

A Csound Algorithmic Composition Tutorial

Michael Gogins

11 August 2009

Introduction

This tutorial will teach you, step by step, how to do algorithmic composition using Csound and the CsoundAC Python extension module for algorithmic composition.

You should be prepared with some basic experience in music composition, computer programming (even if it is only writing a few scripts or spreadsheets), and software synthesis (preferably using Csound).

But above all, you need to have a strong curiosity about algorithmic composition (also known as generative composition or score generation) from a *musical* – not technical – point of view.

This section briefly introduces basic concepts and definitions, provides some references to external resources for algorithmic composition, and presents a series of historically based, tutorial compositions demonstrating basic techniques of composing with algorithms. These pieces are:

- The *Musikalisches Wuerfelspiel* (musical dice game) of 1787 and variations upon it (Chuang, 2007).
- Automatic generation of counterpoints to randomly generated *cantus firmi*, based on the work of Lejaren Hiller (Hiller & Isaacson, 1959) and Bill Schottstaedt (Schottstaedt, 1984).
- Transpositions of star positions and spectral data onto scores, based on John Cage's *Atlas Eclipticalis* (Cage, *Atlas Eclipticalis*, 1961).
- Sustained drones with beating combination and difference tones, inspired by the work of LaMonte Young (Young, 2000).
- Fractal music, inspired by the work of Charles Dodge (Dodge, *Viola Elegy*, 1987).
- Minimalist music, based on the principles of Terry Riley's *In C* (Riley, 2007).

All of these tutorial compositions use Python with the CsoundAC module for both composition and synthesis. All required Csound software, SoundFonts, and instrument definitions are included. All soundfiles are rendered in stereo with floating-point samples at 44,100 frames per second and 1 control sample per audio sample frame, for high-resolution sound and high-precision performance.

The text provides detailed instructions that should enable you to create each piece by writing code from scratch, but pre-written and tested versions of each piece can also be found in the `MusicProjects` directory.

It is assumed that you are working on Windows XP, but similar steps would obtain for Linux or OS X, except that at the time of writing, on OS X you would first need to build CsoundAC on your computer from Csound CVS sources.

What is Algorithmic Composition?

Let us begin at the beginning. What is an *algorithm*? What is *composition*?

The technical definition of *algorithm* is just: *A definite procedure.*

Definite simply means there are no ambiguous steps. Someone who does not understand the field in the least can still perform the algorithm correctly, by following directions one step at a time, and that is why computers can automate the performance of algorithms. *Definite* is not at all the same thing as *deterministic*. Flipping a coin produces a perfectly random sequence – but it is still a perfectly definite procedure.

Examples of algorithms include cooking recipes, MapQuest directions, the steps children are taught for doing long division or taking square roots, and all computer programs.

Philosophers and mathematicians have *proved* that general-purpose computers, such as your own personal computer, are *universal* in the sense that they can perform any algorithm that any other particular computer can perform (as long as they have enough time and enough memory), if necessary by running a program that simulates the other sort of machine. Another term for *universal* is *Turing-complete*. There probably are many ways of making a computer (or a computer language, for that matter) Turing-complete; but in general, if a computer or language implements all of a rather small set of basic instructions, it is Turing-complete. These instructions include reading or writing a value to an arbitrary memory address, interpreting such a value as an instruction and

carrying it out, performing the logical operations AND, OR, and NOT on such values, and performing the instruction at one address if a logical expression is true, but at another address if it is false (branching).

Philosophers and mathematicians also go further, and *assume* – and to date there is absolutely no evidence to the contrary! – that anything at all that *can* be computed, can be computed by a universal computer. The technical term for this assumption is *the Church-Turing thesis* (Kleene, 1967). There is no known counterexample to the thesis, and it is widely accepted, but it has never been proved. The question of its truth has been a central question of philosophy since the 1930s. Proof or disproof would revolutionize our understanding of logic, mathematics, computer science, and even physics.

For a brilliant exposition of these things that cuts to the heart of the matter – there are so infinitely many truths, that almost none of them can actually be proved -- see (Chaitin, 1998).

What is *composition*?

Composition is the combination of distinct elements or parts to form a unified whole; in this context, the art or act of composing a work of music.

Algorithmic composition, then, is the art or act of combining distinct elements or parts to form a single unified work of music – using an algorithm.

Commonly, there is an implication that the algorithm is run on a computer, which enables the composition of music that would be tedious or even impossible to compose by hand. But running on a computer is not necessary to the definition. The first algorithmic compositions far predate computers and so were of course composed by hand, and algorithmic compositions are *still* being composed by hand. An outstanding recent example of an algorithmically composed piece produced completely by hand is Leonid Hrabovsky's *Concerto Misterioso* (1993). It is of course also possible to generate scores by means of a computer, but then print them out in music notation and play them, as Bruce Jacob (2009) has done. But for the sake of speed and self-sufficiency, the approach taken in this book is both to generate, and to render, scores using a computer.

Philosophers and mathematicians have also *proved* that computers can't write their own programs. Or rather, the proof goes: If computers *do* write programs,

they *can't* always debug them. The technical term for this is the *halting theorem* (Turing, 1936).

Actually it *is* possible for computers to write algorithms using *evolutionary programming*, but so far at any rate, the problem domains suited to evolutionary programming are rather small – extremely small, compared to the vast possibilities of musical composition. In music, the fitness function would have to be either a good algorithmic model of educated musical taste, a daunting problem to say the least; or an actual panel of educated and discriminating human listeners, perhaps an equally daunting prospect. (For an introduction to the field see (Banzhaf, Nordin, Keller, & Francone, 1998); for an eye-opening exploration of the creative possibilities of evolutionary programming, including musical applications, see (Bentley & Corne, 2002); for an open-source Java program that evolves graphical designs using various algorithms, see (Jourdan, 2004); for an introduction to genetic programming for musical composition, see (Miranda & Biles, 2007).)

But the decisive point is that even the genetic algorithm still requires a programmer to set up the algorithm and define a fitness function before it begins to evolve solutions – so the setup as a whole is still a program that still needs to be debugged, and that cannot always debug itself.

In short: Computers *execute* algorithms, programmers *write* them. Therefore computers don't, and can't, do algorithmic composition. Only composers who program do.

Now the interesting questions begin.

First, why bother? Composers already hear music in their heads, and write it down. No need for computers or algorithms there.

But even the greatest composers have limited imaginations. Perhaps they can imagine a work of music whole and write it down from memory, but that still takes time, and even for the greatest composers, changes in style come slowly. Perhaps the imagination depends upon unconscious templates that constrain the possible elements and their combinations. Certainly the imagination can hold only a limited number of elements in mind at the same time. Perhaps algorithms could run much faster, or generate new templates and new styles as easily as they generate pieces, or hold more elements together at the same time.

In other words, algorithms might offer, at least in principle, a means of transcending some of the limitations of the human mind.

So, what *kind* of music can composers with computers compose, that composers without computers can't?

Many thinkers have observed that music lends itself to mathematical analysis, beginning with the definition of musical intervals as Pythagorean ratios. For centuries, the rules of voice leading and harmony, and especially canonical forms such as rounds and fugues, have been stated in the form of algorithms. So music seems by its very nature to be suited to an algorithmic approach.

The differences between the human imagination and computer algorithms further suggest that algorithms can be used to make works with more notes than composers can keep track of; works with polymeters like 173/171; works where each level of time has to perfectly mirror every other level of time; works that specify a tiny grain of sound for each star in the sky; works where rhythmic canons have to end together at precisely the same moment after hours of cycling; scores based on mathematical evolutions that composers can't handle; and who knows what...

Finally, the most important question of all: *What is the best way to actually do algorithmic composition?*

For the *art* in algorithmic composition, of course, is writing only those programs that will generate only that wonderful music, while leaving out all boring, mediocre music and, of course, all plain old noise.

The possibilities include evolutionary computing, the use of parametric algorithms controlled by only a few numbers that can rapidly be varied and even mapped for interactive exploration, the reduction of music to simpler forms either by analogy with Schenkerian analysis or in some mathematical form that can be concisely controlled, and no doubt other possibilities that I haven't thought of.

In all cases, the fact that the main advantage of the computer is sheer speed lends great weight to every simplification and speedup of the composing process itself. In my experience, good results in algorithmic composition come from generating a great many pieces, and a great many variations on them, and throwing almost everything away. So the algorithmic composer should learn to

program and compose quickly, and develop an efficient, flexible working environment.

Facilities for Algorithmic Composition

Christopher Ariza maintains an excellent on-line list of references to facilities for algorithmic composition (Ariza, 2007). There is no need for me to reproduce all of his data here. However, I will mention some of the more interesting and important *open source* or *free software* systems below (in alphabetical order, except for Csound).

You should be aware that there also exists a surprising amount of commercial software for algorithmic composition, ranging from what are basically toys to quite serious and capable systems. Ariza's site mentions some of them as well. Others can be found in music industry magazine articles and advertisements, or at the Shareware Music Machine web site (Hitsquad, 2007).

It soon becomes apparent that there is an embarrassment of riches, and that many algorithms are duplicated over and over again in different systems. Furthermore, it is too often the case that some genius wrote a program decades ago like Schottstaedt's species counterpoint generator (Schottstaedt, 1984) in some weird language like SAIL that nobody can run anymore, so their work is effectively lost. I find this situation *quite* frustrating, and would prefer a single standard framework for which different musicians and developers could contribute plugins. However, that would require widespread agreement on a standard protocol for such plugins, not to mention a standard representation for musical event data (MIDI, while a breakthrough and a useful standard, is not sufficiently extensive or precise to represent all styles of music). Don't hold your breath, but that is a future worth working for....

athenaCL

Christopher Ariza's own athenaCL (Ariza, athenaCL, 2007) is a pure Python algorithmic composition environment. It can be used as a command shell, or as a Python extension module. It includes a variety of algorithms from other sources, notably OMDE/PMask (Puxeddu, 2004) which uses probabilistic "tendency masks" to create clouds and sequences of sound grains and other events. athenaCL also contains many ideas and algorithms from "musical set theory," son of serialism. athenaCL can use Csound for making sound.

blue

Steven Yi has written *blue* (Yi, 2007) as a Java-based composition environment that also incorporates Python via the Jython module, which implements the Python language using the Java virtual machine and thus enables the complete inter-operability of Jython scripts and compiled Java classes. *blue* contains a novel approach to computer composition in that it is very concerned with the organization of time along multiple independent lines, and attempts to use more traditional notions of orchestration and performance along with other concepts. *blue* uses Csound for making sound.

Common Music

Common Music (Taube, 2007) is a LISP system for algorithmic composition that is designed to produce MIDI sequences or Csound score files. It is one of the deepest algorithmic composition systems and has been widely used. The author of Common Music, composer Rick Taube, has written a book on algorithmic composition (Taube H. K., 2004), in which all concepts are illustrated with examples in Common Music.

JMusic

jMusic by Andrew Sorensen and Andrew Brown (Sorensen & Brown, 2007) is a pure Java (and therefore cross-platform) framework for computer-assisted composition in Java, and also can be used for generative music, instrument building, interactive performance, and music analysis.

OpenMusic

OpenMusic (Agon, Assayag, & Bresson, 2007) is an object-oriented visual programming environment based on CommonLisp/CLOS. OpenMusic provides libraries and editors that make it a powerful environment for music composition on Unix, Linux, and the Macintosh.

SuperCollider

SuperCollider by James McCartney (McCartney, 2007) (SuperCollider swiki, 2007) is a computer music system that is oriented towards live performance, but it has many compositional algorithms and, since it contains a complete object-oriented computer language with first-class functions and closures, is well suited to algorithmic composition. Originally SuperCollider ran only on the Macintosh, but it has been ported to Linux and now, partly to Windows as well.

SuperCollider actually consists of two parts, `sclang` the composition language, and `scsynth` the synthesis server that makes the sound.

Csound

Although Csound (Vercoe, 2007) was designed primarily as a user-programmable software synthesizer and sound processing language, it has over the years gained sufficient facilities to be suitable for algorithmic composition.

The Score Language

Even the Csound score language contains some limited facilities for algorithmic composition:

1. The carry feature can be used to increment pfields from one `i` statement to the next. The related `ramp`, `next pfield`, and `previous pfield` operators also can be used.
2. Score statements can evaluate arithmetic expressions.
3. The `m` (mark) statement can be used to symbolically identify a section of a score.
4. The `n` and `r` (repeat) statements can be used to repeat symbolically identified sections of scores.
5. The score macro facility can be used for various purposes. It includes the `#define`, `#define(...)`, `#undef`, and `$` operators, which have essentially the same meaning as they do in C.
6. The score `#include` statement can be used to construct a single score from multiple files. Of course, the same file can be included more than once.

Although these facilities are powerful, Csound's score language is *not* Turing-complete, because it does not include an instruction for conditional branching.

The Orchestra Language

The Csound orchestra language, however, *does* include the `if` and `goto` statements for conditional branching; therefore, the orchestra language *is* Turing-complete.

The orchestra language includes opcodes for generating and scheduling score events (`event`, `scoreline`, `schedule`, `schedwhen`, `schedkwhen`). The orchestra language even has user-defined opcodes and subinstruments that can

be used as subroutines. Function tables can be used as arrays. The language has its own `#include` statement and macro facility.

It follows that, in principle, the Csound orchestra language can be used to generate any conceivable score. Unfortunately, most users find the Csound language to be syntactically crude. It lacks blocks, lexical scoping, for loops, classes, lambdas, and currying, to mention only some features of higher-level languages that drastically increase programmer efficiency. It is not *elegant* in the same sense that C, LISP, or Python are elegant, and it is not *powerful* in the same sense that C++, OCaml, or Python are powerful.

CsoundAC as an Environment for Algorithmic Composition

CsoundAC (Gogins, 2007) is a Python extension module for Csound that is specifically designed to support algorithmic composition. CsoundAC is included in the Csound source code CVS archive, and is available ready to run in the Windows installers for Csound.

As I have already indicated, Csound itself already provides some facilities for algorithmic composition in its score language and, especially, its orchestra language. CsoundAC goes much further, to provide not only a Python interface to the Csound API, but also an extensive library of classes for generating and transforming scores. This makes it very easy to write compositions in Python, and to render the generated scores with Csound. You can embed Csound orchestras right in your Python scripts, so all the elements of a composition can be contained in a single file.

Python (Python Software Foundation, 2007) is an interpreted, dynamically typed, object-oriented language, and it is definitely Turing-complete. Python is in the Algol branch of the computer language family tree, with some features borrowed from LISP. Most programmers who know several languages find that Python is very “natural” and that they can write programs in Python quickly and with few errors.

Because it is interpreted, Python does not execute quickly; it runs (very roughly) about 1/30 as fast as C. However, on a contemporary computer, that is more than fast enough for many purposes – certainly including score generation.

Yet this is just the beginning. Not only is Python an easy language to learn and to use, and not only is it a powerful language in its own right, but also Python is very easy to extend by writing modules in C, C++, or other languages.

If you are not familiar with Python, stop right here. Go to www.python.org, find the *Python Tutorial*, and work through it.

Taking advantage of the extensibility of Python, CsoundAC includes a number of additional Python modules for algorithmic composition and synthesis: A class library for generating music graphs (Gogins, 1998); a number of score generating nodes for music graphs, including translating images to scores, chaotic dynamical systems, Lindenmayer systems, and iterated function systems; and novel facilities for working with chords, voice-leading, and chord voicings, based on geometric music theory (Gogins, *Score Generation in Voice-Leading and Chord Spaces*, 2006) (Tymoczko, 2006). CsoundAC's music graphs will be used as the high-level structure for most of the algorithmic compositions in this section, and are explained briefly in the next section.

Because the focus of this section is on algorithmic composition, we will use the same Csound orchestra for most of the compositions (`MusicProjects/CsoundAC.csd`). It is, however, a large orchestra with a number of rather interesting instruments, many of them classic designs from the history of computer music. You can make any number of arrangements of instruments from this large orchestra by re-assigning instrument numbers in the orchestra to the instrument numbers in the score.

Music Graphs

CsoundAC uses my concept of *music graphs* (Gogins, 1998) to create the high-level structure of compositions.

Music graphs are to musical scores, as scene graphs are to 3-dimensional scenes. Scene graphs are very widely used in computer graphics, and form the mathematical basis for 3-dimensional modeling, computer game visuals, computer animations, and so on. Basically, a scene graph draws from a vocabulary of primitive objects, such as spheres, boxes, and cones. Each primitive object is contained in a *node* that is associated with a local transformation of coordinate system. A node can contain not only a primitive object, but also a collection of any number of other nodes. Thus, a scene graph consists of a *directed acyclic graph* (also known as a *tree*) of nodes.

When a scene graph is rendered, the renderer *traverses* the graph depth-first, and at each node, the local transformation of coordinate system is multiplied by the coordinate system inherited from its parent node, to produce a new coordinate system. Or in other words, as the renderer traverses the scene graph,

it uses the cumulative effect of all the transformations on each branch along the tree to move the primitive objects at the leaf nodes around in space until they fit together to compose the scene.

Mathematically, a transformation of coordinate system is just a matrix – usually, a homogeneous affine transformation matrix. Such a matrix can translate (move), and/or scale (stretch or shrink), and/or rotate a set of points (which forms another matrix) on all dimensions at once with a single matrix multiplication.

For example, a simple scene graph might consist of three spheres and a cone. The root node might contain a sphere that expands to form a head, as well as three child nodes: two nodes with smaller spheres that are shrunk and placed on the upper front surface of the head to form eyes, and another node with a cone that is shrunk and rotated to point forward and placed in the middle of the head to form a nose. Once all the transformations are completed, the renderer shades the scene and projects it onto a viewport to form a 2-dimensional rendering of the 3-dimensional scene.

An important concept in scene graphs is *reference*. A node can be named, and then used over and over again in different places in the graph. For example, a node named “tree” might be used over and over again at different locations in the scene to cover a hillside with a wood.

For a complete technical introduction to scene graphs, see (Foley, van Dam, Feiner, & Hughes, 1997). For an open source program with a complete implementation of scene graphs for modeling 3-dimensional images, see POV-Ray (Persistence of Vision Pty. Ltd., 2007). In fact, you can think of CsoundAC as being a sort of POV-Ray for scores.

In music graphs, there is only one primitive object, the *event*, which is normally just a musical note. It is located not in 3-dimensional visual space, but in a 12-dimensional *music space*. The dimensions are:

1. TIME, starting time of the event in seconds.
2. DURATION, duration of the event in seconds.
3. STATUS, corresponding to the high-order nybble of the MIDI status byte; 144 for a “note on” event.
4. INSTRUMENT, corresponding to MIDI channel, from 0 on up.

5. KEY, pitch in MIDI key number, middle C = 60.
6. VELOCITY, loudness in MIDI velocity, 80 = forte.
7. PHASE, in radians (allows events to represent coefficients of time/frequency transforms).
8. PAN, -1 at left through 0 at center stage to +1 at right.
9. DEPTH, -1 at rear through 0 at center stage to +1 at front.
10. HEIGHT, -1 at bottom through 0 at center stage to +1 at top.
11. PITCHES, pitch-class set (i.e. note, interval, chord, or scale) to which this event might be conformed, expressed as the sum of pitch-classes, which in turn are expressed as C = 2 to the 0th power or 1, C#/Db = 2 to the 1st power or 2, E = 2 to 2nd power or 4, and so on.
12. HOMOGENEITY, 1 to make the event a homogeneous vector.

Nodes in music space can contain not only notes and other nodes, but also scores, score generators, and various transforms or filters that act upon the notes produced by child nodes.

For example, a music graph, at the lowest level, might use an *A* phrase in a score node and a *B* phrase in a score node. At the next higher level, a node named "tune" starts with *A*, follows it with *A* again, follows that with *B* transposed up a perfect fourth, and ends with *A* again. At the highest level, a node named "canon" contains one copy of "tune" followed by another copy of "tune" two beats later.

More details of music graphs will be introduced as required to build the compositions in this tutorial.

It should be noted that music graphs, as such, know nothing of traditional music theory – quite in contrast to most composition software, which usually tries to follow or model the rules of counterpoint, harmony, or serialism. This is deliberate. My aim is to provide a general framework for organizing musical events that is deaf to all rules of music theory. Such rules would predetermine the styles possible to realize with the software.

Do not be mistaken: I am very far from indifferent to counterpoint, harmony, serialism, or styles – but I think it is better if these choices are made on a higher level of abstraction, as additional nodes of the system. For example, in the

following I will introduce a Counterpoint node based on Bill Schottstaedt's work (Schottstaedt, 1984) for generating counterpoint over a *cantus firmus*.

Getting Started with Csound in Python

This section will get you started using Csound in Python, and provide a basic pattern for generating scores and rendering them. First, we will embed a piece into Python and render it. Next, we will generate a score using Python and render it using the embedded orchestra.

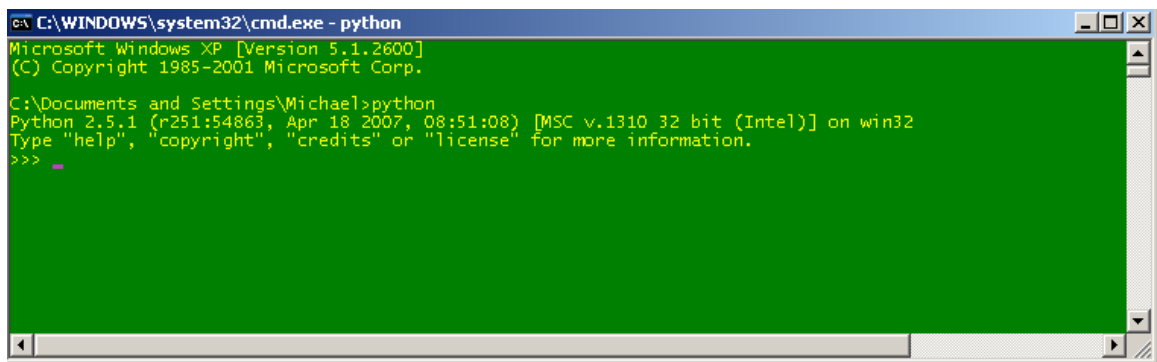
Embedding Csound in Python

The Python interface to Csound consists of two Python extension modules:

1. `csnd` provides an object-oriented Python interface to the Csound API, together with facilities for loading, saving, and manipulating Csound orchestra and score files.
2. `CsoundAC` provides a Python interface to the music graph classes. Basically, `CsoundAC` generates scores, and then uses `csnd` to render them. `CsoundAC` also has convenience functions for all commonly used methods of `csnd`.

If you have not already done so, install Python 2.6 (or whichever version is current for Csound) from (Python Software Foundation, 2007) and Csound and CsoundAC from (Vercoe, 2007).

Run Python from the command line. You should see something like this:

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe - python". The window has a green background and shows the following text:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Michael>python
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Now, attempt to `import CsoundAC`. You should see something like this:

```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Michael>python
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import CsoundAC
Localisation of messages is disabled, using default language.
>>>
>>> dir(CsoundAC)
```

Verify that CsoundAC has been imported correctly by entering `dir(CsoundAC)`:

```
C:\WINDOWS\system32\cmd.exe - python
C:\Documents and Settings\Michael>python
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import CsoundAC
Localisation of messages is disabled, using default language.
>>> dir(CsoundAC)
[Cell, 'Cell_swigregister', 'ChordVector', 'ChordVector_swigregister', 'Chunk', 'Chunk_swigregister', 'Composition', 'Composition_swigregister', 'Conversions', 'Conversions_amplitudeToDecibels', 'Conversions_amplitudeToGain', 'Conversions_amplitudeToMidi', 'Conversions_boolToString', 'Conversions_decibelsToAmplitude', 'Conversions_decibelsToMidi', 'Conversions_doubleToString', 'Conversions_dupstr', 'Conversions_findClosestPitchClass', 'Conversions_gainToAmplitude', 'Conversions_gainToDb', 'Conversions_get2PI', 'Conversions_getMaximumAmplitude', 'Conversions_getMaximumDynamicRange', 'Conversions_getMiddleChz', 'Conversions_getNORM7', 'Conversions_getPI', 'Conversions_getSampleSize', 'Conversions_hzToMidi', 'Conversions_hzToOctave', 'Conversions_hzToSamplingIncrement', 'Conversions_initialize', 'Conversions_intToString', 'Conversions_leftPan', 'Conversions_midiToName', 'Conversions_midiToAmplitude', 'Conversions_midiToDecibels', 'Conversions_midiToGain', 'Conversions_midiToHz', 'Conversions_midiToOctave', 'Conversions_midiToPitchClass', 'Conversions_midiToPitchClassSet', 'Conversions_midiToRoundedOctave', 'Conversions_midiToSamplingIncrement', 'Conversions_modulus', 'Conversions_nameToM', 'Conversions_nameToPitches', 'Conversions_octaveToHz', 'Conversions_octaveToMidi', 'Conversions_octaveToSamplingIncrement', 'Conversions_phaseToTableLengths', 'Conversions_pitchClassSetToMidi', 'Conversions_pitchClassToMidi', 'Conversions_rightPan', 'Conversions_round', 'Conversions_stringToBool', 'Conversions_stringToDouble', 'Conversions_stringToInt', 'Conversions_swapInt', 'Conversions_swapShort', 'Conversions_swigregister', 'Conversions_temper', 'Conversions_trim', 'Conversions_trimQuotes', 'Counterpoint', 'CounterpointNode', 'CounterpointNode_swigregister', 'Counterpoint_swigregister', 'DoubleVector', 'DoubleVector_swigregister', 'Event', 'EventVector', 'EventVector_swigregister', 'Event_swigregister', 'Hocket', 'Hocket_swigregister', 'ImageToScore', 'ImageToScore_swigregister', 'IntVector', 'IntVector_swigregister', 'Lindenmayer', 'Lindenmayer_swigregister', 'Logger', 'Logger_swigregister', 'McRM', 'McRM_swigregister', 'MidiByteVector', 'MidiByteVector_swigregister', 'MidiEvent', 'MidiEventVector', 'MidiEventVector_swigregister', 'MidiEvent_swigregister', 'MidiFile', 'MidiFile_chunkName', 'MidiFile_readInt', 'MidiFile_readShort', 'MidiFile_readVariableLength', 'MidiFile_swigregister', 'MidiFile_toInt', 'MidiFile_toShort', 'MidiFile_writeInt', 'MidiFile_writeShort', 'MidiFile_writeVariableLength', 'MidiHeader', 'MidiHeader_swigregister', 'MidiTrack', 'MidiTrack_swigregister', 'MusicModel', 'MusicModel_generateFilename', 'MusicModel_swigregister', 'Node', 'NodeVector', 'NodeVector_swigregister', 'Node_swigregister', 'PySwigIterator', 'PySwigIterator_swigregister', 'Random', 'Random_seed', 'Random_swigregister', 'Rescale', 'Rescale_swigregister', 'Score', 'ScoreNode', 'ScoreNode_swigregister', 'Score_getScale', 'Score_setScale', 'Score_swigregister', 'Sequence', 'Sequence_swigregister', 'Soundfile', 'Soundfile_swigregister', 'StrangeAttractor', 'StrangeAttractor_swigregister', 'System', 'System_beep', 'System_closeLibrary', 'System_createThread', 'System_createThreadLock', 'System_destroyThreadLock', 'System_execute', 'System_getDirectoryNames', 'System_getFileNames', 'System_getLogFile', 'System_getMessageLevel', 'System_getSharedLibraryExtension', 'System_getSymbol', 'System_getUserData', 'System_notifyThreadLock', 'System_openLibrary', 'System_parsePathname', 'System_setLogFile', 'System_setMessageLevel', 'System_setUserData', 'System_shellOpen', 'System_sleep', 'System_startTiming', 'System_stopTiming', 'System_swigregister', 'System_waitThreadLock', 'System_yieldThread', 'TempoMap', 'TempoMap_swigregister', 'ThreadLock', 'ThreadLock_swigregister', 'Voicelead', 'Voicelead_addOctave', 'Voicelead_areParallel', 'Voicelead_cToM', 'Voicelead_cToP', 'Voicelead_chordToPTV', 'Voicelead_closer', 'Voicelead_closest', 'Voicelead_closestPitch', 'Voicelead_conformToPitchClassSet', 'Voicelead_euclideanDistance', 'Voicelead_initializePrimeChordsPDivisionsPerOctave', 'Voicelead_inversions', 'Voicelead_invert', 'Voicelead_mToc', 'Voicelead_mTocPitchClassSet', 'Voicelead_nameToC', 'Voicelead_nonBijectiveVoicelead', 'Voicelead_normalChord', 'Voicelead_orderedPcs', 'Voicelead_pAndToPitchClassSet', 'Voicelead_pitch', 'Voicelead_pToPrimeChord', 'Voicelead_pc', 'Voicelead_pcs', 'Voicelead_pitchClassSetToM', 'Voicelead_pitchClassSetToPAndT', 'Voicelead_primeChord', 'Voicelead_ptvToChord', 'Voicelead_recursiveVoicelead', 'Voicelead_notate', 'Voicelead_notations', 'Voicelead_simpler', 'Voicelead_smoothness', 'Voicelead_sortByAscendingDistance', 'Voicelead_swigregister', 'Voicelead_toOrigin', 'Voicelead_transpose', 'Voicelead_uniquePcs', 'Voicelead_voicelead', 'Voicelead_voiceleading', 'Voicelead_voicings', 'Voicelead_wrap', 'VoiceleadingNode', 'VoiceleadingNode_swigregister', 'VoiceleadingOperation', 'VoiceleadingOperation_swigregister', 'CsoundAC', 'builtins', 'doc', 'file', 'lshift', 'lt', 'name', 'newclass', 'object', 'object_swigregister', 'swig_getattr', 'swig_property', 'swig_repr', 'swig_setattr', 'swig_setattr_nondynamic', 'csnd', 'cvar', 'new', 'new_instanceMethod', 'sys', 'weakref', 'weakref_proxy']
>>>
```

If you like, you can view all of the classes, variables, and methods of CsoundAC by entering `help(CsoundAC)`.

In a text editor,¹ enter the following script:

```
import CsoundAC

orchestra = '''
sr = 44100
ksmps = 100
nchnls = 2

        instr 1
        ; Sharp attack, but not sharp enough to click.
iattack =          0.005
        ; Moderate decay.
idecay  =          0.2
        ; Fast but gentle release.
irelease =         0.05
        ; Extend the total duration (p3) to include the attack, decay, and
release.
isustain =         p3
p3      =          iattack + idecay + isustain + irelease
        ; Exponential envelope.
kenvelope transeg  0.0, iattack, -3.0, 1.0, idecay, -3.0, 0.25, isustain,
-3.0, 0.25, irelease, -3.0, 0.0
        ; Translate MIDI key number to frequency in cycles per second.
ifrequency =      cpsmidinn(p4)
        ; Translate MIDI velocity to amplitude.
iamplitude =     ampdb(p5)
        ; Band-limited oscillator with integrated sawtooth wave.
aout     vco2     iamplitude * kenvelope, ifrequency, 8
        ; Output stereo signal
outs     aout, aout
        endin
'''

score = '''
i 1 0 10 68 80
'''

command = 'csound -RWfo toot1.wav toot1.orc toot1.sco'

model = CsoundAC.MusicModel()
model.setCsoundOrchestra(orchestra)
```

¹ You can use the IDLE editor that comes with Python, or any other programmer's editor such as EMACS, but I prefer to use SciTE (SciTE: A Free Text Editor for Win32 and X, 2007) together with a Csound mode (solipse, 2007). I prefer SciTE to IDLE because SciTE is easier to stop and restart CsoundAC and Python in SciTE. I prefer SciTE to Emacs because SciTE conforms to Windows user interface guidelines and is a lighter-weight application. All further examples in this section use SciTE.

```

model.setCsoundScoreHeader(score)
model.setCsoundCommand(command)

model.render()

```

This is an extremely simple Csound piece. It plays an A440 sawtooth note for 10 seconds to a soundfile. It is also about the simplest possible way to create and render this piece in Python. (You can find a pre-written version of this script in `MusicProjects\toot1.py`.) Run the script with Python to render the piece:

```

1 import CsoundAC
2
3 orchestra = '''
4 sr = 44100
5 ksmps = 100
6 -nchnls = 2
7
8
9          instr 1
10         ; Sharp attack, but not sharp enough to click.
11         ; Moderate decay.
12         -iattack = 0.005
13         -idecay = 0.2
14         ; Fast but gentle release.
15         -irelease = 0.05
16         ; Extend the total duration (p3) to include the attack, decay, and release.
17         -ps =
18         ; Exponential envelope.
19         -kenvelope transg 0.0, iattack, -3.0, 1.0, idecay, -3.0, 0.25, isustain, -3.0, 0.25, irelease, -3.0, 0.0
20         ; Translate MIDI key number to frequency in cycles per second.
21         -ifrequency = cpsmidinn(p4)
22         ; Translate MIDI velocity to amplitude.
23         -iampitude = aampb(p5)
24         ; Band-limited oscillator with integrated sawtooth wave.
25         -aout vco2 iampitude * kenvelope, ifrequency, 0
26         ; Output stereo signal
27         outs aout, aout
28
29         ...
30
31 score = '''
32 i 1 0 10 68 80
33 ...
34
35 command = 'csound -Rf0 toot1.wav toot1.orc toot1.sco'
36
37 model = CsoundAC.MusicModel()
38 model.setCsoundOrchestra(orchestra)
39 model.setCsoundScoreHeader(score)
40 model.setCsoundCommand(command)
41
42
43 model.render()
44
45

```

```

python -u "toot1.py"
BEGIN CppSound:perform(5, 00EFFF90)...
EBGAN CppSound:compile(5, 00EFFF90)...
Localization of messages is disabled, using default language.
Generated 0 source: ..
addToScore.length(): 16
PortMIDI real-time MIDI plugin for Csound
PortAudio real-time audio module for Csound
Virtual_keyboard real-time MIDI plugin for Csound
0dBFS level = 32768.0
Csound version 5.07 (double samples) Nov  8 2007
libsndfile-1.0.17
orchname: toot1.orc
scorename: toot1.sco
readinfo: PortAudio module enabled ... using callback interface
rtmidi: PortMIDI module enabled
orch compiler:
26 lines read
instr 1
Elapsed time at end of orchestra compile: real: 0.001s, CPU: 0.000s
sorting score ...
Elapsed time at end of score sort: real: 0.052s, CPU: 0.047s
Csound version 5.07 (double samples) Nov  8 2007
displays suppressed
0dBFS level = 32768.0
orch now loaded
audio buffered in 128 sample-frame blocks
writing 1024-byte blocks of floats to toot1.wav (WAV)
SECTION 1:
ENDHD CppSound:compile.
new alloc for instr 1:
B 0.000 .. 5.000 T 5.000 IT 5.000 M: 6524.6 6524.6
E 5.000 .. 10.000 T 10.255 IT 10.255 M: 1647.7 1647.7
end of section 1 sect peak amps: 6524.6 6524.6
inactive allocs returned to freespac
SECTION 2:
B 0.000 .. 5.000 T 5.000 IT 15.254 M: 0.0 0.0
end of section 2 sect peak amps: 0.0 0.0
Score finished in csoundPerformMkmps().
inactive allocs returned to freespac
end of score. overall amps: 6524.6 6524.6
0 errors in performance
Elapsed time at end of performance: real: 0.185s, CPU: 0.188s
5236 1024-byte soundfiles of floats written to toot1.wav (WAV)
Elapsed time = 0.250000 seconds.
ENDHD CppSound:perform.
>Exit code: 0

```

Note the following:

1. The Csound orchestra file, score file, and command line are embedded into the Python script as Python variables. The multi-line orchestra and score text are enclosed in Python's triple quotes, which preserve all line breaks and text formatting.
2. The script creates one instance, `model`, of the `CsoundAC.MusicModel` class. This object manages a music graph (see above), and also contains an instance of Csound.
3. The Csound orchestra, score, and command are sent to the Csound object in the `model` using Python convenience functions.
 1. The `MusicModel.render()` function does the following:

1. Traverse the music graph to generate a Csound score. The graph is empty here, so no score is generated.
 2. Add any Csound score text passed in using the `MusicModel.setCsoundScoreHeader()` function to the front of the generated score. The generated score is empty here, so the score header becomes the entire score. It contains one note.
 3. Parse out the orchestra and score filenames from the Csound command, and save the Csound orchestra and generated score to those files.
 4. Execute the Csound command, to render the generated score using the orchestra.
4. All Csound output messages are printed to the standard output, so they show up in SciTE's output pane, along with any Python output messages.

Generating Scores in Python

We will now change this simple exercise just enough to algorithmically generate a score.

1. Import the `string` module at the top of the script:
2. Replace

```
import string
score = '''
i 1 0 10 68 80
'''
```

with code that iterates the logistic equation, $y = r y (1 - y)$; as the value of r increases, iterating the equation will cause y to converge on a fixed point, to oscillate in a repeating cycle, or to generate a never-ending sequence called a “strange attractor” (Peitgen, Jurgens, & Saupe, 1992, pp. 585-653). With each iteration, advance the time, but allow the notes to overlap by half their duration. Map the value of y in the interval $[0, 1]$ to the interval $[36, 96]$, which is a 5 octave range in terms of MIDI key numbers. Python contains excellent facilities for passing a tuple of values, which can include any type of data, to formatting specifications in a string; use this to format a Csound `i` statement for each iteration of the equation. The generated statements can simply be appended to a list, which can then be joined to form a single string that is the Csound score. The value of r chosen here generates a chaotic attractor.

```

r = 3.974
y = 0.5
time_ = 0.0
duration = 0.25

istatements = []
for i in xrange(1000):
    y = r * y * (1.0 - y)
    time_ = time_ + duration / 2.0
    midikey = int(36.0 + (y * 60.0))
    istatement = "i 1 %f %f %d 80\n" % (time_, duration, midikey)
    print istatement,
    istatements.append(istatement)

score = string.join(istatements)

```

The entire script is then (you can find a pre-written version of this script in MusicProjects\toot2.py):

```

import CsoundAC
import string

orchestra = '''
sr = 44100
ksmps = 100
nchnls = 2

instr 1
; Sharp attack, but not sharp enough to click.
iattack = 0.005
; Moderate decay.
idecay = 0.2
; Fast but gentle release.
irelease = 0.05
; Extend the total duration (p3) to include the attack, decay, and
release.
isustain = p3
p3 = iattack + idecay + isustain + irelease
; Exponential envelope.
kenvelope transeg 0.0, iattack, -3.0, 1.0, idecay, -3.0, 0.25, isustain,
-3.0, 0.25, irelease, -3.0, 0.0
; Translate MIDI key number to frequency in cycles per second.
ifrequency = cpsmidinn(p4)
; Translate MIDI velocity to amplitude.
iamplitude = ampdb(p5)
; Band-limited oscillator with integrated sawtooth wave.
aout vco2 iamplitude * kenvelope, ifrequency, 8
; Output stereo signal
outs aout, aout
endin

```

```

...

r = 3.974
y = 0.5
time_ = 0.0
duration = 0.25

istatements = []
for i in xrange(1000):
    y = r * y * (1.0 - y)
    time_ = time_ + duration / 2.0
    midikey = int(36.0 + (y * 60.0))
    istatement = "i 1 %f %f %d 80\n" % (time_, duration, midikey)
    print istatement,
    istatements.append(istatement)

score = string.join(istatements)

command = 'csound -RWfo toot2.wav toot2.orc toot2.sco'

model = CsoundAC.MusicModel()
model.setCsoundOrchestra(orchestra)
model.setCsoundScoreHeader(score)
model.setCsoundCommand(command)

model.render()

```

A Labor-Saving Pattern for Algorithmic Composition

The pattern that we have established in the previous example can be repeated again and again to generate different pieces. In order to save work and ensure consistency, it is worthwhile to establish a pattern that can be followed with only slight variations in every composition:

1. At the top of the file, in triple quotes, put the title, author, and date of the piece, together with any other explanatory comments you like. It is a Python convention that such a string is a “docstring” that can be used for automatically documenting code. In fact, print out the docstring at the beginning of every performance.
2. In general, put in comments that explain what each section of the script is doing; in fact, *print* these comments during the performance.
3. Import a number of the most useful Python modules.
4. Put in a section that automatically generates all filenames based off the actual name of the script file. This enforces a consistent naming

- convention, and removes any need for coming up with names for soundfiles, Csound files, and so on.
5. Put in a section that defines alternative Csound commands for rendering at various quality levels, or to real-time audio; the reader can select a rendering option by changing the value of a `rendering` variable.
 6. Put in a section to create all CsoundAC objects that may be used in the piece at the top of the script, so that they will be in scope for the remainder of the script.
 7. Put the section for actually generating the score right after that, so it is near the top of the script file and easy to find and edit. Often, this is the only section that you need to edit in order to create a new piece.
 8. Put in a section that embeds a powerful Csound orchestra that uses the same set of pfields in all instrument definitions, and also has high-quality effects and mixer busses (the creation of this orchestra is outside the scope of this section; the orchestra used in the following is an adaption of the one that I myself use).
 9. Put in a section that enables the user to specify an arrangement of selected instruments from the orchestra, and to customize the instrument numbers, loudnesses, and pans of those instruments.
 10. Put in a section to automatically render the generated score, and also automatically save each generated score as a MIDI sequence file. MIDI files take up next to no room, and can be imported into notation software for viewing in traditional music notation, or into a sequencer for rendering with commercial software synthesizers.
 11. End with a section that automatically plays the rendered soundfile, by running Csound with an orchestra that simply reads the output soundfile and renders it again to the real-time audio interface; this method of playing a soundfile works on all operating systems on which Csound itself runs.

Following this pattern ensures that a new piece can be written simply by saving an existing piece to a new filename, changing only the score-generating section of the code, and perhaps also the arrangement and Csound orchestra sections. To illustrate this pattern, the preceding tutorial has been filled out according to the suggested pattern in the following. The script has also been

modified to generate the score by appending events to a `CsoundAC.ScoreNode` (here, the only node) in the music graph:

```
...
TUTORIAL COMPOSITION
Implemented by Michael Gogins
19 November 2007
This code is in the public domain.
...

print __doc__
print 'IMPORTING REQUIRED MODULES...'
print
import CsoundAC
import os
import random
import signal
import string
import sys
import traceback

print 'CREATING FILENAMES...'
print
scriptFilename = sys.argv[0]
print 'Full Python script:      %s' % scriptFilename
title, exte = os.path.splitext(os.path.basename(scriptFilename))
print 'Base Python script:      %s' % title
directory = os.path.dirname(scriptFilename)
if len(directory):
    print 'Working directory:      %s' % directory
    os.chdir(directory)
print 'Working directory:      %s' % directory
orcFilename = title + '.orc'
print 'Csound orchestra:        %s' % orcFilename
scoFilename = title + '.sco'
print 'Csound score:            %s' % scoFilename
midiFilename = title + '.mid'
print 'MIDI filename:           %s' % midiFilename
soundfileName = title + '.wav'
print 'Soundfile name:          %s' % soundfileName
dacName = 'dac'
print 'Audio output name:       %s' % dacName
print

print 'SETTING RENDERING AND PLAYBACK OPTIONS...'
print
print 'Set "rendering" to:       "cd", "preview" (default), or "audio".'
print 'Set "playback" to:       True (default) or False.'
print
rendering = 'preview'
```

```

playback = True
print 'Rendering option:      %s' % rendering
print 'Play after rendering:  %s' % playback
commandsForRendering = {
    'cd':      'csound -r 44100 -k 44100 -m3 -RWZdfo %s %s %s' % (soundfileName,
orcfilename, scofilename),
    'preview': 'csound -r 44100 -k 100 -m3 -RWZdfo %s %s %s' % (soundfileName,
orcfilename, scofilename),
    'audio':   'csound -r 44100 -k 100 -m3 -RWZdfo %s %s %s' % (dacName,
orcfilename, scofilename),
}
csoundCommand = commandsForRendering[rendering]
print 'Csound command line:   %s' % csoundCommand
print

print 'CREATING GLOBAL OBJECTS...'
print
model = CsoundAC.MusicModel()
csound = model.getCppSound()
csound.setPythonMessageCallback()
score = model.getScore()

print 'CREATING MUSIC MODEL...'
print
scoreNode = CsoundAC.ScoreNode()
generatedScore = scoreNode.getScore()
model.addChild(scoreNode)
r = 3.974
y = 0.5
time_ = 0.0
duration = 0.25
istatements = []
for i in xrange(1000):
    y = r * y * (1.0 - y)
    time_ = time_ + duration / 2.0
    midikey = float(36.0 + (y * 60.0))
    generatedScore.append(time_, duration, 144.0, 1.0, midikey, 80.0)

print 'CREATING CSOUND ORCHESTRA...'
print
csoundOrchestra = \
'''
sr          =          44100
ksmps      =          100
nchnls     =           2

instr 1
; Sharp attack, but not sharp enough to click.
iattack    =          0.005
; Moderate decay.

```

```

idecay          =          0.2
                ; Fast but gentle release.
irelease       =          0.05
                ; Extend the total duration (p3) to include the attack, decay, and
release.
isustain       =          p3
p3             =          iattack + idecay + isustain + irelease
                ; Exponential envelope.
kenvelope      transeg      0.0, iattack, -3.0, 1.0, idecay, -3.0, 0.25,
isustain, -3.0, 0.25, irelease, -3.0, 0.0
                ; Translate MIDI key number to frequency in cycles per second.
ifrequency     =          cpsmidinn(p4)
                ; Translate MIDI velocity to amplitude.
iamplitude     =          ampdb(p5)
                ; Band-limited oscillator with integrated sawtooth wave.
aout           vco2         iamplitude * kenvelope, ifrequency, 8
                ; Output stereo signal
outs           aout, aout
                endin
...

print 'CREATING CSOUND ARRANGEMENT...'
print
model.setCsoundOrchestra(csoundOrchestra)
model.setCsoundCommand(csoundCommand)

print 'RENDERING...'
print
model.render()
score.save(midiFilename)

if playback and (rendering != 'audio'):
    print
    print 'PLAYING OUTPUT SOUNDFILE...'
    print
    csound.setCommand('csound -o %s temp.orc temp.sco' % dacName)
    csound.setOrchestra(\
...

; Must be the same sr and nchnls as the soundfile.
sr = 44100
ksmps = 100
nchnls = 2

                instr 1
                ; Extend the note to the duration of the soundfile.
ifileseconds   filelen      "%s"
p3             =            ifileseconds
                ; Simply play the soundfile to the output.
aleft, aright  soundin     "%s"
                outs        aleft, aright
                endin

```

```
''' % (soundfileName, soundfileName))
    csound.setScore('i 1 1 1\n')
    csound.exportForPerformance()
    csound.perform()

print 'FINISHED.'
print
```

This same pattern is followed in all remaining tutorials, but with more elaborate orchestra and arrangement sections. You can find a pre-written version of this script as `MusicProjects\toot3.py`.

Readers with programming experience are probably wondering why I broke a cardinal rule of software engineering, and did not factor out redundant code (e.g. for playing back soundfiles) into a separate module or modules that could be shared by all the composition scripts. Normally this would indeed be the best practice.

However, experience shows that in computer music, it is much wiser to keep everything that is required to re-create a piece in the smallest possible number of files – that is, in just one file. That means that in the future, I don't have to worry that some supporting module will have changed in such a way as to change the sound of the piece when I go back five years later to render it again. I have lost the ability to re-create some of my best pieces in this way! By duplicating all the supporting code in each piece, including especially the entire Csound orchestra, I can count on being able to re-create the piece exactly at any time.

An Historical Tutorial in Algorithmic Composition

Now that you have been introduced to CsoundAC's facilities for algorithmic composition, let us illustrate them and put them to use by implementing a variety of techniques for algorithmic composition.

The remainder of this section does not offer a complete or connected history of algorithmic composition, but it does provide a series of tutorials that draw from formative moments in the history of algorithmic composition. We will not literally re-create all of these historical pieces, partly for reasons of copyright; but we will use techniques borrowed from the original, classic pieces to create new pieces. In the process, we will also create some Python classes that may prove useful for further work in algorithmic composition.

For a brief introductory history of algorithmic composition, see (Roads, 1996). For a somewhat more technical brief introduction, see (Loy, 1990). For a scholarly history, see (Burns, 1993); for a recent overview, see (Nierhaus, 2009).

Mozart (?) and The Musical Dice Game

The Original Musikalisches Wuerfelspiel

This dice game for composing minuets was published anonymously in 1787 and is often attributed to Mozart, who is known to be the author of a slightly later game that resembles part of this one. John Chuang's web site provides an on-line version of this game, a freeware DOS implementation of it, a description of it, and some historical material and links (Chuang, 2007).

The game consists of a set of measures of music, and two tables for specifying how to choose measures for a piece by throwing a die. There is one table for minuets, and another table for trios. For the minuet sections, the die is thrown twice and the number from each throw is summed to provide the dice roll. For the trio sections, the die is thrown once. Each table consists of one row for each possible dice roll, and one column for each of the 16 bars in a section. Each cell in the table contains the number of a precomposed measure of music in waltz time. The bar numbers in the original tables suggest that a minuet of 16 bars is to be followed by a trio of 16 bars. It would have been a common practice of the time to repeat this sequence: *minuet, trio, minuet, trio*.

The algorithm works because each bar (or table column) contains a choice of possible measures that each fit the same harmonic function, so the sequence of columns inevitably forms a suitable harmonic progression for the *galant* style.

It is easy to program this game in Python using CsoundAC. The tables were transcribed from the original by John Chuang, and the measures of music were downloaded from his web site as short MIDI sequence files. They are used here with his permission.

In Python, an object created with `{key0: value0, key1: value1, ...}` is a dictionary of keys that can be used to look up values. Each of the two tables of measures can be done as a dictionary of dictionaries. First you throw the die and use the dice roll as the key for the first dictionary, which returns a second dictionary to which the key is the bar number, which returns the number identifying an actual measure of music.

```
minuetTable = {}
```

```

minuetTable[ 2] = { 1: 96, 2: 22, 3:141, 4: 41, 5:105, 6:122, 7: 11, 8: 30,
9: 70, 10:121, 11: 26, 12: 9, 13:112, 14: 49, 15:109, 16: 14}
minuetTable[ 3] = { 1: 32, 2: 6, 3:128, 4: 63, 5:146, 6: 46, 7:134, 8: 81,
9:117, 10: 39, 11:126, 12: 56, 13:174, 14: 18, 15:116, 16: 83}
minuetTable[ 4] = { 1: 69, 2: 95, 3:158, 4: 13, 5:153, 6: 55, 7:110, 8: 24,
9: 66, 10:139, 11: 15, 12:132, 13: 73, 14: 58, 15:145, 16: 79}
minuetTable[ 5] = { 1: 40, 2: 17, 3:113, 4: 85, 5:161, 6: 2, 7:159, 8:100,
9: 90, 10:176, 11: 7, 12: 34, 13: 67, 14:160, 15: 52, 16:170}
minuetTable[ 6] = { 1:148, 2: 74, 3:163, 4: 45, 5: 80, 6: 97, 7: 36, 8:107,
9: 25, 10:143, 11: 64, 12:125, 13: 76, 14:136, 15: 1, 16: 93}
minuetTable[ 7] = { 1:104, 2:157, 3: 27, 4:167, 5:154, 6: 68, 7:118, 8: 91,
9:138, 10: 71, 11:150, 12: 29, 13:101, 14:162, 15: 23, 16:151}
minuetTable[ 8] = { 1:152, 2: 60, 3:171, 4: 53, 5: 99, 6:133, 7: 21, 8:127,
9: 16, 10:155, 11: 57, 12:175, 13: 43, 14:168, 15: 89, 16:172}
minuetTable[ 9] = { 1:119, 2: 84, 3:114, 4: 50, 5:140, 6: 86, 7:169, 8: 94,
9:120, 10: 88, 11: 48, 12:166, 13: 51, 14:115, 15: 72, 16:111}
minuetTable[10] = { 1: 98, 2:142, 3: 42, 4:156, 5: 75, 6:129, 7: 62, 8:123,
9: 65, 10: 77, 11: 19, 12: 82, 13:137, 14: 38, 15:149, 16: 8}
minuetTable[11] = { 1: 3, 2: 87, 3:165, 4: 61, 5:135, 6: 47, 7:147, 8: 33,
9:102, 10: 4, 11: 31, 12:164, 13:144, 14: 59, 15:173, 16: 78}
minuetTable[12] = { 1: 54, 2:130, 3: 10, 4:103, 5: 28, 6: 37, 7:106, 8: 5,
9: 35, 10: 20, 11:108, 12: 92, 13: 12, 14:124, 15: 44, 16:131}
trioTable = {}
trioTable[ 1] = {17: 72, 18: 6, 19: 59, 20: 25, 21: 81, 22: 41, 23: 89, 24: 13,
25: 36, 26: 5, 27: 46, 28: 79, 29: 30, 30: 95, 31: 19, 32: 66}
trioTable[ 2] = {17: 56, 18: 82, 19: 42, 20: 74, 21: 14, 22: 7, 23: 26, 24: 71,
25: 76, 26: 20, 27: 64, 28: 84, 29: 8, 30: 35, 31: 47, 32: 88}
trioTable[ 3] = {17: 75, 18: 39, 19: 54, 20: 1, 21: 65, 22: 43, 23: 15, 24: 80,
25: 9, 26: 34, 27: 93, 28: 48, 29: 69, 30: 58, 31: 90, 32: 21}
trioTable[ 4] = {17: 40, 18: 73, 19: 16, 20: 68, 21: 29, 22: 55, 23: 2, 24: 61,
25: 22, 26: 67, 27: 49, 28: 77, 29: 57, 30: 87, 31: 33, 32: 10}
trioTable[ 5] = {17: 83, 18: 3, 19: 28, 20: 53, 21: 37, 22: 17, 23: 44, 24: 70,
25: 63, 26: 85, 27: 32, 28: 96, 29: 12, 30: 23, 31: 50, 32: 91}
trioTable[ 6] = {17: 18, 18: 45, 19: 62, 20: 38, 21: 4, 22: 27, 23: 52, 24: 94,
25: 11, 26: 92, 27: 24, 28: 86, 29: 51, 30: 60, 31: 78, 32: 31}

```

Python has a `random.randint(minimum, maximum)` function that returns a random integer in an inclusive range, which perfectly simulates one throw of a die with any number of faces. The following function, in turn, simulates any number of throws of such a die:

```

def rollDice(throws):
    diceroll = 0
    for i in range(throws):
        diceroll += random.randint(1, 6)
    return diceroll

```

CsoundAC can import MIDI files, so the measures are stored in individual MIDI files and imported into CsoundAC . ScoreNode objects as required.

Note the line `scoreNode.thisown = 0`. Python uses a garbage collector for memory management. This means that whenever there are no more variables containing a reference to an object, that object is automatically deleted. In this

case, however, the Python object is only a “wrapper” around an underlying C++ object, which does not use garbage collection. The `thisown` attribute, if non-zero, indicates to Python that the garbage collector can safely delete the object. If 0, the `thisown` attribute indicates to Python that the object should not be garbage-collected. In this case, the newly created `ScoreNode` object is added as a child to another node. This happens in C++, so Python loses track of the reference. The result would be that after a short time, the `ScoreNode` would be deleted – but the C++ code will still attempt to reference it. At that point, the program will crash. Setting `thisown` to 0 prevents this. It is not necessary to set `thisown` to 0 for `CsoundAC` objects that remain in global scope. All `CsoundAC` objects will normally be deleted anyway by the `model` object, which is in global scope, when it is destroyed by the garbage collector at the end of the run.

```
def readMeasure(section, number):
    scoreNode = CsoundAC.ScoreNode()
    scoreNode.thisown = 0
    filename = section + str(number) + '.mid'
    scoreNode.getScore().load(filename)
    return scoreNode, filename
```

Because of the repeats, it is necessary first to generate the sequences of measure numbers, and then to assemble the piece. The `ScoreNode` object, in turn, is placed into a `CsoundAC.Rescale` node. The `Rescale` nodes arrange the measures in the proper temporal sequence. The whole inner loop of the program is as follows:

```
print 'Selecting random measures for minuet and trio by dice roll and bar
number...'
random.seed()
measuresChosen = {}
for bar in xrange( 1, 17):
    measuresChosen[bar] = minuetTable[rollDice(2)][bar]
for bar in xrange(17, 33):
    measuresChosen[bar] = trioTable [rollDice(1)][bar]
print 'Assembling the selected measures with appropriate repeats...'
cumulativeTime = 0
for repeat in xrange(1, 3):
    for barNumber in xrange(1, 17):
        measure, filename = readMeasure('M', measuresChosen[barNumber])
        notes = len(measure.getScore())
        duration = measure.getScore().getDuration()
        print 'Repeat %d: Minuet bar %d measure M%d %d notes at %f seconds' %
(repeat, barNumber, measuresChosen[barNumber], notes, cumulativeTime)
        print measure.getScore().getCsoundScore()
        cumulativeTime += duration
```

```

barTime = CsoundAC.Rescale()
barTime.setRescale(CsoundAC.Event.TIME,      1, 0,  cumulativeTime, 0)
barTime.setRescale(CsoundAC.Event.INSTRUMENT, 1, 0,   1, 0)
barTime.setRescale(CsoundAC.Event.VELOCITY,  1, 1,   75, 5)
barTime.thisown=0
barTime.addChild(measure)
model.addChild(barTime)
for barNumber in xrange(17, 33):
    measure, filename = readMeasure('T', measuresChosen[barNumber])
    notes = len(measure.getScore())
    duration = measure.getScore().getDuration()
    print 'Repeat %d: Trio   bar %d measure T%d %d notes at %f seconds' %
(repeat, barNumber, measuresChosen[barNumber], notes, cumulativeTime)
    print measure.getScore().getCsoundScore()
    cumulativeTime += duration
    barTime = CsoundAC.Rescale()
    barTime.setRescale(CsoundAC.Event.TIME,      1, 0,  cumulativeTime, 0)
    barTime.setRescale(CsoundAC.Event.INSTRUMENT, 1, 0,   1, 0)
    barTime.setRescale(CsoundAC.Event.VELOCITY,  1, 1,   75, 5)
    barTime.thisown=0
    barTime.addChild(measure)
    model.addChild(barTime)

```

Create a Csound arrangement as follows. The CsoundAC.csd file accompanying this tutorial contains a number of Csound instrument definitions that all take the same set of pfields, and all output more or less the same volume of sound. Open this file in a text editor. Select the contents of the <CsInstruments> element, copy it, and paste it into your script, replacing the text of the csoundOrchestra variable. Similarly, select the contents of the <CsScore> element, copy it, and paste it into your script, replacing the text of the csoundScoreHeader variable. Set the Csound command as you would for the standard console version of Csound. This gives you a large orchestra with a number of debugged instrument definitions, a mixer buss, and effects including high-quality reverb.

Now, you can use the arrange function to select instrument definitions for the arrangement.

```

print 'CREATING CSOUND ARRANGEMENT...'
print

model.setCsoundOrchestra(csoundOrchestra)
model.setCsoundScoreHeader(csoundScoreHeader)
#           oldinsno, newinsno, level (+-dB), pan (-1.0 through +1.0)
model.arrange( 1,      7,      0.0,      -0.7 )
model.arrange( 2,      6,      0.0,       0.0 )
model.arrange( 3,      7,      0.0,      +0.7 )
model.setCsoundCommand(csoundCommand)

```

The script for the entire piece is now as follows (the orchestra and the score header have been elided to save space):

```
'''
TUTORIAL COMPOSITION BASED ON MUSIKALISCHES WUERFELSPIEL OF 1787
Implemented by Michael Gogins
6 November 2004 -- 13 October 2007
This code is in the public domain.
'''

print __doc__
print 'IMPORTING REQUIRED MODULES...'
print
import CsoundAC
import os
import random
import signal
import string
import sys
import traceback

print 'CREATING FILENAMES...'
print
scriptFilename = sys.argv[0]
print 'Full Python script:      %s' % scriptFilename
title, exte = os.path.splitext(os.path.basename(scriptFilename))
print 'Base Python script:      %s' % title
directory = os.path.dirname(scriptFilename)
if len(directory):
    print 'Working directory:    %s' % directory
    os.chdir(directory)
print 'Working directory:      %s' % directory
orcFilename = title + '.orc'
print 'Csound orchestra:       %s' % orcFilename
scoFilename = title + '.sco'
print 'Csound score:          %s' % scoFilename
midiFilename = title + '.mid'
print 'MIDI filename:         %s' % midiFilename
soundfileName = title + '.wav'
print 'Soundfile name:       %s' % soundfileName
dacName = 'dac'
print 'Audio output name:    %s' % dacName
print

print 'SETTING RENDERING AND PLAYBACK OPTIONS...'
print
print 'Set "rendering" to:     "cd", "preview" (default), or "audio".'
print 'Set "playback" to:     True (default) or False.'
print
rendering = 'preview'
playback = True
```

```

print 'Rendering option:      %s' % rendering
print 'Play after rendering:  %s' % playback
commandsForRendering = {
    'cd':      'csound -r 44100 -k 44100 -m3 -RWZdfo %s %s %s' % (soundfileName,
orcFilename, scoFilename),
    'preview': 'csound -r 44100 -k 100 -m3 -RWZdfo %s %s %s' % (soundfileName,
orcFilename, scoFilename),
    'audio':   'csound -r 44100 -k 100 -m3 -RWZdfo %s %s %s' % (dacName,
orcFilename, scoFilename),
}
csoundCommand = commandsForRendering[rendering]
print 'Csound command line:   %s' % csoundCommand
print

print 'CREATING GLOBAL OBJECTS...'
print
model = CsoundAC.MusicModel()
csound = model.getCppSound()
csound.setPythonMessageCallback()
score = model.getScore()

print 'CREATING MUSIC MODEL...'
print

minuetTable = {}
minuetTable[ 2] = { 1: 96,  2: 22,  3:141,  4: 41,  5:105,  6:122,  7: 11,  8: 30,
9: 70, 10:121, 11: 26, 12:  9, 13:112, 14: 49, 15:109, 16: 14}
minuetTable[ 3] = { 1: 32,  2:  6,  3:128,  4: 63,  5:146,  6: 46,  7:134,  8: 81,
9:117, 10: 39, 11:126, 12: 56, 13:174, 14: 18, 15:116, 16: 83}
minuetTable[ 4] = { 1: 69,  2: 95,  3:158,  4: 13,  5:153,  6: 55,  7:110,  8: 24,
9: 66, 10:139, 11: 15, 12:132, 13: 73, 14: 58, 15:145, 16: 79}
minuetTable[ 5] = { 1: 40,  2: 17,  3:113,  4: 85,  5:161,  6:  2,  7:159,  8:100,
9: 90, 10:176, 11:  7, 12: 34, 13: 67, 14:160, 15: 52, 16:170}
minuetTable[ 6] = { 1:148,  2: 74,  3:163,  4: 45,  5: 80,  6: 97,  7: 36,  8:107,
9: 25, 10:143, 11: 64, 12:125, 13: 76, 14:136, 15:  1, 16: 93}
minuetTable[ 7] = { 1:104,  2:157,  3: 27,  4:167,  5:154,  6: 68,  7:118,  8: 91,
9:138, 10: 71, 11:150, 12: 29, 13:101, 14:162, 15: 23, 16:151}
minuetTable[ 8] = { 1:152,  2: 60,  3:171,  4: 53,  5: 99,  6:133,  7: 21,  8:127,
9: 16, 10:155, 11: 57, 12:175, 13: 43, 14:168, 15: 89, 16:172}
minuetTable[ 9] = { 1:119,  2: 84,  3:114,  4: 50,  5:140,  6: 86,  7:169,  8: 94,
9:120, 10: 88, 11: 48, 12:166, 13: 51, 14:115, 15: 72, 16:111}
minuetTable[10] = { 1: 98,  2:142,  3: 42,  4:156,  5: 75,  6:129,  7: 62,  8:123,
9: 65, 10: 77, 11: 19, 12: 82, 13:137, 14: 38, 15:149, 16:  8}
minuetTable[11] = { 1:  3,  2: 87,  3:165,  4: 61,  5:135,  6: 47,  7:147,  8: 33,
9:102, 10:  4, 11: 31, 12:164, 13:144, 14: 59, 15:173, 16: 78}
minuetTable[12] = { 1: 54,  2:130,  3: 10,  4:103,  5: 28,  6: 37,  7:106,  8:  5,
9: 35, 10: 20, 11:108, 12: 92, 13: 12, 14:124, 15: 44, 16:131}

trioTable = {}
trioTable[ 1] = {17: 72, 18:  6, 19: 59, 20: 25, 21: 81, 22: 41, 23: 89, 24: 13,
25: 36, 26:  5, 27: 46, 28: 79, 29: 30, 30: 95, 31: 19, 32: 66}
trioTable[ 2] = {17: 56, 18: 82, 19: 42, 20: 74, 21: 14, 22:  7, 23: 26, 24: 71,
25: 76, 26: 20, 27: 64, 28: 84, 29:  8, 30: 35, 31: 47, 32: 88}
trioTable[ 3] = {17: 75, 18: 39, 19: 54, 20:  1, 21: 65, 22: 43, 23: 15, 24: 80,
25:  9, 26: 34, 27: 93, 28: 48, 29: 69, 30: 58, 31: 90, 32: 21}

```

```

trioTable[ 4] = {17: 40, 18: 73, 19: 16, 20: 68, 21: 29, 22: 55, 23: 2, 24: 61,
25: 22, 26: 67, 27: 49, 28: 77, 29: 57, 30: 87, 31: 33, 32: 10}
trioTable[ 5] = {17: 83, 18: 3, 19: 28, 20: 53, 21: 37, 22: 17, 23: 44, 24: 70,
25: 63, 26: 85, 27: 32, 28: 96, 29: 12, 30: 23, 31: 50, 32: 91}
trioTable[ 6] = {17: 18, 18: 45, 19: 62, 20: 38, 21: 4, 22: 27, 23: 52, 24: 94,
25: 11, 26: 92, 27: 24, 28: 86, 29: 51, 30: 60, 31: 78, 32: 31}

```

```

def readMeasure(section, number):
    scoreNode = CsoundAC.ScoreNode()
    scoreNode.thisown = 0
    filename = section + str(number) + '.mid'
    scoreNode.getScore().load(filename)
    return scoreNode, filename

```

```

def rollDice(throws):
    diceroll = 0
    for i in range(throws):
        diceroll += random.randint(1, 6)
    return diceroll

```

```

print 'Selecting random measures for minuet and trio by dice roll and bar
number...'

```

```

random.seed()
measuresChosen = {}
for bar in xrange(1, 17):
    measuresChosen[bar] = minuetTable[rollDice(2)][bar]
for bar in xrange(17, 33):
    measuresChosen[bar] = trioTable [rollDice(1)][bar]
print 'Assembling the selected measures with appropriate repeats...'
cumulativeTime = 0
for repeat in xrange(1, 3):
    for barNumber in xrange(1, 17):
        measure, filename = readMeasure('M', measuresChosen[barNumber])
        notes = len(measure.getScore())
        duration = measure.getScore().getDuration()
        print 'Repeat %d: Minuet bar %d measure M%d %d notes at %f seconds' %
(repeat, barNumber, measuresChosen[barNumber], notes, cumulativeTime)
        print measure.getScore().getCsoundScore()
        cumulativeTime += duration
        barTime = CsoundAC.Rescale()
        barTime.setRescale(CsoundAC.Event.TIME, 1, 0, cumulativeTime, 0)
        barTime.setRescale(CsoundAC.Event.INSTRUMENT, 1, 0, 1, 0)
        barTime.setRescale(CsoundAC.Event.VELLOCITY, 1, 1, 75, 5)
        barTime.thisown=0
        barTime.addChild(measure)
        model.addChild(barTime)
    for barNumber in xrange(17, 33):
        measure, filename = readMeasure('T', measuresChosen[barNumber])
        notes = len(measure.getScore())
        duration = measure.getScore().getDuration()

```

```

        print 'Repeat %d: Trio   bar %d measure T%d %d notes at %f seconds' %
(repeat, barNumber, measuresChosen[barNumber], notes, cumulativeTime)
        print measure.getScore().getCsoundScore()
        cumulativeTime += duration
        barTime = CsoundAC.Rescale()
        barTime.setRescale(CsoundAC.Event.TIME,      1, 0,  cumulativeTime, 0)
        barTime.setRescale(CsoundAC.Event.INSTRUMENT, 1, 0,   1, 0)
        barTime.setRescale(CsoundAC.Event.VELOCITY,  1, 1,  75, 5)
        barTime.thisown=0
        barTime.addChild(measure)
        model.addChild(barTime)

print 'CREATING CSOUND ORCHESTRA...'
print

csoundCommand = 'csound -m3 -RWZdfo %s %s %s' % (soundfileName, orcFilename,
scoFilename)
print 'Csound command line: %s' % csoundCommand
print

csoundOrchestra = \
'''
; Csound orchestra goes here...
'''

csoundScoreHeader = \
'''
; Csound score header goes here...
'''

print 'CREATING CSOUND ARRANGEMENT...'
print

model.setCsoundOrchestra(csoundOrchestra)
model.setCsoundScoreHeader(csoundScoreHeader)
#           oldinsno, newinsno, level (+-dB), pan (-1.0 through +1.0)
model.arrange( 1,      7,      0.0,      -0.7 )
model.arrange( 2,      6,      0.0,      0.0 )
model.arrange( 3,      7,      0.0,      +0.7 )
model.setCsoundCommand(csoundCommand)

print 'RENDERING...'
print
model.render()
score.save(midiFilename)

if playback and (rendering != 'audio'):
    print
    print 'PLAYING OUTPUT SOUNDFILE...'
    print

```



```

        csound.setCommand('csound -o %s temp.orc temp.sco' % dacName)
        csound.setOrchestra(\
'''
; Must be the same sr and nchnls as the soundfile.
sr = 44100
ksmps = 100
nchnls = 2

        instr 1
        ; Extend the note to the duration of the soundfile.
ifileseconds filelen      "%s"
p3           =             ifileseconds
        ; Simply play the soundfile to the output.
aleft, aright soundin     "%s"
        outs             aleft, aright
        endin
''' % (soundfileName, soundfileName))
        csound.setScore('i 1 1 1\n')
        csound.exportForPerformance()
        csound.perform()

print 'FINISHED.'
print

```

Rendering the Piece

You can find a pre-written version of the piece as `MusicProjects\01_DiceGame\wuerfelspiel.py`. You can run the piece either from the command line, or using IDLE, which comes with Python, or from a text editor such as SciTE as previously discussed, or from the Windows console.

The script must be run from the directory containing it, so that the relative paths used to load the SoundFonts used by Csound instruments, and any included Python modules, will all work. The piece should print out messages about what it is doing and terminate, leaving a soundfile player running and ready to play `wuerfelspiel.py.wav`.

Abstracting the Algorithm

The algorithm as presented completely captures the original musical dice game. You could easily make a copy of the program and change the code to make a different piece. However, this is not enough. It is not *abstract* enough.

In CsoundAC, it is possible to define a Python class that derives from a C++ class with virtual functions, so that when a C++ base class pointer actually points to an instance of a Python derived class (through a SWIG-generated *director*

object), calling the base class virtual C++ function invokes the Python overridden function.

Therefore derive a Python class, `DiceGameNode`, derived from `CsoundAC.Node`, and put the musical dice game code into a `DiceGameNode.generate` method, like this:

```
'''
MUSIKALISCHES WUERFELSPIEL OF 1787 AS A NODE
Implemented by Michael Gogins
6 November 2004
This code is in the public domain
'''

import CsoundAC
import random

## Inherit Python DiceGameNode class
## from CsoundAC.Node C++ class.

class DiceGameNode(CsoundAC.Node):
    def __init__(self):
        print 'DiceGameNode.__init__()'
        CsoundAC.Node.__init__(self)
        self.minuetTable = {}
        self.minuetTable[ 2] = { 1: 96, 2: 22, 3:141, 4: 41, 5:105, 6:122,
7: 11, 8: 30, 9: 70, 10:121, 11: 26, 12: 9, 13:112, 14: 49, 15:109, 16: 14}
        self.minuetTable[ 3] = { 1: 32, 2: 6, 3:128, 4: 63, 5:146, 6: 46,
7:134, 8: 81, 9:117, 10: 39, 11:126, 12: 56, 13:174, 14: 18, 15:116, 16: 83}
        self.minuetTable[ 4] = { 1: 69, 2: 95, 3:158, 4: 13, 5:153, 6: 55,
7:110, 8: 24, 9: 66, 10:139, 11: 15, 12:132, 13: 73, 14: 58, 15:145, 16: 79}
        self.minuetTable[ 5] = { 1: 40, 2: 17, 3:113, 4: 85, 5:161, 6: 2,
7:159, 8:100, 9: 90, 10:176, 11: 7, 12: 34, 13: 67, 14:160, 15: 52, 16:170}
        self.minuetTable[ 6] = { 1:148, 2: 74, 3:163, 4: 45, 5: 80, 6: 97,
7: 36, 8:107, 9: 25, 10:143, 11: 64, 12:125, 13: 76, 14:136, 15: 1, 16: 93}
        self.minuetTable[ 7] = { 1:104, 2:157, 3: 27, 4:167, 5:154, 6: 68,
7:118, 8: 91, 9:138, 10: 71, 11:150, 12: 29, 13:101, 14:162, 15: 23, 16:151}
        self.minuetTable[ 8] = { 1:152, 2: 60, 3:171, 4: 53, 5: 99, 6:133,
7: 21, 8:127, 9: 16, 10:155, 11: 57, 12:175, 13: 43, 14:168, 15: 89, 16:172}
        self.minuetTable[ 9] = { 1:119, 2: 84, 3:114, 4: 50, 5:140, 6: 86,
7:169, 8: 94, 9:120, 10: 88, 11: 48, 12:166, 13: 51, 14:115, 15: 72, 16:111}
        self.minuetTable[10] = { 1: 98, 2:142, 3: 42, 4:156, 5: 75, 6:129,
7: 62, 8:123, 9: 65, 10: 77, 11: 19, 12: 82, 13:137, 14: 38, 15:149, 16: 8}
        self.minuetTable[11] = { 1: 3, 2: 87, 3:165, 4: 61, 5:135, 6: 47,
7:147, 8: 33, 9:102, 10: 4, 11: 31, 12:164, 13:144, 14: 59, 15:173, 16: 78}
        self.minuetTable[12] = { 1: 54, 2:130, 3: 10, 4:103, 5: 28, 6: 37,
7:106, 8: 5, 9: 35, 10: 20, 11:108, 12: 92, 13: 12, 14:124, 15: 44, 16:131}

        self.trioTable = {}
        self.trioTable[ 1] = {17: 72, 18: 6, 19: 59, 20: 25, 21: 81, 22: 41,
23: 89, 24: 13, 25: 36, 26: 5, 27: 46, 28: 79, 29: 30, 30: 95, 31: 19, 32: 66}
        self.trioTable[ 2] = {17: 56, 18: 82, 19: 42, 20: 74, 21: 14, 22: 7,
23: 26, 24: 71, 25: 76, 26: 20, 27: 64, 28: 84, 29: 8, 30: 35, 31: 47, 32: 88}
```

```

        self.trioTable[ 3] = {17: 75, 18: 39, 19: 54, 20: 1, 21: 65, 22: 43,
23: 15, 24: 80, 25: 9, 26: 34, 27: 93, 28: 48, 29: 69, 30: 58, 31: 90, 32: 21}
        self.trioTable[ 4] = {17: 40, 18: 73, 19: 16, 20: 68, 21: 29, 22: 55,
23: 2, 24: 61, 25: 22, 26: 67, 27: 49, 28: 77, 29: 57, 30: 87, 31: 33, 32: 10}
        self.trioTable[ 5] = {17: 83, 18: 3, 19: 28, 20: 53, 21: 37, 22: 17,
23: 44, 24: 70, 25: 63, 26: 85, 27: 32, 28: 96, 29: 12, 30: 23, 31: 50, 32: 91}
        self.trioTable[ 6] = {17: 18, 18: 45, 19: 62, 20: 38, 21: 4, 22: 27,
23: 52, 24: 94, 25: 11, 26: 92, 27: 24, 28: 86, 29: 51, 30: 60, 31: 78, 32: 31}

```

```

def readMeasure(self, section, number):
    scoreNode = CsoundAC.ScoreNode()
    scoreNode.thisown = 0
    filename = section + str(number) + '.mid'
    print 'Reading: %s' % filename
    scoreNode.getScore().load(filename)
    return scoreNode, filename

```

```

def rollDice(self, throws):
    diceroll = 0
    for i in range(throws):
        diceroll += random.randint(1, 6)
    return diceroll

```

```

def generate(self):
    print 'BEGAN generate...'
    print 'First, select random measures for minuet and trio by dice roll and
bar number.'
    random.seed()
    measuresChosen = {}
    for bar in xrange( 1, 17):
        measuresChosen[bar] = self.minuetTable[self.rollDice(2)][bar]
    for bar in xrange(17, 33):
        measuresChosen[bar] = self.trioTable [self.rollDice(1)][bar]
    print 'Next, assemble the selected measures with appropriate repeats:'
    cumulativeTime = 0
    for repeat in xrange(1, 3):
        for barNumber in xrange(1, 17):
            measure, filename = self.readMeasure('M',
measuresChosen[barNumber])
            notes = len(measure.getScore())
            duration = measure.getScore().getDuration()
            print 'Repeat %d: Minuet bar %d measure M%d %d notes at %f
seconds' % (repeat, barNumber, measuresChosen[barNumber], notes, cumulativeTime)
            print measure.getScore().getCsoundScore()
            cumulativeTime += duration
            barTime = CsoundAC.Rescale()
            barTime.setRescale(0, 1, 0, cumulativeTime, 0)
            barTime.setRescale(5, 1, 1, 70, 10)
            barTime.thisown=0
            barTime.addChild(measure)
            self.addChild(barTime)
        for barNumber in xrange(17, 33):

```

```

measure, filename = self.readMeasure('T',
measuresChosen[barNumber])
    notes = len(measure.getScore())
    duration = measure.getScore().getDuration()
    print 'Repeat %d: Trio bar %d measure T%d %d notes at %f
seconds' % (repeat, barNumber, measuresChosen[barNumber], notes, cumulativeTime)
    print measure.getScore().getCsoundScore()
    cumulativeTime += duration
    barTime = CsoundAC.Rescale()
    barTime.setRescale(0, 1, 0, cumulativeTime, 0)
    barTime.setRescale(5, 1, 1, 70, 10)
    barTime.thisown=0
    barTime.addChild(measure)
    self.addChild(barTime)
print 'ENDED generate.'
```

The reason for abstracting the dice game to a `Node`, of course, is that later on you can use the musical dice game as a source of raw material for further algorithmic transformations by *other* nodes in the music graph. Then, you can do things like this:

```

print 'CREATING MUSIC MODEL...'
print

import DiceGameNode

model.setConformPitches(True)
diceGameNode = DiceGameNode.DiceGameNode()
diceGameNode.generate()
rescale = CsoundAC.Rescale()
rescale.addChild(diceGameNode)
rescale.setRescale(CsoundAC.Event.PITCHES, 1, 0, CsoundAC.Conversions_nameToM('Bb
bebop'), 1)
model.addChild(rescale)
```

In this case, the music generated by the `DiceGameNode` is transposed from its original key of D minor to a “Bb bebop” scale. It is not literally transposed, but rather its pitch-classes are permuted. This is a *very* basic example of what is possible. Multiple transformations of multiple copies and variations of the node could easily be performed.

You can find a pre-written version of the piece as `MusicProjects/01_DiceGame/wuerfelspiel2.py`. You can run the piece either from the command line, or using IDLE, which comes with Python, or from a text editor such as SciTE as previously discussed, or from the Windows console.

The script must be run from the directory containing it, so that the relative paths used to load the SoundFonts in the Csound instruments, and any included Python modules, will all work. The piece should print out messages about what it is doing and terminate, leaving a soundfile player running and ready to play the output soundfile.

By the way, the story of the musical dice game does not end in the 18th century! The measures of Mozart's game were used by John Cage and Lejaren Hiller as source material for their algorithmically composed piece *HPSCHD* (Cage & Hiller, 1969). And I use the 1787 measures again in a later section of this section, as source material for a tutorial on using the minimalist technique of cellular accretion in algorithmic composition.

Lejaren Hiller, Bill Schottstaedt, and Counterpoint Generation

In 1959 Lejaren Hiller and Leonard Isaacson published a technical monograph, *Experimental Composition* (Hiller & Isaacson, 1959), summarizing the results of several years' research at the University of Illinois on composing music with computer programs. This book is a founding text of algorithmic composition (and indeed of computer music as such, since it predates by a year or so Max Mathews' work at Bell Laboratories on synthesizing sound with computer programs (Mathews, 1969)).

Hiller and Isaacson used information theory as the conceptual basis for their work. Although information theory remains valid and relevant for computer music, it is no longer the most important or most widely used basis for algorithmic composition. Nevertheless, *Experimental Composition* is still essential reading because of the deep thought its authors gave to fundamental problems of algorithmic composition. All of the algorithms and approaches invented in this book are still in use today.

Hiller and Isaacson worked with the Illiac I computer, a room-filling, vacuum-tube beast somewhat less powerful than a programmable pocket calculator is today. Still, even a programmable calculator can multiply quite a bit faster than an unaided person!

The fruit of Hiller and Isaacson's researches is the *Illiac Suite* of 1957 (Hiller & Isaacson, 1959a) for string quartet. They made this piece using a toolkit of probabilistic, information-theoretic algorithms, with one master algorithm per each of 4 "Experiments," or movements, in the quartet, plus a "Coda." To my

ears, the most fascinating part of the suite is “Experiment Two”, in which Hiller and Isaacson began with randomly generated *cantus firmi* and used an algorithm to compose four-part, first-species counterpoints to them. As the movement progresses, I find it interesting in the extreme to hear the counterpoint become increasingly refined with the addition of one of Fux's (Mann, 1971 [1943]) rules after another to the algorithm, until the music becomes entirely smooth and correct.

Hiller and Isaacson's book does not contain source code and if it did, it would be in an obsolete language. In an effort to come up with a more modern approach to counterpoint generation in the spirit of Hiller, I found Bill Schottstaedt's report (and program) on *Automatic Species Counterpoint* (Schottstaedt, 1984). The original program is also in a language nobody uses any more, SAIL, but Schottstaedt has translated the algorithm to C, and he has graciously given me permission to use his C code here.

I have used Schottstaedt's code to make a new `CsoundAC.Node` class, `CounterpointNode`, which can extract a *cantus firmus* from a previously generated score, and then generate a counterpoint to fit the *cantus*.

In adapting Schottstaedt's code, I made every effort to not change any of the program logic at all, since I know from experience how easy it is to break somebody else's code by modifying it, and how hard it is to figure out what went wrong. I made one change at a time, and I tested the program after each change to ensure that it continued to produce the same results. I changed Schottstaedt's code as follows:

1. I changed the counterpoint generator from a set of C functions to a C++ class. This was done merely by enclosing all of Schottstaedt's original C functions as member functions of a single `Counterpoint` C++ class.
2. I changed all symbolic constants in Schottstaedt's C code from `#define` directives into C++ `enum` declarations. I changed those that needed to be configurable (such as the maximum number of voices and notes) to `int` member variables.
3. In particular, the original code used symbolic constants to specify the numerical weights used to penalize breaking various rules of counterpoint. I changed these numbers to member variables, so that the user of the code could change the weights.

4. I changed all data storage for the *cantus firmus*, the generated counterpoint, and intermediate results of calculations into dynamically allocated C++ matrixes and vectors. Fortunately, Schottstaedt's original C functions do not assume any fixed sizes for storage arrays.
5. I created two versions of the code based on the Counterpoint class, first a stand-alone counterpoint program that generates a Csound score, and then the CsoundAC.CounterpointNode version. The stand-alone program made it easier to debug my changes and test the limits of the algorithm.
6. The original code has a function named fux that is designed to produce a counterpoint for a given *cantus firmus*, species, mode, number of voices, and choice of beginning note in each voice. I wrote a new function, Counterpoint::counterpoint, that follows the same logic:

```
void Counterpoint::counterpoint(int OurMode, int *StartPitches, int CurV, int
cantuslen, int Species, int *cantus)
{
    initialize((cantuslen * 8) + 1, CurV + 1);
    if(StartPitches)
    {
        for (int i = 0; i < CurV; i++)
        {
            vbs[i] = StartPitches[i];
        }
    }
    int i;
    for (i=1;i<=cantuslen;i++)
    {
        Ctrpt(i,0) = cantus[i-1];
    }
    for (i=0;i<3;i++)
    {
        Fits[i]=0;
    }
    AnySpecies(OurMode,&vbs[0],CurV,cantuslen,Species);
}
```

7. In the new CounterpointNode class, I was then able to write CounterpointNode::produceOrTransform to override the base class Node::produceOrTransform function as follows:

```
void CounterpointNode::produceOrTransform(Score &score, size_t beginAt, size_t
endAt, const ublas::matrix<double> &globalCoordinates)
{
    // Make a local copy of the child notes.
```

```

Score source;
source.insert(source.begin(), score.begin() + beginAt, score.begin() + endAt);
System::message("Original source notes: %d\n", source.size());
// Remove the child notes from the target.
score.erase(score.begin() + beginAt, score.begin() + endAt);
// Select the cantus firmus.
source.sort();
std::vector<int> cantus;
std::vector<int> voicebeginnings(voices);
// Take notes in sequence, quantized on time, as the cantus.
// If there are chords, pick the best fitting note in the chord and discard
the others.
std::vector< std::vector<int> > chords;
double time = -1;
double oldTime = 0;
for (size_t i = 0; i < source.size(); i++)
{
    oldTime = time;
    time = std::floor((source[i].getTime() / secondsPerPulse) + 0.5) *
secondsPerPulse;
    if (oldTime != time)
    {
        std::vector<int> newchord;
        chords.push_back(newchord);
    }
    chords.back().push_back(int(source[i].getKey()));
}
for(size_t i = 0, n = chords.size(); i < n; i++)
{
    int bestfit = chords[i].front();
    int oldDifference = 0;
    for(size_t j = 1; j < chords[i].size(); j++)
    {
        int difference = std::abs(bestfit - chords[i][j]);
        oldDifference = difference;
        if (difference > 0 && difference < oldDifference)
        {
            bestfit = chords[i][j];
        }
    }
    cantus.push_back(bestfit);
}
System::message("Cantus firmus notes: %d\n", source.size());
if(voiceBeginnings.size() > 0)
{
    voicebeginnings.resize(voices);
    for (size_t i = 0; i < voices; i++)
    {
        voicebeginnings[i] = voiceBeginnings[i];
    }
}

```



```

        System::message("Voice %d begins at key %d\n", (i + 1),
voicebeginnings[i]);
    }
}
else
{
    voicebeginnings.resize(voices);
    int range = HighestSemitone - LowestSemitone;
    int voicerange = range / (voices + 1);
    System::message("Cantus begins at key %d\n", cantus[0]);
    int c = cantus[0];
    for (size_t i = 0; i < voices; i++)
    {
        voicebeginnings[i] = c + ((i + 1) * voicerange);
        System::message("Voice %d begins at key %d\n", (i + 1),
voicebeginnings[i]);
    }
}
// Generate the counterpoint.
counterpoint(musicMode, &voicebeginnings[0], voices, cantus.size(), species,
&cantus[0]);
// Translate the counterpoint back to a Score.
double duration = 0.;
double key = 0.;
double velocity = 70.;
double phase = 0.;
double x = 0.;
double y = 0.;
double z = 0.;
double pcs = 4095.0;
Score generated;
for(size_t voice = 0; voice <= voices; voice++)
{
    double time = 0;
    for(int note = 1; note <= TotalNotes[voice]; note++)
    {
        time = double(Onset(note,voice));
        time *= secondsPerPulse;
        duration = double(Dur(note,voice));
        duration *= secondsPerPulse;
        key = double(Ctrpt(note,voice));
        // Set the exact pitch class so that something of the counterpoint
will be preserved if the tessitura is rescaled.
        pcs = Conversions::midiToPitchClass(key);
        System::message("%f %f %f %f %f %f %f %f %f %f\n", time, duration,
double(144), double(voice), key, velocity, phase, x, y, z, pcs);
        generated.append(time, duration, double(144), double(voice), key,
velocity, phase, x, y, z, pcs);
    }
}
// Get the right coordinate system going.

```

```

        System::message("Total notes in generated counterpoint: %d\n",
generated.size());
    ublas::matrix<double> localCoordinates = getLocalCoordinates();
    ublas::matrix<double> compositeCoordinates = getLocalCoordinates();
    ublas::axpy_prod(globalCoordinates, localCoordinates, compositeCoordinates);
    Event e;
    for (int i = 0, n = generated.size(); i < n; i++)
    {
        ublas::axpy_prod(compositeCoordinates, generated[i], e);
        generated[i] = e;
    }
    // Put the generated counterpoint (back?) into the target score.
    score.insert(score.end(), generated.begin(), generated.end());
    // Free up memory that was used.
    Counterpoint::clear();
}

```

Generating the Score

In “Experiment Two” of the Illiac Suite, randomly generated *cantus firmi* have counterpoints generated for them algorithmically. As the music progresses, more and more of Fux's rules are added to the counterpoint generator, until at the conclusion of the piece the counterpoint sounds smooth and competent. Let us write a piece of our own along these lines.

1. Write a Python function to generate a random *cantus firmus* of 8 whole notes. The `CsoundAC.Random` node can easily do this. Place each `Random` node inside a `CsoundAC.Rescale` node, and use it to conform the randomly generated pitches to a named pitch-class set, in this case the C major scale. Make sure that the first and last notes of each *cantus* are the tonic of the scale.

```

randomNode = CsoundAC.Random()
print randomNode.minimum
print randomNode.maximum
randomNode.createDistribution("uniform_real")
now = int(time.time())
randomNode.seed(now)

cmajor = CsoundAC.Conversions_nameToM("C major")
wholeNote = 1.0

def createCantus(notes):
    scoreNode = CsoundAC.ScoreNode()
    scoreNode.thisown = 0
    tonic = 36.0
    event = None
    for i in xrange(notes):

```

```

event = CsoundAC.Event()
event.thisown = 0
event.setTime(float(i))
event.setDuration(1.0)
event.setStatus(144)
event.setInstrument(1)
key = tonic + (randomNode.sample() * 12.0)
# print 'key:', key
event.setKey(key)
# Randomize the loudness a little.
event.setVelocity(70 + (randomNode.sample() * 8.0) - 4.0)
event.setPitches(cmajor)
event.conformToPitchClassSet()
# Make sure it ends on the tonic.
if i == (notes - 1):
    event.setKey(tonic)
scoreNode.getScore().append(event)
return scoreNode

```

2. Define a function to create a counterpoint node from a mode, species, number of voices, and list of voice beginnings. It should create a `CsoundAC.CounterpointNode`, assign the relevant properties, and set the voice beginnings.

```

def createCounterpointNode(mode, species, voices, voicebeginnings):
    counterpointNode = CsoundAC.CounterpointNode()
    counterpointNode.thisown = 0
    counterpointNode.LowestSemitone = int(24)
    counterpointNode.HighestSemitone = int(108)
    counterpointNode.musicMode = mode
    counterpointNode.species = species
    counterpointNode.voices = voices
    counterpointNode.secondsPerPulse = 1.0 / 4.0
    print 'Counterpoint : mode %d species %d voices %d' %
(counterpointNode.musicMode, counterpointNode.species, counterpointNode.voices)
    for key in voicebeginnings:
        counterpointNode.voiceBeginnings.append(int(key))
    return counterpointNode

```

3. Define a function to create a section: take a music model, a note count, a counterpoint node, a cumulative time pointer, the duration of a whole note, and a transposition factor; generate a *cantus* with the specified number of notes; add it to the counterpoint node; add the counterpoint node to a rescale node; use the rescale node to move the counterpoint to the appropriate point in time and the specified transposition; and return the new value of the time pointer:

```

def createSection(model, notes, counterpoint, cumulativeTime, wholeNote,
transposition):
    cantus = createCantus(notes)

```

```

print cantus.getScore().toString(),
counterpoint.addChild(cantus)
rescale = CsoundAC.Rescale()
rescale.thisown = 0
rescale.addChild(counterpoint)
football = wholeNote * 4.0 / counterpoint.secondsPerPulse
    rescale.setRescale(CsoundAC.Event.TIME, True, False, float(cumulativeTime),
0.0)
        rescale.setRescale(CsoundAC.Event.KEY,      True, False, float(36 +
transposition), 0.0)
model.addChild(rescale)
cumulativeTime += (notes * wholeNote * 2.0)
print 'Cumulative time is:',cumulativeTime
print
return cumulativeTime

```

4. Generate 13 of these counterpoints. For the first one, set all of the basic penalties in the CounterpointNode class to 0. For each successive counterpoint, set many of the penalties back to their default values. For the last 4 counterpoints, set all the weights back to their defaults. In the last counterpoint but two, generate a 2nd species counterpoint. For the last but one, generate 3rd species. For the very last, go back to 1st species. Here is the first section:

```

cumulativeTime = 0.0
sections = 0

voices = int(3)
notes = int(8)
species = int(1)
voiceBeginnings = [57, 62, 69, 84, 98]

sections = sections + 1
print 'SECTION', sections
counterpoint = createCounterpointNode(CsoundAC.CounterpointNode.Ionian, species,
voices, voiceBeginnings)
counterpoint.UnisonPenalty = 0 # counterpoint.Bad;
counterpoint.DirectToFifthPenalty = 0 # counterpoint.RealBad;
counterpoint.DirectToOctavePenalty = 0 # counterpoint.RealBad;
counterpoint.ParallelFifthPenalty = 0 # counterpoint.infinity;
counterpoint.ParallelUnisonPenalty = 0 # counterpoint.infinity;
counterpoint.EndOnPerfectPenalty = 0 # counterpoint.infinity;
counterpoint.NoLeadingTonePenalty = 0 # counterpoint.infinity;
counterpoint.DissonancePenalty = 0 # counterpoint.infinity;
counterpoint.infinity;
counterpoint.OutOfRangePenalty = 0 # counterpoint.RealBad;
counterpoint.OutOfModePenalty = 0 # counterpoint.infinity;
counterpoint.TwoSkipsPenalty = 0 # 1;
counterpoint.DirectMotionPenalty = 0 # 1;
counterpoint.PerfectConsonancePenalty = 0 # 2;

```

```

counterpoint.CompoundPenalty           = 0 # 1;
counterpoint.TenthToOctavePenalty      = 0 # 8;
counterpoint.SkipTo8vePenalty          = 0 # 8;
counterpoint.SkipFromUnisonPenalty     = 0 # 4;
counterpoint.SkipPrecededBySameDirectionPenalty = 0 # 1;
counterpoint.FifthPrecededBySameDirectionPenalty = 0 # 3;
counterpoint.SixthPrecededBySameDirectionPenalty = 0 # 8;
counterpoint.SkipFollowedBySameDirectionPenalty = 0 # 3;
counterpoint.FifthFollowedBySameDirectionPenalty = 0 # 8;
counterpoint.SixthFollowedBySameDirectionPenalty = 0 # 34;
counterpoint.TwoSkipsNotInTriadPenalty = 0 # 3;
counterpoint.BadMelodyPenalty          = 0 # counterpoint.infinity;
counterpoint.ExtremeRangePenalty       = 0 # 5;
counterpoint.LydianCadentialTritonePenalty = 0 # 13;
counterpoint.UpperNeighborPenalty      = 0 # 1;
counterpoint.LowerNeighborPenalty      = 0 # 1;
counterpoint.OverTwelfthPenalty        = 0 #
counterpoint.infinity;
counterpoint.OverOctavePenalty          = 0 # counterpoint.Bad;
counterpoint.SixthLeapPenalty           = 0 # 2;
counterpoint.OctaveLeapPenalty         = 0 # 5;
counterpoint.BadCadencePenalty         = 0 #
counterpoint.infinity;
counterpoint.DirectPerfectOnDownbeatPenalty = 0 # counterpoint.infinity;
counterpoint.RepetitionOnUpbeatPenalty  = 0 # counterpoint.Bad;
counterpoint.DissonanceNotFillingThirdPenalty = 0 # counterpoint.infinity;
counterpoint.UnisonDownbeatPenalty     = 0 # 3;
counterpoint.TwoRepeatedNotesPenalty   = 0 # 2;
counterpoint.ThreeRepeatedNotesPenalty = 0 # 4;
counterpoint.FourRepeatedNotesPenalty  = 0 # 7;
counterpoint.LeapAtCadencePenalty       = 0 # 13;
counterpoint.NotaCambiataPenalty       = 0 # counterpoint.infinity;
counterpoint.NotBestCadencePenalty     = 0 # 8;
counterpoint.UnisonOnBeat4Penalty      = 0 # 3;
counterpoint.NotaLigaturePenalty       = 0 # 21;
counterpoint.LesserLigaturePenalty     = 0 # 8;
counterpoint.UnresolvedLigaturePenalty = 0 #
counterpoint.infinity;
counterpoint.NoTimeForaLigaturePenalty = 0 #
counterpoint.infinity;
counterpoint.EighthJumpPenalty         = 0 # counterpoint.Bad;
counterpoint.HalfUntiedPenalty         = 0 # 13;
counterpoint.UnisonUpbeatPenalty       = 0 # 21;
counterpoint.MelodicBoredomPenalty     = 0 # 1;
counterpoint.SkipToDownBeatPenalty     = 0 # 1;
counterpoint.ThreeSkipsPenalty         = 0 # 3;
counterpoint.DownBeatUnisonPenalty     = 0 # counterpoint.Bad;
counterpoint.VerticalTritonePenalty     = 0 # 2;
counterpoint.MelodicTritonePenalty     = 0 # 8;
counterpoint.AscendingSixthPenalty     = 0 # 1;
counterpoint.RepeatedPitchPenalty      = 0 # 1;

```

```

counterpoint.NotContraryToOthersPenalty          = 0 # 1;
counterpoint.NotTriadPenalty                     = 0 # 34;
counterpoint.InnerVoicesInDirectToPerfectPenalty = 0 # 21;
counterpoint.InnerVoicesInDirectToTritonePenalty = 0 # 13;
counterpoint.SixFiveChordPenalty                 = 0 # counterpoint.infinity;
counterpoint.UnpreparedSixFivePenalty            = 0 # counterpoint.Bad;
counterpoint.UnresolvedSixFivePenalty            = 0 # counterpoint.Bad;
counterpoint.AugmentedIntervalPenalty            = 0 # counterpoint.infinity;
counterpoint.ThirdDoubledPenalty                 = 0 # 5;
counterpoint.DoubledLeadingTonePenalty            = 0 #
counterpoint.infinity;
counterpoint.DoubledSixthPenalty                 = 0 # 5;
counterpoint.DoubledFifthPenalty                 = 0 # 3;
counterpoint.TripledBassPenalty                  = 0 # 3;
counterpoint.UpperVoicesTooFarApartPenalty       = 0 # 1;
counterpoint.UnresolvedLeadingTonePenalty          = 0 # counterpoint.infinity;
counterpoint.AllVoicesSkipPenalty                = 0 # 8;
counterpoint.DirectToTritonePenalty              = 0 # counterpoint.Bad;
counterpoint.CrossBelowBassPenalty               = 0 # counterpoint.infinity;
counterpoint.CrossAboveCantusPenalty             = 0 # counterpoint.infinity;
counterpoint.NoMotionAgainstOctavePenalty        = 0 # 34;
cumulativeTime = createSection(model, notes, counterpoint, cumulativeTime,
wholeNote, 0) + 1.0

```

And here is the final section:

```

sections = sections + 1
print 'SECTION', sections
voices = int(3)
notes = int(8)
species = int(1)
counterpoint = createCounterpointNode(CsoundAC.CounterpointNode.Ionian, species,
voices, voiceBeginnings)
cumulativeTime = createSection(model, notes, counterpoint, cumulativeTime,
wholeNote, 0) + 1.0

```

5. Now create an arrangement to render the piece.

```

print 'CREATING CSOUND ARRANGEMENT...'
print

model.setCsoundOrchestra(csoundOrchestra)
model.setCsoundScoreHeader(csoundScoreHeader)
#          oldinsno, newinsno, level (+-dB), pan (-1.0 through +1.0)
model.arrange( 1,      13,      0.0,      0.0 )
model.arrange( 2,       6,      0.0,     -0.7 )
model.arrange( 3,       6,      0.0,      +0.7 )
model.setCsoundCommand(csoundCommand)

```

Rendering the Piece

You can find a pre-written version of the piece as `MusicProjects\02_Counterpoint\counterpoint.py`. You can run the piece either from the command line, or using IDLE, which comes with Python, or from a text editor such as SciTE as previously discussed, or from the Windows console.

The script must be run from the directory containing it, so that the relative paths used to load the SoundFonts in the Csound instruments, and any included Python modules, will all work. The piece should print out messages about what it is doing and terminate, leaving a soundfile player running and ready to play `counterpoint.py.wav`.

This piece uses a great deal of time and memory, and it might even fail on some computers. Some of the more demanding sections have been commented out in the script, to make sure that it will run to the end. If you have a powerful computer, you may wish to try un-commenting these sections to render the entire piece..

I find that if Hiller's algorithm runs without any rules, the counterpoint is generated at random pitches, whereas if Schottstaedt's algorithm runs without any weights, the counterpoint is generated using a deterministic lookup table. As the weights are added, the notes from the table are moved around. With most of the weights in place, the Schottstaedt algorithm produces results that sound quite similar to the Hiller algorithm, and the generated counterpoint sometimes sounds completely finished. The program managed to generate good-sounding 2nd species counterpoint, but it gave up after the first few notes of 3rd species.

This and other experiments with the `counterpoint` program and the `CounterpointNode` demonstrate the following results of interest:

1. The counterpoint generator can indeed generate correct-sounding counterpoint – but only for relatively short *cantus firmi*. This is perhaps not surprising, since the texts upon which the code is based universally use examples of 10 or so whole notes. Also, Schottstaedt's algorithm involves selecting a potential solution to the counterpoint, attempting to generate a counterpoint that ends with a cadence, evaluating its fitness according to weights for each rule and how badly it is broken, and then re-starting with a different initial note if a good solution cannot be found, but saving prior results on a stack for comparison. This recursive strategy is designed to

cut down the combinatorial explosion of paths to be searched, but it cannot entirely eliminate the exponential nature of the problem, nor can it guarantee a solution. Indeed, the rules of counterpoint themselves cannot guarantee a solution for an arbitrary *cantus firmus*.

2. Consequently, although I have extended the potential duration of the *cantus firmus* indefinitely, the longer the *cantus* is, and the higher the species of counterpoint or the number of voices, the more likely the generator is to fail. It fails in several different ways:
 1. Because it is recursive, sometimes the search goes so deep that it uses all available stack space or memory, causing the program to crash.
 2. The generator sometimes fails to generate the upper voices of the counterpoint, presumably because the search runs into a “brick wall” – i.e., beginning with the initial note for a voice, none of the 12 possible choices for the next voice leads to an acceptable solution.
 3. The generator sometimes simply runs on and on. It is impossible to tell if it is going in circles, or if the search time has exploded to an impractical duration thanks to the combinatorial explosion of possible routes to a cadence.
3. The texture of species counterpoint is pleasing, but only within a narrow range. It does not produce anything like the gamut of rhythms and textures that can be generated algorithmically in other ways, or that a composer might wish to use in his or her music.
4. The algorithm takes no account of imitative techniques such as canon or fugue, or serial techniques such as inversion and retrograde, which are almost as important in music history as the techniques of voice leading.

In spite of these limitations, the algorithm has very real musical usefulness. It can often generate pleasing counterpoints with correct voice leading for notated or algorithmically generated *cantus firmi* ranging in length from 8 to 100 or so notes. This might prove handy in a number of contexts.

The algorithm is also quite useful in demonstrating some important current limitations of the *imitative approach* to algorithmic composition. In other words, human thought is poorly understood and in some areas it is obviously much more effective than current computer programs. In fact, using a computer to do in a limited fashion what a trained human being can do extremely well may not

seem the wisest use of the computer itself, or of the computer programmer's time.

But it is important to try, for in no other way will such limitations become apparent. Refining the algorithmic imitation of human procedures is an important part of the *scientific* aspect of algorithmic composition.

Some recent work in counterpoint generation has involved evolutionary computing, but I have not seen any evidence that the results are more useful than the examples here.

Cage and Transposition from Nature

John Cage changed music by composing without hearing music or, at least, without *pre*-hearing it. He did this in many ways, but one of the notable pieces in this vein is *Atlas Eclipticalis* for chamber ensemble (Cage, *Atlas Eclipticalis*, 1961). Cage took a star atlas for the middle band of the sky – where the planets wander and the constellations of the zodiac are found – and overlaid sections of this atlas onto staff paper. He transcribed each star as a note of music. The brightness of each star became the loudness of the note. Cage also added durations for the notes and sections, and instructions for assigning them to players.

For a long time I was skeptical of the musical worth of such methods, but after hearing the right musicians play Cage, I was astonished to find it as compelling as any other music. In fact it has become a deep influence on my own work. Cage's work proves that even if the polyphonic and harmonic techniques of Western art music should fall into disuse, serious music will always have many other fields in which to flower.

Most of Cage's music is not computer music – with the notable exception of *HPSCHD* composed in collaboration with Lejaren Hiller (Cage & Hiller, *HPSCHD*, 1969) – but, obviously, much of it has a distinctly algorithmic component. Anyone can take a star chart and make a piece the same way Cage did. And each piece will be different.

Cage's approach could be called “transposition of nature into music,” and many pieces of modern and contemporary music have been made this way.

Yet one suspects that musical experience and taste *must* have a profound influence on just how the whole procedure works out: from choosing particularly interesting parts of the atlas, to making the cutouts the right size and proportion,

to laying them down just so on the staff paper, and so on, most especially not forgetting the orchestration.

It is highly instructive to follow a similar procedure in CsoundAC.

Ours is a golden age of observational astronomy, and the amount of star catalogues, images, and tables of data from terrestrial and extra-terrestrial astronomical telescopes and other instruments is simply staggering. The Centre de Donnees astronomiques de Strasbourg (CDS) (Centre de Donnees astronomiques de Strasbourg, 2006) provides a central clearing house for this data, and some Web services for accessing it. Anyone interested in astronomy ought to know about this site. The services most relevant to the present purpose are:

1. VizieR, a service for interactively querying a large number of astronomical catalogues and returning the results to the local computer in various forms.
2. Aladin, a Java applet or application for interactively querying astronomical catalogues and images and viewing, plotting, or analyzing the results.

Generating the Score

Here we shall make a piece inspired by Cage's *Atlas Eclipticalis* and following a roughly similar procedure. We shall query CDS for lists of stars surrounding the brightest star in each of the 13 zodiacal constellations, download the lists of stars into delimited text files, and translate each of the brightest stars in the text files into SCORENODEs that we will place in RESCALE nodes and arrange by trial and error to make a piece. In more detail the procedure is as follows:

1. Identify the 13 zodiacal constellations and the brightest star in each one:

Constellation	Brightest star
Virgo	Spica
Libra	Alpha Libris
Scorpius	Alpha Scorpii
Ophiuchus	Rasalhague

Saggitarius	Rukbat
Capricorn	Alpha Capricornis
Aquarius	Alpha Aquarii
Pisces	Alpha Piscium
Aries	Alpha Arietis
Taurus	Aldebaran
Gemini	Pollux
Cancer	Alpha Cancri
Leo	Regulus

- For each of these stars, query the *All-Sky Compiled Catalog of 2.5 Million Stars* catalog using the VizieR service to find all stars in a 1 degree box centered on the star, and save the data to your disk as a tab-delimited text file. Name the files `Spica.tsv`, `Alpha_Libris.tsv`, and so on.
- Write a Python script to read the star data files, separate the fields, and translate the field values as follows:

Astronomical datum	Musical dimension
Right ascension	Time
Declination	MIDI key
Magnitude	MIDI velocity and duration
Spectral class	MIDI channel

- Create a Python table of these data files and how to arrange them (including arranging them in time, pitch range, and dynamic range), for the script to read:

```

print 'CREATING MUSIC MODEL...'
print

print 'Specifying star catalogs to use...'
sections = []
# 0      1      2      3      4      5      6      7      8
# Filename, start, duration, bass, range, softest, dynamic range, left, width
time = 0.
duration = 30.
sections.append(['Alpha_Piscium.tsv',          time, duration, 36., 60., 60., 15.,
                -.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Alpha_Arietis.tsv',        time, duration, 36., 60., 60., 15.,
                -.75, 1.5])

```

```

time = time + duration + 4
duration = 30.
sections.append(['Aldebaran.tsv',           time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Pollux.tsv',             time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Alpha_Cancri.tsv',       time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.

sections.append(['Spica.tsv',             time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Alpha_Libris.tsv',       time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Alpha_Scorprii.tsv',     time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Rasalhague.tsv',        time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Rukbat.tsv',           time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Alpha_Capricornis.tsv',  time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.
sections.append(['Alpha_Aquarii.tsv',      time, duration, 36., 60., 60., 15.,
-.75, 1.5])
time = time + duration + 4
duration = 30.

sections.append(['Regulus.tsv',          time, duration, 36., 60., 60., 15.,
-.75, 1.5])

```

5. Also create a Python table for specifying the Csound instrument number to assign to each spectral class:

```

instrumentsForSpectralTypes = {' ':0, '0':1, 'B':2, 'A':3, 'F':4, 'G':5, 'K':6,
'M':7}

```

6. Define a function to read a section of the star catalog table, and translate the stars into notes. The function should create a `CsoundAC.ScoreNode` to contain the notes. The function can read the file one line at a time, skipping over all non-data lines (those beginning with `--`). The data lines consist of fields separated by tab characters, so it is easy to split each line into fields and translate them into the fields of a musical note. Place each translated `ScoreNode` into a `Rescale` node in order to position it as specified by the section parameters. Then return the finished `Rescale` node.

```
def readCatalog(section):
    print 'SECTION', section
    score = CsoundAC.ScoreNode()
    score.thisown = 0
    f = file(section[0])
    # Read the file until we run into '--'
    # which marks the last line of non-data.
    while(True):
        line = f.readline()
        if string.find(line, '--') >= 0:
            # Read the file until done.
            # The fields we want are:
            # Right ascension    1
            # Declination        2
            # Visual magnitude    8
            # Spectral type      9
            while(True):
                line = f.readline()
                if not line:
                    break
                fields = string.split(line, '\t')
                if not len(fields) > 8:
                    break
                # print fields
                time = float(fields[1])
                key = float(fields[2])
                velocity = float(fields[8])
                pan = float(fields[6])
                if(len(fields) > 9):
                    instrument = float(instrumentsForSpectralTypes[fields[9][0]])
                else:
                    instrument = 8
                score.getScore().append(time, velocity * 0.001, 144.0, instrument,
                key, velocity, 0.0, pan)
            print score.getScore().toString()
            # Put the section into a Rescale node to position it in the piece.
            rescale = CsoundAC.Rescale()
```

```

        rescale.thisown = 0
        scoreTime = section[1]
        scoreDuration = section[2]
        shortest = 4.0
        durationRange = 8.0
        key = section[3]
        range = section[4]
        lowest = section[5]
        dynamicRange = section[6]
        leftmost = section[7]
        width = section[8]
        print 'time:          ',scoreTime
        print 'duration:       ',scoreDuration
        print 'shortest:        ',shortest
        print 'duration range:',durationRange
        print 'key:           ',key
        print 'range:          ',range
        print 'lowest:         ',lowest
        print 'dynamic range: ',dynamicRange
        print 'leftmost:      ',leftmost
        print 'width:         ',width
        print
        rescale.setRescale(CsoundAC.Event.TIME,          True, True, scoreTime,
scoreDuration)
        rescale.setRescale(CsoundAC.Event.DURATION,     True, True, shortest,
durationRange)
        rescale.setRescale(CsoundAC.Event.KEY,          True, True, key,
range)
        rescale.setRescale(CsoundAC.Event.VELOCITY,    True, True, lowest,
dynamicRange)
        rescale.setRescale(CsoundAC.Event.PAN,          True, True, leftmost,
width)
        rescale.addChild(score)
        return rescale

```

7. In the body of the script, assemble the sections into a single piece. Create a master `Rescale` `rescale` node to hold the sections, and add it to the `model`. For each section in the star catalog table, read the chart and add the finished sub-score to the master `rescale` node. Then add the `rescale` node to the music model.

```

print 'Assembling sections into a piece inside a master Rescale node...'
rescale = CsoundAC.Rescale()
model.addChild(rescale)
rescale.setRescale(CsoundAC.Event.DURATION,    True, True, 6.0, 8.0)
rescale.setRescale(CsoundAC.Event.INSTRUMENT, True, True, 1.0, 7.0)
rescale.setRescale(CsoundAC.Event.VELOCITY,   True, True, 57.0, 12.0)
for section in sections:
    subscore = readCatalog(section)
    rescale.addChild(subscore)

```

8. Now render the generated score using Csound. Use your musical judgment to create an arrangement of 12 Csound instruments in the orchestra. Here, I used SoundFont instruments (the fluid opcodes require a separate instrument definition to actually collect and output the sound; this is taken care of in the score header section).

```
print 'CREATING CSOUND ARRANGEMENT...'
print

model.setCsoundOrchestra(csoundOrchestra)
model.setCsoundScoreHeader(csoundScoreHeader)
#           oldinsno, newinsno, level (+-dB), pan (-1.0 through +1.0)
model.arrange( 0,      51 )
model.arrange( 1,      7 )
model.arrange( 2,     12 )
model.arrange( 3,     16 )
model.arrange( 4,     51 )
model.arrange( 5,      9 )
model.arrange( 6,     10 )
model.arrange( 7,     51 )
model.arrange( 8,     51 )
model.arrange( 9,     51 )
model.arrange(10,     51 )
model.arrange(11,     51 )
model.arrange(12,     51 )
model.arrange(13,     51 )
model.setCsoundCommand(csoundCommand)
```

Rendering the Piece

You can find a pre-written version of the piece as `MusicProjects\03_Zodiac\zodiac.py`. You can run the piece either from the command line, or using IDLE, which comes with Python, or from a text editor such as SciTE as previously discussed, or from the Windows console.

The script must be run from the directory containing it, so that the relative paths used to load the SoundFonts in the Csound instruments, and any included Python modules, will all work. The piece should print out messages about what it is doing and terminate, leaving a soundfile player running and ready to play `zodiac.py.wav`.

Test the piece by listening to it carefully, perhaps several times. If you are not satisfied, go back to step 1 and change the order of sections, or their durations, or ranges, instrument assignments, dynamic range, and so on, then re-generate the piece.

I believe you will find that a few hours of the “generate and test” cycle improves the music drastically, even though each individual decision might seem to have little obvious or foreseeable musical effect. The exercise might prove illuminating to some more traditionally-minded composers.

Although trial and error improves the music, it is instructive that the most pleasing arrangement that I ended up with had each of the 13 segments rescaled to exactly the same duration, range, and dynamic range. This seems to validate Cage's trust of chance and natural pattern over human choice in some situations – such an arrangement does not obscure the natural order with arbitrary human re-arrangements.

Yet, I re-ordered the constellations to begin with Pisces instead of Virgo, so that the densest and therefore loudest textures came about 3/4 of the way through the piece, in the constellation of Sagittarius where the Zodiac passes through the heart of the Milky Way, in line with the traditional arch form of music. This was definitely more pleasing. Also, finding the most effective duration for the total piece (tempo, density of texture) had a huge impact on the quality of the music.

In my opinion, these compositional decisions do not so much re-arrange the natural order, as serve to bring it into clearer focus.

LaMonte Young and Just Intonation

American composer LaMonte Young was a founder of the minimalist school. In a radical break with Western musical tradition, and particularly with the then-dominant serialist movement, the minimalists rejected key changes, the twelve-tone method, the sonata and variation forms, regular meters, and much else to focus on building music out of small, repeating elements that could interlock in surprising ways. This was and is an essentially algorithmic approach to composition.

At the very beginning, Young rejected not only all of the above, but even melody and the equally tempered scale. Young focused, instead, and obsessively, on long chords of notes tuned to just intonation according to very precisely specified ratios of prime numbers (but avoiding ratios of 5 or divisible by 5, which would produce intervals of a third or a sixth). In such chords, the partials of the overtone series interlock to produce surprising combination and difference tones that can form a sheen or scrim of virtual melodies and even counterpoints not at all present in the individual tones of the chord (Young, 2000).

It is important to note that although pieces based on drones and pieces based on interlocking cells may seem on the surface to be in very different styles, beneath the surface they share the same idea: repeat a regular pattern against another similar pattern of different duration. In the case of a drone, the regular patterns are sine tones of different frequencies, i.e. whose cycles are of different duration. In the case of something like Terry Riley's *In C* (Riley, 2007), the regular patterns are short musical motives, or cells, of different duration.

Young's pieces of his early period were either free improvisations on the chosen chord tones, or music consisting of the mere persistence of long loud tones in a reverberant space. Young came to prefer stable electronic oscillators to produce these tones, so this was electronic music.

Csound is, of course, ideally suited to produce this kind of music. Csound's oscillators are extremely precise, much more so than any electronic oscillator, Csound can synthesize tones with varying degrees of harmonic content or varying degrees of distortion, and Csound's high-quality digital reverberation can simulate many kinds of reverberant spaces.

The following piece generates 3 two minute drone chords in succession: a C minor seventh chord, a G dominant seventh chord, and a C minor seventh chord (all in just intonation, all without fifths, and all with septimal octaves (63/32)).

The chord tones are produced by the `poscil3` opcode, a high-quality digital oscillator, using a wavetable generated from the fundamental, 2nd partial and 5th partial.

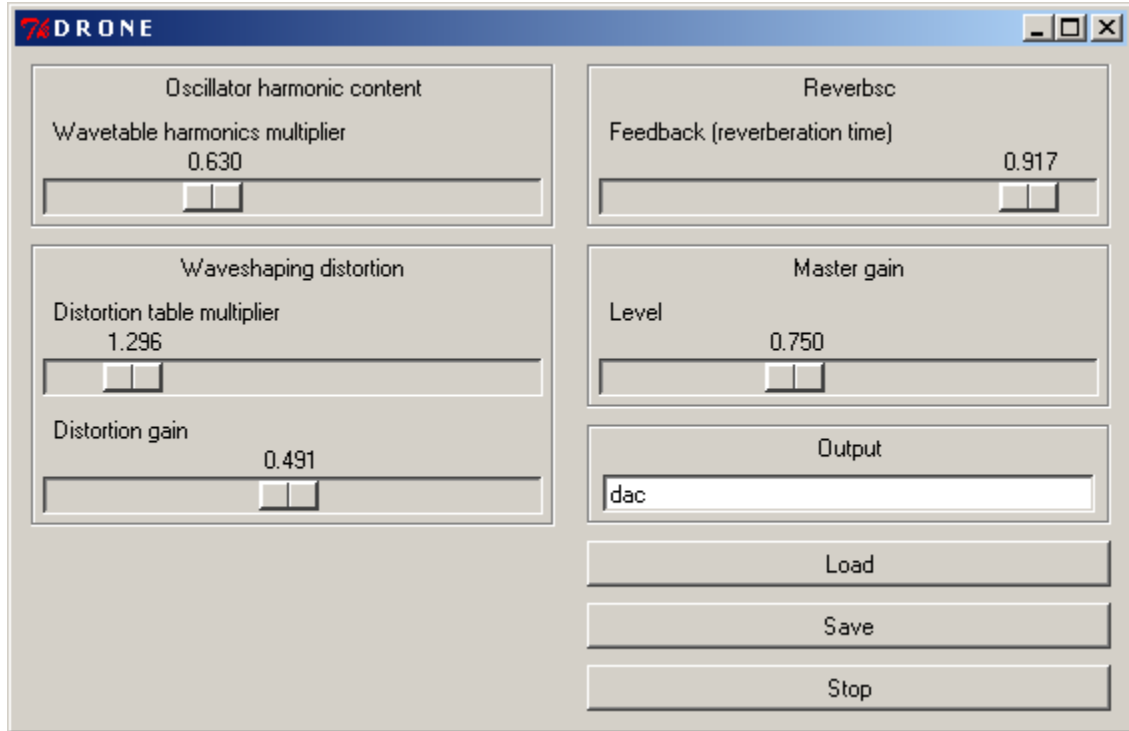
The chord tones are then modulated by the `distort` opcode, which uses a wavetable that is generated by a Chebyshev polynomial to produce a distorted reflection of the signal waveform.

Finally, the waveshaped tones are globally reverberated by the high-quality `reverb3c` opcode, which takes a feedback parameter that controls the reverberation time.

In this style of music, which consists of only a few long sustained chords, it is imperative to make certain that every detail of the production is exactly correct: the frequencies of the notes, the timbres of the sound generators, and the quality of the listening space.

To that end, it is very instructive to realize this piece in an interactive form. The musician can then use sliders to fine-tune the harmonic content of the

oscillator, the amount of waveshaping, and the amount of reverberation while the music is sounding. Small changes in these parameters turn out to have large effects on the effect of the piece. When the user has adjusted the parameters to his or her satisfaction, it should be possible to save the settings, restart the piece from the beginning, and so on, so that many possible realizations of the basic idea of the piece can be intensively explored in a short time. The user interface for the piece should look something like this:



Csound contains its own user interface toolkit, in the form of the FLTK opcodes, and it would quite possible to use the FLTK opcodes to realize this piece. However, those who are familiar with graphical user interface programming in other contexts may prefer to use a regular graphical user interface toolkit, which I personally find more intuitive. Regular toolkits also prove to be more powerful in the long run.

As it happens, Python comes with its own built-in graphical user interface toolkit (actually, a toolkit in several parts: the Tkinter, Tix, ScrolledText, and turtle modules). It is possible to make quite sophisticated graphical applications with this toolkit. In fact, the IDLE development environment that comes with Python is implemented with them. The user interface for this piece uses only a fraction of the kit.

In an application like this piece, it is wisest to start with the Csound orchestra and score, establish channels for the orchestra to communicate with the user interface, and then implement the user interface.

Orchestra, Score, and Command

The orchestra and score for this piece can be quite simple. There only needs to be one instrument, a simple wavetable oscillator with a very gradual envelope and some waveshaping distortion. Run the instrument output to global variables, run the global variables through more or less reverberation, and run the final result out to the sound card. Use this instrument to play 3 two-minute chords in just intonation. To achieve perfect just intonation, specify the pitch of each note as a fundamental tone multiplied by a frequency ratio. As Young did, use the omnipresent 60 cycle hum of our society as the fundamental tone. Also as Young did, avoid intervals divisible by 5. Since the parameters to the opcodes will need to be controlled from the user interface, declare them as global variables in the orchestra header. Naturally, the Csound orchestra, score, and command are defined as global variables in your `Drone.py` script:

```
csoundOrchestra = '''\
sr                =          44100
ksmps             =          100
nchnls            =           2
0dbfs             =         40000

gkDistortFactor    init      1.0
gkReverbScFeedback init      0.9
gkMasterLevel     init      1.0

gareverb1         init      0
gareverb2         init      0

                                instr 1
iattack           init      20
idecay            init      20
p3                =         p3 + (iattack + idecay) / 2.0
isustain          =         p3 - (iattack + idecay)
ifundamental      =         p4
inumerator        =         p5
idenominator      =         p6
ivelocity         =         p7
ipan              =         p8
iratio            =         inumerator / idenominator
ihertz           =         ifundamental * iratio
iamp              =         ampdb(ivelocity)
```

```

kenvelope      transeg      0.0, iattack, 0.0, iamp, isustain, 0.0, iamp,
idecay, 0.0, 0.0
asignal        poscil3      kenvelope, ihertz, 1
asignal        distort      asignal, gkDistortFactor, 2
gareverb1      =            gareverb1 + aleft
gareverb2      =            gareverb1 + aright
                endin

```

```

instr 30
aleft, aright      reverbsc      gareverb1, gareverb2, gkReverbscFeedback,
15000.0
aleft              =            gkMasterLevel * (gareverb1 + aleft * 0.8)
aright             =            gkMasterLevel * (gareverb2 + aright * 0.8)
                outs          aleft, aright
gareverb1          =            0
gareverb2          =            0
                endin
'''

```

```

csoundScore = '''\
;                A few harmonics...
f 1 0 8193 10 3 0 1 0 0 2
;                ...distorted by waveshaping.
f 2 0 8193 13 1 1 0 3 0 2
;                Change the tempo, if you like.
t 0 2

```

```

; p1 p2 p3 p4 p5 p6 p7
p8
; insno onset duration fundamental numerator denominator velocity
pan

;                Fundamental
i 1 0 60 60 1 1 60
0.75
;                Major whole tone
i 1 0 60 60 9 8 60
0.25
;                Septimal octave
i 1 0 60 60 63 32 60
0.00
;                Fundamental
i 1 60 60 53.5546875 1 1 63
0.75
;                Perfect fifth
i 1 60 60 53.5546875 3 2 62
0.25
;                Harmonic seventh
i 1 60 60 53.5546875 7 4 61
0.00
;                Fundamental

```

```

i 1 120 60 60 1 1 60
0.75
; Major whole tone
i 1 120 60 60 9 8 60
0.25
; Septimal octave
i 1 120 60 60 63 32 60
0.875
; Septimal octave
i 1 120 60 60 32 63 60
0.125

i 30 0 -1

s 10.0
e 10.0
'''

csoundCommand = '''\
csound -o %s temp.orc temp.sco
'''

```

Real-time Control

How, then, to get the user interface to control the orchestra while it is running?

By using the channel opcodes, which provide access to variables both from inside the running Csound orchestra, and from external software using the Csound API. *All* that is necessary is to bind each global variable to a control channel that has the same name, using the `chnexport` opcode:

```

; Bind named control channels to global variables.

gkDistortFactor      chnexport  "gkDistortFactor", 3
gkDistortFactor      init       1.0
gkReverbFeedback     chnexport  "gkReverbFeedback", 3
gkReverbFeedback     init       0.9
gkMasterLevel        chnexport  "gkMasterLevel", 3
gkMasterLevel        init       1.0

```

Then a simple API call can set (or get) the value of any exported global variable by name while the orchestra is running:

```

def on_gkReverbFeedback(self, value):
    self.csound.SetChannel("gkReverbFeedback", float(value))
    self.configuration.gkReverbFeedback = value

```

By using the `CppSound.inputMessage` API, which permits external software to send any score statement to a running orchestra for instantaneous or deferred execution. In this case, we will send new function tables to the orchestra

in order to change both the harmonic content of the oscillator, and the shape of the waveshaping function:

```
def on_gkHarmonicTableFactor(self, value):
    f = float(value)
    self.configuration.gkHarmonicTableFactor = f
    message = 'f 1 0 8193 10 3 0 %f 0 0 %f\n' % (f * 1.0, f * 2.0)
    self.csound.inputMessage(message)
```

As it happens, with a fast computer, small enough wavetables, and a reasonable kperiod such as 100, function tables can be replaced in a running orchestra without causing any clicks or other noise. Of course, the message could also have been a note, a tempo change, or any other score statement.

To simplify the code and reduce the number of variable names that need to be remembered, in the following let us adopt a convention of using the name of the global Csound variable not only for the name of its control channel, but also for its associated Tkinter widget and widget callback.

User Interface

The most rudimentary possible Tkinter program in Python looks like this:

```
from Tkinter import *

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.label = Label(
            self,
            text = 'Hello, world!'
        )
        self.label.grid(row=0, column=0, sticky=N+E+W+S)

tk = Tk()
application = Application(master=tk)
application.mainloop()
```

Add an instance of Csound to this application. For this application, we do not need CsoundAC; we can use the csnd module directly.

```
import csnd
from Tkinter import *

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        master.title('D R O N E')
        self.pack()
```

```

self.csound = csnd.CppSound()
self.csound.setPythonMessageCallback()

```

Let us modify this program to add all the widgets shown in the previous figure. We need to manage the layout of these widgets. In Tkinter, the easiest way to do that is to use a grid layout, and to put groups of widgets inside forms that are grouped inside larger forms, and so on. Here is the program with the two columns of the layout and the first slider in each column.

```

class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        master.title('D R O N E')
        self.pack()

        self.csound = csnd.CppSound()
        self.csound.setPythonMessageCallback()
        self.playing = False

        self.configuration = Configuration()

        r = 0
        c = 0

        self.leftFrame = Frame(self)
        self.leftFrame.grid(row=r, column=c, sticky=N+E+W, padx=4, pady=4)

        self.harmonicsFrame = Frame(
            self.leftFrame,
            bd = 2,
            relief = 'groove'
        )
        self.harmonicsFrame.grid(row=r, column=c, sticky=N+E+W+S, padx=4, pady=4)

        self.harmonicsLabel = Label(
            self.harmonicsFrame,
            text = 'Oscillator harmonic content'
        )
        self.harmonicsLabel.grid(row=r, column=c, sticky=N+E+W+S)
        r = r + 1

        self.gkHarmonicTableFactor = Scale(
            self.harmonicsFrame,
            from_ = 0.0,
            to = 2.0,
            resolution = 0.001,
            length = 250,
            orient = HORIZONTAL,

```

```

        label = 'Wavetable harmonics multiplier',
        command = self.on_gkHarmonicTableFactor
    )
self.gkHarmonicTableFactor.grid(row=r, column=c, sticky=E+W)
r = r + 1

r = 0
c = 1

self.rightFrame = Frame(self)
self.rightFrame.grid(row=r, column=c, sticky=N+E+W, padx=4, pady=4)

self.reverbFrame = Frame(
    self.rightFrame,
    bd = 2,
    relief = 'groove'
)
self.reverbFrame.grid(row=r, column=c, sticky=N+E+W, padx=4, pady=4)

self.reverbLabel = Label(
    self.reverbFrame,
    text = 'Reverb'
)
self.reverbLabel.grid(row=r, column=c, sticky=E+W)
r = r + 1

self.gkReverbFeedback = Scale(
    self.reverbFrame,
    from_ = 0.0,
    to = 0.9999,
    resolution = 0.001,
    length = 250,
    orient = HORIZONTAL,
    label = 'Feedback (reverberation time)',
    command = self.on_gkReverbFeedback
)
self.gkReverbFeedback.grid(row=r, column=c, sticky=E+W)
r = r + 1

```

Note that the various properties of each widget are passed to the widget constructor as named arguments; this is a convention in Tkinter. Then set the layout position using the `grid` function. You can simply increment the row number, so that you don't have to keep it straight in the code; reset the row number to 0 when you start the second column.

Go ahead and add the other widgets. Each slider and button gets a callback.

```
def on_play(self):
```



```

def on_load(self):

def on_save(self):

def on_gkHarmonicTableFactor(self, value):

def on_gkDistortTableFactor(self, value):

def on_gkDistortFactor(self, value):

def on_gkReverbscFeedback(self, value):

def on_gkMasterLevel(self, value):

```

Implementing the Callbacks

Programs with graphical user interfaces are what is called *event-driven*. The program just idles in circles (called the main loop) until the user does something with a widget. Then the main loop catches an event from the widget and dispatches it to the registered callback. All the actual work in such a program is done in the widget callbacks. Thus, we write our program by implementing the widget callbacks.

Begin with saving and restoring the run-time configuration. Use Python's facility for saving and restoring the contents of any Python object to or from a file, which is called *pickling*. Gather all the real-time control variables into a simple Python class named `Configuration`:

```

'''
A class that contains all control channel configuration values,
in order to simplify saving and restoring configurations.
Naming convention:
Csound global variable name equals Csound control channel name,
equals Tkinter widget name, equals configuration variable name,
and the widget command handler name is the same but prefixed with "on_".
'''

class Configuration(object):
    def __init__(self):
        self.gkHarmonicTableFactor = 0.5
        self.gkDistortTableFactor = 0.5
        self.gkDistortFactor = 0.125
        self.gkReverbscFeedback = 0.8
        self.gkMasterLevel = 0.25
        self.output = 'dac'

```

Add an instance of this class to your application:

```

self.configuration = Configuration()

```

Of course, you need a function to apply the values of the configuration variables to your widgets when the configuration is restored. Note how using a consistent naming convention makes the code easier to read.

```
'''
Set initial control channel values.
'''
def configure(self):
    self.gkHarmonicTableFactor.set(self.configuration.gkHarmonicTableFactor)
    self.gkDistortTableFactor.set(self.configuration.gkDistortTableFactor)
    self.gkDistortFactor.set(self.configuration.gkDistortFactor)
    self.gkReverbScFeedback.set(self.configuration.gkReverbScFeedback)
    self.gkMasterLevel.set(self.configuration.gkMasterLevel)
    self.outputStringVar.set(self.configuration.output)
```

The `tkFileDialog` module provides easy to use file dialogs. Use them in the `on_save` and `on_load` callbacks to get filenames for opening files to pickle and unpickle the configuration object:

```
def on_load(self):
    try:
        filename = tkFileDialog.askopenfilename(filetypes=[('Drone files',
            '*.pickled'), ('All files', '*')])
        if filename:
            picklefile = open(filename, 'rb')
            self.configuration = pickle.load(picklefile)
            picklefile.close()
            self.configure()
            print 'Loaded configuration: "%s".' % filename
    except:
        traceback.print_exc()

def on_save(self):
    try:
        filename = tkFileDialog.asksaveasfilename(filetypes=[('Drone files',
            '*.pickled'), ('All files', '*')])
        if filename:
            picklefile = open(filename, 'wb')
            pickle.dump(self.configuration, picklefile)
            picklefile.close()
            print 'Saved configuration: "%s".' % filename
    except:
        traceback.print_exc()
```

The `on_play` callback will load the orchestra, score, and command into the `csound` object, export the score and orchestra for performance, compile the orchestra, and apply the initial configuration. Then – and this is very important –

when using the Tkinter toolkit,² the application must take turns running the Csound orchestra, and dispatching Tkinter events; this is done by calling the Csound API `CppSound.performKsmps(False)`, which runs one kperiod of the Csound orchestra, followed by the Tkinter API `Frame.update()`, which dispatches any pending widget callbacks, in a tight loop. The loop exits either when the `self.playing` flag goes to `False`, or when `performKsmps` returns `True`. Note that after compiling the orchestra, and before actually compiling, the application sets the values of all the global variables in the Csound score to the current values of the sliders on the user interface. When Csound has started playing, set the label of the play button to *Stop*; when the user clicks on the button again, set the `self.playing` flag to `False` so that the application will exit the performance loop and shut down Csound, and set the label of the play button back to *Play*.

```

    ...
    Play if stopped,
    and stop if playing.
    ...
    def on_play(self):
        if not self.playing:
            self.playButton['text'] = 'Stop'
            self.playing = True
            try:
                print 'Started playing...'
                self.csound.setOrchestra(csoundOrchestra)
                self.csound.setScore(csoundScore)
                self.csound.setCommand(csoundCommand % self.outputStringVar.get())
                self.csound.exportForPerformance()
                self.csound.compile()
                # Apply initial control channel values before actually starting
synthesis.
                f = self.configuration.gkHarmonicTableFactor
                message = 'f 1 0 8193 10 3 0 %f 0 0 %f\n' % (f * 1.0, f *
2.0)
                self.csound.inputMessage(message)
                f = self.configuration.gkDistortTableFactor
                message = 'f 2 0 8193 13 1 %f 0 %f 0 %f\n' % (f * 1.0, f
* 2.0, f * 3.0)
                self.csound.inputMessage(message)
                self.csound.SetChannel("gkDistortFactor",
float(self.gkDistortFactor.get()))
                self.csound.SetChannel("gkReverbScFeedback",
float(self.gkReverbScFeedback.get()))

```

² This is because Tkinter has trouble if other threads are started in a Tkinter application. With other GUI toolkits, it is better to run Csound in a separate thread of execution, which is less likely to be interrupted by GUI events, using the `Csnd.CsoundPerformanceThread` class.

```

float(self.gkMasterLevel.get()))
self.csound.SetChannel("gkMasterLevel",
# Tkinter only likes 1 thread per application.
# So, we hack the rules and switch back and forth between
# computing sound and handling GUI events.
# When the user closes the application, self.update will raise
# an exception because the application has been destroyed.
while self.playing and not self.csound.performKsmps(0):
    try:
        self.update()
    except TclError:
        traceback.print_exc()
        self.playing = False
        return
    self.csound.stop()
except:
    traceback.print_exc()
else:
    try:
        print 'Stopping...'
        self.playButton['text'] = 'Play'
        self.playing = False
    except:
        print traceback.print_exc()

```

There is one final trick in using Tkinter with Csound, and that is stopping Csound from the user interface. If the user closes the application while Csound is running, Tkinter destroys the application, but the performance loop will still call the Tkinter update method. This causes a TclError exception that you must catch; in the exception handler you should simply set `self.playing` to `False` to exit from the performance loop in the normal way.

To implement real-time orchestra control using control channels, do something like this:

```

def on_gkDistortFactor(self, value):
    self.csound.SetChannel("gkDistortFactor", float(value))
    self.configuration.gkDistortFactor = value

def on_gkReverbFeedback(self, value):
    self.csound.SetChannel("gkReverbFeedback", float(value))
    self.configuration.gkReverbFeedback = value

def on_gkMasterLevel(self, value):
    self.csound.SetChannel("gkMasterLevel", float(value))
    self.configuration.gkMasterLevel = value

```

To implement real-time orchestra control using run-time score messages, do something like this:

```

def on_gkHarmonicTableFactor(self, value):
    f = float(value)
    self.configuration.gkHarmonicTableFactor = f
    message = 'f 1 0 8193 10 3 0 %f 0 0 %f\n' % (f * 1.0, f * 2.0)
    self.csound.inputMessage(message)

def on_gkDistortTableFactor(self, value):
    f = float(value)
    self.configuration.gkDistortTableFactor = f
    message = 'f 2 0 8193 13 1 %f 0 %f 0 %f\n' % (f * 1.0, f * 2.0,
f * 3.0)
    self.csound.inputMessage(message)

```

Rendering the Piece

You can find a pre-written version of the piece as `MusicProjects\04_Drone\drone.py`. You can run the piece either from the command line, or using IDLE, which comes with Python, or from a text editor such as SciTE as previously discussed, or from the Windows console.

You can run the piece from your text editor, or by simply executing it with Python. In either case, there are default values for the controls, so you can begin the piece simply by clicking on the *Play* button. You should see Csound messages in the output panel of your text editor, or in the Windows console, and the sound should gradually get louder.

Experiment with changing various sliders. Save settings that you like under different filenames. When you have found a configuration that you can't seem to improve, change the name of the output from `dac` to a soundfile name, click on the *Play* button, and render the piece to a soundfile.

I suggest that you spend some time listening to various settings before making up your mind about what works and what does not. You may find that some initially appealing settings wear thin, whereas other settings that do not at first sound compelling end up sounding better.

As a result of experimenting with different settings, I have concluded that many of the shifting tones heard in some versions of this piece arise from combinations and differences, not of the pure just intervals, but of nonlinear distortions of them introduced by both the waveshaper and the reverberator.

This piece could be elaborated in many ways. It would probably be very interesting to put in sliders to control the ratios of the just intonation intervals, instead of setting them in the score. You could set up sliders to control the individual variables in the Chebyshev polynomial, or the individual partials in

the oscillator wavetable. You could add a filter. You could put in buttons to generate notes, or chords, at run time. You could use MIDI to play the instrument. And so on....

Charles Dodge and Fractal Music

In 1987 Charles Dodge composed *Viola Elegy* for that instrument and computer-generated tape (Dodge, 1987). The music was made using an algorithm that computes a melody based on $1/f$ noise -- a random fractal -- and then on top of each note of that melody, recursively computes another shorter melody based on $1/f$ noise, and so on for several layers. Dodge then arranged the music by doubling the computer parts on viola, ending with a *cadenza* for viola solo. It is a moody and affecting piece.

In an article (Dodge, Profile: A Musical Fractal, 1998), Dodge explains this procedure as a variation on the Koch snowflake curve. In the Koch snowflake, each segment of a hexagon is inscribed with a "generator," a triangle sitting in the middle third of a line segment. Then, each segment of the resulting curve is inscribed with a smaller copy of the generator, and so on *ad infinitum*.

Generating the Score

A similar procedure can be used to generate a score, as follows.

1. Define the generator as a set of movements in time, pitch, and loudness: a factor to lengthen or shorten duration, half-steps to add to pitch, and decibels to add to loudness. Each point in the generator can be a 3 element Python list. The complete generator can be a list of points.

```
generator = [  
    [1.500, 3, 1],  
    [1.250, 9, 2],  
    [1.000, 11, 3],  
    [1.375, 4, 1],  
]
```

2. Define a function to normalize the duration factors in a generator.

```
def normalize(notes, dimension):  
    print 'Normalize dimension', dimension  
    total = 0.0  
    for note in notes:  
        total = total + note[dimension]  
    factor = 1.0 / total  
    for note in notes:
```

```

    note[dimension] = note[dimension] * factor
    print note[dimension]

```

3. Define a recursive function that applies the generator to each point in a normalized generator, and then translates each point into a note that is stored in a score. In addition to the generator, the function will take a time, duration, and pitch at which to place the generator, and a score in which to place generated notes. When the level of recursion reaches the bottom, return without recursing further.

```

print 'CREATING MUSIC MODEL...'
print

# A recursive function for generating a score
# in the form of a Koch curve.
# Each note in a generating motive
# will get a smaller copy of the motive nested atop it,
# and so on.

def generate(score, notes, levels, level, time, duration, key, velocity):
    localLevel = level
    localTime = time
    localPan = ((float(localLevel) / float(levels)) - 1.0) *.75
    localVelocity = velocity
    for note in notes:
        localDuration = note[0] * duration
        localKey = key + note[1]
        localVelocity = localVelocity - (note[2] * 0.35)
        print 'Adding note: level %3d of %3d time %7.3f duration %7.3f key %3d
velocity %3d pan %7.3f' % ((localLevel + 1), levels, localTime, localDuration,
localKey, localVelocity, localPan)
        score.append(localTime, localDuration, 144.0, localLevel + 1, localKey,
localVelocity, 0, localPan)
        if localLevel > 0:
            generate(score, notes, levels, (localLevel - 1), localTime,
localDuration, localKey, localVelocity)
            localTime = localTime + localDuration

```

4. Generate a composition: define a generator that assigns each level of recursion to a different Csound instrument (a sustained instrument for the slowest, lowest level of structure, a more percussive sound for the faster, higher levels of structure), normalize your generator, and call the generation function. The entire score generation section is then:

```

print 'CREATING MUSIC MODEL...'
print

# A recursive function for generating a score
# in the form of a Koch curve.
# Each note in a generating motive

```

```

# will get a smaller copy of the motive nested atop it,
# and so on.

def generate(score, notes, levels, level, time, duration, key, velocity):
    localLevel = level
    localTime = time
    localPan = ((float(localLevel) / float(levels)) - 1.0) *.75
    localVelocity = velocity
    for note in notes:
        localDuration = note[0] * duration
        localKey = key + note[1]
        localVelocity = localVelocity - (note[2] * 0.35)
        print 'Adding note: level %3d of %3d time %7.3f duration %7.3f key %3d
velocity %3d pan %7.3f' % ((localLevel + 1), levels, localTime, localDuration,
localKey, localVelocity, localPan)
        score.append(localTime, localDuration, 144.0, localLevel + 1, localKey,
localVelocity, 0, localPan)
        if localLevel > 0:
            generate(score, notes, levels, (localLevel - 1), localTime,
localDuration, localKey, localVelocity)
            localTime = localTime + localDuration

def normalize(notes, dimension):
    print 'Normalize dimension', dimension
    total = 0.0
    for note in notes:
        total = total + note[dimension]
    factor = 1.0 / total
    for note in notes:
        note[dimension] = note[dimension] * factor
    print note[dimension]

generator = [
    [1.500, 3, 1],
    [1.250, 9, 2],
    [1.000, 11, 3],
    [1.375, 4, 1],
]
normalize(generator, 0)

scoreNode = CsoundAC.ScoreNode()
scoreNodeScore = scoreNode.getScore()
model.addChild(scoreNode)
generate(scoreNodeScore, generator, 4, 4, 0, 150, 30, 75)

```

5. Create an arrangement, placing more sustained sounds in the lower, long-lasting notes and more percussive sounds in the higher, shorter notes:

```

print 'CREATING CSOUND ARRANGEMENT...'
print

```



```

model.setCsoundOrchestra(csoundOrchestra)
model.setCsoundScoreHeader(csoundScoreHeader)
#           oldinsno, newinsno, level (+-dB), pan (-1.0 through +1.0)
model.arrange( 1,      61 )
model.arrange( 2,      7 )
model.arrange( 3,     59 )
model.arrange( 4,     12 )
model.arrange( 5,     12 )
model.setCsoundCommand(csoundCommand)

```

Rendering the Piece

You can find a pre-written version of the piece as `MusicProjects/05_Koch/koch.py`. You can run the piece either from the command line, or using IDLE, which comes with Python, or from a text editor such as SciTE as previously discussed, or from the Windows console.

The piece must be run from the directory containing it, so that the relative paths used to access the SoundFonts and included Python modules will all work. The piece should print out messages about what it is doing and terminate, leaving a soundfile player running and ready to play `koch.py.wav`.

Try changing the generator one element at a time, to see how such changes, thanks to recursion, produce global effects throughout the generated score. Also try adding and removing elements from the generator.

Terry Riley and Cellular Accretion

From the late 1960s through the 1980s the minimalist school of composition was very influential. Although generally speaking the minimalists were not computer musicians, their compositional approach was certainly algorithmic, as indicated by their frequent use of the term “process music,” which implies that one may compose not a piece, but a process for making a piece, so that when the process is performed, the result is music.

One of the commonest techniques in minimalism is the use of independent motivic cells. These are constructed so that they can be played together in various combinations. The process often involves playing sequences of cells in difference voices such that they overlap canonically. Frequently the canon is not symmetrical, but one voice repeats in a period a beat or so longer than another other voice, so that the offset of the canon shifts as each period repeats. This could be called an overlap or differential canon.

There are various ways of determining the sequence of cells, the number of repeats, the amount of shift, and so on, ranging from completely predetermined, to improvised during performance, to chosen purely by chance.

Terry Riley's *In C* was a formative piece in this movement. It consists of 53 cells of varying lengths with an eighth-note beat. All 53 cells are assigned to a group of players. Each player is instructed to play through all the cells in sequence and in meter, but to play each cell as many times as desired before moving on to the next. A piano keeps the eighth-note beat with its two top C's throughout the piece. Thus, the actual combinations of cells are completely unpredictable and form a vast field of more or less subtle variations on a theme.

Copyright forbids reproducing this piece literally in a CsoundAC program, but the basic idea is easy and rewarding to reproduce. Playing a sequence of cells with varying randomly chosen numbers of repeats is quite easy to program, and harks back to the musical dice game. Perhaps the algorithm can never reproduce the full effect of Riley's original piece, because that depends on human performers who listen to each other as they play and keep a more flexible style of time. On the other hand, the algorithm has no trouble at all keeping perfect time and playing as fast and as precisely as you like.

Generating the Score

Be that as it may, let us produce a piece similar to *In C*, except using the measures of the musical dice game as raw material for the cells, as follows:

1. Copy the entire minuet table from the musical dice game into the script, and create a function to read a measure from the minuet table into a cell.

```
print 'CREATING MUSIC MODEL...'
print

minuetTable = {}
minuetTable[ 2] = { 1: 96,  2: 22,  3:141,  4: 41,  5:105,  6:122,  7: 11,  8: 30,
9: 70, 10:121, 11: 26, 12:  9, 13:112, 14: 49, 15:109, 16: 14}
minuetTable[ 3] = { 1: 32,  2:  6,  3:128,  4: 63,  5:146,  6: 46,  7:134,  8: 81,
9:117, 10: 39, 11:126, 12: 56, 13:174, 14: 18, 15:116, 16: 83}
minuetTable[ 4] = { 1: 69,  2: 95,  3:158,  4: 13,  5:153,  6: 55,  7:110,  8: 24,
9: 66, 10:139, 11: 15, 12:132, 13: 73, 14: 58, 15:145, 16: 79}
minuetTable[ 5] = { 1: 40,  2: 17,  3:113,  4: 85,  5:161,  6:  2,  7:159,  8:100,
9: 90, 10:176, 11:  7, 12: 34, 13: 67, 14:160, 15: 52, 16:170}
minuetTable[ 6] = { 1:148,  2: 74,  3:163,  4: 45,  5: 80,  6: 97,  7: 36,  8:107,
9: 25, 10:143, 11: 64, 12:125, 13: 76, 14:136, 15:  1, 16: 93}
minuetTable[ 7] = { 1:104,  2:157,  3: 27,  4:167,  5:154,  6: 68,  7:118,  8: 91,
9:138, 10: 71, 11:150, 12: 29, 13:101, 14:162, 15: 23, 16:151}
minuetTable[ 8] = { 1:152,  2: 60,  3:171,  4: 53,  5: 99,  6:133,  7: 21,  8:127,
9: 16, 10:155, 11: 57, 12:175, 13: 43, 14:168, 15: 89, 16:172}
```

```

minuetTable[ 9] = { 1:119, 2: 84, 3:114, 4: 50, 5:140, 6: 86, 7:169, 8: 94,
9:120, 10: 88, 11: 48, 12:166, 13: 51, 14:115, 15: 72, 16:111}
minuetTable[10] = { 1: 98, 2:142, 3: 42, 4:156, 5: 75, 6:129, 7: 62, 8:123,
9: 65, 10: 77, 11: 19, 12: 82, 13:137, 14: 38, 15:149, 16: 8}
minuetTable[11] = { 1: 3, 2: 87, 3:165, 4: 61, 5:135, 6: 47, 7:147, 8: 33,
9:102, 10: 4, 11: 31, 12:164, 13:144, 14: 59, 15:173, 16: 78}
minuetTable[12] = { 1: 54, 2:130, 3: 10, 4:103, 5: 28, 6: 37, 7:106, 8: 5,
9: 35, 10: 20, 11:108, 12: 92, 13: 12, 14:124, 15: 44, 16:131}

```

```

def readMeasure(number):
    scoreNode = CsoundAC.ScoreNode()
    scoreNode.thisown = 0
    filename = 'M' + str(number) + '.mid'
    print 'Reading "%s"' % (filename)
    scoreNode.getScore().load(filename)
    return scoreNode

```

2. Define a function to build a track that takes a sequence, a channel or instrument number, a bass pitch, and a stereo pan as arguments. The function can use a time accumulator to keep track of time as the sequence is built. For each measure of the minuet table from 1 through 16, for row of the table from 2 through 6, pick a number of repeats from 1 through 13 at random and repeat the measure that number of times. Put each measure into a Rescale node and position it at the proper time, instrument number, pitch, loudness, and pan. Don't forget to set thisown=0 to keep Python from destroying the Rescale nodes. If you want a longer piece, you can use all the rows in the minuet table up through 12.

```

def buildTrack(sequence, channel, bass):
    print 'Building track for channel %3d bass %3d...' % (channel, bass)
    cumulativeTime = 0.0
    for i in xrange(1, 16):
        for j in xrange(2, 6):
            repeatCount = 1 + int(random.random() * 12)
            for k in xrange(repeatCount):
                measure = readMeasure(minuetTable[j][i])
                duration = 1.5
                rescale = CsoundAC.Rescale()
                rescale.setRescale(CsoundAC.Event.TIME, 1, 0, cumulativeTime, 0)
                rescale.setRescale(CsoundAC.Event.INSTRUMENT, 1, 0, channel, 0)
                rescale.setRescale(CsoundAC.Event.KEY, 1, 0, bass, 0)
                rescale.thisown = 0
                rescale.addChild(measure)
                print 'Repeat %4d of %4d at %8.3f with %3d notes of duration
                %7.3f...' % (k + 1, repeatCount, cumulativeTime, len(measure.getScore()), duration)
                sequence.addChild(rescale)
                cumulativeTime = cumulativeTime + duration

```

- Use another `Rescale` node to hold the sequence of measures. Conform the pitches in the sequence to a specific key or chord (here, the Bb major scale). Then call the track-building function a number of times with different instrument numbers, bass notes, and pans to separate out the voices. Reseed the random number generator so a different sequence is generated for each performance.

```

sequence = CsoundAC.Rescale()
model.addChild(sequence)
model.setConformPitches(True)
sequence.setRescale(CsoundAC.Event.KEY,          1, 1, 34, 66)
sequence.setRescale(CsoundAC.Event.VELOCITY,    1, 1, 50, 12)
sequence.setRescale(CsoundAC.Event.PITCHES,     1,          1,
CsoundAC.Conversions_nameToM("Ab major"), 0)
sequence.setRescale(CsoundAC.Event.INSTRUMENT,  1, 1,  0, 5)
sequence.setRescale(CsoundAC.Event.TIME,        1, 1,  0, 700)
sequence.setRescale(CsoundAC.Event.DURATION,    1, 1,  1.0, 3.0)
buildTrack(sequence, 1, 36)
buildTrack(sequence, 2, 48)
buildTrack(sequence, 3, 36)
buildTrack(sequence, 4, 48)
buildTrack(sequence, 5, 60)
model.setCppSound(csound)
random.seed()

```

- This kind of piece works well with percussive, sampled sounds. In the Python file, define a Csound orchestra that creates an arrangement of grand piano, marimba, Rhodes electric piano, tubular bell, and harpsichord using SoundFonts.

```

print 'CREATING CSOUND ARRANGEMENT...'
print

model.setCsoundOrchestra(csoundOrchestra)
model.setCsoundScoreHeader(csoundScoreHeader)
#          oldinsno, newinsno, level (+-dB), pan (-1.0 through +1.0)
panIncrement = (.875 * 2.0) / 5.0
pan = -.875
model.arrange( 0,          8,          0.0,          pan )
pan = pan + panIncrement
model.arrange( 1,         20,          0.0,          pan )
pan = pan + panIncrement
model.arrange( 2,         51,          6.0,          pan )
pan = pan + panIncrement
model.arrange( 3,         14,          3.0,          pan )
pan = pan + panIncrement
model.arrange( 4,          7,          0.0,          pan )
pan = pan + panIncrement
model.arrange( 5,          9,          0.0,          pan )

```

```
model.setCsoundCommand(csoundCommand)
```

Rendering the Piece

You can find a pre-written version of the piece as `MusicProjects\01_DiceGame\cellular.py`. You can run the piece either from the command line, or using IDLE, which comes with Python, or from a text editor such as SciTE as previously discussed, or from the Windows console.

The piece must be run from the directory containing it, so that the relative paths used to access the SoundFonts used by the Csound instruments and any included Python modules will all work. The piece should print out messages about what it is doing and terminate, leaving a soundfile player running and ready to play `cellular.py.wav`.

Experiment with different durations for the measures, ranges, choices of instruments, and especially tempos to develop a pleasing texture.

Afterword

Let's go back to the beginning of this section and dig a little more deeply into it. Computability is a metaphysical question: Is an entity definable, knowable, finite, or not? – So, any question concerning the *computability* of music is a question of the *metaphysics* of music.

My considerations are grounded in several obvious and unassailable facts, from which far-reaching inferences can be drawn.

In the first place, works of music are physical phenomena – sounds. Each work of music is a sound of finite tessitura, dynamic range, and duration, which persists without gaps until it ends. Human hearing, too, has limited resolution. Therefore, each work of music can be considered to be completely represented by a digital recording of sufficient bandwidth: a certain finite series of 0s and 1s. Or again, a work of music is a sequence of finite complexity. So of course all such sequences are computable, though not necessarily in a reasonable period of time.

But works of music have no meaning unless they are heard, and hearing music is an inward, subjective experience. It would be a mistake to suppose that the experience of music is not caused by the physical sound, yet it would equally be a mistake to suppose that the physical sound has any meaning without the experience of hearing it. Forgetting the physical causation of music leads to the error of cultural relativism sometimes found in postmodernism, and forgetting

the inward hearing of music leads to the error of objectifying music typical of scientism.

Now although any *given work* of music is a finite object, the *art of music* consists of a sequence of such works, each distinct from all others. Mathematically speaking, this series could continue indefinitely. The art of music can thus be considered to be an *infinite* series of 0s and 1s.

In other words, considered in the concrete, as a completed series, from the viewpoint of God if you will, the art of music is a series of countably infinite length and countably infinite complexity. But considered in the abstract, as an uncompleted series, from the viewpoint of creatures if you will, the art of music is an *infinite set of possible series, each of countably infinite length*. This set of possible series can be diagonalized (exactly as in the halting theorem) to prove that it is of *uncountable* infinity and *uncountable* complexity. Since the series *has not* been completed and the art of music *has not* been finished, *the reality of our situation is that the art of music is effectively uncomputable*.

The most important implication of this, for us, is that it would be absurd to hope for computers to generate the art of music as a whole. But as I have tried to show, they can be very useful indeed in computing works of music.

The field of algorithmic composition, at least in its computer-assisted form, is only 50 years old. Yet I hope this section has demonstrated that algorithmic composition already has more than one string to its lyre. There are styles of algorithm, just as there are styles of song. I believe that this lyre is young, and has only just begun to play music that will sound ever more sweet and strange.

References

- Agon, C. A., Assayag, G., & Bresson, J. (2007). *OpenMusic: a Visual Programming Language*. Retrieved from <http://recherche.ircam.fr/equipes/repmus/OpenMusic/>
- Apple Developer Connection. (2007). *Core Audio Technologies*. Retrieved from Developer Connection: <http://developer.apple.com/audio/coreaudio.html>
- Apple. (2007). *Logic Pro 8*. Retrieved from Apple: <http://www.apple.com/logicstudio/logicpro/>
- Ariza, C. (2007). *Algorithmic.net*. Retrieved from flexatone: <http://www.flexatone.net/algoNet/>

- Ariza, C. (2007). *athenaCL*. Retrieved from <http://www.flexatone.net/athena.html>
- Audacity. (2007). *The Free, Cross-Platform Sound Editor*. Retrieved from Audacity: <http://audacity.sourceforge.net/>
- AudioNerdz. (2007). *AudioNerdz*. Retrieved from AudioNerdz: <http://www.audionerdz.com/>
- Banzhaf, W., Nordin, P., Keller, R. E., & Francone, F. D. (1998). *Genetic Programming ~ An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. San Francisco: Morgan Kaufman Publishers.
- Bencina, R. (2007). *PortAudio - portable cross-platform Audio API*. Retrieved from PortAudio: <http://www.portaudio.com/>
- Bentley, P. J., & Corne, D. W. (2002). *Creative Evolutionary Systems*. San Francisco: Morgan Kaufman Publishers.
- boost.org. (2007). *Welcome to boost.org*. Retrieved from boost C++ Libraries: <http://www.boost.org/>
- Burks, P. (2007). *Computer Music Links*. Retrieved from SoftSynth: <http://www.softsynth.com/links/>
- Burns, K. H. (1993). *The History and Development of Algorithms in Music Composition, 1957-1993*. Ball State University.
- Buzzwiki. (2007). Retrieved from Buzzwiki: http://wiki.buzzmusic.de/index.php/Main_Page
- C++ Standards Committee. (2007). Retrieved from JTC1/SC22/WG21 - The C++ Standards Committee: <http://www.open-std.org/jtc1/sc22/wg21/>
- Cage, J. (1961). *Atlas Eclipticalis*. London: Edition Peters.
- Cage, J., & Hiller, L. (1969). *HPSCHD*. London: Edition Peters.
- Cakewalk. (2007). *Introducing: Sonar 7 Producer Edition*. Retrieved from Cakewalk: <http://www.cakewalk.com/Products/SONAR/default.asp>
- Cakewalk. (2007a). *Project 5*. Retrieved from Project 5: <http://www.project5.com/>

- Centre de Donnees astronomiques de Strasbourg. (2006). *Astronomer's Bazaar*. Retrieved from The CDS Service for astronomical Catalogues: <http://cdsweb.u-strasbg.fr/Cats.html>
- Chaitin, G. J. (1998). *The Limits of Mathematics: A Course on Information Theory and the Limits of Reasoning*. Singapore: Springer-Verlag.
- Chuang, J. (2007). *Mozart's Musikalisches Würfelspiel*. Retrieved from <http://sunsite.univie.ac.at/Mozart/dice/>
- Cook, P. R. (2002). *Real Sound Synthesis for Interactive Applications*. Natick, Massachusetts: A. K. Peters.
- Cook, P. R., & Scavone, G. K. (2007). *The Synthesis ToolKit in C++ (STK)*. Retrieved from The Synthesis ToolKit in C++ (STK): <http://ccrma.stanford.edu/software/stk/>
- Creative. (2007). *Developer Central*. Retrieved from Creative USA: <http://developer.creative.com/landing.asp?cat=2&sbcat=34&top=51>
- Dannenberg, R. B. (2007). *Nyquist, A Sound Synthesis and Composition Language*. Retrieved from Nyquist, A Sound Synthesis and Composition Language: <http://www.cs.cmu.edu/~music/nyquist/>
- Digidesign. (2007). *Digidesign*. Retrieved from Digidesign: <http://www.digidesign.com/>
- Dodge, C. (1998). Profile: A Musical Fractal. *Computer Music Journal* , 12 (3), 10-14.
- Dodge, C. (Composer). (1987). *Viola Elegy*. On *Any Resemblance is Purely Coincidental* [CD]. New Albion Records.
- Embree, P. M., & Danieli, D. (1999). *C++ Algorithms for Digital Signal Processing*. Upper Saddle River, New Jersey: Prentice Hall.
- Fishman, S. (1998). *Software Development: A Legal Guide, 2nd Edition*. Berkeley: Nolo Press.
- Fitz, K., & Haken, L. (2007). *Loris Software for Sound Modeling, Morphing, and Manipulation*. Retrieved from Loris: <http://www.cerlsoundgroup.org/Loris/>
- FLTK. (2007). *FLTK - Fast Light Tool Kit*. Retrieved from FLTK: <http://www.fltk.org/>

- Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1997). *Computer Graphics: Principles and Practice, Second Edition* in C. Reading, Massachusetts: Addison-Wesley.
- Free Software Foundation. (2007). *Free Software Foundation*. Retrieved from <http://www.fsf.org/>
- Future Publishing Limited. (2007). *Computer Music Make Music Now!* Retrieved from Computer Music: <http://www.computermusic.co.uk/>
- Gogins, M. (2007). *Csound on SourceForge*. Retrieved from <http://csound.sourceforge.net/>
- Gogins, M. (1998). Music Graphs for Algorithmic Composition and Synthesis with an Extensible Implementation in Java. *Proceedings of the 1998 International Computer Music Conference* (pp. 369-376). San Francisco: International Computer Music Association.
- Gogins, M. (2006). Score Generation in Voice-Leading and Chord Spaces. *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association.
- Grabit, Y. (2007). *The Yvan Grabit Developer Resource*. Retrieved from The Yvan Grabit Developer Resource: http://ygrabit.steinberg.de/~ygrabit/public_html/index.html
- Green Oak. (2007). *Crystal Soft Synth*. Retrieved from Crystal Soft Synth.
- Hammer, T. (2007). *HammerSound*. Retrieved from HammerSound: <http://www.hammersound.net/>
- Hanappe, P., & Green, J. (2007). *Fluidsynth*. Retrieved from <http://fluidsynth.resonance.org/trac>
- Harmony-Central. (2007). *Home*. Retrieved from Harmony Central: <http://www.harmony-central.com/>
- Heckmann, U., & de Jong, B. (2007). *VST Source Code Archive*. Retrieved from VST Source Code Archive: <http://www.u-he.com/vstsource/>
- Hiller, L. A., & Isaacson, L. M. (1959a). *Illiac Suite for String Quartet*. New York: New Music Editions.

- Hiller, L., & Isaacson, L. M. (1959). *Experimental Music: Composition with an Electronic Computer*. New York: McGraw-Hill.
- Hitsquad. (2007). *Shareware Music Machine*. Retrieved from Hitsquad Musician Network: <http://www.hitsquad.com/smm/>
- Holm, F. (1992, Spring). Understanding FM Implementations: A Call for Common Standards. *Computer Music Journal* , 16 (1), pp. 34-42.
- Home. (2007). Retrieved from FL Studio : <http://www.flstudio.com>
- Hrabovsky, Leonid. (1977). *1448 Concerto Misterioso: The Music of Leonid Hrabovsky (b. 1935), Vol. 1*. Las Vegas, Nevada: TNC Recordings.
- Intel. (2007). *Intel Compilers*. Retrieved from Intel: <http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm>
- Jacob, Bruce. (Retrieved August 11, 2009). *VARIATIONS: Algorithmic Composition for Acoustic Instruments*. http://www.ece.umd.edu/~blj/algorithmic_composition.
- Jourdan, T. (2004). *Kandid: A Genetic Art Project*. Retrieved from <http://kandid.sourceforge.net>
- Kleene, S. C. (1967). *Mathematical Logic*. New York: Wiley.
- KristalLabs Software Ltd. (2007). *Kristal Audio Engine*. Retrieved from Kristal Audio Engine: <http://www.kreatives.org/kristal/>
- Lassfolk, K., & Halmkrona, J. (2006). *Sound Processing Kit Home Page*. Retrieved from <http://www.music.helsinki.fi/research/spkit/>
- linuxaudio.org. (2007). *apps*. Retrieved from linuxaudio.org: <http://apps.linuxaudio.org/>
- Lopo, E. d. (2007). *libsndfile*. Retrieved from lbsndfile: <http://www.mega-nerd.com/libsndfile/>
- Loy, G. (1990). Composing with computers: a survey of some compositional formalisms and programming. In M. Mathews, & J. Pierce, *Current Directions in Computer Music*. Cambridge, Massachusetts: The MIT Press.

- Maldonado, G. (2007). *Home Page of Gabriel Maldonado*. Retrieved from Gabriel Maldonado: <http://web.tiscali.it/G-Maldonado/>
- Mann, A. (1971 [1943]). *The Study of Counterpoint from Johann Joseph Fux's Gradus ad Parnassum*. New York: W. W. Norton.
- Mark of the Unicorn. (2007). *Digital Performer Overview* . Retrieved from MOTU: <http://www.motu.com/products/software/dp>
- Mathews, M. V. (1969). *The Technology of Computer Music*. Cambridge, Massachusetts: The MIT Press.
- Max/MSP. (2007). Retrieved 2007, from 74: <http://www.cycling74.com/products/maxmsp>
- McCartney, J. (2007). *SuperCollide: real-time audio synthesis and algorithmic composition*. Retrieved from <http://supercollider.sourceforge.net/>
- Microsoft. (2007). *Home*. Retrieved from Microsoft Developer Network: <http://msdn2.microsoft.com/en-us/default.aspx>
- MSDN. (2007). *DirectX Resource Center*. Retrieved from MSDN: <http://msdn2.microsoft.com/en-us/directx/default.aspx>
- Muse Research. (2007). *Audio Plugin News*. Retrieved from KVR Audio: <http://www.kvraudio.com/>
- Music Technology Group of Pompeu Fabra University. (2007). *News*. Retrieved from CLAM: <http://clam.iaa.upf.edu/>
- Native Instruments. (2007). *Reaktor 5 - The Future of Sound*. Retrieved from NI: <http://www.nativeinstruments.de/index.php?id=reaktor5>
- Nierhaus, Gerard. (2009). *Algorithmic Composition: Paradigms of Automated Music Generation*. New York: SpringerWien.
- Peitgen, H.-O., Jurgens, H., & Saupe, D. (1992). *Chaos and Fractals: New Frontiers of Science*. New York: Springer-Verlag.
- Persistence of Vision Pty. Ltd. (2007). *POV-Ray - The Persistence of Vision Raytracer*. Retrieved from <http://www.povray.org/>
- Pope, S. (2007). Retrieved from The CREATE Signal Library (CSL) Project: <http://fastlabinc.com/CSL/>

- Pouet. (2007). *Your online demoscene resource*. Retrieved from Pouet:
<http://www.pouet.net/>
- Propellerhead Software. (2007). *Propellerhead News*. Retrieved from
Propellerhead: <http://www.propellerheads.se/>
- psychedelics-. (2007). *Psyche*. Retrieved from Psyche:
<http://psyche.pastnotecut.org/portal.php>
- Puckette, M. (2007). *About Pure Data*. Retrieved 2007, from pd~:
<http://puredata.info/>
- Puxeddu, M. U. (2004). *OMDE/PMask*. Retrieved from
<http://pythonsound.sourceforge.net>
- Python Software Foundation. (2007). *Python Programming Language -- Official Website*. Retrieved from Python Programming Language -- Official Website:
<http://www.python.org/>
- Roads, C. (1996). *The Computer Music Tutorial*. Cambridge, Massachusetts: The MIT Press.
- Sapp, C. S. (2007). *Sig++: Musical Signal Processing in C++*. Retrieved from <http://sig.sapp.org/>
- savannah. (2007). *Savannah*. Retrieved from Savannah:
<http://savannah.gnu.org/>
- scene.org. (2007). Retrieved from scene.org: <http://www.scene.org/>
- Schottstaedt, B. (1984). *Automatic Species Counterpoint*. Center for Computer Research in Music and Acoustics, Stanford, California.
- SciTE: A Free Text Editor for Win32 and X. (2007). Retrieved from
<http://www.scintilla.org/SciTE.html>
- Seppänen, J., & Kananoja, S. (2007). *Sonic Flow*. Retrieved from Sonic Flow:
<http://sonicflow.sourceforge.net/>
- Silicon Graphics Incorporated. (2007). *Standard Template Library Programmer's Guide*. Retrieved from <http://www.sgi.com/tech/stl/>
- Smith, J. O. (2007). *Julius Orion Smith III Home Page*. Retrieved from
<http://ccrma.stanford.edu/~jos/>

- solipse. (2007). *noise*. Retrieved from http://solipse.free.fr/site/index_csound.html
- Sorensen, A., & Brown, A. (2007). *jMusic*. Retrieved from jMusic: <http://jmusic.ci.qut.edu.au/>
- Sound on Sound Ltd. (2007). *Sound on Sound*. Retrieved from Sound on Sound: <http://www.soundonsound.com/>
- sourceforge.net. (2007). *sourceforge.net*. Retrieved from sourceforge.net: <http://sourceforge.net/>
- Steiglitz, K. (1996). *A Digital Signal Processing Primer: With Applications to Digital Audio and Computer Music*. Upper Saddle River, New Jersey: Prentice-Hall.
- Steinberg Media Technologies GmbH. (2007). Retrieved from Steinberg: <http://www.steinberg.net>
- Steinberg Media Technologies GmbH. (2007a). *3rd Party Developers*. Retrieved from Steinberg: http://www.steinberg.net/324_1.html
- SuperCollider swiki*. (2007). Retrieved from <http://swiki.hfbk-hamburg.de:8888/MusicTechnology/6>
- Synapse Audio Software. (2007). *Home*. Retrieved from Synapse Audio Software: <http://www.synapse-audio.com/>
- Täips, R. (2007). *Rumpelrausch Täips*. Retrieved from Rumpelrausch Täips: <http://zr-3.sourceforge.net/>
- Taube, H. K. (2004). *Notes from the Metalevel: An Introduction to Computer Composition*. Oxford, England: Routledge.
- Taube, R. (2007). *Common Music*. Retrieved from Common Music: <http://commonmusic.sourceforge.net/doc/cm.html>
- The GIMP Team. (2007). *GNU Image Manipulation Program*. Retrieved from GIMP: <http://www.gimp.org/>
- Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* , 42 (1), 230-265.
- Tymoczko, D. (2006). The Geometry of Musical Chords. *Science* , 313, 72-74.

- Vercoe, B. (2007). *Csound on SourceForge*. Retrieved from Csound on SourceForge:
<http://csound.sourceforge.net/>
- Ward-Bergeman, M. J. (2007). *Csound Compositions*. Retrieved from cSounds.com:
<http://www.csounds.com/compositions/>
- Wavosaur Team. (2007). *Audio Editor with VST Support*. Retrieved from
Wavosaur: <http://www.wavosaur.com/>
- WebRing. (2007). *Cubase*. Retrieved from WebRing: [http://d.webring.com/hub?
ring=cubase&list](http://d.webring.com/hub?ring=cubase&list)
- www.mingw.org. (2007). *MinGW - Home*. Retrieved from MinGW - Minimalist
GNU for Windows: <http://www.mingw.org/>
- wxWidgets. (2007). *wxWidgets Cross-Platform GUI Library*. Retrieved from
wxWidgets Cross-Platform GUI Library: <http://www.wxwidgets.org/>
- Yi, S. (2007). *blue*. Retrieved from Steven Yi:
<http://www.csounds.com/stevenyi/blue/>

