On Teaching Digital Signal Processing to Composition Students

by

Matthew Barber

Submitted in Partial Fulfillment
of the
Requirements for the Degree
Master of Arts

Supervised by
Professor Allan Schindler

Department of Composition
Eastman School of Music

University of Rochester
Rochester, New York

2008

Contents

List of Figures

1 Introduction

COMPUTER MUSIC is no longer a new field. Its sounds are rife in popular culture – and the consequences of its ideas rifer yet, for the ubiquity of portable music. Courses devoted to the subject are commonplace in university and conservatory programs; because digital media has grown so important in musical spheres, an educator could even claim that a lack of training in at least some techniques of computer music in music curricula does as egregious a disservice to music students as the lack of a proper training in music theory and history. Whether or not this is an extreme position (especially within traditional music programs), there is no doubt that the technology associated with musical pursuits has historically changed musical, and therefore pedagogical values.

Definitions of computer music and related topics are rather fuzzy in the computer-music literature, though it is unlikely that this will cause any serious confusion. "Computer music" may be said to intersect semantically with "electronic music," but there is no agreed-upon distinction between the two.[1] Similarly, while "music technology" may refer colloquially to the familiar hardware and software used to record music and to play it back, or to create electronic music, proper usage also requires an account of the theoretic underpinnings of the involved systems.[2] The theoretic backbone of computer audio is "digital signal processing" (hereafter "**DSP**").[3] A would-be computer musician requires various levels of DSP knowledge in order to competently operate his or her software. The fluency required will be determined partially by the types of problems the user wishes to solve, and partly by the nature of the software environment in use.

[1] Some authors (e.g. Chadabe, 1997, chs. 5–11) treat computer music as a subset of electronic music for historical reasons; here computer music is electronic music for which a digital computer was used in some stage of its production. Others (e.g. Chadabe, 1997, pp. 277-285 for historical purposes, and Roads, 1996, chs. 18–19 for pedagogical reasons) distinguish between music for which electronics or a computer was used only to create or modify sounds and that for which a computer was used as a compositional aid. For an introduction to this distinction, see Puckette, 2006a. "Algorithmic composition," as it has come to be called, is a subset of the latter, as is "spectral music." (For history and techniques of spectral music, see Anderson, 2000 and Fineberg, 2000.) These two classes intersect in the case that a computer was simultaneously used as a compositional aid and a sound generator or modifier; a subset of this intersection has sometimes been called "interactive music" – see Chadabe, 1997, ch. 11 and Winkler, 1998). I prefer the term "computer audio" for sound created or processed by computers, and this thesis concerns itself primarily with the application of computers for sound generation; these other distinctions are less important. Another approach would be to include as part of the field those topics which are discussed in peer-reviewed journals like the Computer Music Journal.

[2] Thus the "technology" of contemporary music theory includes not only the familiar techniques of musical speculation, analysis, and synthesis, but has come to locate its foundation in propositional logic, abstract algebra, statistics, physics, cognition, and related branches of philosophy, science, and mathematics.

[3] Although the "processing" in computer-music literature sometimes informally implies the modification of existing sound, the sphere of DSP is much greater, and encompasses the synthesis or manipulation of any digital signal, which may be an audio signal, an image, or any other signal which may be represented digitally.

In the following sections of this thesis I offer an introduction to some of the recurrent problems involved with teaching DSP to musicians, and especially to university and conservatory composition students. This thesis can be considered the first part of a two-part study on the topic. In this first part I discuss the background matters, academic and intellectual tensions, and difficult pedagogical decisions inherent to teaching DSP to composers. Sections 2–4 (pages 2–7) speak to methodological issues specific to composers. Sections 5–7 (pages 7–22) are about specific computer-related problems facing an instructor when he or she is making decisions about how to assemble computer resources for a course. In a forthcoming second part I plan to set out a well-designed DSP core curriculum which could be worked into a year-long graduate computer-music course for composers. I provide a brief and rather informal outline of the design of this curriculum in section 8 beginning on page 22.

2 Why Teach DSP Fundamentals?

Perhaps the most justifiable argument *against* including a substantial DSP emphasis in computer-music curricula is the fact that nearly every modern computer-audio environment includes a wide array of intuitive and fairly easy-to-use tools. If I know how to *use* a tool competently, why should I need to know how it *works* "under the hood?" Furthermore, if I have access to a wide array of ready-made tools, surely I will spend my time more wisely by using them to create compelling music rather than reproducing something someone with a great deal more knowledge and experience has already built? Nobody needs to know about TCP to send an email, nor the intricacies of PostScript in order to print it, so why learn the mathematics it would take to build a band-pass filter from scratch when I can employ a similar filter with better performance and greater efficiency out of the box?[4]

There are at least two motivations for this line of reasoning that are unfortunately perniciously easy to conflate. One combines a rather healthy desire to avoid cliché with a caution against clumsily reinventing extant technology. The other, somewhat more insidious basis for the argument above orbits Gilbert Ryle's famous epistemological distinction between "knowing how" (sometimes referred to as "procedural knowledge") and "knowing that" (sometimes called "propositional knowledge").[5] That is, knowledge of *how* to perform a given task and knowledge *that* a given proposition is true are

[4] One analogy I have heard in this vein says, "I want to be the one driving the Ferrari, not the one building and repairing it!"

[5] Ryle, 1946; Ryle, 1949

different types of knowledge.[6] The most important upshot of this distinction is that it is possible to apply rules of operation intelligently without necessarily knowing what the rules are, how to formulate them, or even that the rules exist. It is the basis for a great deal of philosophical work on natural language, as well as some empirical cognitive research where it is necessary to induce unconscious "mental rules" from their past implicit applications.[7]

Implicitly valuing the procedural and the practical over the propositional and the theoretical is the essence of the second motivation for the argument at the head of this section. Conflation of the two occurs when one believes that because it would be distasteful and a waste of time and effort to reinvent the wheel, one can or should confidently refuse to learn or apply any theoretical knowledge about it.

I believe there are difficulties with both of these attitudes even when taken separately. The first implicitly undervalues the capacity of theoretical knowledge to act as a foundation for future original work and even to reinforce even already-learned procedural ability. The second either values theoretical knowledge only to the extent that one can immediately convert it to practical knowledge, or disavows its ability to contribute to practical ability entirely. In my experience composers who share these attitudes in any significant degree may have some academic inertia to overcome when presented with difficult technical material, and any computer-music teacher will need to have some strategies for dealing with this problem.

Fortunately, many tactics are available. However, since it can be very difficult to tease apart knowledge-that and complex forms of knowledge-how for any reasonably sophisticated set of intelligent behaviors, and since this is especially the case with music composition given the wide range of activities traditionally subsumed under "music theory," it may be advantageous to avoid a direct epistemological discussion in lectures. While this may seem counter to the spirit of my foregoing discussion, I believe that convincing students of the value of theoretical knowledge may be done Socratically, or

[6] Ryle's analysis is quite a bit more sophisticated than this, and primarily adheres to the following logical investigation. He wishes to refute both the notion that knowledge-how and knowledge-that are members of the same type of knowledge, and the idea that all intelligent actions are the result of some kind of propositional knowledge or have a purely mental and rational basis. His argument hinges on the fact that the deliberate contemplation of a known fact is itself an operation the execution of which can be carried out more or less well. If this is the case, then if any knowledge-how results from a prior knowledge-that, then the simple act of contemplating the propositional knowledge in question – a knowledge-how operation – would itself require a previous consideration of another, more complex proposition. This leads to a causality dilemma (a "chicken or the egg" problem) and infinite regress that is impossible to break.

[7] Music students are likely to be familiar with a related (but knobbier) distinction that permeates intellectual and music history, that between *theory* and *practice*. See Christensen (2002, pp. 2–13) for an introduction to the problem in the history of musical thought. A great deal of traditional "music theory" should properly be considered procedural knowledge, e.g. "how to correctly resolve a leading-tone" or "how to double the correct pitch in a deceptive cadence to avoid parallel fifths and octaves."

simply with clear teaching. The instructor's aim should be to help students expand their compositional freedom both by shifting the balance of constraint from software designers to them,[8] and by helping them become aware of the constraints which are their own but which may not be self-evident. This can only happen if students can learn to "do computer music" *and* explain how and why what they are doing works; however, I believe this to be a necessary but not always a sufficient condition for maximal compositional freedom.

3 Achievement and Artistic Productivity

If a course that emphasizes DSP is introduced into a music-composition curriculum for advanced undergraduate and graduate students, a portion of the frustration involved with such a course can usually be attributed to the fact that few of the musical skills the students will have cultivated previously will be immediately applicable to the material. That is, the students may be used to reaching a level of understanding and achievement in the majority of their courses that they may have difficulty reaching at first – they may feel like they are starting from scratch. An instructor should make this clear throughout the course, and should foster an environment in which confusion is acceptable and even expected but not unconquerable, and in which we would rather explain mystery away than delight in it.

Some course units can be made more efficient if the technical material, e.g. a specific math skill, is not explained abstractly beforehand but is woven implicitly in the discussion and then extracted explicitly after the students have a grasp of the more concrete instances of the idea. Language and definition should be as conservative as possible (i.e. conservative in the semantic sense), so that words have the same meaning from unit to unit – DSP is cobbled together from topics in a large number of fields, each of which has its own terminology, the potential for words "overloaded" with several definitions is high. Indeed much of the confusion I have seen students experience with DSP arises from the reckless application of DSP concepts like "frequency domain" in beginning synthesis classes, encouraging them to form intuitions about problems which are neither accurate nor pertinent. The only remedy is to be conservative with definitions, distinctions, and concepts from the very beginning.[9]

The computer-music instructor should be aware that many composition students will be turned off or bored by material that does not immediately stimulate compositional

[8] See section 7, page 20.

[9] Of course, this will be nearly impossible to maintain completely among the various texts and online resources used for course readings. In this case it is important for students to learn which terms are synonymous, and which have possibly ambiguous meanings.

ideas, or proclaim the existence of concepts that are exciting but too theoretical to apply. This is where the utility of compositional études is paramount. If the students are asked to write a single compositional étude for each topic covered in the course, and are provided with an adequate and intuitively arranged computer-audio environment in which to do it, they will tend to be more appreciative of the knowledge they have applied. Some students may even feel that through compositional exercise some of the difficult abstract concepts which previously seemed ethereal or protean will have reified themselves into something rather concrete and well-defined.

I have taught many students who have said, "it needs to make sound before it can make sense to me." To reassure students that what they are learning is not a waste of time, I have often found it useful to explain such activity as analogous to the pianist learning scales and études in order to become a more expressive musician, or to a composer learning ancient styles of counterpoint and recent techniques used by famous composers: these skills are not what "it" is ultimately about, but they serve to bolster the musician's facility and fluency. To be successful in computer music, however, a student may need to study quite a bit more technical material than in other musical fields.

The instructor should also be cautious not to underscore the potential tension between learning fundamentals and creating interesting and original music. I say "potential" because although I believe most interesting computer music is fundamentally sound from a DSP point of view, compositions written while the student is in the process of learning fundamentals may not attain the level of quality the student may feel he or she is capable of. While avoiding the banal and the dated may be artistically commendable, the composition student must not become paralyzed by fear of cliché; one way to do this is to decouple technical learning from the "necessity" of artistic production. In fact, it can even be useful for a computer-music student to attempt to recreate and extend the sounds of an existing piece by another composer as an exercise – the student can learn a great deal about computer-music history and analysis, composing with computers, and sound itself, while in the process also actively learning which techniques have already been used and may have run their aesthetic course.[10]

A composer may find that his or her orientation toward composition may change greatly when writing computer music. While I believe that every possible road into computer composition should remain open to a student, since each road is potentially uncertain I also believe it is important to spur their ways of thinking about composition in directions that may seem counterintuitive at first. For instance, for a composer who always has a sound in mind that he or she wants to achieve, it may be interesting to

[10] An example of this kind of project may be found in Puckette, 2006b. In this paper Miller Puckette investigates some interesting modern methods of reproducing the sounds in Charles Dodge's *Speech Songs*, a seminal composition which uses an early form of digital vocal synthesis.

induce that composer to start instead with a focused DSP concept to see what surprising sound might result from its application, and why.

4 Mathematics and Intuition

INVOKING "COMPOSERS" as though it were a class of people with similar abilities and interests is an error. When teaching composition students material that falls outside of normal musical training, it is impossible to count on any kind of shared education. Digital signal processing is a field which is inextricably interwoven with mathematics and physics. Even the most simple topics rely heavily on trigonometry, algebra, and statistics; the most important topics depend upon complex analysis, linear algebra, vector calculus, and numerical analysis, most of which are usually far beyond the math a composition student will have experienced in high school and college. Mathematical training among composition students tends to be very diverse, though, and almost all of them will have had some experience with basic algebra; some will have experience with trigonometry, and a few may have more advanced training yet. To complicate matters, sophisticated math is a prerequisite for most of the pedagogical DSP literature, and most of them are given to using algebraic proof and skipping "simple" algebraic steps. Unfortunately many students will find algebraic proofs inscrutable, and those who comprehend it may not feel they have a "tangible" understanding of the concept – they may understand the algebraic manipulations without having a feel for what they "mean."[11]

There are at least two ways a computer-music instructor may address this problem. First, he or she should try to make certain that the students are up to speed with any mathematical concepts they need to apply by requiring them to perform a page of exercises.[12] In order to be as intellectually inclusive as possible, a course needs to be structured in such a way that no skill or value is taken for granted and resources for quick review and remediation of basic concepts is available and clear. When I feel I must proceed with assumed mathematical knowledge, I have often given my students a brief mathematical primer at the beginning of the unit which covers the basic mathematical assumptions of that unit.

Second, and probably most importantly, the instructor should strive to tell an intuitive story about every topic contained in the course and any equation used in class or found in required reading. This is above all the purpose of the following sections of this thesis. In order to teach intuitively the instructor must always engage in active

[11] A corollary to this notion is that while an equation can be copied and therefore readily utilized in computer-audio environments, explaining why to use it, what it does, and how to extend it can be formidable.

[12] Hopefully the odium associated with doing math exercises in a composition course will not outweigh the exercises' pedagogical utility!

"demystification" of topics, which usually means he or she will have to reach slightly beyond the scope of common DSP textbooks. Many composers rely heavily on pattern recognition and location of symmetry in their own musical work, and so they may find explanations which induce them to use these already refined skills the most intuitive.

Nearly every perplexing concept can be illustrated with a cogent thought-experiment, a visual picture, and if appropriate an audible example. Equations may be more conceptually efficient, but a picture and a sound can provide more tangible information than a thousand lines of even the gentlest algebra. Ideally the thought-experiment-with-illustration excursion would lead the students through the steps necessary to enable them to express an idea mathematically. Put another way, the mathematical representation can most often serve as a distilled shorthand of an already "informally" understood concept, not as the springboard for the discussion. An instructor may find it helpful to discuss the topics at hand with an engineer or computer science specialist at his or her institution to learn about oft-used teaching techniques.

Third, as with any technical subject, DSP topics must be presented in an appropriate order. Finding a satisfactory order is complicated by the nature of the subject material, which resists "linear" explanation – some DSP topics form "explanatory loops," where concepts in each topic are needed to explain concepts in the other. For example, in order to discuss convolution, one must first have a knowledge of frequency-domain signals and how they are computed from time-domain signals. But in order to discuss the implications of any operation in the time domain or frequency domain one must understand convolution. When the explanatory chain is circular rather than linear, a group of concepts that would normally be subsumed under one "topic" may need to be broken into two or more parts. In my experience I have found it problematic to introduce a concept and then promise the students that it will be explained later; the amount of information students are willing to take on faith alone is limited, and asking them to do so effectively amounts to asking them to temporarily suppress intellectual curiosity.[13]

5 Computers

COMPUTERS are nearly omnipresent in university settings. Most university students will now have spent much of their lives operating computers, but this is a mixed blessing for computer-music pedagogy. On one hand students may find the computer-related

[13] Concepts which have to be logically deduced from axioms must by definition depend upon taking the axioms on faith, and most axioms are postulated on intuitively "true" ideas; deciding which ideas are to be taken axiomatically is a primary pedagogical activity.

tasks easier and more straightforward, and may have less of a learning curve for complex applications. On the other hand, though, a student might "already know how computers work and what they are for," and may thus be resistant to learning new ways of using a computer, or may balk at using programs which do not follow the standards of typical (usually commercial) user interfaces.

The latter reaction is especially typical for open-source software,[14] much of which is powerful and well-designed but may not conform to commercial standards. A contingency of open-source programmers regard these commercial standards as flawed in the first place. Some also require a greater knowledge of the operating system for which they were written than most proprietary software; for instance many open-source programs for computer audio require the user to be comfortable with issuing commands via the command-line interface. A large percentage of open-source software is also poorly or clumsily documented.

Certainly the advantages of open-source software greatly outweigh the disadvantages in pedagogical settings. In fact, I believe that open-source software is the only ethically tenable option for teaching; where possible no student should be made to be dependent on the resources in the music studio, nor should they be made to purchase expensive software in order to learn and be productive at home. If two applications, one proprietary and one open-source, share a common purpose, design, and interface, the open-source version should be the primary one introduced and used in any educational setting even if it lacks a few features of the proprietary version. Alternatively, if an instructor wants his or her students to be aware of proprietary software to ensure they are familiar with the state of the art, or for purposes of job placement, a commercial application can be taught side-by-side with an open-source counterpart. This also affords students a choice in the matter.[15]

[14] That is, software for which the programmers' source code is available to peruse, compile, modify, and develop, and which is usually free to download, use, and distribute to others. For example see http://www.gnu.org/licenses/gpl.html for the Gnu Public License and http://www.opensource.org/licenses/bsd-license.php for the Berkeley Software Distribution License, two popular open-source/free-software licenses.

[15] There are also open-source operating systems which may be downloaded and used for free, and for which software packagers maintain repositories of open-source multimedia software. The most prominent open-source operating system for multimedia is Linux (http://www.linux.org/); currently the most popular multimedia Linux platforms include "Ubuntu Studio" (http://ubuntustudio.org/), which uses "Ubuntu" Linux (http://www.ubuntu.com/) as its main operating system, "Planet CCRMA at Home" (http://ccrma.stanford.edu/planetccrma/software/), which runs on "Fedora" Linux (http://fedoraproject.org/), and "pure:dyne" (http://code.goto10.org/projects/puredyne/), a "live distribution" or operating system that is not installed on a computer but which is instead booted and run entirely from removable media.

6 Computer-Audio Environments

There are generally two types of computer software one can use to learn audio-based DSP: applied math environments like MATLAB and Octave,[16] or computer-audio environments. While both are potentially useful, composers will generally want to learn DSP *and* use it for composition; therefore computer-audio environments find their way into computer-music curricula more often. One extremely important decision a computer-music instructor will need to make is which computer-audio applications to include in the course. An ideal environment should support both pedagogical uses and "production" uses equally, but this can be a difficult balance; not all environments support solutions to all DSP problems equally well, and each presents different advantages and challenges for compositional use.

6.1 Unit Generators and Block Diagrams

Most musical DSP problems involve manipulation of audio data using a large set of individual, archetypal tools called "unit generators" (hereafter "**UG**(s)"), the purpose of which is to break problems into several small modular pieces.[17] UGs usually have inputs and outputs, as well as control handles; the output of one UG can be input into another, allowing them to be chained together to create DSP networks, which in some cases can be thought of as more complex UGs. DSP networks are called "instruments" or "patches" in some common computer-audio environments. UGs can be made for any conceivable audio-signal generation or manipulation.

The UG concept also allows solutions to many problems to be written in a "DSP block diagram" – a kind of "flow chart" showing how the UGs relate to one another, as well as the order of operations. Documentation for nearly every computer-audio environment/language uses block diagrams for pedagogical purposes, and some of them even provide the user with a "graphical user interface" or "GUI" (pronounced "gooey") to connect unit generators in a block-diagram format.[18] To the extent that a block

[16] MATLAB is proprietary, while GNU Octave is open-source. See http://www.mathworks.com/ and http://www.gnu.org/software/octave/ for details.

[17] The unit-generator concept was first proposed and implemented by Max Mathews in the computer-music language "Music III," which was programmed by Mathews and Joan Miller in 1960 (Roads, 1996, p. 89).

[18] These languages are sometimes called "dataflow programming languages." Open-source applications using a block-diagram design include "Pure Data" or "Pd" (http://puredata.info/), designed and distributed by Miller Puckette; "Pd-extended" is a set of external libraries for Pd written maintained by other developers.

"Desire Data" (http://artengine.ca/desiredata/), is based on Pd and implemented by Matthieu Bouchard and Chun Lee.

"jMax" (http://freesoftware.ircam.fr/rubrique.php3?id_rubrique=14) is also based on Miller Puckette's work at IRCAM, and is developed by Françoise Déchelle et al. and distributed by IRCAM.

The "CLAM Network Editor" (http://iua-share.upf.es/wikis/clam/index.php/Network_Editor_tutorial)

diagram is useful for representing the solution to the DSP problem at hand, software which implements the same representation on-screen is exceedingly useful for Socratically "discovering" and improving a solution to the problem piece-by-piece in a lecture or individual instruction.

6.2 Graphical Dataflow Environments – Pure Data

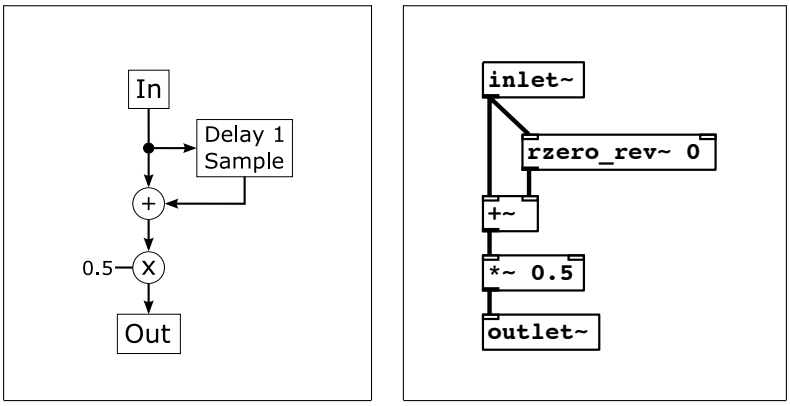


Figure 1: **Left.** A block diagram for a very simple low-pass filter.
Right. The same filter implemented in Pd (`rzero_rev~ 0` is a UG for a 1-sample delay).

Figure 1 shows an example of a block diagram for a very simple low-pass filter, and the same filter implemented in Pd, an open-source computer-audio environment (see note 18). The details are unimportant; the point is to notice the similarity between the two. This is a DSP example that can run in "real-time" – that is, a live sound source may be fed into the filter and the processing can occur almost instantaneously. Most of the graphical programs consist of a sophisticated sound "server" which is responsible for performing all of the DSP calculations, determining the order of those calculations, and for passing audio samples to and from microphones and speakers via the operating system, and a graphical "client" which provides an environment for a user to create and edit patches, sends mouse clicks and keyboard events to the server, and records information from the server to present to the user in graphical form.

Not all problems are easily represented in a single, static block diagram. Some problems require dynamic solutions where a block-diagram representation would need to change over time. More typically the problem will require the DSP network associated with the block diagram to be used as a template called a "class," "object class," or "ab-

by Xavier Amatriain et al., builds on previous work by Xavier Serra.

Proprietary software includes "Max/MSP" after work by Miller Puckette, developed by David Zicarelli, and distributed by Cycling '74 (http://www.cycling74.com/), and "Reaktor," distributed by Native Instruments (http://www.native-instruments.com/index.php?id=reaktor_us).

straction," which needs to be copied and loaded – "instantiated as an object" – multiple times with different values for the control parameters. For example imagine a primitive "glockenspiel" synthesizer, which can be generated by a simple sine-wave oscillator the control parameter of which is **frequency**, multiplied by a decay envelope the control parameters of which are peak **amplitude** and the **length** of the decay.Figure 2 shows a block diagram for this simple "instrument class."

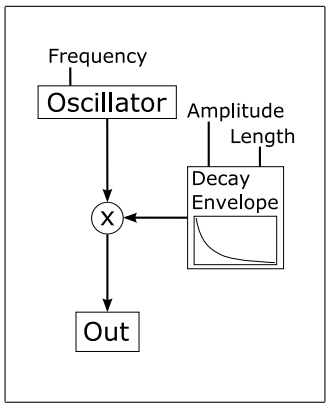


Figure 2: A block diagram for a primitive "glockenspiel" instrument class.

In order for this to be a useful synthesizer, we need to be able to ensure that one "note" can begin before another ends; that is we need to allow for simultaneity and polyphony. This becomes difficult in most visual block-diagram languages, because there is usually a one-to-one mapping from what is visible in the network to what is actually in the network. This means that the user must decide in advance how many voices of polyphony are available by loading several instances of the object class (represented by the block diagram) and sending control messages to them individually and sequentially;[19] this model is also common among vintage keyboard synthesizers that also have a polyphonic upper limit.

Figure 3 shows how this might be done in Pd. Again, the specific details of the Pd code are immaterial to the discussion at hand; it is more important to look at the structural details of the graphs to see how they correspond to the problem described. The patch on the left-hand side is a Pd implementation of the block diagram from figure 2. The three `inlet` boxes receive messages for the three control parameters frequency, amplitude, and length from, say, a MIDI keyboard, a note list ("score"), or an automated compositional algorithm. The decay envelope is generated by the chain on the right, and it is applied to the oscillator – `osc~` – on the left. The `outlet~` sends the resulting audio out. On the right-hand side of figure 3, the patch on the left-hand

[19]Most of these enviroments now have tools to automate this process so that the individual instances of the instrument are out of sight, but nonetheless the number of voices is not fully dynamic.

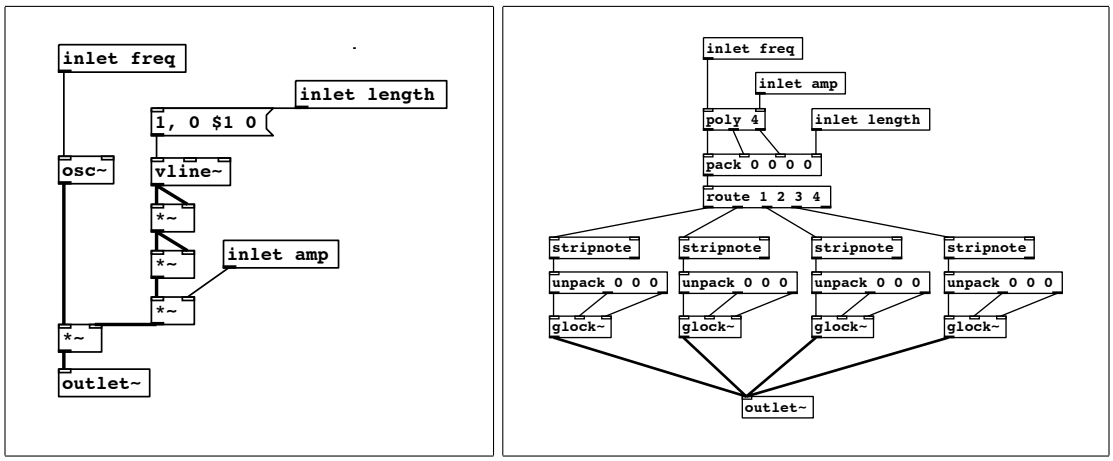


Figure 3: **Left.** Pd implementation of the glockenspiel block diagram, known as a "patch." **Right.** A more complex glockenspiel instrument with four-voice polyphony.

side is instantiated four times as `glock~`, providing four voices of polyphony (in this graph, notice that `glock~` visually has three inlets and one outlet). The rest of the patch sends messages to them sequentially so that all four of them can play "notes" that overlap in time. Also, it is worth noting that it still has the same three `inlet` boxes and one `outlet~` box – it has "inherited the interface" of the monophonic version, meaning it can receive the same messages but perform differently (in this case polyphonically). Whichever one is used, the instrument will be computing samples even during silence; it is always "on," but audible sound is generated only when it receives a message to apply the amplitude envelope. The "always on" feature is a characteristic of most of the graphical languages, and stems from the "server-client" model discussed above. Users of most of these languages now enjoy the option to turn off a patch when it is not in use.

While this polyphony predicament may seem trivial for something like a glockenspiel, there are synthesis techniques which can require as many as a few thousand voices of polyphony on demand (e.g. granular synthesis). To overcome this obstacle a program will need to be able to load and unload copies of the template dynamically (or to use common computer parlance, the program will need to "instantiate" or "allocate" the class as an "object" – instance of the class – and then will need to "free" or "deallocate" or "collect" it when it has performed its task). An application which can do this will usually run more efficiently because it will only instantiate as many objects as it needs at a given moment saving both space in computer memory and computations on the processor.

There is no reason that a graphical block-diagram-type environment could not do this, but the user would have to simply believe that it was working correctly, since in the process of allocating and deallocating the instrument it would not become visible; some

who regard this type of language as more or less WYSIWYG[20] may believe that anything that is present but not even potentially visible is a violation of that model.

Since there are some DSP techniques for which a graphical program may be less efficient pedagogically (as well as computationally), and since students will benefit from learning more than one computer-audio environment, I will address two important non-graphical models here. The first follows an important thread of computer-music history beginning with Max Mathews's research at Bell Labs, which can be traced through the various MUSIC-N languages to one culmination in Barry Vercoe's program Csound.[21] Csound is an especially rich environment because of its long and distinguished history; it also boasts the greatest collection of UGs likely available in any software environment, at over 1200. The second model is a text-based server-client model with one manifestation in the excellent SuperCollider computer-audio language by James McCartney.[22] Both Csound and SuperCollider are open-source.

6.3 Csound

In the traditional MUSIC-N/Csound model, the user codes a text-based set of "instruments" called an "orchestra," and then provides it with a text-based "score" to render. The score is simply an event list that tells the program what instruments to instantiate, at what times, and with what initial control parameters; an event is often called a "note." When translated into English, a set of notewise instructions for an instrument might say, "play a note which begins 2.3 seconds from the beginning, which lasts exactly 3.4 seconds, on middle C, and at half the maximum amplitude." This set of instructions would require 5 control parameters: 1. the instrument which is to play the event,[23] 2. the time at which the event begins, 3. the duration of the event, 4. the pitch of the event, 5. the amplitude of the event. For more control over complex events, more "parameter fields," or "p-fields" are required – some instruments require more than a hundred.[24]

Csound can render a score to a file (e.g. a .wav file), or it may play its samples directly to an audio interface in real-time. Csound is also fully polyphonic: when a voice of polyphony is needed, it allocates it automatically.[25] Figure 4 is an example of how the

[20] "What you see is what you get."

[21] http://www.csounds.com/. I say "one culmination" because nearly every DSP/synthesis language is heir to the ideas pioneered by Mathews and others, particularly the unit-generator innovation.

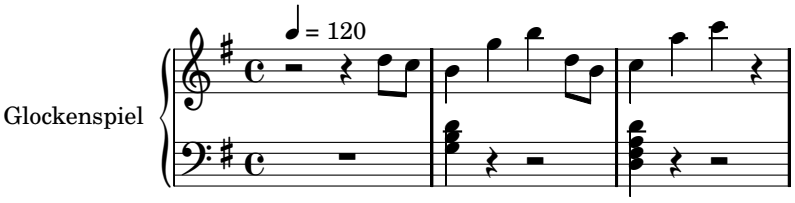
[22] http://supercollider.sourceforge.net/.

[23] It is possible that the "orchestra" might have more than one instrument from which to select.

[24] In Csound the upper limit for the number of p-fields in an individual instrument is currently 800. (ffitch, 2000, p. 113)

[25] Its polyphony is limited only by the capabilities of the hardware on which it is running; Csound will continue to allocate instruments until the machine runs out of usable memory or, if it is running in real-time, until the machine's CPU is no longer able to keep up with the computations required to generate the samples.

previous glockenspiel instrument might be implemented in Csound. On the left-hand side is an "orchestra" with one instrument, the code of which has almost exactly the same structure as the block diagram from figure 2.[26] Lines 10, 11, and 12 are respectively analogous to the "length," "amplitude," and "frequency" control parameters in figure 2, and line 22 is analogous to the `Out` box. An exponential decay envelope is generated in line 16 using the length and amplitude values from lines 10 and 11.[27] Line 19 is the oscillator setup.




```
01  ; header
02  sr = 44100
03  kr = 22050
04  nchnls = 1
05  0dbfs = 1
06
07
08  instr 1
09
10     ilength = p3
11     iamp = p4
12     ifreq = p5
13     itable = 1
14
15   ; envelope:   amp   length   endamp
16     kenv expseg iamp, ilength, .0001
17
18   ; glock setup:  env.  freq.  table
19     aglock oscil3 kenv, ifreq, itable
20
21   ; output
22     out aglock
23
24  endin
```

```
01  ; table time size gen harm-amp.
02  f 1     0    1024 10  1
03
04  ; "notes"
05  ; instr time  length amp.  freq.
06  i 1     0.000 3.293  0.050 2349.318
07  i 1     0.250 2.565  0.064 2093.005
08  i 1     0.500 3.027  0.078 783.990
09  i 1     0.512 2.294  0.078 987.767
10  i 1     0.526 4.253  0.078 1174.659
11  i 1     0.540 2.364  0.078 1975.533
12  i 1     1.000 2.946  0.108 3135.964
13  i 1     1.500 2.757  0.138 3951.567
14  i 1     2.000 3.939  0.168 2349.318
15  i 1     2.250 3.127  0.184 1975.533
16  i 1     2.500 3.301  0.200 587.330
17  i 1     2.512 4.676  0.200 739.990
18  i 1     2.526 3.710  0.200 880.000
19  i 1     2.540 3.027  0.200 1174.659
20  i 1     2.556 2.827  0.200 2093.005
21  i 1     3.000 2.098  0.232 3520.000
22  i 1     3.500 2.675  0.266 4186.010
23
24  e
```

Figure 4: **Left.** Csound implementation of the glockenspiel block diagram, known as an "orchestra." **Right.** A Csound event list – or "score" – for the instrument, in this case a phrase from a glockenspiel solo in *Die Zauberflöte*, shown in musical notation above.

[26] All lines which begin with the `;` character are "comments" and are ignored by Csound. Lines 2-5 are unimportant to this discussion; they simply define some global parameters for the resulting file (its sample rate, how many channels, etc.).

[27] The `.0001` indication gives the envelope a target amplitude to reach over the length of the decay.

Oscillators in Csound are a bit more sophisticated than in Pd, and require independent controls for amplitude, frequency and the waveform through which to oscillate; thus the multiplication from the block diagram is unnecessary since the amplitude envelope can be used directly in the oscillator's control statement. Note how much more efficient text-based instrument definitions can be – what took more than 10 boxes in Pd is done here in just a few simple (but rich) lines of code.

On the right-hand side is an event list – or "score" – for the instrument, a stirring rendition of an antecedent phrase from a famous glockenspiel solo from Mozart's Singspiel, *Die Zauberflöte*, shown also in music notation above the example. Line 2 defines a table of numbers which stores the sinusoidal waveform, and lines 6-22 are each notes. The columns of the notes – the "p-fields" described above correspond to the control parameters defined in the instrument. The first column indicates an "i" statement, meaning it will define an event. In Csound the first p-field (second column) always indicates which instrument plays the event (in this case, "instr 1," our glockenspiel), the second p-field defines the onset time for the note counted in seconds from time=0 (the beginning), and the third p-field is the duration of the note in seconds. Subsequent p-fields are defined by the user in the instrument; in this case the only other parameters are amplitude and frequency.

In the musical notation it seems as though the greatest number of polyphonic voices required in this passage is five. However, since this is an instrument with a fairly long decay, we must be prepared to allocate many voices of polyphony to accommodate both decaying notes and new attacks; it turns out that all eight notes from the first five beats of sound will still be "ringing" when the five notes at the beginning of the third bar enter. Fortunately Csound handles this automatically and virtually instantaneously. There are other features of the score that may be of some interest. One can tell by the second p-field (onset time) that lines 8-11 and 16-20 are rolled chords. The third p-field seems to define durations selected at random from between 2 and 5 seconds; giving each decay a slightly different length will increase the "realism" of the instrument. The fourth p-field indicates a rather steady crescendo, and one can follow the pitch contour of the passage in the fifth p-field.

Csound's historically built-in "score" idea makes it a bit cumbersome to use by itself for some projects, and affects composition in at least three ways. First, some improvements could be made to the score to make it easier to follow – for instance the fifth p-field could be written in a specialized pitch notation and converted to frequency in the instrument – but in general Csound scores are not optimized for human readability especially when many p-fields are required. Second, time representation in Csound is problematic; since all events are referenced to the beginning of the score in seconds, it

becomes difficult to use for long passages.[28] Several score preprocessors for Csound exist to aid in the generation of event lists like this from a more human-readable format.[29] Third, although Csound is a computer-audio environment that fully supports real-time processes, the score makes some real-time processing tasks difficult. For instance, something like a reverberator which in an environment like Pd could simply sit at the end of a DSP chain and run until the program is turned off, in Csound would need to be instantiated – almost perversely – as a "note" in the score with a specific duration; there are tools in Csound which can be used to avoid this kind of difficulty, but the score's legacy remains. However, Csound can be "front-ended" by (that is, run from within) other computer-audio or algorithmic composition environments.[30] In many cases a Csound front-end can be used to generate note-events automatically and in real-time, so that it resembles the server-client model discussed briefly above.

6.4 SuperCollider

The computer-audio environment SuperCollider combines the real-time server-client model found in GUI programs like Pd or in Csound front-ends, the efficiency and expressivity of text-based code, and the conceptual power of "object-oriented programming."[31] In my opinion SuperCollider is among the most the most versatile and powerful environments for several reasons. First and most importantly, it behaves like a real programming language,[32] supports and encourages many types/styles of programming, and thus does not force a particular way of thinking about music; there are many stylistically appropriate ways to represent time, pitch, or most any musical parameter. Second, this program employs one of the most robust and elegant separations of client from server, making it more reliable in performance situations. This design permits the server to be used as a stand-alone program, of which the language itself is a client; this means that many other clients can be written for the server, and some programs, such as Pd, can already be used to control it. While Pd and its graphical cousins are unwieldy tools for sophis-

[28] This fact also limits the length of the score, since there is a limit to the largest number one can represent on a computer. For instance, if one were to use the settings from the "header" of the orchestra in figure 4, the longest score one could create would "only" last just over 27 hours and 3 minutes. (Vercoe et al., http://www.csounds.com/manual/html/ScoreTop.html)

[29] For example Aleck Brinkman's "score11" (http://ecmc.rochester.edu/ecmc/docs/score11/score11.html) and Mikel Kuehn's "ngen" (http://mustec.bgsu.edu/~mkuehn/ngen/).

[30] See Vercoe et al., http://www.csounds.com/manual/html/OviewFrontEnds.html and also http://www.csounds.com/frontends/for some examples; because Csound's history is so rich, there are far too many front-ends to list. One particularly useful case is the `csoundapi~` object class for Pd, which allows a user to control Csound from within Pd.

[31] Also of note is "ChucK," another open-source, cross-platform, and object-oriented computer-audio environment by Ge Wang and Perry Cook. ChucK will become increasingly important in the field of computer music as its code matures. See http://chuck.cs.princeton.edu/.

[32] In fact its syntax is based upon the object-oriented programming language "Smalltalk."

ticated polyphonic synthesis, and the event list is an encumbrance for those who would use Csound for real-time processing, SuperCollider's language design is equally at home for both uses.

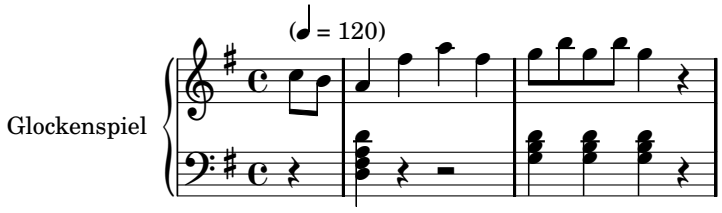
To show one possible way that SuperCollider could be used, figure 5 is a SuperCollider implementation of the glockenspiel instrument, which is programmed to play the consequent portion of the *Zauberflöte* phrase from before (the musical notation is again given above the figure). In line 2 a server is instantiated.[33] Lines 5–12 are the by now familiar glockenspiel instrument definition. Line 5 says that the instrument (or "Synth") will be named "Glock," and it will expect messages for frequency, amplitude, and length. Line 7 defines the decay envelope: `Env.perc` is an envelope method which will generate a decay envelope that one might expect for a percussion instrument, using the given length and amplitude (the `-8` argument is a measure of how sharply the envelope decays – one might want something with a sharper decay, say `-16`, to synthesize something resembling a marimba). Line 8 is the oscillator, to which the frequency and decay envelope are applied, and line 9 is analogous to the `Out` box in the block diagram in figure 2; line 10 sends the instrument definition to the server to await event instructions. Line 12 instantiates a "TempoClock" which is used to schedule the events at two beats per second.

Lines 14–48 create a function named `~notes`, which is like a little program which defines and executes the events. In order to show how a function like this might be used in composition, I decided to break the excerpt into several conceptual chunks. First, I have separated right-hand figures from left-hand figures. Lines 16 and 17 define `rhFreq`, a "pattern sequence" (or "`Pseq`") of all the frequencies for the right hand in order.[34] The beginning of line 17 – `Pseq([3135.964,3951.567], 2)` – is itself a pattern sequence which repeats twice, indicated by the `2` before the close-parenthesis, and corresponding to the repeated G-B in the 2nd full bar of the excerpt. The rhythm sequence `rhRhythm` in line 19 takes even more advantage of this kind of chunking, being a a sequence of two half-beat events, four full-beat events, four half-beat events, and one full beat event (the trailing `0` stops the stream of events).

The left-hand events are even more redundant. For the frequency components I have defined a D-major group of frequencies (`dMaj`) and a G-major group of frequencies (`gMaj`) (the latter repeats three times) in lines 24 and 25, respectively. They are nested into the full pattern in line 26 (the leading `0` is there to indicate a one-beat rest at the beginning). Following this in bars 28–30 is a similar grouping of the left-hand rhythmic events. First

[33] Any line which begins with `//` is a comment and is ignored by the SuperCollider language interpreter

[34] As with the Csound example, in a real-world example one would probably enter the notes in a more human-readable fashion, inducing the software to convert those values to frequency. I have eschewed this in order to keep the instrument definition as conservative as possible.




```
// Setup
s=Server.local.boot;

(
SynthDef("Glock", { arg freq, amp, length;
  var glock, env;
  env = EnvGen.ar(Env.perc(0,length,amp, -8),1.0,doneAction:  2);
  glock = SinOsc.ar(freq, 0, env);
  Out.ar(0, [glock, glock]);
}).send(s);

~clock = TempoClock(2);

~notes = { var rhFreq, rhRhyth, rhAmp, dMaj, gMaj, lhFreq, fourRoll, threeRolls, lhRhyth, lhAmp;
// right hand
  rhFreq = Pseq([2093.005, 1975.533, 1760.000, 2959.996, 3520.000, 2959.996,
            Pseq([3135.964, 3951.567], 2), 3135.964]).asStream;

  rhRhyth = Pseq([Pseq([0.5], 2), Pseq([1], 4), Pseq([0.5], 4), 0]).asStream;

  rhAmp = 0.250;

// left hand
  dMaj = Pseq([587.330, 739.990, 880.000, 1174.659]);
  gMaj = Pseq([783.990, 987.767, 1174.659], 3);
  lhFreq = Pseq([0, dMaj , gMaj, 0]).asStream;

  fourRoll = Pseq([0.05, 0.07, 0.08, 3.8]);
  threeRolls = Pseq([0.04, 0.06, 0.9], 3);
  lhRhyth = Pseq([1, fourRoll, threeRolls, 0]).asStream;

  lhAmp = 0.200;

// sequence patterns
  ~clock.schedAbs(~clock.beats,
    {Synth.new("Glock",
      [\amp, rhAmp=rhAmp*0.9,
      \freq, rhFreq.next,
      \length, rrand(2.0,5.0)]);
    rhRhyth.next});

  ~clock.schedAbs(~clock.beats,
    {Synth.new("Glock",
      [\amp, lhAmp=lhAmp*0.85,
      \freq, lhFreq.next,
      \length, rrand(2.0,5.0)]);
    lhRhyth.next});
};
)

~notes.value;
```

Figure 5: SuperCollider implementation of the glockenspiel block diagram, with a phrase from the *Zauberflöte* solo generated using "pattern sequences."

I implement a four-note roll (`fourRoll`), and then a three-note roll repeated three times (`threeRolls`); each is nested into the `lhRhythm` pattern in line 30.

Lines 35–47 use the `~clock` from line 12 to assemble, sequence, and schedule the patterns. In the first group of statements in lines 35–42, the code is telling the `~clock` to schedule (`~clock.schedAbs`) an event at time=now (`~clock.beats` indicates the current time). The event will consist of a new allocation of the "Glock" instrument that was stored previously on the server (`Synth.new("Glock"...)`, iterating over the right-hand frequency pattern once per event (`rhFreq.next`), and scheduling a new event the number of beats found in the ordered sequence of right-hand rhythms (`rhRhythm.next`). Amplitude for the right hand is initialized in line 21 (`rhAmp = 0.250`); in line 37, the `rhAmp=rhAmp*0.9` statement says, "use the current value of `rhAmp`, and multiply it by 0.9 – this will be its value at the beginning of the next event." Therefore, since the right-hand amplitude values will decrease for each successive note, the passage will follow a decrescendo. As in the Csound example, the length of each event is a random value between 2.0 and 5.0 seconds (`rrand(2.0,5.0)`). The left-hand events scheduled in lines 42–47 follow the same logic. Finally, one can play the example by evaluating the function, shown here in line 51; note that SuperCollider is also fully polyphonic – it has no trouble allocating even thousands of overlapping events, to within the limits of the hardware it is running on. In fact, one could even evaluate the `~notes` function many times in sequence to generate multiple, independent, and simultaneous instances of this polyphonic function – a "polyphony of polyphonies."

While the foregoing may seem laborious for such a small piece of music, one should be able to imagine the expressive power of the language for larger projects, or for algorithmic composition. The user is free to allow the programming for instrument definition and event timing to intermingle or to be separate. One can schedule events sequentially as in a Csound score, create functions similar to the one in the example to assemble events in chunks, or create routines to automatically algorithmically generate events. Another important feature of SuperCollider that results from its nearly complete separation of server from client is the ability to begin a process and then write the code for another process during a performance. This "live coding" allows an experienced SuperCollider performer to improvise computer music intelligently and with facility; it is much more difficult in the other two language models discussed above.

6.5 Summary and a Dilemma

To summarize, each of three models discussed here has its assets and liabilities. GUI models like Pd are useful for teaching DSP because they graphically show the structure of a DSP chain in the same manner one might draw them in block diagrams. They are

also very useful for building interfaces for musicians to use to play back sound files and to process live sound. However, they are often burdensome to use for note-based or algorithmic synthesis, because it is difficult to polyphonically allocate new instances of instrument definitions. Csound is useful for teaching because of its historical importance, its rich potential for expressive synthesis, its computational efficiency and quality of sound, and its relatively simple syntax for designing instruments and scheduling events. However, Csound's historically built-in score idea presents real difficulties when using it for real-time processing. This is mitigated in part by the inclusion of UGs designed for real-time use, and can be eliminated entirely when running Csound from within another environment, for instance using `csoundapi~` in Pd. SuperCollider represents an environment closer to computer programming. SuperCollider is useful for teaching because it it is highly versatile, and encourages imagination and ingenuity. It is equally useful for sound synthesis, creating GUIs for musicians to use in real time, and algorithmic composition. Its sound quality is as high or higher than Csound's and Pd's, and it is just as efficient as Csound, and several orders more efficient than Pd.

A dilemma might be apparent from the foregoing. In order to learn how to implement a given DSP concept, a student will need to learn not just the material involved with the DSP concept, but will also need to know a computer-audio environment well enough to implement the concept. Sometimes this means that a student will need to spend as much or more time learning the technicalities of an environment as learning DSP fundamentals. This is not a bad thing, for facility with one or more environments will help the student be more productive once he or she has learned the fundamentals, but it does mean that class time will need to be devoted to teaching language syntax, and the instructor will need to show patience while the students learn it. Also, the instructor has the option of using one piece of software primarily while merely introducing others, or of creating examples for each of two or three languages centered on each particular DSP topic, in the meantime highlighting the useful and tricky features of each application. I strongly favor the latter strategy if there is time in the course and if the students are up to the challenge, since they may embark upon their respective journeys through the fundamentals of DSP and computer-audio programming concomitantly.

7 The Black-Box Problem

YET ANOTHER serious dilemma presents itself here. Intelligent and eager programmers continue to add to the tools of computer-audio software, making them greater in both multitude and sophistication. Even in relatively "low-level" programs like the three

discussed in the last section,[35] the sheer availability of good tools often tempts students to use the most advanced ones available before they have mastered the fundamental ideas germane to them. They may wish to use them for fear that there is not much of interest one can produce with lower-level tools, or simply because they carry potential for creating engaging music. I call this the "black-box" problem: an operose machine which boasts many handles to manipulate, but the inside workings, intended use, and/or the nature of the end product of which are mostly unknown. Additionally, many of these tools begin to experience "feature bloat," that is, their creators tried to build a tool capable of doing everything.

This obviously echoes deeply the range of topics discussed previously in sections 2 and 3, but I discuss it here for another reason. Aside from being detrimental to learning, composition with black-boxes encourages application dependence – if students become attached to particular tools, when they are forced to move for whatever reason[36] to an environment in which that tool does not exist in the same form, they will find it difficult to get anywhere close to the same level with the new environment. But if, on the other hand, they know what the tool does and how, they will be able to cobble something like it together using the new set of tools. The feature-bloat problem is probably more pernicious; if a tool is designed to do too many things, it tends to be inflexible with regard to *how* those things are accomplished, and so forces one to think in one way when using it. Furthermore, if the number of handles to control becomes overwhelming, it becomes tempting to rely on the built-in defaults. For pedagogical purposes a collection of smaller tools which can be strung together to solve a particular problem is much more preferable.

With this in mind, it is also obvious that there is no way anyone could learn the details of every process underlying his or her favored means, so it is up to the student and the instructor to agree upon the line at which a tool becomes a black box. It is usually the case that lower-level UGs in computer-audio languages (for instance, phasors, oscillators, filters, etc.) tend to function in very similar or identical ways from one application to the next, while higher-level UGs (for instance, reverberators, chorusers, pitch-shifters, etc.) do not. However, the higher-level UGs can almost always be built using the lower-level ones as building blocks. For this reason, I find it appropriate to treat the shared UGs from among several prominent computer-audio environments in the manner of "axioms" from which higher-level devices may be deduced. In addition, employing an environment with relatively few UGs, like Pd, can help stave off both black-box and feature-bloat problems.

[35] Low-level, that is, when compared with the ubiquitous "digital audio workstation" (DAW) environments such as Pro Tools, which typically present with a preponderance of "plug-ins."

[36] For instance if the program they are used to using is no longer developed or supported to run on new operating systems.

After students learn the low-level concepts, they can be unfettered and allowed to use any existing tool they please, as long as they can explain what it does, and if possible, how it works.[37]

8 An Outline for a DSP Curriculum

In this section I provide a brief outline of a DSP core curriculum for a computer-music course at the graduate level. The specific details and content of the course will be the subject of a forthcoming second part of the current paper, and therefore the topics are covered in less depth here. The following thus requires some knowledge of the topics involved. Where appropriate I give an illustrative example, but for the most part in this outline terms are left undefined. The DSP core is meant to fit into a broader course which covers some history of computer music, as well as other techniques which rely on DSP but are not properly part of the underlying DSP fundamentals. In particular, the reader may notice the absence of *modulation synthesis* – ring modulation and amplitude modulation are best explained as a result of convolution, while there are few good ways to intuitively explain the results of frequency modulation without venturing into difficult mathematics,[38] except to show what occurs when nearly identical signals are created using frequency modulation on one hand and piecewise additive synthesis on the other, indicating a complex phase interaction. Also perhaps conspicuously missing is a section on sound localization, room simulation, and other kinds of physical modeling. These topics involve basic DSP building blocks, including delay lines, convolution, filtering, and also vector analysis. Each also relies heavily on psychoacoustics. Therefore they are also not part of DSP proper but draw upon its fundamentals; any course with units on these topics will benefit from solid DSP fundamentals.

A word about course readings belongs here. It is unlikely that one textbook will suffice for a course like the one outlined below. There are only a few which are aimed at musicians (e.g. Roads, 1996), and those tend to avoid some of the most important but

[37] If time is of the essence in the course, the instructor may again need to strike a balance between insisting upon fundamentals and allowing students to be compositionally productive.

[38] Specifically, there is no pleasant way to derive an accurate picture of the fallout of frequency or phase modulation without appeals to differential equations and Bessel functions. A formula for frequency modulation which is derived from several trigonometric identities and an infinite expansion is

$$cos(\omega_c t + I sin(\omega_m t)) = \sum_{n=-\infty}^{\infty} J_n(I) cos((\omega_c + n\omega_m)t)$$

where ω_c is a frequency measurement of the carrier signal, ω_m is a frequency measurement of the modulating signal, and J_n is the n^{th} order Bessel function of the first kind. Note that this explicitly shows the amplitude and frequency of the various sidebands associated with frequency modulation. See Loy, 2007, pp. 389-402.

difficult topics outright, confine them to appendices, or point to other sources for further reading. On the other hand, most DSP books are written with scientists or engineers in mind (e.g. Smith, 1997), or are written for mathematicians with the requisite rigor of proof. The best set of readings is likely to be a host of excerpts from a number of sources. In the spirit of the foregoing, the instructor should make students aware of a great deal of free online DSP resources of varying quality and aimed at various audiences.

8.1 Unit 1: Signals, Sound, and Sampling

In this unit students learn or review the very most basic fundamentals of acoustics, learn how the physical concepts of sound (e.g. *frequency*) correlate to psychoacoustic concepts of sound (e.g. *pitch*). Finally they learn how these properties of sound are represented digitally, and what the consequences of "digitizing" the sound are for the information contained in the signal. Beginning the course with discussion of familiar phenomena should allow students to ease into thinking analytically about sound. It would be a good idea to coordinate this unit with any instruction of microphone and recording techniques, to provide a theoretical basis for decisions one might make while recording sound with microphones. The unit is split into four parts.

Signals

In a subunit on **Signals**, the students learn about prototypical signals such as *periodic signals*, *sinusoidal signals*, *noisy signals*, and even self-similar or *fractal signals*. I find it helpful to have students see how the visual characteristics of a digital signal (e.g. its amplitude envelope characteristics, periodicity, noisiness, specific waveforms, etc.) correlate or not with its audible sound. At the very least this allows students to become more comfortable thinking of sound as a signal. At this point there need not be any physical explanation of the signals, as they will be spending significant time on these matters very soon.

Amplitude, Loudness, and Bit Depth

The content of a subunit on **Amplitude, Loudness, and Bit Depth** begins with a detailed exploration of how sound travels through air, and how we measure it physically. Acoustic concepts such as *amplitude*, *intensity*, and *decibel* are fully deduced from simplified Newtonian principles, including *force*, *work*, *energy*, *power*, *pressure*, *impedance*, and the like. Care must be taken to fully explain concepts which are usually taken as methodically "given" in the DSP literature with proper mathematical and physical motivation. For instance, when discussing *root-mean-square amplitude* (or RMS amplitude),

the instructor should avoid casting it as a simple convenience that allows us to make positive numbers of negative ones. Instead, he or she should explain how it is derived from the statistical measure *standard deviation*, contrast it with the *average absolute deviation*, and explain that RMS measurement is more in keeping with how amplitude relates to *acoustic power*.

The physical concepts of the subunit orbit a psychoacoustic phenomenon we call *loudness*. The auditory pathways which encode loudness are studied and discussed, as well as some psychoacoustic scales of loudness like *phon* and *sone*. The instructor should emphasize that amplitude and intensity are not the only correlates of pschoacoustic loudness, and should put the psychoacoustic scales of loudness in their proper context by discussing some timbre-based correlations with perceived loudness. This should further emphasize that psychoacoustics involves a much more complex set of measurements and considerations than acoustics and DSP, and should allow students to view *using* these concepts in composition as just as much an "artistic" endeavor as "scientific."

Finally, the fundamental DSP concept *bit depth* is shown to be the most pertinent digital concept associated with amplitude. A full discussion about how one converts live sound into a digital signal, the amplitude properties of digital signals, and the fallout from decisions made about the bit depth of the signal affects the fidelity of the information in the signal. The prevailing concepts include *quantization noise*, *signal-to-noise ratio*, and *dithering*.

Frequency, Pitch, and Sample Rate

In the third subunit, which is about **Frequency, Pitch, and Sample Rate** we attack the physical and digital correlates of the psychoacoustic phenomenon *pitch* in a similar way to the previous subunit. Acoustic concepts like *periodicity*, *positive and negative frequency*, and *phase* are fully explored. One very important point to make is that a sinusoid of any phase may be synthesized from a sine signal and a cosine signal of the same frequency but different amplitudes. After this a discussion of auditory pathways involved with encoding pitch perception ensues, with a brief treatment of tuning and temperament. The instructor should note that like loudness, *pitch perception* involves timbre, since sinusoids of many frequencies can collapse into one perceived timbre with one perceived pitch.

Then, as with bit depth and amplitude, the fundamental DSP concept *sample rate* is shown to be very important to frequency, and other important concepts like *bandwidth*, and *aliasing* are introduced. There is one rather consequential aspect of digital signals that may become frustrating for students as they work with them; it is the fact that a visual representation of a digital signal will not hold any information about a specific

sample rate or any specific frequencies in *Hertz*. For example a sine wave at 480Hz at a sample rate of 48,000Hz will *look* exactly the same as a 960Hz sine wave at a 96,000Hz sample rate, or indeed a 1Hz signal at a sample rate of 100Hz. Therefore in many cases it will make sense to remain relatively agnostic about specific frequencies and sample rates, and instead employ *normalized frequency* – frequency in relation to the sample rate. Instead of Hertz (that is, cycles per second), for digital signals frequency one may use *cycles per sample*. For technical reasons beyond the scope of this paper, cycles-per-sample is in fact eschewed in favor of *radians per sample*, designated ω, with values from $-\pi$ to π.[39]

Multirate Techniques

In the fourth and final subunit, which is on **Multirate Techniques**, the students learn that since bit depth and sample rate both confer *information* to the signal, they can be interchanged using special techniques. This investigation is performed rather naïvely (that is, informally); in a future unit on *Interpolation* these ideas are further developed (see section 8.6, page 32).

8.2 Unit 2: Envelopes

This is a relatively short unit, and probably occurs best when the course also covers *additive synthesis*. The main theme of the *envelope* unit is that nearly any conceivable parameter of sound can vary over time. An instructor may appeal to classical *Gestalt psychology* to show how individual sounds either attain a greater sense of independence by disassociating their aural parameters using envelopes, or to make them collude into something resembling a single sound by using envelopes to give their parameters a "common fate."

Amplitude envelopes have a profound effect upon spectrum and timbre,[40] especially in granular synthesis techniques. This is a good point to introduce granular synthesis if it has not already been introduced in the course, and to help the students catalogue some effects of different kinds of grain envelopes. It is also a good idea for the students to begin thinking about envelopes that are a function of a domain other than time. For instance a common *lowpass* signal can be thought of as a sound with a "spectral" envelope – amplitude decreases over frequency rather than over time.

[39] $\omega = \pi$ radians per sample corresponds to the Nyquist frequency.
[40] See Unit 4, page 26

8.3 Unit 3: Delay Lines

Delay Lines are prerequisite knowledge for *convolution*, and are the building blocks of digital filters.[41] They are usually fairly simple to understand, but students often come into class with two prominent misconceptions about them. The first, probably due to their ubiquitous use in popular music for echo effects, is the notion that delay lines must also employ feedback. The second is much more subtle, but just as easy to address simply by discussing the design of delay line implementations; it is the idea that a delay is somehow implemented in such a way that an incoming sample is given a "time tag" which schedules it to occur at a later time – that delay is something one *does* to a sample to "throw it into the future." This is erroneous because a standard implementation of delay "looks into the past" to grab samples that already occurred. The former could be called *write-based* delay, while the latter, traditional version could be called *read-based* delay.

This distinction/misconception comes to the fore with *variable delays*; as its name implies, the delay time of a variable delay line is not fixed but variable. They can be used to simulate moving sounds by introducing doppler shift. This is one of the best ways to "prove" that a delay is read-based: for a read-based system that is looking at past samples, if one envelopes the amount of delay to get a doppler shift, the doppler shift should occur immediately upon applying the envelope. If it is a write-based system, the doppler shift should occur just a bit after applying the envelope, because the first sample affected by the envelope would not be scheduled until a bit later. One last subject to introduce here is *causality*; if a system only allows delay lines to read into the past, they are said to be "causal." This is the case for real-time environments. However, if the processing is not occurring in real time, there is no reason a "delay" line could not read samples from a "future" event; such a system is said to be "acausal" – acausal systems can be useful for some kinds of filtering.

8.4 Unit 4: Fourier Transforms and Convolution

Unit 4 is by far the most difficult, and its topics need to be treated with utmost care. *Fourier transforms* and *convolution* form the theoretic backbone of DSP. These topics must be covered together because each depends on the other – it is very difficult to find a place to begin a discussion. To make matters worse, the mathematics involves vector analysis in the complex plane, which is usually far beyond the math a typical graduate composer will have studied recently. In general I approve of the approach in Smith, 1997, which is to "translate" the math into problems which do not require

[41] See Unit 4, page 26, and Unit 5, page 31

imaginary numbers to solve, and then to introduce how imaginary numbers apply in subsequent chapters. In the process they lose some of their symmetry and the math becomes dirtier, but this is a small price to pay for the associated pedagogical relief – this is one place where a less-than-full explanation is probably warranted. Nonetheless, students should get used to pairing *cosine* with *real*, and *sine* with *imaginary* early on.

At the beginning of the discussion it will be important to review some of the topics from Unit 1, especially the definitions involving *sinusoids* and *periodicity*, and distinctions between *continuous* and *discrete* signals. One very important point of review should be synthesis of sinusoids with arbitrary phase by combining sine and cosine signals of the same frequency but different amplitude. As with the first unit, it makes sense to break this one into four subunits, the first and third of which are primarily about Fourier transforms, and the second of which introduces convolution, and fourth of which is about dynamic spectra. Also, because initially the consequences of the topics will be more conceptual and more "visible" than "audible," it makes sense to employ a graphical program like Pd to show what is occurring in the process.

The Frequency Domain and the Discrete Fourier Transform

The first purpose of the first subunit is to introduce the **frequency domain**. At this point in the course students should be familiar with additive synthesis methods involving sinusoids, so I believe the most intuitive way to introduce the frequency domain is to begin with *Fourier synthesis*. The instructor can show that many differently shaped periodic signals can be made by adding harmonically related digital *sine* and *cosine* signals, slowly introducing cataloging of frequencies in tables. Afterward we can make the claim that the catalog of frequencies is itself a signal, but instead of a function of time it is a function of frequency – we can now distinguish between *time domain* and *frequency domain*. This is not a trivial claim – in order for it to be a signal everything we need to know about the sound must be encoded in it. A particularly good experiment is to try to build a pulse train of a given frequency in the time domain using cosine signals, and then to see what kind of frequency-domain signal results (it will be another pulse train, but in the frequency domain) – each can be considered a discrete signal.

Once we have established the existence of the frequency domain, we can begin to derive **Fourier transforms**. First, we may now posit that signals in the frequency domain may be either continuous or discrete. This will not be an easy point to grasp, and it might be one of the few things that a student will need to take on faith: it is possible to think of sound as the combination of an infinite number of infinitely long sinusoids. We know from Unit 1 that discrete signals can alias; we can define this

phenomenon as a periodicity in the frequency domain – frequency domain signals are periodic. Putting all of what we know together so far, we will find that a discrete signal in the frequency domain implies a periodic time domain (that is, individual sine waves related harmonically will always produce a periodic time-domain signal), and that a discrete time domain implies a periodic frequency domain. Meanwhile a continuous time- or frequency-domain signal implies an aperiodic signal in the other domain. This implies only four different possible pairings of different types of time and frequency domains, and four Fourier transforms to move between them. Because we are using digital equipment, we will be interested in the one which is discrete (and thus periodic) in both domains – the *discrete Fourier transform*, or **DFT**.

In order to begin discussion of the DFT, the instructor must remind the students of the definition of a *transform* – naïvely it may be defined as an operation which does not lose information.[42] This makes sense because we are trying to convert between two signals each of which holds just as much information as the other. In order to teach how the DFT works, we must come to terms with one particular assumption: because it will output a discrete frequency-domain signal, the DFT assumes that whatever it is analyzing at the time is one period of a periodic signal.

In order to make this point persuasively, I have students imagine that a digital recording of Wagner's *Parsifal*, if somehow repeated exactly from the beginning of time to the end of time, would be a periodic signal (with an impossibly "low" pitch) that could be decomposed into *individual* sine and cosine components arranged harmonically: the sounds of the entire piece, from beginning to end, would "merely" be the result of phase interaction among the components. In this case the important point is that all of the relevant information for a periodic signal may be found in just one period. This should also stress that the DFT alone gives no way to represent *dynamic spectra*, that is, spectra that change over time – a frequency-domain signal on its own has no time information, just as a time-domain signal has no frequency information. It should also emphasize that this goes counter to our intuitions about music, as well as the content of the unit on envelopes – it should be apparent that analyzing signals in terms of perception or technique of synthesis is difficult to do formally.

When the underlying conceptual symmetries of Fourier analysis are mastered, the discussion may move to the steps required to perform a DFT. The major concept at play are *heterodyne filters* which are used to *correlate* a given signal with several predefined

[42]Note that not all "information" is relevant to ordinary perception. After all, white noise, which has a rather consistent perceptual phenomonology from an information standpoint is all news, no redundancy. Meanwhile, such "lossless" transforms as the DFT may be contrasted with "lossy" compression algorithms which purport to discard only that information which is perceptually irrelevant, for instance mp3 compression.

sinusoidal basis functions. If the students are mathematically inclined, the motivation for the heterodyning/correlation may be made clear by performing 4- and 8-point DFTs by solving a system of linear equations using the *Gaussian method*. Students learn how to quantify *phase* and *magnitude* from the real (cosine) and imaginary (sine) components, and learn to distinguish between "amplitude" and "magnitude." It should be useful to investigate how different signals in one domain look in the other; for instance a sinusoid in the time domain shows up as an *impulse* in the frequency domain, while an impulse in the time domain shows up as a sinusoid in the frequency domain.

To end the unit, students learn about the *fast Fourier transform* (or **FFT**). The instructor may either simply allow students to know that it exists – "it is an algorithm that allows computers to do the DFT much more quickly, and which some smart people named Cooley and Tukey made available in the 1960s"[43] – or if time allows and the students are ambitious, some of the details may be covered.

Introduction to Convolution

Convolution can be very tricky to teach, but mostly because the chapters devoted to convolution in some DSP literature begin with a rather opaque and intimidating equation rather than deriving the equation using intuitive examples.[44] Smith, 1997, Roads, 2001, and Loy, 2007 all do a very good job of mitigating this circumstance. I find the best way into convolution is to begin with an intuitive discussion of reverberation. Then, convolution may be discussed mathematically as conceptually related to the *distributive law* in algebra. The *convolution theorem*, which states that "convolution in the time domain equals signal multiplication in the frequency domain, and the converse" is introduced. It is appropriate to address the convolution theorem by using it to put *ring modulation* and *amplitude modulation* in their proper context as a multiplication in the time domain and a convolution in the frequency domain. After they can handle this material students learn that all time-domain processes, such as amplitude envelopes have spectral consequences because of the convolution theorem, and that concepts like frequency-domain amplitude envelopes (what we usually call "digital filters") also have significant consequences in the time domain.

For ambitious students, the instructor may show the students how to perform FFT-based convolution, discuss why it is more efficient than direct convolution, reveal and

[43] In fact, the algorithm was likely invented by Gauss in the early 19th century, but this fact was not known until after the Cooley-Tukey algorithm was published and named.

[44] The equation in question is:

$$y[n] = x[n] * h[n] \quad \equiv \quad y[i] = \sum_{j=0}^{M-1} h[j] \cdot x[i-j]$$

emphasize the distinction between the normal form of convolution and *circular convolution*, and if there is time, provide a model for real-time FFT-based *partitioned convolution*.

DFT: Leakage and Windowing

Once *convolution* is understood, a discussion of **DFT leakage** ensues. The best way to do this is to examine what happens when a DFT tries to analyze something that looks like one and a half periods of a sine or cosine wave. Here is where the periodic assumption of the DFT takes its greatest consequence, and the instructor should emphasize this by synthesizing the 1.5-period sinusoid as a periodic signal, examining the waveform in a sound-file editor, playing it back for the class – they will clearly hear more than one sinusoidal component in the signal. This "extra information" can then be located in the DFT of the signal, and is designated *leakage*.

After this, **windowing** is described. First, students should be aware that the *discontinuity* in the previous example was responsible for much of the "noise" in the signal. If a "fade in – fade out" amplitude envelope is applied to the samples in the period (e.g. a triangular window), the students will see and hear how the discontinuities lead to what seems to be a more accurate DFT, and what sounds like a less noisy signal. At this point convolution can be revealed as the mathematical underpinning of leakage: the first signal was noisy because of its implied signal multiplication with a *rectangular window*. Taking the DFT of the rectangular window alone, and then convolving it with the hypothetical frequency-domain representation in the frequency-domain on the blackboard will show students how the "extra information" came about. The discussion can end with some properties of other important window functions, in particular the ones they are likely to use in FFT-based applications, e.g. the *Hann*, *Hamming*, *Blackman*, *Gaussian*, and *Kaiser-Bessel* windows.

Dynamic Spectra

Dynamic Spectra will be the culmination of this unit, and the one with the most practical fallout. This subunit should give students the theoretical tools to be able to think about dynamic (time-varying) spectra, which was not possible with just the DFT. The students first learn about the *short time Fourier transform* (the **STFT**), which uses the DFT to analyze successive overlapping chunks of a time-domain signal in order to "track" changes in spectrum. The students should begin with an STFT called the *overlap-add method*, which amounts to a form of synchronous granular synthesis with an intervening DFTand inverse DFT. Provided neither the time-domain nor the frequency-domain

signals are processed (for instance by using a window), with a good STFT algorithm the resulting signal should be identical to the original. The output of the magnitude spectrum of an STFT can be plotted on a *spectrogram*.

After this the instructor should discuss the *Heisenberg Uncertainty* inherent in the DFT/FFT (and thus in the STFT): if one is employing a very short window size for the FFT, he or she will be forced to overlap the windows much more frequently in the STFT, and thus we will know exactly *when* certain events occurred – we will have terrific *time resolution*. But with a small window, there is a tradeoff: our spectral information will be minimal, so we will have a very poor description of *what* the event was when it occurred – we will have poor *frequency resolution*. The converse holds as well – with a larger window we will have better frequency resolution at the cost of time resolution. The phenomenon is made especially plain using a spectrogram. The practical fallout of this is profound, because it will force different sizes of FFT to be used for different sound sources; the STFT– like much of the practical fallout of DFT – is as much art as it is science. After this, some steps to mitigate the problem can be shown. For instance *zero-padding* the data before analysis can help not by increasing the frequency resolution of each DFT, but by interpolating smoothly between frequency-domain samples. This can be useful as a visualization tool for frequency-domain signals with poor resoultion, since it shows the contour of signals much more clearly.

In the next part of the lesson, students learn how to apply phase and magnitude information from successive DFT windows; this is the basis for *phase vocoders* and *oscillator-bank resynthesis*. At this point the students will have the tools they need to understand the effects of directly processing spectra. At the end of the unit, if the students are inclined, the instructor may introduce *wavelet* analysis and synthesis, which is a kind of dynamic-spectrum system based on the seminal granular work of Dennis Gabor and also Iannis Xenakis.

8.5 Unit 5: Digital Filters

Digital filters are another very difficult subject in DSP; entire books and courses are devoted to filter design. There are actually several reasons not to delve too deeply into this topic in a course for graduate composers. First, nearly every computer-audio environment has a wide assortment of built-in filters, and because they tend to behave very similarly with few overarching differences, they may be considered relatively low-level UGs. The basics of the familiar filter types – lowpass, highpass, bandpass, notch, comb, etc. are easy enough to learn. Second, in order to really tackle the theory behind digital filtering, one must learn about both the *Laplace transform* and the *Z-transform*,

which are like generalized Fourier transforms. This set of mathematics is probably beyond the scope of a graduate composer course. Nevertheless, there are important filtering topics which are near the surface and worthwhile to discuss in class.

All filters are specialized chains of delay lines, some with feedback and some not. The spectral characteristics of these delay-line chains can be explained as the product of phase cancellation resulting from time-domain convolution. Different classes of impulse responses may be associated with different kinds of filters. Probably the most interesting and the most informative to any non-formal discussion is the class of filters whose impulse response looks like an exponentially decaying sinusoid with a given frequency between 0Hz (i.e. DC-offset) and the Nyquist.[45] If we take into account the periodic nature of the frequency domain, when we take the DFT of the impulse response in question, we will find what looks like a band-pass filter response centered on the frequency of the decaying sinusoid, with the bandwidth depending on the length of the decay – a longer decay will have a narrower bandwidth and the converse.

This can be attributed to the fact that an exponentially decaying envelope can be shown to have a low-pass frequency response, which looks like a band-pass response centered on 0Hz, half of the bandwidth extending into the positive frequencies and half into the negative frequencies. Multiplying the impulse response by a sinusoid ring-modulates it, splitting the former frequency response into two, each centered on a positive frequency and the corresponding negative frequency. Multiplying it by the Nyquist frequency will center it on the Nyquist, making it a high-pass filter. This shows that in some cases it is simpler to think of all filters of this kind as band-pass filters which can turn into low- and high-pass filters depending upon the frequency of the decaying sinusoid.

8.6 Unit 6: Interpolation

Interpolation, which forms the basis of *sample rate conversion* (and many other common DSP operations) is also a very difficult subject to treat with considerable depth. This is because it relies on numerical analysis to approximate derivatives, and some elementary linear algebra to solve the systems of interpolation equations. Linear interpolation is trivial to solve, but there are six possible four-point polynomial interpolations which are more difficult to derive, but also more interesting. Two of them in particular, both of

[45] Of course we know that technically a sinusoid extends to infinity in both directions. The purpose of the Laplace and Z-transforms is to provide a way of considering the exponentially decaying and exploding variety as also primary waveforms. In this case, infinitely periodic sinusoids are shown to be a special case of a sinusoid with an exponential envelope – they form the boundary exactly between "decaying" and "exploding." Also, the continuous Fourier transform may be shown to be a subset of the Laplace transform, and the DFT may be shown to be a subset of the Z-transform.

them cubic, are used in computer-audio applications. The first, a Lagrange interpolator, finds a cubic curve to pass through all four points, and uses that curve to interpolate between the inner two points; the Lagrange interpolator is used in both Pd and Csound for their cubic-interpolating table-reading UGs. The second, an Hermite interpolator, finds a cubic curve which passes through only the two inner points, and uses the two outer points in combination with the two inner points to approximate the first derivative. An Hermite interpolator is "smoother" because its first derivative is continuous – there are no sharp edges. This kind of interpolator is used in SuperCollider for any UG which requires cubic interpolation.

All interpolators have an associated impulse response – imagine an interpolator moving through a signal that was entirely silent except for a single impulse somewhere in the middle of the signal – so they all may be considered a form of filtering by convolution. If the students have particular fortitude, they may be introduced to sinc interpolation, which is a theoretical form of interpolation that underlies the validity of analog-to-digital and digital-to-analog conversion. This kind of interpolation is very difficult to come to terms with; since the interpolator's impulse response is infinitely long, this means that the value of the very last sample in the signal has some effect on the way a sinc interpolator interpolates from the first to the second sample. It is a nice, if simplistic, example of quantum entanglement – a "spooky action at a distance."

9 Final Thoughts

I OPENED THIS THESIS with a claim that computer music is no longer a new field. During its infancy composers who would pursue computer music had to rely on the computer technicians who had the knowledge and ambition to stake out the new territory. Meanwhile only a very few computers and digital-to-analog converters were available anywhere in the world, and programming methods and languages were also just being developed – FORTRAN, one of the first "high-level" programming languages appeared in the same year (1957) as Max Mathews's *Music I*. Unless the composer was him- or herself a technician with access to computer-music resources, that composer was in stark compositional dependence.

Obviously the situation has improved. Where collaborative relationships between composers and technicians used to be normal, I have noticed in the generation of composers to which I belong a stigma attached to such dependence. This is only the case because the computer hardware is general enough, and the computer-music environments high-level enough, for the majority of decision to be delivered to the hands of the composer; the contemporary computer-music composer is self-reliant. I would argue

that computer-music has thoroughly ceased to be a new field not because its theory is so well developed and documented, nor because of its rich history, but because composers have the ability to be autonomous in the field. Providing composers with appropriate tools and the knowledge underpinning them is, I believe, the best way to foster an educationally, and thus compositionally flourishing environment.

In discussing the programme contained in these pages with my peers, the patterns of criticism coalesced around two related ideas: 1) the curriculum is probably too ambitious, might inspire frustration and discontent rather than excitement, and may interfere with rather than benefit or complement a student's compositional activity; and 2) there is little reason to be skeptical of so-called "black-box" software if it allows students to be compositionally creative and productive, or prepares them to enter a job market which seems to teem with this kind of software; and isn't all software a black-box on some level? To address the first problem, I believe it is unconscionable for an instructor to assume that students will not be able to handle certain material, and irresponsible to assume what students' interests may be. Where I am a little less sure of my plan is in the ethics of requirement from the standpoint of a general educator – to what extent is it appropriate to require a student to learn something he or she is not interested in? Educational philosopher Harry Brighouse opines:

> . . . we should not, as a general matter, presume that we know better than other people how they should best lead their lives. But becoming a teacher, a school administrator, or a parent, is adopting a role in which you have power over a child's life, and you know that the child is highly imperfectly informed about what will make for a flourishing life, and spectacularly ill-equipped to pursue one. If one is uncomfortable with the role, one should either avoid it, or carry it out despite one's discomfort.
>
> Being more knowledgeable than, and having legitimate power over, a child, does not, however, give us a right to impose our particular view of how they will flourish on them. The paternalistic role is very complicated. We should not be guided by our own pre-existing views; rather, our views should be guided by our best judgements about the child and her interests – the kinds of things that will tend to her long-term flourishing.[46]

In this passage one may substitute "graduate student" for "child" more or less successfully, but in the matter at hand the dilemma is even more concentrated, because the average graduate student will usually know a great deal about his or her interests, and will have specific goals as well as expectations of the often expensive education he or she is seeking. In ideal circumstances a computer-music course would be implemented primarily as a set of individual lessons, much like composition lessons, in which the student and instructor meet each other half-way. Where a course must be taught as a

[46] Brighouse, 2006, p. 43

lecture, instructors should of course allow some flexibility in their syllabi, and should aim to teach to and complement students' common interests. As I argue in the sequel, I believe students should expect instructors to challenge their intellectual boundaries; the problem remains, but students who expect to journey through their higher education without having to learn difficult or potentially uninteresting material should not be seeking such education.

To address the other criticism, that "black-box" software is not the bogeyman I made it out to be in section 7, I would make the following four arguments. First, to the extent that we regard autonomy as an objective good and a primary aim of education, students may need to have their current interests challenged to discover others. Autonomy does not imply the absence of constraint, but it does require awareness of constraint: what it is, and where it is located. A common feature of black-box environments is their propensity toward hiding the constraints of the system from the user. While I have no specific qualms with a student using this kind of software, I believe it makes a poor tool for education. To put it in starker terms, for students to be intentionally encumbered with an environment that imposes too many extra unnecessary (and equally unprofessed) constraints on their freedom of thought about computer music does not seem educationally responsible.

Second, the stigma attached to dependence on a technician I alluded to earlier strangely extends less to dependence on black-box software. I say "strangely" because to me the situations are quite similar. In the former case, however, the composers' dependence is a two-way relationship – the composer and technician are collaborators – whereas in the latter case the composer is depending on a technician (the programmer) just as much to handle the difficult things ahead of time, except now the relationship has fewer dimensions. An application which hides crucial information from the user but manages to maintain a high degree of flexibility is closer to the two-way collaboration, but all too often decisions which have great effect on composition are made not by the composer but by the distant programmer, hard-coded into the design of the program.

Third, I am acutely aware that this is all a matter of degree – one person's unconfined sand box is another's stuffy black box. But simply noting this does not preclude an instructor from finding an appropriate degree at which to draw the line. In my view the best tool computer-music instructors have to expand the compositional intuition of their students is by fervently demystifying the theory underlying the involved techniques of the field. Therefore I believe that any language or environment which includes an axiomatic set of DSP tools,[47] like the three I reviewed in the body of this thesis, are the most appropriate. I do not, of course, believe that music composed in one of these environments is necessarily superior to one created in, say, Pro Tools, nor do I believe

[47] One might instead say, a historically agreed-upon minimal set of DSP tools.

that a composer will meet exposure to theory with automatic or measurable compositional "improvement." My point is about the responsibilities of an instructor to promote individual students' autonomy, not just creativity and productivity.

Finally, I am sensitive to the fact that a student in my program might not learn the commercial software required so often from the job market in academia and elsewhere. However, I would argue that an individual who has solid grounding in one of the lower-level environments I discuss will have a much easier time learning how to use a less flexible application than someone who needs to perform the reverse transition. I have already stated my ethical views on requiring competence on commercial software in the computer-music studio from students, but I am also aware of the instructor's role in adequately preparing students for job placement. This is a dilemma for which I have no consistent solution, but the most likely compromise is to augment students' choice by having state-of-the-art commercial software available to them in the studio, and perhaps some tutorial time in the classroom. Even with this compromise, if a balance must be struck between requiring students to learn universally applicable conceptual knowledge and requiring them to become proficient in specific expensive proprietary or black-box environments, the scale should tip in favor of knowledge.

Extremely important note: This thesis was written entirely using open-source software. The project was typeset using LaTeX. Most graphical figures were designed and rendered in Inkscape; others were created using the PostScript export feature of Pd. Musical notation was engraved with LilyPond. Some features of the paper are meant to be enjoyed by reading the PDF on a computer screen; please save paper by not printing it unless it is necessary.

My thanks to Allan Schindler for his years of devotion to teaching computer music and composition, to Kevin Ernste for spurring my interests in open-source software and clear pedagogy, to Robert Morris for helping me think clearly and honestly about composition, and to my following dear friends: to Scott Petersen and Scott Worthington for their helpful tutelage in SuperCollider, to Baljinder Sekhon, Paul Coleman, and Robert Pierzak for their help with editing prior drafts of this thesis, and to Christina Crispin for everything.

Works Cited

Julian Anderson. A provisional history of spectral music. *Contemporary Music Review*, 19, Issue 2 — *Spectral Music: History and Techniques*: 7–22, 2000.

Harry Brighouse. *On Education*. Routledge, London and New York, 2006.

Joel Chadabe. *Electric Sound: The Past and Promise of Electronic Music*. Prentice–Hall, Inc., Upper Saddle River, NJ, 1997.

Thomas Christensen. Introduction. In Thomas Christensen, editor, *The Cambridge History of Western Music Theory*, pages 1–23. Cambridge University Press, Cambridge, UK, 2002.

John ffitch. What happens when you run csound. In Boulanger, Richard, editor, *The Csound Book: perspectives in software synthesis, sound design, signal processing, and programming*, pages 99–121. The MIT Press, Cambridge, MA, 2000.

Joshua Fineberg. Guide to the basic concepts and techniques of spectral music. *Contemporary Music Review*, 19, Issue 2 — *Spectral Music: History and Techniques*: 81–113, 2000.

Gareth Loy. *Musimathics: The Mathematical Foundations of Music, Volume 2*. The MIT Press, Cambridge, MA, 2007.

Miller Puckette. Preface: Computing while composing. In Carlos Agon, Gérard Assayag, and Jean Bressard, editors, *The OM Composer's Book, Volume 1*. Editions Delatour France / IRCAM, 2006a.
http://crca.ucsd.edu/~msp/Publications/om-reprint.pdf.

Miller Puckette. Phase-bashed packet synthesis: a musical test. In *Proceedings of the ICMC*, pages 507–510, 2006b.
http://crca.ucsd.edu/~msp/Publications/icmc06-reprint.pdf.

Curtis Roads. *The Computer Music Tutorial*. The MIT Press, Cambridge, MA, 1996.

Curtis Roads. *Microsound*. The MIT Press, Cambridge, MA, 2001.

Gilbert Ryle. *The Concept of Mind*. University of Chicago Press, Chicago, IL, 1949.

Gilbert Ryle. Knowing how and knowing that. *Proceedings of the Aristotelian Society*, 56, 1946.

Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, 1997.
http://www.dspguide.com/pdfbook.htm.

Barry Vercoe et al. *The Canonical Csound Reference Manual*, version 5.09.
http://www.csounds.com/manual/html/index.html.

Todd Winkler. *Composing Interactive Music: Techniques and Ideas Using Max*. The MIT Press, Cambridge, MA, 1998.