)
(MEDIUM 10 5) (HIGH 10 3) (MEDIUM 10 6)(LOW 10 5) (HIGH 11 3)
(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 19: 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.*

Set Manipulation

As discussed, *cheat sheet* proceeds harmonically from guitar fingerings. With the purpose of obtaining extra pitches for the ensemble that lie above and below the guitar's range, the intervallic relationship of the guitar chord is mapped onto its highest and lowest notes. This process has been genericised in the *stack* method: The interval structure of a set is duplicated, interleaving from the top note upwards, and the bottom note downwards. Going downwards proceeds symmetrically, so for example if we have a major triad, with ascending intervals of a major and minor third, this will be reflected downwards with the same intervals in the same order, thus resulting in a minor triad (Figure 20).

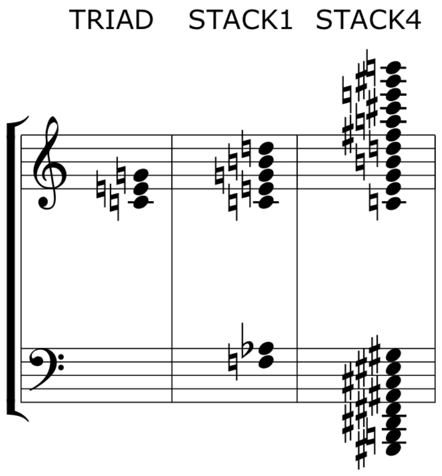


Figure 20: The *stack* algorithm applied once and four times to a C-Major triad.

Due to *cheat sheet's* microtonal guitar scordatura, the *stack* extensions would normally result in a high incidence of quartertones. In practice, the extensions were contracted by a quartertone so as to result in less microtones and thus simpler pitching and fingering for the ensemble players. See Figure 21 for examples of this process using the six guitar fingering patterns discussed.

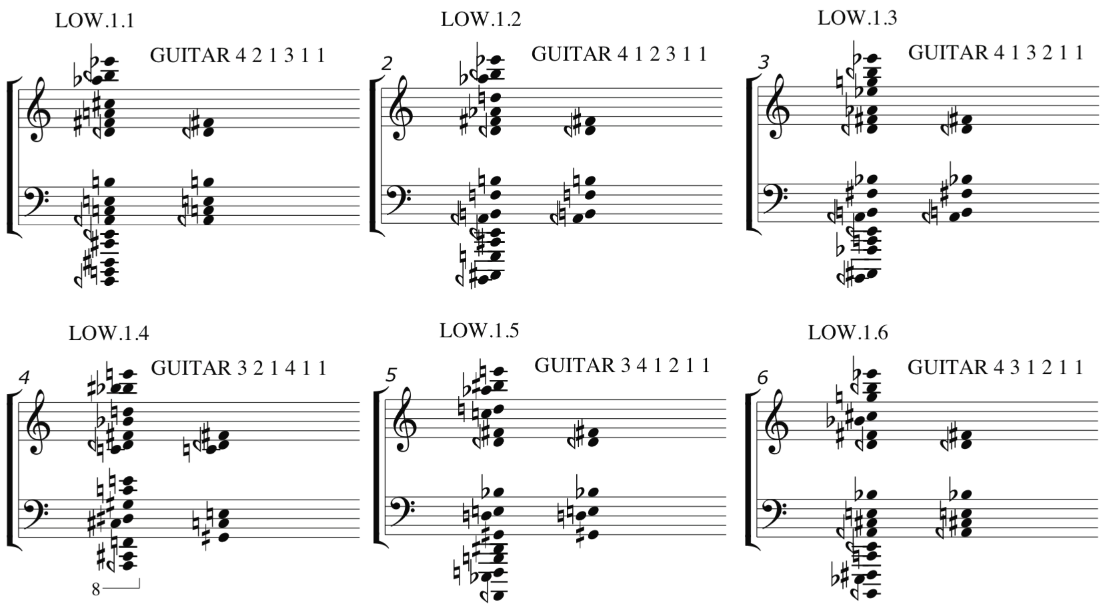


Figure 21: Guitar chords and ensemble extensions for the author's *cheat sheet*.

set limits

The range of these harmonies is quite similar, but it is unlikely that all the notes in them will ever sound simultaneously; rather, they form the *pitch sets* from which notes and chords will be selected. Nevertheless, were this range to remain so similar for each *pitch set* in the piece, then the textural development could become quite stagnant. We have seen that transitions are an important part of the *slippery chicken* concept; this applies as equally to pitch material as to other musical parameters. *set limits* can be applied to a piece in the form of two optional breakpoint functions, one for the designated highest, the other for the lowest pitch. The breakpoint functions will be scaled over the whole duration of the piece, so an arbitrary x-axis range is possible. Thus, despite a current set perhaps occupying a range of five octaves from a low bass note, the piece's current lower limit as defined through the interpolating breakpoint function might restrict this to notes above middle C. The limiting process is essentially the same as that applied when choosing notes for each instrument: the set in that case is limited, temporarily, to the instrument's range. Furthermore, *set limits* can be applied on a global as well as on a player-by-player basis. It is thereby possible to set an instrument's tessitura to be high, low, middling, or full range in different parts of the piece, independently of the other instruments and the range of the pitches available from the current set.

Rhythm Chains

Another *slippery chicken* innovation, *rhythm chains* were first used in the author's *altogether disproportionate*, for piano and computer (Edwards 2010). They were used more elaborately in *you are coming into us who cannot withstand you* (Edwards 2011c). This algorithmic work is for eight-piece ensemble only, i.e. no computer or electronics are used during the performance.

Deceptively simple on the page, *you are coming into us who cannot withstand you* gains its impetus from the combination of small, simple rhythmic units that form larger, sometimes repeating sequences. These are placed in potentially polymetric opposition to similarly constructed, contrapuntally combined sequences. As used in *you are coming*, the tempi are quick, the energy level is high, and the perception of multiple pattern streams moving at different rates is the main intended feature of the music.

In *rhythm chains*, there are an arbitrary number of one beat rhythmic units of arbitrary complexity. An example of one such unit is simply a quaver (eighth note) rest followed by a quaver note; another is even simpler: a single crotchet (quarter note) note. These rhythmic units are then strung together to form the chain, which internal to *slippery chicken* means a series of *rhythm sequences* and an automatically generated map. The order in which the units are algorithmically combined into *rhythm sequences* is determined either by a *Fibonacci Transition* or by the *procession* algorithm. The latter 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. A piece using *rhythm chains* may make use of both *Fibonacci Transitions* and *procession* for organising the rhythmic units.

However many such rhythmic units there are, there must be a matching number of partner units towards which a transition is made over the course of the whole piece. The transition between the two unit collections (or more if desired) is handled by the *Fibonacci Transition* algorithm. In *you are coming*, the transition is from exactly such preponderantly simple binary rhythms as described above, to preponderantly ternary rhythms, i.e., those characterised by triplets (Figure 22).

The simple one-beat units are intended to form faster-moving parts; these are counterpointed by usually slower moving units of two or three beats. The technique is therefore essentially two-part contrapuntal, though this can be scaled up to any number of parts—in *you are coming*, eight. Again, we use *Fibonacci Transitions* or the *procession* algorithm in the generation of the 2/3-beat rhythm order, i.e., the order in which each 2/3-beat unit is used, not the order in which we select either a 2-beat or a 3-beat unit.[14]

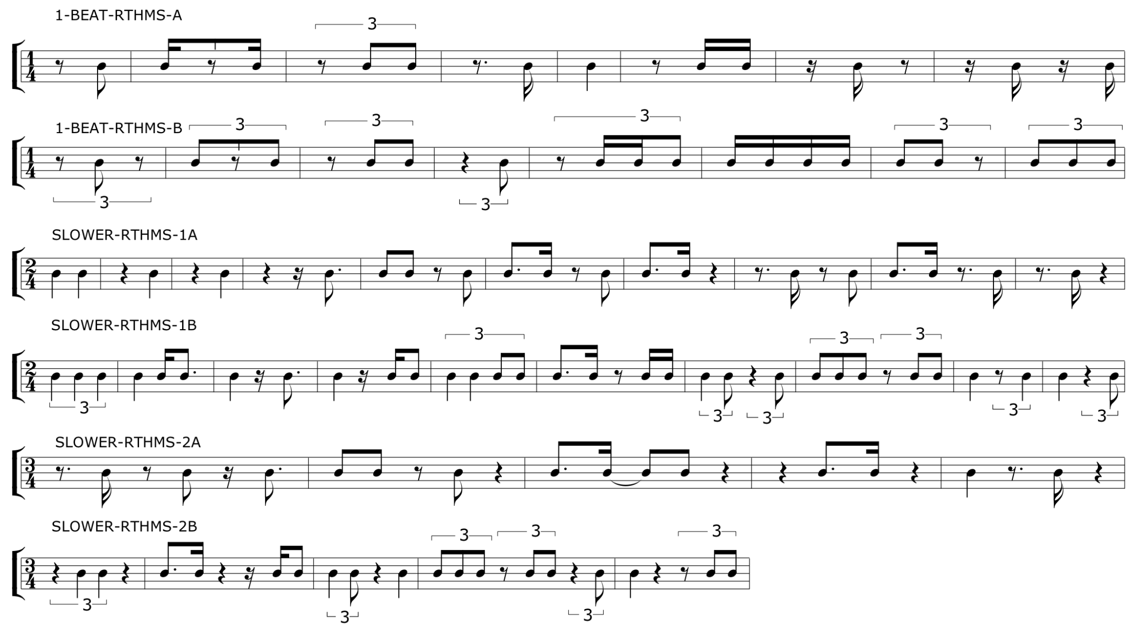


Figure 22: The rhythmic cells used for *rhythm chains* in the author's *you are coming into us who cannot withstand you.*

To clarify Figure 22, the chaining of the faster-moving units in 1-BEAT-RTHMS-A (mainly simple binary rhythms) transitions over the course of the piece to be replaced by those in 1-BEAT-RTHMS-B (mainly ternary rhythms). Similarly, with the slower-moving 2-beat rhythm chaining, we transition from SLOWER-RTHMS-1A to SLOWER-RTHMS-1B when the algorithm calls for a 2-

[14] This is decided more simply cycling through the list 2 3 2 2 3 2 2 3 3 3.

beat unit, and from SLOWER-RTHMS-2A to SLOWER-RTHMS-2B when it calls for a 3-beat unit.

The grouping of an arbitrary number of slower moving units into shorter or longer *rhythm sequences* (with, as always, one *pitch set* per *rhythm sequence*) is determined by a user-defined harmonic rhythm: this may change during the piece and is controlled by a breakpoint function. The repetitions of both units and sequences, with or without the repetition of their associated pitches, creates identifiably recurring material which, through the additional insertion of rests and new contrapuntal combinations, constantly varies an essentially consistent, flowing musical structure.

Variation of density and activity is provided by the algorithmic control of rests in two forms: through the interaction of multiple cycles of rests, and user-defined *activity levels*. Together these determine the relative mix of rests to notes: this can vary from very sparse to unbroken. Both approaches are controlled by the use of breakpoint functions which map over the whole generated structure.

Figure 23 shows the process in action at the opening of the flute and clarinet parts of *you are coming*. In this case, the flute is the slower-moving part.

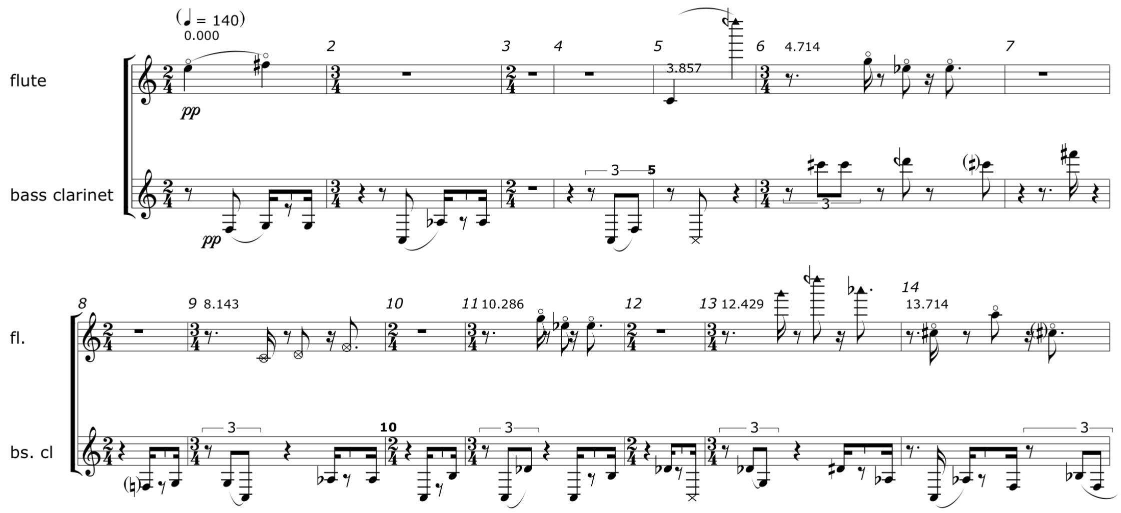


Figure 23: Flute and clarinet parts at the opening of the author's *you are coming into us who cannot withstand you.*

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 creating 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.

More generally speaking, *slippery chicken* is a bridging technology. As traditional composition training does not usually include algorithmic or computer music techniques, but assuming that their incorporation into compositional practice is inevitable, it is essential—as with any new technological approach—that bridging mechanisms are found.[15] In industry, a leap from one technology to another presents a business risk, even if the main task is shared, as with the shift from the typewriter to the word processor. It is essential to proceed with caution, if the customer base is to be retained. Thus we have transition solutions, often with computer software and even hardware that have a rather 'analogue feel' to them. Most word processors, for instance, have a virtual sheet of paper and virtual tab stops; and we type with a keyboard that is not by accident related to the typewriter. However, when extended over decades and in different contexts this approach can become a hindrance. A significant part of the music software industry still offers interfaces and concepts developed from, and in some cases ever more reminiscent of, analogue studio equipment: virtual tracks, patch cords, synthesizer keyboards, knobs, even power buttons. Nostalgia marketing would seem to be more the motivation behind the interface

[15] The author learned this, perhaps surprisingly but most conspicuously, whilst working as a software engineer, on a project that had nothing to do with music, but rather business document recognition.

here than the oft-heard argument that analogue processing is superior to digital, ergo an analogue emulation is preferable in the absence of such hardware. Irrespective of the veracity of that argument, the utility of such music software interfaces and their resultant workflow is often questionable: sometimes they are quite simply cumbersome. For music composition, as opposed to sound processing/sequencing/mixing, such interfaces are most often inappropriate. Systems based around or including programming interfaces (such as *slippery chicken* or the *Scheme* scripting interface to *Common Music 3*) offer the potential to move further along the technology bridge.

Perhaps the developments arising from the debate surrounding the relative merits of analogue versus digital studio technology has most convincingly shown that combining the old with the new is the best solution. This applies just as well to music composition and performance. Formats that continue to include rather than bypass the talented and highly-trained acoustic musicians which our musical infrastructure has at its disposal arguably yield the most impact, particularly when viewed, for better or for worse, from the audience's perspective. To this end, hybrid works combining digital and acoustic instrumental technologies are ideal. *slippery chicken* is focussed on exactly such musical bridging solutions: using the computer to combine and meld together electronic and acoustic resources at both the structural and formal level.

Acknowledgements

Thanks to Johannes Goebel and the Center for Art and Media Technology (ZKM), Karlsruhe, Germany, for Guest Artist stipends in 2000 and 2001, during which the bulk of the design and programming was completed for the main algorithmic system of *slippery chicken*.

Thanks also to Rick Taube, author of Common Music, for allowing the inclusion of CM 2.6.0 in the *slippery chicken* release and subsystem. The author's debt to Common Music is considerable in that his introduction to algorithmic composition was through CM and therefore its approach made an indelible stamp on his algorithmic music thinking, especially in its approach to output-independent processes. As no doubt some users will be interested in using CM in conjunction with *slippery chicken*, the latter includes the whole of CM 2.6.0 as part of its installation procedure. It also uses a few of CM’s routines for pitch/MIDI/frequency conversion and MIDI file writing.

Bill Schottstaedt must be thanked for CLM and CMN. Though these are not included in the *slippery chicken* installation, its use in combination with them is all the richer for it.

It goes without saying that *slippery chicken* is intended to complement, not duplicate or replace CM, CLM, or CMN. Any deficiencies of *slippery chicken* are the result of the author’s failures, not those of Bill or Rick.

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

References

Alsop, R. 1999. “Exploring the self through algorithmic composition.” *Leonardo Music Journal*, 9 (1), 89 – 94.

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

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

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).

Cope, D. 1996. “Experiments in Musical Intelligence.” Madison, WI: A-R Editions.

Edwards, M. 2000. “slippery when wet.” (Musical composition.) http://www.sumtone.com/work.php?workid=5.

Edwards, M. 2004. “breathing Charlie”. (Musical composition.) http://www.sumtone.com/work.php?workid=132.

Edwards, M. 2005. “in limine”. (Musical composition.) http://www.sumtone.com/work.php?workid=131.

Edwards, M. 2007a. “cheat sheet”. (Musical composition.) http://www.sumtone.com/work.php?workid=182.

Edwards, M. 2007b. “I Kill by Proxy”. (Musical composition.) http://www.sumtone.com/work.php?workid=178.

Edwards, M. 2009. “who says this, saying it’s me?” (Musical composition.) http://www.sumtone.com/work.php?workid=261.

Edwards, M. 2010. “altogether disproportionate”. (Musical composition.) http://www.sumtone.com/work.php?workid=275.

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

Edwards, M. (2011b). don’t flinch. (Musical composition.) http://www.sumtone.com/work.php?workid=304.

Edwards, M. 2011c. “you are coming into us who cannot withstand you” (Musical composition.) http://www.sumtone.com/work.php?workid=307.

Essl, K. 2007. *The Cambridge Companion to Electronic Music,* 107–125. Cambridge: Cambridge University Press.

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

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

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

Monro, G. 1997. “This is art, not science.” *Leonardo Music Journal,* 7 (1), 77.

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

Nierhaus, G. 2009. *Algorithmic Composition*. New York: Springer-Verlag.

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

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

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, Massachusetts: MIT Press.

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

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

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

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