

|| Computer Music Composition with RTcmix ||

Jerod Sommerfeldt

|| copyright - Jerod Sommerfeldt, 2015 ||

This book is dedicated to all past, current, and especially future users of RTcmix.

## || Table of Contents ||

Foreword.....	5
Prelude: A brief history of RTcmix.....	9
Sonata I: Downloading and installing RTcmix.....	12
Interlude I: Basic UNIX for the RTcmixer.....	28
Sonata II: The anatomy of an RTcmix score file.....	37
Interlude II: Exploring maketable( ).....	48
Sonata III: Elements of C, with arrays and conditionals.....	56
Interlude III: RTcmix and Python.....	71
Sonata IV: Adding various effects to your scripts.....	82
Interlude IV: A detailed look at bus_config( ) and connections.....	89
Sonata V: Using RTcmix the “wrong” way.....	100
Interlude V: More synthesis and modulation.....	116
Sonata VI: Randomness and algorithms.....	132
Interlude VI: Pitch transformations.....	143
Sonata VII: RTcmix in Pure Data and Max/MSP.....	158
Interlude VII: Interactivity.....	174
Postlude: Customization.....	189
Appendix: 100 Progressive Score Files.....	199
Afterword.....	256
Useful links in the world of RTcmix.....	261
Index of score file commands and instruments.....	264

## || Foreword ||

In today's fast-paced world of digital technology, composers interested in creating digital music finds themselves very far from a shortage of tools to assist them in the realization of their art. As a student, I recall very clearly during a lesson with Mara Helmuth, my teacher at CCM, that I was fast becoming frustrated with the sheer volume of software suites — both free and proprietary — at my disposal. Prior to my arrival in Cincinnati for my DMA, I had already been introduced to electroacoustic and computer music while privately studying composition with Christopher Burns at the University of Wisconsin-Milwaukee: I had taken a number of classes using the graphical programming language Pure Data, a seminar on algorithmic composition using Scheme and Common Music LISP, and spent two years making music on my laptop with the Milwaukee Laptop Orchestra. With regard to digital audio workstations, I'd taken classes on Digital Performer, owned my own copy of Pro Tools, was working in a studio that used Logic, and was at the time enrolled in a class that used Audacity! With so many options, I was becoming confused about how I could better streamline my work, stop worrying about the idiosyncrasies of each individual tool, and begin focusing solely on the music at hand.

Eventually, I found that I wanted to write music that would allow me to express myself in a sound world limited only by my imagination, would afford me the chance to explore and study the basic tenets of digital signal processing, and would provide me the opportunity to feel as if I was working with my computer at a deeper level than big-box, proprietary software — with its graphical user interfaces and point-and-click methodology — would allow.

Enter RTcmix.

Here was a program — well, more of a “language” (but more on that later) — that took the text I typed into my computer and processed the sounds by way of a bash shell! (More on that later, too.) All of a sudden, I was using my Mac OS X Terminal,

learning some cool UNIX commands, constructing for( ) loops, setting up arrays...all of the things that I yearned for as a budding computer musician who admittedly has zero computer science background. I was finally “under the hood”, tinkering with the engine, creating exciting sounds and understanding not only how I made them, but what I could do with them.

Moreover, I was using a language that shares a lineage to the very genesis of computer music and has been utilized by some of the pioneers in the field. RTcmix’s authors continue to maintain, update, and expand the program so that it can now be integrated into iPhone and Android applications or Pure Data and Max/MSP. With a supportive community of like-minded musicians and developers, questions about using RTcmix are never more than an email away. In short, RTcmix is a dynamic, exciting tool for anyone interested in creating digital music, be it interactive, electroacoustic, installation work, fixed media, or music that is playable on mobile platforms.

So, why this book?

Outside of the terrific tutorials and documentation on its website — as well as course notes from its authors — RTcmix lacks a concise, all-in-one introductory text that is especially useful for those being introduced to the program for the first time. The pages that follow fill these needs, serving as a comprehensive introduction to the world of RTcmix. The language is approached methodologically and takes the reader through the basics of installation, explains some very useful commands in UNIX, demonstrates how to write score files and play them back, carefully explains various RTcmix instruments and commands, and finally delves into more advanced topics, such as running RTcmix in a variety of platforms, in conjunction with the Python programming language, and designing your own RTcmix instruments.

The mantra of this book is musical creation and is divided into a series of sonatas and interludes. Following a brief prelude that surveys the history of RTcmix, each sonata focuses on a specific task in RTcmix that culminates in a set of suggested directions for composing the reader’s own, brief work. Interludes will cover topics introduced in each

sonata and seek to explain those concepts in greater detail.<sup>1</sup>

Nothing is assumed w/r/t prior experience in the world of computer science or computer music. For most of my teaching life, I've had the absolute pleasure of working with beginners who find great joy in learning about computer music using RTcmix (and Pure Data) from the ground up. While topics such as synthesis, MIDI, visual programming, and envelopes are discussed, those discussions are not at all comprehensive and may hope is that with the wealth of access to information in our digital world, you'll find it rewarding to pause and further research any topic in this book on your own. I can say without hesitation that that has been my biggest asset in learning more about topics at hand in my own work and composition.

This book and its accompanying score files were all written using an Apple MacBook Pro running the Macintosh Operating System 10.10 Yosemite. Thus, I will refer the user to the Terminal utility and some useful, ancillary downloads and applications that have been tested and used extensively on the Mac platform. Moreover, I prefer using the free text editor TextWrangler for writing score files, although it is fun to write in the Terminal using nano or pico.<sup>2</sup>

And now, an apologia. I am not a computer programmer. While I enjoy writing code and am still endlessly fascinated by the sound world I can achieve using lines of text, I know very little about the world of computer science outside of the tiny space I've cultivated for myself. I do know that I am first and foremost and will always be a composer. I just happen to be a composer whose sonic palette and compositional world is enriched by MINC, C, Python, RTcmix, Pure Data, Max, Processing, and others.

---

<sup>1</sup> I'll also add that this book was used in my course "Computer Music Programming" at the Crane School of Music and can be divided into the requisite 15-week semester, with the tasks at the end of each sonata and interlude functioning as assignments. Meeting three times per week, we start with the Prelude and Sonata I in the first week and finish up with the Postlude, supplementing each class with relevant listening examples, for which the RTcmix Soundcloud group is an invaluable resource.

<sup>2</sup> Quick note about syntax. General writing is done in this hip serif font, but score file examples and UNIX commands will always be in **this font**. Like, **grep** for example. Or **sudo**. Or **WAVETABLE(0, 10, 20000, 440, 0.5)**.

Furthermore, this book is not intended to be used as a complete manual for the vast world of digital synthesis and computer music in general. My hope is that the words that follow are written in an inviting, welcoming way for everyone, regardless of prior knowledge or background. Where details in signal processing procedures are glossed over, my hope is that this text — focusing solely on just a few of the many ways to compositionally approach RTcmix — instills a reliance on self-research for the reader. I often refer to RTcmix’s online documentation for further reading and often find it most useful to work with that documentation in an open browser window for quick reference. Moreover, there are far too many really, really great books written by far more capable minds than mine to enlighten and enrich your experience with computer music: Miller Puckette, Curtis Roads, and Charles Dodge come immediately to mind, and a listing of their books and those by others are included at the end of this text.

If you’ve read this far, you probably know that some thank you’s are coming and until I actually took the time to sit down and write a book, I never realized just how important those thank you’s are to include. This book would not exist were it not for the invaluable work of two of my teachers, who I am proud to call mentors and friends: Mara Helmuth and Christopher Burns. Brad Garton and John Gibson are two of the authors of RTcmix and their documentation, music, correspondence, and support and encouragement are central to not only me, but to the RTcmix community at-large. Joel Matthys and David McDonnell are two dear friends who share the love I have for this program and our continued correspondence and sharing cool score files and answering each others programming questions makes me endlessly happy and completely and totally grateful. My wife Christi is amazingly supportive of the things that I do. Our two kitties Emma and Oscar would often try desperately to add in their two cents by walking over my keyboard, though they still leave the room when I start composing or making noise.



## **|| Prelude: A Brief History of RTcmix ||**

It's impossible to discuss the history of any generative music language for the computer without first introducing the work of Max Mathews. An engineer, programmer, and musician, Mathews' pioneering work in the digital synthesis of sound began at Bell Laboratories in the 1950s. Interested in the musical possibilities of emerging digital technology, he developed MUSIC I in 1957, a generative program that was able to realize sound using a triangle wave at specified durations, amplitudes, and frequencies. Further refining his series of programs (dubbed MUSIC-N), Mathews completed MUSIC V in 1968, which was an efficient, universal release that met the demands of the fast-evolving technology of computers. With each subsequent release of MUSIC-N, by either Mathews or others in the field who were adapting it for their own use, the notion of portability was being realized. While at first, synthesized sound and computer music was only capable of being executed at Bell Labs, it was soon being disseminated to Princeton, IRCAM, MIT, and Columbia (among others) and today finds itself in the homes of anyone with interest and a computer or tablet device.

Throughout the 1980s, Paul Lansky was at Princeton, working on refining a program he called MIX, which was written using Fortran. Music composed with MIX was realized using punch cards and magnetic tape, and MIX was capable of mixing together various sound files. Lansky decided to add libraries of C functions to the program and make it capable of running on UNIX machines, eventually dubbing his work CMIX. With the versatility of C, users were able to create dynamic scripts that contained loops and conditional tests to further sculpt and refine their sounds.<sup>3</sup> Moreover, CMIX contained powerful instruments written by Lansky, including linear predictive coding.

CMIX was being further developed as the decade moved along and was ported to

---

<sup>3</sup> CMIX didn't use C verbatim, and neither does RTcmix. They use a parser called MINC, which stands for "MINC is not C." This language is easy to use and understand, was written by Lars Graf, and borrows many of the strong features of C, but not all of them.

a variety of personal computers, including NeXT systems. As computers became more and more portable and powerful, it was becoming of interest to utilize CMIX and other languages in the moment. In 1995, NeXT computers failed, and Brad Garton and Dave Topper — two major contributors to the development of CMIX — began work on a version of the program for the Silicon Graphics platform and eventually Linux. Their work uncovered something quite cool: Following their successful porting to the new platforms, many CMIX instruments were capable of being processed in real-time. They were able to find a solution that directly connected the synthesis engine of CMIX to the powerful digital-to-analog converters that were intrinsic to the new machines of the time. Thus, it was possible to process live sound and manipulate data on-the-fly.

RTcmix was born.

As the 1990s and 2000s moved along to the present, many new contributors came on board for the RTcmix project. Thanks to the efforts of John Gibson and Doug Scott, instruments could be chained together, several new instruments were added to the RTcmix library, scores could be written in Perl or Python, and a body of documentation became available for everyone to peruse online. Mara Helmuth added not only powerful probability functions, but also a stochastic granular synthesis instrument. Running on the command-line, RTcmix was distributed to and maintained by a community of users, who continually added to and tweaked the program for their use. As graphical programming environments began to gain popularity among the computer music community, Luke DuBois and Brad Garton developed an [rtcmix~] object for Max/MSP and Joel Matthys wrote the object for Pure Data, further widening RTcmix's audience. In fact, RTcmix and its gorgeous array of sounds can also be used as the audio engine for both iPhone and Android apps.

With its versatility and portability, RTcmix exists as a powerful program designed for composing highly customizable computer music in any of a variety of platforms. At its heart lies the work of Paul Lansky's MIX and CMIX programs and to an extent Max Mathews' MUSIC-N languages, thus giving RTcmix a branch on the

MUSIC-N tree shared by CSound and SuperCollider.

Because of its long and rich history of flexibility, there is no reason to suspect that RTcmix will suddenly become irrelevant as the next-greatest-iDevice or platform hits the market. In fact, I'm already imagining a world where I can be eating dinner and coding RTcmix scores using my retinas and stealth mind control in my Apple Glasses or, who knows, DSP microchip implant...

## || Sonata I: Downloading and Installing RTcmix ||

An ongoing theme in this book is the notion that RTcmix exists in various guises and that this flexibility is part of its appeal. The heart of RTcmix lies in its score files, or scripts. The suite of instruments, commands, and the anatomy of the score file will remain the same whether you decide to use RTcmix on the command line, in the standalone application, with Pure Data or Max/MSP, or on your iOS or Android device. In this first sonata, we're going to explore three ways to begin using RTcmix: Within its standalone application, a so-called “front end,” and on the command line.

Not to oversimplify the process, but when it comes to purchasing a home, the buyer is more or less left with two choices: Would they rather have a “turn key” or something customizable? There are advantages to both, of course. The turn key property affords the buyer the opportunity to merely move in furniture, hang pictures, and start grilling. You've probably heard of houses with “good bones” or something that needs a little TLC<sup>4</sup>, but the general idea is that the buyer can do some custom work to help make their new purchase look and feel exactly to their tastes.

With the standalone application in RTcmix, we essentially have a “turn key” that is ready to go. It's the easiest to download and allows us to start making music right away, which is important. We're going to start by downloading the application, exploring its features, and testing out a basic score file to get some sound out of our machines.

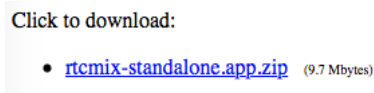
Brad Garton is one of the founders of RTcmix and is not only an author of several of the instruments that we'll be using, but also created the standalone version that we're going to explore first. Downloading the program is as simple as visiting the following URL: <http://music.columbia.edu/~brad/rtcmix-standalone>

You'll find a link to download a file called “rtcmix-standalone.app.zip” and will

---

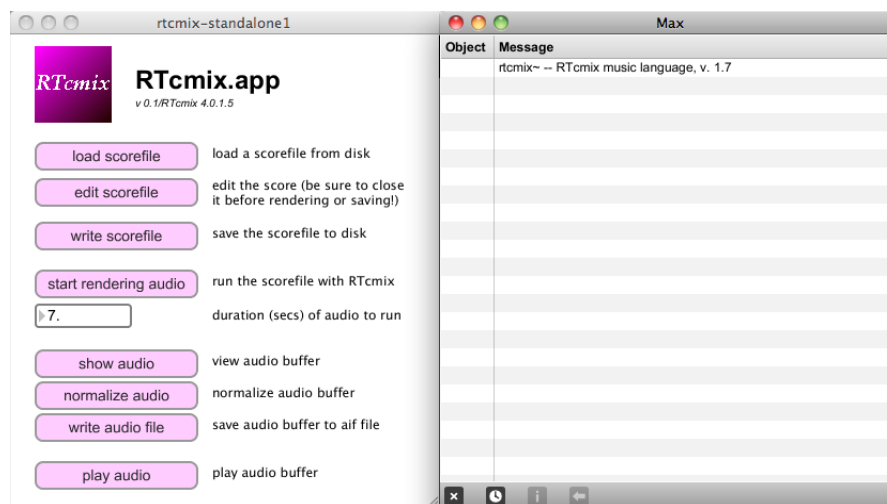
<sup>4</sup> Not that RTcmix is some antiquated program that needs fixing!

likely be saved to your /Downloads<sup>5</sup> folder.



Double-click on that .zip file and watch as your computer briefly unpacks the .zip contents into a neat and tidy, purple icon called rtcmix-standalone. If you'd like, feel free to drag that icon to both your /Applications folder and also to your Dock to create a shortcut.

Opening RTcmix in this way for the first time will create two windows, shown in Figure 1.



---

<sup>5</sup> If you're wondering about the slash in front of "Documents" above, let me quickly interject. As we begin working with RTcmix on the command line, we're going to start talking about various directories on your computer. When I save a file to my /Documents folder on my computer, it may look, for example, like I'm dragging a .doc file, or something similar, to my Documents folder icon, but in reality, I've created a new directory, or path, on my computer. If the document was called mygreatdocument.doc, then the new path will be /Users/jerod\_s/Documents/mygreatdocument.doc. Obviously, your computer isn't named jerod\_s, so it's common in texts like this to see /Users/yourcomputer/mygreatdocument.doc, where "yourcomputer" stands in as a placeholder for whatever name you've given to your machine. Moreover, when discussing long paths to files, as you'll see in many examples of the RTcmix documentation, it's common to see /path/to/your/file.sco as a generic address.

The window on the left is the graphical user interface for the program. A graphical user interface, or GUI (“gooey”), is a carefully designed canvas that allows us to quickly navigate our software intuitively. Right away, we see boxes to click, comments that tell us what to do with those boxes, and even a shiny RTcmix icon at the top. The window on the right (called “Max<sup>6</sup>”) is going to speak directly to us, letting us know if we have any errors, if our scripts were saved properly, what egregious things we’ve done to the program, etc.

Let’s start by making some sound. Go ahead and click on the purple button marked “start rendering audio” and listen to the awesome result! What you heard was an RTcmix script written by Brad Garton himself! To see this script, click on “edit scorefile.” A new window opens with a lines of text that for now might seem a little daunting. However, this is the score file, or script, that helps create the sounds we just heard. Rather than customize this script, we’re going to make our own and briefly discuss each step in creating our own score files, knowing that another sonata in this book is dedicated to a more detailed explanation of an RTcmix score file’s constituent parts.<sup>7</sup>

Clicking *command-a* will select all of the text in the window and from there, go

---

<sup>6</sup> Second interruption, but an important one. This RTcmix application was created using Max/MSP, which I’m guessing many of you reading this have heard of or used in some way before. For those who don’t know, we’ll be exploring Max/MSP later on in this book, but for now, just know that it is another, separate, powerful application that is used to graphically program digital music and video, among many other applications. Max/MSP is proprietary software, created by Miller Puckette and named after Max Mathews, the godfather of computer music. (If you don’t know who Max Mathews is and are interested after hearing his name both here and in the Prelude, I won’t be offended if you place your bookmark here and spend a great deal of time researching who he was, what he did, and why he is so important, because you’ll find that none of what we are discussing in this book would be possible without his groundbreaking work.) Max/MSP has a closely related twin, called Pure Data, which we’ll also explore in the book because both of these programs can be used in conjunction with RTcmix.

<sup>7</sup> Script is to score file as score file is to script. Apologies in advance for the interchangeability of these two concepts, but please know that in each case I’m referring to the text you are currently engaging in the “script\_0” window. It’s commonplace to say either “write an RTcmix score file” or “email a few RTcmix scripts.”

ahead and delete it, leaving us with a blank canvas, so to speak. Now, type the following into the window, being careful to write it exactly as I'm about to type it below.

```
rtsetparams(44100, 2)
load("WAVETABLE")
```

Thus far, we've accomplished two critical tasks. First, we've told RTcmix that we'll be working with the standard, CD-quality audio sampling rate of 44,100 samples/second or 44.1 kHz and we've also stated that we'll be working with two channels of audio output. This is the *raison d'être* of `rtsetparams`, which stands for "real time set parameters." Later on, we'll be tweaking a few more things using `rtsetparams( )`, but please note that this will be the standard header on every score file that you'll write.

We've also loaded the `WAVETABLE( )` instrument, which is a powerful wavetable oscillator that we'll encounter several times in this book. If you didn't include the quotations in `load("WAVETABLE")` or perhaps didn't include the capitalization, you're going to notice that your script won't work, so it's important to get in the habit of understanding the proper syntax. Like Ron Burgundy, RTcmix interprets text very literally, and all of the quotes, caps, commas, and parentheses need to be exact in order to run. After setting the parameters for your script, your next step will always be to load the instruments that you'd like to use, so this whole `load( )` business will become second hand in no time.

Hit return twice to give yourself a double space and type in the following:

```
WAVETABLE(start = 0, duration = 5, amplitude = 20000, frequency = 500, pan = 0.5)
```

Each time you call on `WAVETABLE( )` to execute a command, you need to tell it what to do, in the form of its parameters or "p-fields". In essence, you've told it to start making sound right away, make a sound for five seconds, at an amplitude of 20000 and a frequency of 500 Hz, directly in the middle of the stereo field. Again, we'll get into much

more detail in the next chapter, but for now, we just want to hear some sound!

In sum, your score file should look like this...

```
rtsetparams(44100, 2)
load("WAVETABLE")

WAVETABLE(start = 0, duration = 5, amplitude = 20000, frequency = 500, pan = 0.5)
```

*Score file 1: Simple sine wave<sup>8</sup>*

...and now you're ready to close the score file window, click "start rendering audio," and bask in the warm sonic bath of a 500Hz sine wave oscillator.

Congratulations on your first sound in RTcmix! Such beauty!

If you'd like to save this watershed moment of digital music and show it off to your family at Thanksgiving, please click on "write scorefile," which will open a save panel dialog box. Call it sinewave.sco — or something of your choice — and save it to your preferred location. You might want to create a folder on your /Desktop to save all of your score files into, so that you can quickly find them later.

The remaining features of the RTcmix application are pretty self explanatory, thanks to the comments that Brad has left us next to each purple icon. However, a brief summary of each feature, from top down is explained here:

1. [load scorefile] - Load a previously saved .sco scorefile for playback and editing
2. [edit scorefile] - Open a new window to further manipulate the current scorefile
3. [write scorefile] - Save the current scorefile to your preferred location
4. [start rendering audio] - Listen to the scorefile
5. The number box below [start rendering audio] changes the total duration that your scorefile will play. You can double click on this box and type in a number, or hover over

---

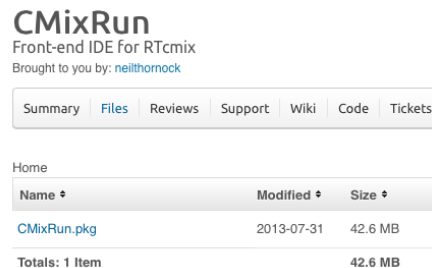
<sup>8</sup> All of the example score files are included online for downloading, though it's highly recommended to copy them out line by line rather than copying and pasting.



it with your arrow, click and hold, then scroll up or down to change the durations.

6. [show audio] - Open a new window displaying the current scorefile waveform
7. [normalize audio] - Perform audio normalization, which expands the peak amplitude of the current waveform and applies a constant shift in gain to maintain the integrity of the original.
8. [write audio file] - Save the current scorefile's audio waveform as an audio file
9. [play audio] - Playback for the current waveform

Neil Thornock is the author of CMixRun, a front-end application that includes all of RTcmix in another easy to use, graphical interface. You can download the application from his site on Sourceforge: <http://sourceforge.net/projects/cmixon/files/?source=navbar> and navigate to the link for his “CMixRun.pkg” file, which contains everything that you’ll need.

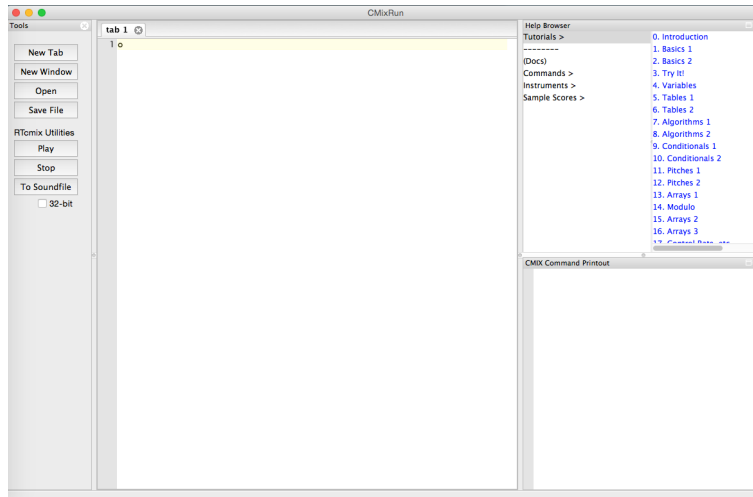


The screenshot shows the SourceForge project page for CMixRun. At the top, it says "CMixRun" and "Front-end IDE for RTcmix". Below that, it says "Brought to you by: neilthornock". There is a navigation bar with links: Summary, Files, Reviews, Support, Wiki, Code, and Tickets. The "Files" tab is selected. Below the navigation bar, there is a table with columns: Name, Modified, and Size. The table contains one row: "CMixRun.pkg" with a modified date of "2013-07-31" and a size of "42.6 MB". At the bottom of the table, it says "Totals: 1 Item" and "42.6 MB".

CMixRun		
Front-end IDE for RTcmix		
Brought to you by: neilthornock		
Summary Files Reviews Support Wiki Code Tickets		
Home		
Name	Modified	Size
CMixRun.pkg	2013-07-31	42.6 MB
Totals: 1 Item		42.6 MB

Once you’ve downloaded it, double click on the icon to begin the installation process. This will automatically place the CMixRun application into your Applications folder.

Go ahead and open the program, which has by far the coolest Dock icon you’ll find. When the program opens, you’ll find a really useful interface with a series of windows.



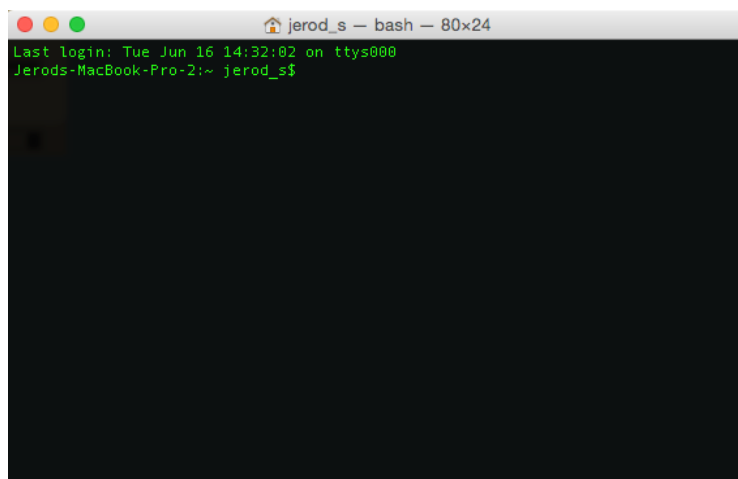
The leftmost window includes all of the tools that you'll need to either open new tabs for working on multiple score files at once, create a new window, prompt your Finder to search and open .sco files, save files to disk, as well as play, stop, and save scripts as audio files. Your main window will serve as your scripting window and includes line numbers and text highlighting to better view the commands and instrument calls that we'll be utilizing in our work. To the upper right, you'll find a series of tutorials written by Neil, which I highly encourage you to scour while working on this text. The window to the lower right will let you know if your score files have run successfully or if they have any errors.

Go ahead and double-click on "0. Introduction" and the script will open in your main window. Follow Neil's instructions and you'll hear one of his score files in action!

As we saw with the RTcmix standalone application and CMixRun, our operating system has a wonderful set of tools to streamline the process of downloading and installing software to your computer. Moreover, Apple currently boasts an App Store which serves as a one stop shop for all of your software needs and generally requires little more than a simple point-and-click to initiate and execute your download. Because RTcmix isn't proprietary software (meaning that you don't have to pay for it) and is open source (meaning that you'll be able to alter the program itself in any way you

wish), we won't find it in the App Store. Instead, we will have to download the program from its website and execute the installation manually, which is more fun, anyway.

We will be running our installation of RTcmix from the command line using UNIX. This is a powerful computer programming language that is integral to the Macintosh OS. In fact, using UNIX is a central part of our work with RTcmix and we'll be exploring and utilizing UNIX with a utility on your Mac called Terminal, which can be found at `/Users/User/Applications/Utilities/Terminal.app`.<sup>9</sup> Here's an example of the default Terminal window, which is black with green lettering, though you can change your window's appearance in Terminal's preferences.



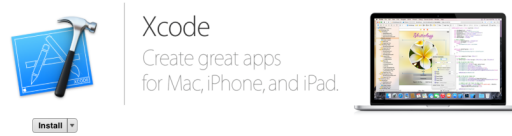
We'll also be compiling RTcmix manually, so there are a few steps we'll need to complete in order to get us started, to wit:

1. Download XCode
2. Download the RTcmix package
3. Install RTcmix from the Terminal Utility

---

<sup>9</sup> My Terminal Utility is on the directory `/Users/jerod_s/Applications/Utilities/Terminal.app`. We'll be getting around on our Mac using UNIX and Terminal in a bit, so for now of course you'd want to simply click on your Applications folder and find Utilities from there.

Our first task is to head to the aforementioned App Store and search for the program XCode.



We will be downloading this powerful program designed to facilitate — among many tasks — the building of both OSX software and iOS apps using a variety of computer programming languages. We’re not targeting those tasks quite yet (RTcmix has a number of useful tutorials for building iPhone apps using RTcmix as your audio engine), but we are for now interested in a component of XCode, called the Developer Tools Library.<sup>10</sup>

While XCode is downloading (which is going to take a bit of time as the size of the program is over 2GB) let’s explore exactly what it is that we are adding to our machines.

The instrument libraries, scorefile functions, and other intrinsic components that make up the RTcmix program were all written in the C and C++ computer programming languages. Because RTcmix is distributed freely and open-source, you (the user) will be able to take a look at all of this code that went into designing the program itself. However, C and C++ are examples of so-called high-level programming languages, meaning they borrow terms and syntax from our spoken language in order to help create a somewhat intuitive programming experience. As mentioned in the Prelude, RTcmix scripts are written in a high-level language called MINC, which stands for “MINC-is-not-C.” Once we delve into the details that constitute the anatomy of an RTcmix scorefile, we’ll be able to better differentiate MINC and C, but for now it’s important to

---

<sup>10</sup> Alternatively, you can open a Terminal window and type `gcc`. You’ll receive a note that you don’t have these tools, but hit enter and a dialog box will appear in your Finder, prompting you to download them directly. It is nice, however, to have XCode for other projects, including using RTcmix in iOS.

understand their relationship to low-level languages.

Also referred to as assembly-language or machine code, low-level languages correspond and interact closely to the computer's unique hardware architecture. An incredibly detailed sequence of instructions, these assembly-languages get at the heart of the machine: Streams of binary data. Represented as the decimals 0 and 1, these strings of numbers assist with complex tasks that your computer executes each time you put it to use. However, because of the level of detail and level of foreignness to our written and spoken languages, it is useful for us to have a translator, which we refer to as the compiler.

Within the Developer Tools Library exists an important compiler, GCC. The GCC compiler serves as the bridge between the code that built RTcmix (C and C++) and the machine-language that supports and executes commands in the Macintosh Operating System. So, in order to install RTcmix on our OS, we need the GCC compilers to help our machine understand what it is that RTcmix will be asking of it. What follows are the steps to take in order to make that process happen.

Open your preferred internet browser and navigate to the RTcmix website at <http://www.rtcmix.org>. From the home page, you'll be able to find the link for the "Standalone" version.<sup>11</sup>

- **Standalone**  
A command line version that runs on most Unix-like systems, including various flavors of Linux, Mac OS X, IRIX, and FreeBSD.

Once you click on that link, you'll be taken to another page that gives you two options for downloading RTcmix from the GitHub website. Choose the first option.

---

<sup>11</sup> I know, this is probably confusing. All this time was spent installing and playing around with the "standalone" application, and now here on the RTcmix page is yet (another?) "standalone" version. In this instance, the authors of RTcmix are referring to RTcmix's ability to be used on the command line, without any GUI interface or front end application to assist you while using it. To alleviate confusion, I'll be referring to this version of RTcmix as "command line" or more generally, the version that uses the Mac Terminal.

1. Go to the [RTcmix Releases page](#), and download the most recent (or other) release, using the "Source code (zip)" or "Source code (tar.gz)" buttons. Unpack this archive, if necessary, by double-clicking its icon. This should produce a folder called "RTcmix-4.1.0" (or a similar version number). Then move this folder into the place where you keep RTcmix source code. The usual place for such things is `/usr/local/src`, but it can be anywhere.

Once you are on RTcmix's GitHub site, click on the link that will download the source code .zip file.



RTcmix most likely downloaded to your `/Downloads` directory, and although you are welcome to leave the "RTcmix-master" folder named as-is, I've always gotten into the habit of renaming it something much more concise, and simply "RTcmix" works well for me.

Once you have RTcmix in `/Applications`, open a Terminal window and type

```
cd /Applications
```

and hit enter. At this point, your Terminal should have three lines on it, which look something like this:

```
Last login: Fri Aug 30 11:34:10 on ttys000
localhost:~ mycomputer$ cd/Applications
localhost:Applications mycomputer$
```

Congratulations again! You just executed your first UNIX command! Do you feel like Neo from the Matrix yet?<sup>12</sup> Now, type in

---

<sup>12</sup> I remember the first time that I successfully executed a UNIX command and while at the time I was furiously keeping up with the in-class example I never really did stop and ask myself, "OK this is cool, but what exactly did I just do? `cd`? What does that even mean?" As alluded to earlier, our next chapter will detail UNIX in much greater detail (in a way that is designed specifically for the RTcmix user) but it's also important to know that the `cd` command stands for "change directory." More on that later.

```
cd /RTcmix
```

provided that you changed the name of the RTcmix download folder to something simple, in which case you'd put in the original name after the cd command.

At this point, you are now working inside the RTcmix folder and have access to all of the items within it. This is important, because now we're going to use those GCC compilers to build the program. To get that process going, type

```
./configure
```

and watch what happens. Your Terminal window should now be scrolling through line after line after line of syntax, which details the play-by-play of the build process. This is the internal workings of your computer that many don't ever see when installing software from a Setup Wizard or other GUI download assistant. Depending on the speed of your machine, this might take a minute, so hang tight.

Once the process is complete, your Terminal readout should look something like this.

Summary...

```
Building 64-bit binaries
Build with multi-threaded support..... : false
Build with Perl support..... : false
Build with Python support..... : false
Build with ALSA support..... : false
Build with JACK support..... : false
Build with NetPlay support.....: false
Build with FFTW support..... : false
Build with OSC support..... : false
```

This is RTcmix's way of letting you know that the process was successful and complete. However, we didn't install any of the ancillary components of the program,

such as the ability to send OSC data or write our score files using the Python programming language. If, for example, you are more comfortable working in Python or Perl, you can choose these options during the configuration process by typing the following into Terminal:

```
./configure --with-perl --with-python
```

You are of course able to select as many options as you'd like, though you'll also need some ancillary downloads that will be explored later on. For now, there is no need to to configure with anything else, thus allowing us to make music right away using the standard MINC parser.

Now that RTcmix has been configured, we need to “make” all of the instruments and scorefile commands in its library. This is a simple process that only involves typing

```
make
```

and hitting enter. This process should take longer than configuration, so hang tight for a bit. Once that is done, the last step is to actually install those instruments and commands onto your computer, by typing in

```
make install
```

and voila! RTcmix is now installed on your machine!

We are interested in an executable file found in RTcmix's /bin folder called CMIX. To verify that the entire installation process worked correctly, execute the following in your Terminal, hitting [enter] after each command:

```
cd bin  
CMIX
```



You'll know that RTcmix successfully installed if after having typed in CMIX, your Terminal prints out the following:

```
-----> RTcmix 4.0.1 (CMIX) <-----
```

All of our installation work went into ensuring that this executable command is working properly, which allows us to have our MINC score files interpreted by our computer and realized as actual sounds. Type control-c to “interrupt” the executable and return to your working Terminal window.

Use your preferred method of searching for files on your machine<sup>13</sup> and look for any .sco files that you now have. RTcmix comes with a variety of help files, useful documentation, and examples to draw from, which you'll find in the RTcmix folder. More specifically, they can be found in /Users/yourcomputer/Applications/RTcmix/docs/sample\_scores.

Any score file will work for now, so don't worry if you don't know what it is or what it does at this point. If you want to hear a score file similar to the one we sampled in the standalone application, check out the STRUM( ) library of sounds. When you have a file in mind, double-click to open it, most likely by way of your TextEdit application.

Provided that you are still working in the RTcmix/bin directory and have access to the CMIX command, type

```
CMIX <
```

---

<sup>13</sup> Some prefer the Spotlight magnifying glass on the top right corner of your screen and others might prefer to open a Finder window to search your folder, files, and directories. In the next interlude, we're going to do it the stealth, UNIX-y way by using find and grep.

in your Terminal window. The < sign acts as a pointer to the score file that you'd like to execute. Your TextEdit window should have an icon with the .sco file's name seated above the text window with all of the MINC code. By clicking on that icon and holding it down, you'll be able to move it across your screen, much like you would do if you wanted to open a file with an application found in your dock. This is a shortcut to tell RTcmix (or any application on your computer, really<sup>14</sup>) in which directory to find the file that you are attempting to open. By dragging that /.sco icon in TextEdit to your Terminal window, you'll see a green "plus" icon appear, which indicates that you can release your click and have Terminal understand the directory of the file in question. For example, if I wanted to open Brad Garton's STRUM1.sco file, I would open it in TextEdit, drag its icon to Terminal, and see the following in my window:

```
CMIX < /Users/jerod_s/Applications/RTcmix/docs/sample_scores/STRUM1.sco
```

Provided that your window reads something similar, you can now hit [enter] and RTcmix will play the script!

Because we don't want to always change directories into our RTcmix folder and the /bin directory, we need to be able to access the CMIX from anywhere. Luckily, there is a root /bin directory on your machine, which houses all of the fun commands we've been working with thus far, like cd, and others that we'll see, such as pwd, ls, and sudo.

---

<sup>14</sup> Never one to get into the habit of shameless plugs, I can't help but introduce TextWrangler, from Bare Bones Software. You can find the download link at <http://www.barebones.com/products/textwrangler/> which will take you through an automatic installation process. I like TextWrangler because it, like RTcmix, is free software, and is highly customizable. I recommend viewing your .sco files in TextWrangler with the line numbers (which show up on the left-hand side of the application). When we inevitably have a syntax error while writing our MINC scores, RTcmix will tell us (in the Terminal) that we have an error at "line X." These errors can be maddening to seek out in TextEdit, but much easier in TextWrangler. Moreover, Joel Matthys created a brilliant syntax highlighter for RTcmix and TextWrangler (<https://github.com/jwmatthys/rtcmix-mode-textwrangler>) which will display native scorefile commands and instrument names in beautiful, wonderful colors. This too is a great way to seek out specific elements in your MINC code with minimum strain and parsing through line after line of mundane black on white.

We're going to make a copy of the CMIX executable file and place it with all of the others, thus allowing us to use it and play scripts wherever we happen to be working while on the Terminal. Typing

```
sudo cp CMIX /bin/CMIX
```

is a quick way to complete this task, but it will ask you to enter the password that you use to log into your machine. `sudo` is one of my favorite UNIX commands as it allows you to execute commands as a super user. In this case, we need to ask an administrator (you) if it is okay to add commands to your root directory.<sup>15</sup>

If everything up to this point has worked smoothly and your computer is executing CMIX and making wonderful sounds, then we've completed the installation process and this, our first sonata. Later on, we'll explore a variety of ancillary downloads to further enhance your RTcmix experience - such as viewing sets of data in actual graphs or using your mouse or trackpad to change variables in real time - but for now, we have all the tools that we need to make music and start exploring the details of the program.

## **Sonata I: Successful installation**

In a text file, write down a set of directions (in your own words) that will assist you — or others that you might someday show — in the RTcmix installation process from the Terminal. Then, see if you are able to install RTcmix on another machine, or have a friend install RTcmix on their machine from your set of directions.

---

<sup>15</sup> I highly encourage you to visit the following xkcd comic, which explains this concept brilliantly: <http://xkcd.com/149/>.

## || Interlude I: Basic UNIX for the RTcmixer ||

Each time we sit down to work on an RTcmix score, we'll first head to our Applications folder, search for Utilities, and open Terminal, a small, unassuming window, that allows you to type in a variety of commands and have them executed on your operating system, hence why it is often referred to as the "command line."<sup>16</sup> The purpose of this interlude is not to explore every feature of the Terminal<sup>17</sup> at our disposal, but to get a quick feel for the types of commands that you'll be using each time you use RTcmix. While going over all of the intricate details and walking the entire labyrinth of UNIX isn't exactly our goal here either, getting some useful, basic information from UNIX will greatly enhance not only our experience with RTcmix, but also our experience in working with our computers.

At the heart of the Macintosh Operating System (OSX) lies UNIX, a capable operating system that often goes unnoticed in your day-to-day computer business, because Apple has lovingly created OSX with a gorgeous user interface. For example, if you were to create a new folder on your Desktop (by going to your top menu and selecting File -> New Folder) you might not be aware that it is possible to accomplish the same task in Terminal, by typing the command `mkdir`. You are probably familiar with iCal (now called Calendar), Apple's native calendar application. UNIX has its own, albeit spartan, calendar and can be accessed by opening up Terminal and typing

```
cal
```

---

<sup>16</sup>As another tidbit of commonplace terminology, you'll often hear the phrase "RTcmix on the command line." This refers to our using RTcmix with the Terminal, rather than with the standalone application, Pure Data, Max/MSP, or other front-end applications that are able to execute CMIX commands. This is, to borrow a phrase from Kirk McElhearn, our way to use our computers "under the hood" and without any reliance on the graphical user interface.

<sup>17</sup>Okay, so let me officially plug a beautiful book by Kirk here. If you're like me and you like to have a lot of books on your bookshelf for quick reference or you just want to have the best source of learning UNIX on Mac OSX for beginners, please, please check out *The Mac OS X Command Line: UNIX Under the Hood*. It is a brilliant, easy to follow introduction to the variety of tasks that you can accomplish from the Terminal window.

Your window should give you a view of the calendar for the current month. You can also check to see what the calendar looked like for the month and year that you were born. For example, if I wanted to know what it looked like in December of 1982, I would need to type:

```
cal -m -y 12 1982
```

In order to change the view of our calendar, we had to use a few so-called flags. In this case, we wanted to view a particular month in the future (or past), so we needed to let the Terminal know that we would be specifying the month, as well as the year, with the flags `-m` and `-y`, with a space between each individual flag. After that, we wrote, in order, the month and year in question. For some, the configuration process while installing RTcmix was done with flags, especially if you chose or will be choosing to write your score files in the Python programming language, in which case you wrote `./configure --with-python`.

Each UNIX command (like `cal`) contains its own help file, which is a great way to learn more about the processes that you are executing. For example, type the following into your Terminal and read the corresponding printout.

```
man cd
```

You just prompted your Terminal to give you some more information about the command that allows you to change directories, `cd`. When you are done reading, type the letter “q” and you’ll exit the help menu. As mentioned in the section on downloading and installing RTcmix, it’s important to remember that as you navigate around your computer — moving from your Applications to Documents to your Downloads and back — you could move in the same way using `cd`. Now if you installed RTcmix into your

Applications folder, take a moment to practice navigating your way there.<sup>18</sup> It might help to type `pwd` in order to print your working directory, or see where you are currently working. Calling `cd` all by itself will send you back to your home directory and if you want to be super stealth, try typing two periods like this

```
cd ..
```

and after doing another `pwd` you'll notice that you'll find yourself up one level, back where you came from.

While it's all well and good to navigate around directories, we need some commands that will be useful for us in conjunction with RTcmix. For example, navigate to your RTcmix folder and type in the following:

```
ls
```

You should see a number of items listed in your Terminal window, which should look something like this:

```
Macintosh:~ jerod_s$ cd /Applications/RTcmix/
Macintosh:RTcmix jerod_s$ ls
AUTHORS          config.h.in      lib
ChangeLog        config.log       liblo-0.27
INSTALL          config.status    makefile.conf
LICENSE          config.sub       makefile.conf.in
Makefile         configure        pkg
NEWS             configure.ac     pkg-config
README           defs.conf        scores
THANKS           defs.conf.in     shlib
aclocal.m4       docs             site.conf
```

---

<sup>18</sup> Now, you might need to add a slash in front of applications, so you'll be typing `cd /Applications` for this particular example. This can be confusing, as you don't need the same slash for Desktop or Documents. If you have a long list of things to get through — such as the `sample_scores` folder in RTcmix — you'll want to take advantage of the “tab complete” feature in Terminal. Just start typing the first two or three letters of the directory that you are trying to move to and UNIX will do its best to complete the rest of the typing for you. How cool!

apps	genlib	snd
bin	include	src
config.guess	install-sh	test
config.h	insts	utils

We've listed all of the items in our directory and for now many of them might be confusing. You can probably guess what the AUTHORS or NEWS, or README, or THANKS<sup>19</sup> files are all about, but there are certainly a few listed in here that while now seem at first blush vague and perhaps cryptic, are going to be of interest to us, to wit:

1. /docs contains all of the sample scores and sample code for RTcmix
2. /lib houses the source code for RTcmix's instruments, written by its authors
3. /snd contains sound files for you to sample and please listen to Looch
4. /bin has the CMIX executable, as well as PYCMIX for Python, etc.
5. /src keeps all of RTcmix's source files and source code<sup>20</sup>

Take a moment to navigate to /docs/sample\_scores and type in an ls command to view all of the sample scores that are available to you right out of the box. Each one of these scores are there for you to alter, explore, and utilize in creating your own music. Moreover, each one was created by one of RTcmix's authors, which is a great way for you to see how they write their code. As you move along in the program, you'll find syntactical tendencies with your own coding and will quickly notice that despite the

---

<sup>19</sup>If you're curious, go ahead and type in open THANKS (or any of the others) and your TextEdit application should open up and display their contents. For those especially curious, now is a good time to read the AUTHORS file and google (still can't believe that that is a noun) their names. You'll find a wealth of fantastic music written by all of them and know that all of our work in the program wouldn't be possible without their time, care, and efforts.

<sup>20</sup> For those who plan to alter/change the program in some way, this will be a place for you to check out. While the scope of this book is to use RTcmix as a compositional tool, it's always a joy to scour through the work of others' often thankless efforts in creating such great digital tools, which is a unique and wonderful task when working with open source software.

program itself not really changing in any way, many different users will have very different approaches to coding their music in RTcmix.

Using `cd ..`, find your way back to your home folder for RTcmix and navigate to `/insts`. When you list the contents, you should see something like this on your screen:

```
Makefile      bgg           joel          std
stk           base       jg            maxmsp std-04
vccm
```

Within this folder are more example score files and documentation for RTcmix, straight from the authors who created each respective instrument. For example, you can navigate to John Gibson's instrument folder, Brad Garton's, Joel Matthys', or the standard RTcmix instruments, those in the synthesis tool kit, or those that can be used in conjunction with Max/MSP (which we'll cover a bit later.)

As we move along and create many, many new score files, we'll definitely want to be able to search for certain commands that we've previously used so that we can recall it later. For example, let's say that we are trying to make some sounds using granular synthesis, and we want to use a previously coded Hanning window for our grain envelope. Although we know we used it, we can't quite remember the exact score file that it was utilized within. This is where some UNIX search tools will come in handy, including the powerful `grep` command.

Navigate your way into the `/RTcmix/docs/sample_scores` directory and type the following into your Terminal window:

```
grep hanning *
```

and you'll see a readout that looks like this:

```
Macintosh:sample_scores jerod_s$ grep hanning *
CONVOLVE1_2.sco:src_env = maketable("window", 1000, "hanning")
CONVOLVE1_2.sco>window = maketable("window", 1000, "hanning")
```



```

GRANSYNTH1.sco:envtab = maketable("window", 2000, "hanning")
GRANSYNTH2.sco:granenv = maketable("window", 2000, "hanning")
GRANSYNTH3.sco:envtab = maketable("window", 2000, "hanning")
GRANULATE1.sco:envtab = maketable("window", 1000, "hanning")
GRANULATE2.sco:envtab = maketable("window", 1000, "hanning")
GRANULATE3.sco:envtab = maketable("window", 1000, "hanning")
GRANULATE4.sco:envtab = maketable("window", 1000, "hanning")
JCHOR1.sco:grainenv = maketable("window", 1000, "hanning")
JCHOR2.sco:grainenv = maketable("window", 1000, "hanning")
JCHOR_8chan.sco:grainenv = maketable("window", 1000, "hanning")
JGRAN1.sco:genv = maketable("window", 1000, "hanning")
JGRAN2.sco:genv = maketable("window", 10000, "hanning")
JGRAN3.sco:genv = maketable("window", 10000, "hanning")
JGRAN4.sco:genv = maketable("window", 1000, "hanning")
JGRAN_FLANGE_REVERBIT.sco:genv = maketable("window", 10000, "hanning")
JGRAN_JDELAY.sco:genv = maketable("window", 10000, "hanning")
JGRAN_REVERBIT.sco:genv = maketable("window", 10000, "hanning")
SHAPE1.sco:indexguide = maketable("window", 1000, "hanning")    // bell curve
STEREO3.sco:env = maketable("window", 10000, "hanning")
grep: disk-based: Is a directory

```

We’ve just queried our directory to seek out and return all of the instances where the string “hanning” is located. So, within all of the score files, it managed to find quite a few instances of the Hanning window! Looking at the first returned line, we can see that the string “hanning” was used in the CONVOLVE1\_2.sco score file, where it was used to declare the variable `src_env` (presumably the audio “source” envelope shape in convolution) and was utilized in a `maketable` command (which we’ll use A LOT) and its requisite variables ("window", 1000, "hanning"). You’re probably wondering why we used the `*` character for our search: This is a so-called “wild card” that can be used in the `grep` command. In this case, we’re using our wild card to search only in the `/sample_scores` directory.<sup>21</sup>

While the world of UNIX is in and of itself varied and exciting and well worth exploring in more detail, we’re now equipped with most of the basics that we’ll be using

---

<sup>21</sup> For more on wild cards, you might want to refer to UNIX Under the Hood or run a quick online search. Suffice it to say, typing in `grep` without that wild card will take a long, long time, as our entire computer will be searched.

in conjunction with our work in RTcmix. Here and there we might come across a few new UNIX commands, but with this Interlude, we should be ready to make some music!

### **Interlude I: Scavenger hunt**

Using the Terminal and your new UNIX wizardry, locate and open a file called `randfuncs.c` on your computer. Once open, see how much of the code you might be able to understand and how it relates to use with RTcmix and then rewrite one of the functions in a new TextWrangler window.

## || Sonata II: The Anatomy of an RTcmix Score File ||

While certainly daunting at first blush to look at a score file that contains hundreds of lines of code, each RTcmix script begins in exactly the same way, calls instruments just like any other, and sets up routines for your computer to execute. We're going to start small and bit by bit construct something fun, all the while gaining a good understanding of what exactly goes into an RTcmix score file.

As mentioned earlier, each script will intrinsically begin by setting up our audio sound card and loading instruments.<sup>22</sup> In a new TextWrangler window<sup>23</sup> type in the following text:

```
rtsetparams(44100, 2)
load("WAVETABLE")
```

Again, we are setting our sampling rate to 44,100 samples per second and utilizing two channels of audio output. If, say, we have a super fancy new FireWire sound card that we'd like to try out and we know that it can sample at much higher rates, we could declare 96,000 or something even higher, if it is supported. Moreover, let's say we're working in a studio that boasts eight channel surround sound, in which case we could opt to declare eight channels of output.

One feature that we will start using now and is highly, highly recommended for your work is the comment. By placing two slashes in front of any text, RTcmix will leave that bit out during execution. So, I can comment the code we've written to help me remember what each line does. If you have lots of text that you'd like to use as a comment, you can use block comments instead.

```
rtsetparams(44100, 2) //set parameters, rtsetparams(samplerate, channels)
```

---

<sup>22</sup> Unless, that is, you are running your scripts in Pure Data or Max/MSP, in which case you don't need to do any of the steps to set parameters or load instruments.

<sup>23</sup> Or if you'd like to keep things in Terminal, don't forget about pico or nano.

```
load("WAVETABLE")      //load the WAVETABLE instrument
```

```
/*
```

A slash and then a star will make a block comment.

Anything inside of those characters

will be

commented out.

Use a star then slash to finish the block comment.

```
*/
```

Were we to omit those slashes after `rtsetparams( )`, we'd get an error in our Terminal readout, which would look something like this:

```
Macintosh:~ jerod_s$ cmix < /Users/jerod_s/Desktop/mygreatfile.sco
-----> RTcmix 4.0.1 (cmix) <-----
=====
rtsetparams:  44100 2
Audio set:  44100 sampling rate, 2 channels

*** ERROR [parser-yyerror]: near line 1: syntax error
```

Looking back, we'd find our way to line 1 and make note of our error, in this case the fact that we forgot to include our slashes for the comments.

`WAVETABLE( )` is a great instrument to use when starting out with `RTcmix`, as it is able to synthesize all of the standard waveforms (sine, sawtooth, triangle, square) and when used in conjunction with `maketable( )`, gives the user the ability to create custom waveforms, subtract and add respective partials and adjust the amplitudes of those partials, or even make some fun, grainy speaker clicks to use as textural material. In short, it is a highly versatile instrument and one that we will utilize often while working through the program.

Earlier, we tested our installations of `RTcmix` by implementing a simple sine wave at 500 Hz. Let's rehash that, but keep it a bit more bare bones for now.

```
rtsetparams(44100, 2) //set parameters, rtsetparams(samplerate, channels)
load("WAVETABLE")      //load the WAVETABLE instrument

WAVETABLE(0, 5, 20000, 500, 0.5) //play a sine wave for 5 seconds
```

*Score file 2: Simple sine wave with comments*

At this point, it is useful to introduce the notion of variables and how they can be used in RTcmix, because as we can already see, housed within WAVETABLE( ) parameters are a series of values separated by commas that for now are somewhat meaningless to us. A variable is nothing more than a declaration inside of your script. For those who are more familiar with C programming, we don't need to specify whether or not our variable will be an integer or a float or anything else, which is quite handy in the MINC parser. As a simple example, I can create variables for integers to print to the Terminal window by typing the following code:

```
x = 2
y = 3

sum = x + y
print(sum)
```

*Score file 3: Print to Terminal window using variables*

In the case of our example using WAVETABLE( ), it is going to be useful to substitute variables in place of our integer values so that we better understand how they function.

Each RTcmix instrument contains a series of parameters, or p-fields, which need to be declared in order to properly execute sound. They are listed in order within the parentheses and are separated by commas. In this case, WAVETABLE( )'s p-fields are

determine the following parameters<sup>24</sup>:

p0 = start time  
p1 = duration, in seconds  
p2 = amplitude, or loudness  
p3 = frequency  
p4 = placement of sound in the stereo field (optional p-field command)  
p5 = waveform specification (also optional)

If you're like me, you probably find it helpful to remember what these p-field commands are, not necessarily by memorizing them, but by putting names or variable names to them within your script. We can do this within the parentheses that call the instrument, like so:

```
WAVETABLE(start = 0, duration = 5, amplitude = 20000, frequency = 500, pan = 0.5)
```

However, its not always going to be the case that we want to localize those variables so specifically<sup>25</sup>. Once we add a number of other instruments and effects to our scripts, we might want a global start time, or a duration that works for a variety of different sounds. So, here is a nice looking script that first specifies our p-field parameters as variables and then puts those variables into the call to WAVETABLE( ).

```
rtsetparams(44100, 2)
load("WAVETABLE")

start = 0
```

---

<sup>24</sup> As we will encounter later on, many of the p-fields for RTcmix's instruments can be updated in real-time, or through the use of tables, defined by the maketable( ) command. For now, we're only interested in understanding what exactly a p-field is, but it will always be clear in the documentation when you are free to update those parameters can can be changed with tables or various connections.

<sup>25</sup> Once we get to Python, we won't be able to do this at all.

```
duration = 5
amplitude = 20000
frequency = 500
pan = 0.5

WAVETABLE(start, duration, amplitude, frequency, pan)
```

*Score file 4: Sine wave with variables*

You'll notice that p4, panning, is one optional parameter for you to specify. Each RTcmix instrument that you encounter will have some p-fields that are absolutely necessary for their function and some p-fields that are optional. If we didn't specify a pan value (0.5 in the middle, 0 to the right, 1 to the left), RTcmix would send a default 0.5. Likewise, p5 is optional and defaults to a sine wave. If we want to utilize one of our other available waveforms, we need to introduce the maketable( ) command. Try typing this after the declaring the variable for pan:

```
waveform = maketable("wave", 1000, "saw")
```

and include it in your WAVETABLE( ) parameters. You should have a line that now looks like this,

```
WAVETABLE(start, duration, amplitude, frequency, pan, waveform)
```

and sounds the same 500 Hz wave as before, but instead of a sine wave, now plays a sawtooth wave. Each time that you call on maketable( ), you need to specify a few parameters. First, you need to indicate the type of table that you need and put it in quotations. Since we want to draw distinct waveforms, we need to write "wave." The

1000 indicates the number of points on the x-axis of the table that we are drawing.<sup>26</sup>

1000 is a nice, easy number to remember, but that's not to say that you can't write tables of 2000, 3000, or 4000. While a more detailed explanation of the various types of `maketable( )`s will follow in the next interlude (and there is another special section dedicated to `WAVETABLE( )` itself) note for now that "wave" can provide you with the following waveforms:

"sine" -- sine wave

"saw" -- sawtooth wave

"sawX" -- a sawtooth wave with X number of harmonics

"sawdown" -- sawtooth wave whose first curve moves downward

"sawup" -- sawtooth wave that goes upward

"square" -- square wave

"squareX" -- square wave with X harmonics

"tri" -- triangle wave

"triX" -- triangle wave with X harmonics

"buzz" -- pulse wave

"buzzX" -- pulse wave with X harmonics

Thus far, we've approached our score file from a fairly methodological angle. Our goal was to make a simple sine wave, have it sound for a few seconds, then turn off. What if, however, we wanted to start constructing scales and modes and impressing our friends with endlessly rising whole-tone-scale-dream-sequence sounds? For that, we can call a full gamut of `WAVETABLE( )` instruments:

```
rtsetparams(44100, 2)
```

---

<sup>26</sup> If you have RTcmix's `plottable( )` command (it requires a separate download of two separate programs called AquaTerm and gnuplot), you can see these tables for yourself by including, in this case, `plottable(waveform)` in your script. A new window will then pop up with a graphic representation of your table.



```

load("WAVETABLE")

start = 0
duration = 5
amplitude = 10000
pan = 0.5
waveform = maketable("wave", 1000, "sine")
WAVETABLE(start, duration, amplitude, cpslet("C4")27, pan, waveform)
WAVETABLE(start+1, duration, amplitude, cpslet("D4"), pan, waveform)
WAVETABLE(start+2, duration, amplitude, cpslet("E4"), pan, waveform)
WAVETABLE(start+3, duration, amplitude, cpslet("F#4"), pan, waveform)
WAVETABLE(start+4, duration, amplitude, cpslet("G#4"), pan, waveform)
WAVETABLE(start+5, duration, amplitude, cpslet("A#4"), pan, waveform)
WAVETABLE(start+6, duration, amplitude, cpslet("C5"), pan, waveform)
WAVETABLE(start+7, duration, amplitude, cpslet("D5"), pan, waveform)

```

*Score file 5: Whole tone scale*

which if you typed out in full and didn't take advantage of at least copy and paste, can get pretty maddening pretty quickly. The snippet of code above does in fact work: With a new duration of one second, it plays C4, then after one second plays D4, then after another second E4, and on and on and on. There really isn't any limit to the number of notes that you can play in sequence, but my hope is that its evident that we're doing more work than necessary. We're on a computer after all, and computers are designed to make routine tasks easier on us. For that, we'll need to construct a loop.

Loops give us the ability to repeat a number of tasks that we designate for however many iterations we desire. These loops have a pretty detailed construction that we'll utilize quite a bit over the course of our work, so its best to just take the leap of faith now, explain the concept, and commit it to memory.

---

<sup>27</sup> What kind of sorcery is this, you ask? Well, RTcmix doesn't always need to represent pitch in terms of frequency, or cycles per second. In fact, I can't imagine a music theory class where notes were represented in only Hz and nothing else! This `cpslet( )` command translates note letter and octave designations to cycles per second. You can literally read the call as "To cycles per second (cps) , translate note letter names (let)." There are a variety of ways to represent pitches and notes and frequencies in RTcmix, which we'll of course explore in detail.

```

rtsetparams(44100, 2)
load("WAVETABLE")

start = 0
duration = 1
amplitude = 10000
note = 60 //the MIDI note number for middle C, or C4
pan = 0.5
waveform = maketable("wave", 1000, "tri")

for(start = 0; start < 7; start = start + 1){
    WAVETABLE(start, duration, amplitude, cpsmidi(note)28, pan, waveform)
    note = note + 2 //increment by a whole step
}

```

*Score file 6: Whole tone scale from a loop*

Okay so here's what we just accomplished. Our loop construction is literally interpreting us saying, "Make the variable 'start' equal zero. Then, as long as start is less than seven, increment start by one."<sup>29</sup> You'll notice that because we also utilized "start" in our call to WAVETABLE( ), our first start will equal zero, then one, then two, then three..., just like it did above when we had to physically write it over and over. Way easier. Moreover, starting a loop will give us the chance to change variables within the loop structure,<sup>30</sup> as we did when we wrote that we wanted "note" to equal itself, but add two. Thus, our variable "note" will first equal 60, then 62 (60+2), then 64 (62+2), then 66 (64+2), etc.

---

<sup>28</sup> Yet another way to designate pitch. MIDI note numbers are a great way to represent pitch and cpsmidi( ) will do the translation for us. Think of this one as stating, "To cycles per second (cps), translate MIDI note numbers (midi)."

<sup>29</sup> Readers who are super familiar with the C programming language are probably wondering why our increment didn't use start++ (Not to mention the absence of semicolons). MINC doesn't include that functionality and after all, MINC-is-not-C.

<sup>30</sup> That is, any commands and text that are placed within the curly braces { }. The indentation is a matter of style and preference, in MINC. Indentations are super important in Python, however.

We're incrementing by one each time through the loop and all of our durations are also equal to one. Thus, we're getting fairly clear, articulated notes, but what if we wanted to overlap those a bit? Try changing your value for duration to 1.5 or even 2 or 3 and listen to the result. If you find the sound is starting to sound distorted as more and more waveforms pile on one another, try turning your amplitude down from 20000.

```
rtsetparams(44100, 2)
load("WAVETABLE")

start = 0
duration = 3
amplitude = 10000
note = 48
pan = 0.5
waveform = maketable("wave", 1000, "buzz")

//two octave whole tone scale
for(start = 0; start < 14; start = start + 1){
    WAVETABLE(start, duration, amplitude, cpsmidi(note), pan, waveform)
    note = note + 2
}
```

*Score file 7: Whole tone scale with overlapping notes*

Each of the previous two scores contains a slight click, or pop, at the onset of each note, and another at the point where the note stops playing. This is due to the fact that we're literally asking RTcmix to generate a pitch with robust amplitude and one that is without any semblance of shape, or envelope. For example, a piano is unable to gradually fade into a note (unless each string inside the instrument is bowed) due to the sheer mechanics of the hammer striking the strings each time we depress a key on the keyboard. RTcmix is more or less acting in the same way for us right now. Unlike a piano, RTcmix is able to sculpt sounds using various envelope shapes. If each sound has, according to the ADSR envelope, a particular attack, decay, sustain, and release time, we will be able to manipulate our sounds using RTcmix and its maketable( ) command.

We'll start by having each individual note rise to its peak amplitude over a gradual amount of time, rather than start with full amplitude right away, and fade out. Let's do this in the shape of a triangle, so that our sound will fade in half way through its total duration (since we're using 3, it will fade in over 1.5 seconds) and immediately fade out over the remainder. We can think of this in terms of a graph, with the x-axis representing time and the y-axis designating amplitude, from 0.0 to 1.0.<sup>31</sup><sup>32</sup>

To draw our envelope using `maketable()`, use the following code, which specifies a graph using line point segments:

```
envelope = maketable("line", 1000, 0,0, 0.5,1.0, 1.0,0)
```

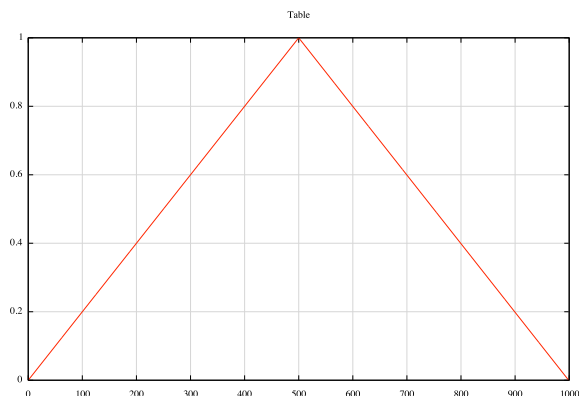
You'll note that we're creating a variable called "envelope" with 1000 points. Instead of designating "wave" as we did before, we'll use "line" to give us the line segment graph. What follows are pairs of numbers, in this case 0,0, 0.5,1.0, 1.0,0. I deliberately added spaces between each pair of numbers, which is a stylistic trait that isn't necessary to use, but one that I hope you'll also adopt for line graphs, as it makes

---

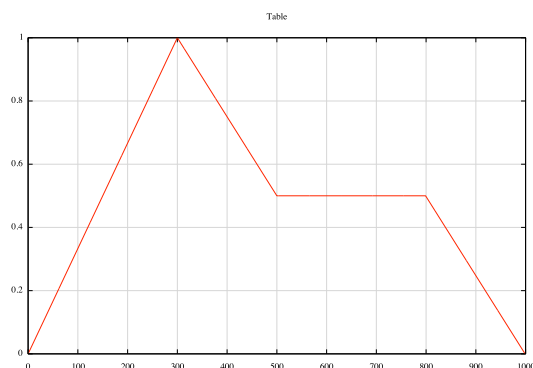
<sup>31</sup> Let's pause for a second and talk about amplitude in more detail. By now, you know that when we refer to amplitude we're discussing volume or loudness, but you've seen it referenced as a number like 20000 (which we call *absolute* amplitude) and now on a scale from 0.0 to 1.0 (called *relative* amplitude). Without diving too far into the theory of digital audio signals, think about a Cartesian coordinate plane (x,y) graph where the x-axis is time and the y-axis is amplitude. Now, we can represent amplitude relatively, with points -1 to 0 to 1 on the y-axis (knowing that signals [waveforms for example] have points of positive energy transfer called *compression* and negative energy transfer called *rarefaction*), or we can think of it absolutely, with 65536 points on the y-axis (-32768 to 0 to 32767). Why 65536? Maybe you've heard of *bit depth*, which for standard CD quality audio is 16 bit and determines how accurately (think y-axis again) we can represent amplitude values in the digital domain.  $16^2 = 32,768$ , which is the standard formula for determining those integer values.

<sup>32</sup> Note 31 was quite a handful. Again, this text isn't meant to fill the purpose of an all-in-one digital audio composition text, but some explanations are unavoidable and my hope is to introduce some of these heavier topics in a super conversational and as basic a way as possible. If you're interested, you might seek out *Computer Music: Synthesis, Composition, and Performance* by Charles Dodge and Thomas Jerse, which is, while older, pretty much a go-to and can be found on almost any electronic musician's bookshelf. Not for the faint of heart (but you're total encyclopedia) would be *The Computer Music Tutorial* by Curtis Roads.

understanding the segments that we are using much clearer. In essence, I'm stating that at the beginning of my graph, point 0, use a value of 0. Then, half way through the graph at point 0.5, declare a value of 1.0 (full amplitude). Finally, at the end of the graph, point 1.0, use 0, which is no amplitude, thus no sound.



Something seems fishy with this, however. Although I am using 1000 points in my graph, I'm using points 0, 0.5, and 1.0. Wouldn't it make sense to use points 0, 500, and 1000? Unless you use a special designation for maketable("line") called "nonorm", RTcmix will always normalize your values to something between 0.0 and 1.0. So, even if we did use 500 and 1000, the program will still interpret that as 0.5 and 1000. While it might be confusing at first, I find this pretty useful, especially when using graphs that have a large number of points. Here is our whole tone scale, which uses a graph for sculpting our notes using the standard ADSR envelope:



```

rtsetparams(44100, 2)
load("WAVETABLE")

start = 0
duration = 3
amplitude = 10000
envelope = maketable("line", 1000, 0,0, 0.3,1.0, 0.5,0.5, 0.8,0.5, 1.0,0)
note = 48
pan = 0.5
waveform = maketable("wave", 1000, "sawup")

for(start = 0; start < 14; start = start + 1){
    WAVETABLE(start, duration, amplitude*envelope, cpsmidi(note), pan, waveform)
    note = note + 2
}

```

### *Score file 8: Adding an ADSR envelope*

In just a few scripts, we’ve covered quite a bit of ground. Intrinsically, RTcmix — when used on the command line — requires that we set our parameters for sampling rate and the number of channels for audio output. From there, we load our instruments in question and have the chance to declare variables to fill in the respective instruments’ p-field values. Most p-field parameters are absolutely necessary, while others are optional. By using the WAVETABLE( ) instrument, we were able to use a variety of waveforms through the maketable(“wave”) command. Further, when put into a loop structure, we were able to repeat commands, thus only calling on WAVETABLE( ) once in our score file, but also incrementing pitch values - which can be represented in any of a variety of ways - as we moved along. Lastly, we sculpted our individual pitches with the ADSR envelope, utilizing maketable(“line”).

Here is one final score file that introduces a bit of randomness to our palette and you are encouraged to research the commands that might be unfamiliar to you and

understand how they are working in the script.<sup>33</sup>

```
rtsetparams(44100, 2)
load("WAVETABLE")

amplitude = 10000
envelope = maketable("line", 1000, 0,0, 0.1,1.0, 0.8,1.0, 1.0,0)

increment = 1.0
for(start = 0; start < 50; start += increment){
    duration = irand(4,10)
    note = trand(60,72)
    pan = random()
    WAVETABLE(start, duration, amplitude*envelope, cpsmidi(note), pan)
    increment = irand(0.25,4.0)
}
```

*Score file 9: Chromatic sine wave bath*

## **Sonata II: Random pitch melody**

Create a 30 second etude that uses at least two different graph shapes for envelopes, as well as two different ways to represent pitch. Then, randomly select pitches for the melodic material inside of a loop. Don't feel constrained to only use WAVETABLE( ), but scour the documentation in order to find a new instrument to use in this sonata, should you choose to do so.

---

<sup>33</sup> The RTcmix website's documentation is a great place to start seeking out this information. You'll note that in this script, I'm using a variable for our increment value, which will change inside of the loop. You might also see that instead of stating `start = start + increment`, I decided to use `start += increment`, which is exactly the same command. I prefer the latter since it is shorter and still easy to understand.

## **|| Interlude II: Exploring maketable( ) ||**

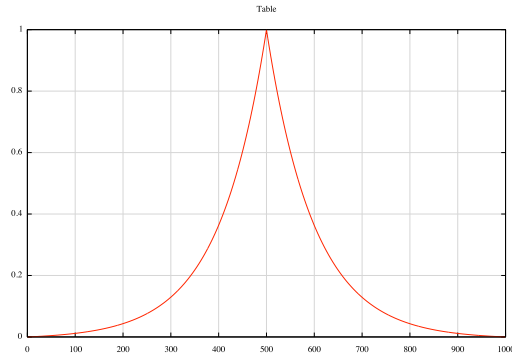
As we've already seen, `maketable( )` is very useful in the sculpting and refinement of our sounds and compositions. It has the ability to generate waveforms, graph line segments or curves, import data from text files, create tables from weighted, randomly distributed numbers, and window functions, among other useful possibilities. We'll be exploring most, but not all, of the `maketable( )` commands in detail here, but the documentation found in the index will always be your best guide to utilizing any of them in the future.

Filling a graph with line segments is a quick, easy to understand, and helpful way to manipulate data in your scripts. In score file 8, we were manipulating amplitude data, creating a standard ADSR envelope. When we consider that the outgoing amplitude of our `MAKETABLE( )` had a peak of 10000, it's easy to understand that if we wanted to have no amplitude, or 0, we would multiply  $10000 * 0$ . By subjecting our peak amplitude to the strictures of the ADSR envelope, we are able to generate streams of numbers that altered our volume during playback. We'll find later on, especially when used in conjunction with Pure Data or Max/MSP, that we can further manipulate amplitude in real-time (perhaps even with your iDevice...), but it's important for now to understand that `maketable( )`'s versatility and easy to understand syntax will greatly enhance your experience with RTcmix and the sounds you are generating.

I can't imagine a world where every detail is visually composed of straight line segments and thankfully the world of `maketable( )` isn't limited to "line" either. Let's amend score file 8's envelope shape from the standard ADSR envelope to a triangle shape that has curved line segments in its place.

```
envelope = maketable("curve", 1000, 0,0,5.0, 0.5,1.0,-5.0, 1.0,0)
```





Rather than working with pairs of numbers, as we did when using “line”, we are instead working with three sets of numbers for each plot, which as a syntactical preference I’ve spaced accordingly in my list. What we have for each segment in “curve” is a value for x, a value for y, and a curvature value. In essence, I’m asking for the same 0,0, 0.5,1.0, 1.0,0 sequence of points as before, but I’m adding curvature values, in this case 5.0 and -0.5. You can think of it in this way:

```
graph = maketable("curve", 1000, x1,y1,curve1, x2,y2,curve2..., x,y)
```

noting that at the end of our series of numbers, we need only call on a single (x,y) pair with no value for curve, thus the 1.0,0. In our example for the envelope, we used both positive and negative values for the curvature: Negative values will generate convex curves and positive values will create concave curves. The greater the values that you use, the more pronounced your curves will be. Take the envelope example that we generated and try playing around with the values, changing some from positive to negative and back, adding more line segments, using 0 as a curve value (straight line), or other transformations that will help you to better understand the possibilities that “curve” will provide for you. Moreover, read through the documentation for “spline” and see how it relates to “curve”, as well as how “linebrk” relates to “line”.

The graphs that we are generating aren’t limited for use with envelopes or

transformations in amplitude. Because many of the p-field parameters for many RTcmix instruments can be updated dynamically through tables, it stands to reason that we could use maketable( ) for them as well. In the following score file, we'll explore frequency modulation through the FMINST( ) instrument, which performs FM synthesis using a carrier frequency, modulator frequency, and index. Rather than perform the most basic form of FM - in which we have static values for its requisite components, we'll update them using three different types of maketable( )'s: "line", "curve", and "expbrk".

```
rtsetparams(44100, 2)
load("FMINST")

start = 0
duration = 30
amplitude = 10000
carrier = maketable("line", "nonorm", 100, 0,25.0, 0.25,100.0, 0.75,10000.0, 1.0,20.0)
modulator = maketable("curve", "nonorm", 10, 0,15.0,-5.0, 0.75,28.0,5.0, 1.0,0)
index_low = 1.0
index_high = 10.0
pan = 0.5
waveform = maketable("wave", 1000, "tri20")
index_envelope = maketable("expbrk", 250, 0,50, 0.75,150, 0.90,50, 1.0)

FMINST(start, duration, amplitude, carrier, modulator, index_low, index_high, pan,
waveform, index_envelope)
```

### *Score file 10: Dynamic updates for FM*

Thus far, we've kept one important parameter for our sounds quite vanilla. Because our pan values have been set to 0.5, all of our sounds have been reaching us in the middle of the stereo field at all times. RTcmix interprets panning using values of 0.0 (hard right) to 1.0 (hard left) and more often than not pan values can be dynamically updated, so we can use maketable( ) to alter our sounds in time.

While any of the various types of `maketable( )` commands will work for pan values, one very nice feature of RTcmix is the `makeLFO( )` command, which oscillates between low and high input values in a way that mimics a low frequency oscillator. For example, if we want to pan between 0.0 and 1.0, we would need to call on `makeLFO( )`, choose a waveform and a corresponding frequency<sup>34</sup>, and our low value (0.0) and high value (1.0).

```
pan = makeLFO("sine", 5.0, 0.0,1.0)
```

One nice addition to `makeLFO( )` is the fact that the p-field for frequency can be dynamically updated! Thus, it's possible to change how fast or how slow the low frequency oscillator is causing our sounds to sweep across the stereo field. You might even try dynamically updating the range of values that `makeLFO( )` is sweeping through?

```
pan_frequency = maketable("line", "nonorm", 10, 0,5.0, 0.25,15.0, 0.625,2.5, 1.0,5.0)
pan = makeLFO("tri", pan_frequency, 0.0,1.0)
```

While we teased the notion of randomness using `random( )`, `irand( )`, and `trand( )`<sup>35</sup> earlier, there may be times when a table of random numbers is more desirable.

---

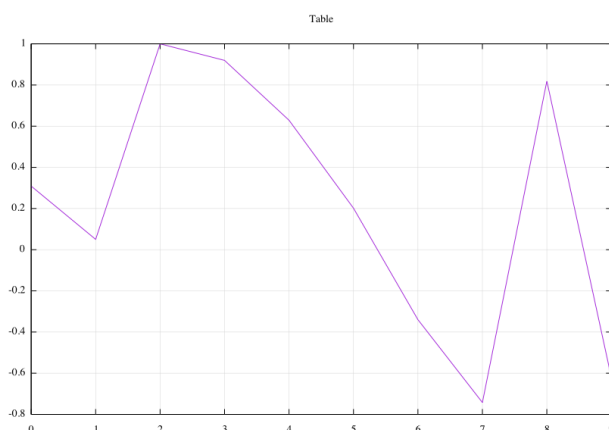
<sup>34</sup> In order for your oscillator to truly act as an LFO, specify a frequency that is less than 20 Hz, which is the generally agreed-upon low range of human hearing. Anything greater than or equal to 20 Hz will start to distort your sounds in a way that is akin to a ring modulator, which can also be quite interesting to hear. As with anything discussed in the book, it's always best to try things out and play with parameters to not only sculpt and refine your music, but also to better understand how these instruments and commands are functioning.

<sup>35</sup> Hopefully you were able to successfully uncover the workings of both of these random number generators using the index or online documentation. If not, `irand( )` returns random floating point numbers, or those with decimal points, and `trand( )` returns random integers, or whole numbers. Simply using `random( )` will return random floating point numbers between 0.0 and 1.0, which is super useful for panning or relative amplitude. A detailed account of random distributions will be covered in time, but for now understand that both of these commands evenly choose a random number between the range that you specify.

For example, rather than sticking with the standard set of waveforms available through `maketable("wave")`, we can create our own waveforms that are constructed randomly.

Because a waveform oscillates between moments of positive energy transfer (compression) and negative energy transfer (rarefaction), we construct waves using both positive and negative numbers between 1.0 and -1.0.<sup>36</sup>

```
waveform = maketable("random", 10, "even", -1.0,1.0)
```



We need to declare the command `maketable("random")` with the following p-fields: The number of points on the graph (10)<sup>37</sup>, the type of distribution for those random numbers (even), and both a low value and high value. The distributions are particularly useful, which allow you to choose random numbers that are weighted. For

---

<sup>36</sup> If you'd like to bookmark this page and skip ahead to Interlude V, please feel free to do so. In that interlude, waveforms and synthesis are explored in greater detail, but for now we're only interested in the constructs of waves within `maketable( )`, especially this notion of -1.0 and 1.0 and how those values relate to amplitude.

<sup>37</sup> Note that we're only using ten points on this graph. The more and more points we use while constructing random waveforms, the more noise we'll introduce into our sound. White noise is a sound that you've probably encountered before, either through TV static (Does that even exist any longer?) or the whirring of a fan. In the digital realm, white noise exists as 44,100 random frequencies sampled each second, which corresponds to the sampling rate. Thus, to get a waveform that acts similarly to white noise (RTcmix also offers a `NOISE( )` instrument), choose 44100 for the size of your table.

example, if you wanted to choose random numbers between 0 and 11 (for a twelve tone melody, perhaps?) but you wanted to choose more often from the lower numbers, you'd specify "low". Choosing "high" will pick randomly, but more often choose the higher numbers.

There are certain random distributions that have been proven to choose numbers from interesting shapes and curves. You'll find them by using "gaussian", "cauchy", or my favorite, "triangle". Each of these will favor numbers toward the middle of your range, but in ways that look like a bell curve or, well, a triangle. If you are excited about these random distributions or probability in general, Mara Helmuth created a particularly fun "prob" option to use with maketable("random"). In it, you specify minimum and maximum values, as well as a midpoint. Then, a value for tightness will determine how closely the random numbers are chosen near the midpoint value.

```
low_value = 0.0
high_value = 10.0
midpoint = 5.0
/*
The values for tightness work in this way:
0 = randomly choose between only the low value or the high value
1 = evenly choose between any number in the range specified
Using numbers greater than one (up to 100) will begin to "tighten" toward the
midpoint.
If 100 is used, almost all numbers will be at or very near your midpoint value
*/
tightness = 75

random_values = maketable("random", 100, "prob", low_value,high_value,midpoint,
tightness)
```

Each of our tables can be further transformed using the makefilter( ) command, which is especially useful when utilizing one table to determine a variety of parameters in your script. For example, we could choose to generate an array of amplitude and pan values between 0.0 and 1.0, later transforming them to generate frequencies falling in a

range of our choice.

```
amplitudes = maketable("line", 100, 0,0.0, 0.33,0.75, 0.66,1.0, 1.0,0)
pan = amplitudes
frequencies = makefilter(amplitudes, "fitrange", 220,1530)
```

In this instance, we need to call on `makefilter( )`, first directing it to the table in question (`amplitudes`), the type of transformation that we'd like to accomplish ("`fitrange`"), and finally a range of values to utilize. Even though our original values fall between 0.0 and 1.0<sup>38</sup>, "`fitrange`" is able to expand those numbers into a new range of your choice. It is also possible to invert your values around a declared midpoint using "`invert`", or ensure that your values will stay within a specified range using "`clip`".

In all, RTcmix offers a variety of ways for you to create tables, lists of data, and transformations of those tables or data to enhance your experience, by using `maketable( )` and `makefilter( )`. As a historical note, you might encounter documentation or example scripts that utilize a system of table declarations called `makegen( )`. This is an antiquated system of creating tables that is no longer being actively updated for future use. It's not out of the realm of possibility that some older RTcmix instruments will still rely upon `makegen( )` as the sole way of creating tables, but for the most part, if you'd like to do data transformations, use only `maketable( )`. In fact, many of the older instruments have recently been revised to use `maketable( )`, so feel free to try it out. The worst that can happen is an error and maybe you'll be off to the mailing list to either ask about an updated version of the instrument in question or maybe even try updating it yourself? As with anything in this book, the best resource for finding information on any RTcmix instrument, as well as a complete list of `maketable( )` and `makefilter( )` commands is the documentation found online.

## Interlude II: Graph constraints

---

<sup>38</sup> "`fitrange`" will also work with values between -1.0 and 1.0.

Compose a 30 second script that uses only one instrument, but every single p-field of that instrument that can be updated using a `maketable( )` must be updated. Choose at least two types of `maketable( )` and transform each of them using a `makefilter( )`.

## **|| Sonata III: Elements of C with arrays and conditionals||**

Computers are able to execute routine tasks in ways that make our lives easier, which includes our compositional work days. In score file 5, we constructed a whole tone scale using separate calls to the WAVETABLE( ) instrument, later streamlining that task into a for( ) loop. Using arrays, nested loops, and conditional tests, we can further enhance our experience with RTcmix and the music we hope to make while utilizing it.

Major and minor scales play a large role in the vocabulary of anyone undertaking the serious study of western art music. For those who might be unfamiliar with the constructs of a major scale, a piano is a great resource. When looking at the keyboard of the piano, you'll notice that it is divided into a series of black and white keys. The black keys are further divided into groups of two and three. Find the grouping of two and play the white note that rests just before the first black note in the two grouping. You are playing the note C and any white note that falls in the same place along the keyboard will always be the note C.<sup>39</sup> Now, play only the white notes of the piano, ascending, from your original C to the next one and then back down. This is a major scale in the key of C.

There are many, many ways to represent the note C. One is just, well, C, which is a letter designation, useful for reading and writing music. Moreover, there is a special C, called "middle C", which has a frequency of about 261.62 cycles/second, or 261.62 Hz. To further confuse the matter, when using the musical instrument digital interface (MIDI), we represent that same middle C by the MIDI note 60. Not to be outdone, middle C is also referred to as C4 when using octave designators. To make things even more confusing, RTcmix offers octave point pitch class designation, for which middle C is

---

<sup>39</sup> It's a complete coincidence that the title of this sonata is "Elements of C:", which actually has nothing to do with the fact that we'll obsess for a bit on the musical note C. The point of the title was to shift our focus toward the constructs of the C programming language. I suppose I could've gone with "Elements of MINC:", but MINC-is-not-C, so that's confusing. To me at least.



7.00.<sup>40</sup> To wit:

```
// middle C
frequency = 261.62 //in cycles per second, or Hz
note = 60 //this is the MIDI note number
pitch = C4 //the letter and octave designator
pitch_class = 7.00 //the octave, followed by the pitch. 7.01 = C#, 7.02 = D, etc.
```

In order to construct a C major scale, we have a variety of options in RTcmix. The notes of a C major scale (which you can say aloud to yourself while playing them, ascending, on the piano) are C D E F G A B, followed by the next C, which will put you in the next octave. Since each of those notes can be thought of as an element and all of the elements together can be thought of as a list of elements, we can put them into an array, which we'll do using their MIDI note numbers.

```
c_major_scale = {60, 62, 64, 65, 67, 69, 71, 72}
```

Each element in our list is part of the major scale, which we've called "c\_major\_scale". It is important to remember that each of our elements in the array (designated as the elements within the curly braces and separated by commas) is also kept at a specific place within the list, or array, itself. Let's pick out just the note E in our list, which is MIDI note number 64.

```
E = c_major_scale[2]
```

Take a moment to think about how this is working and don't be afraid if some confusion ensues. We can see that we made a variable called "E" because we wanted to

---

<sup>40</sup> If you were left feeling confused at the piano, unable to find the note C in the first place, a quick online search should help you find that note. Any college level music theory text will also include a far more detailed account of this discussion, likely in its first chapter, though I highly doubt that they'll get into octave point pitch class. You might, however, find them explaining cycles/second, especially with regard to the note A4.

grab the note E from our array. Next, we called on the array named “c\_major\_scale” because that’s where our particular note lies. Finally, you’ll note the brackets and the number 2: This is the address for our element within the array. This is where confusion might onset.

It looks to me as if E (64) is the third element in our array, which begs the question: Why did we call on 2? Wouldn’t that return the second element, D (62)? While we do have eight elements in our array, they are numbered from 0 - 7, rather than 1-8, so you would say that C (60) is the 0th element in the array, D (62) is the 1st element, etc.

```
[ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 ]  
array = {alpha, beta, charlie, delta, echo, foxtrot, hotel, golf}  
alpha = array[0]  
echo = array[4]  
golf = array[7]    //even though its technically the 8th element.
```

Arrays are particularly useful when put into a loop, as we can cycle through their elements in the same way as playing the ascending notes for C major on the piano.

```
rtsetparams(44100, 2)  
load("STRUM2")  
  
pitch_array = {7.00, 7.02, 7.04, 7.05, 7.07, 7.09, 7.11, 8.00} //C major, in octave  
point pitch class  
pitch_length = len(pitch_array) //the len( ) command returns the number of elements  
in the array (8)  
  
//——STRUM2 p-fields  
start = 0  
duration = 1.0  
amplitude = 30000  
envelope = maketable("window", 1000, "hanning")  
pitch = pitch_array[0] //can you guess which number we’re calling on at this point?  
squish = 1.0  
decay = 1.25
```

```

pan = 0.5

for(start = 0; start < pitch_length; start += duration){
    STRUM2(start, duration, amplitude*envelope, pitch_array[start], squish, decay,
        pan)
}

```

### *Score file 11: Ascending major scale<sup>41</sup>*

Because our value for “start” will increment by one each time through the loop, it is an incredibly useful way to play through elements in an array, provided that you move from left to right. However, what if we wanted to play the scale both forward and backward, like we might do when warming up on our instrument?

```

rtsetparams(44100, 2)
load("STRUM2")

pitch_array = {7.00, 7.02, 7.04, 7.05, 7.07, 7.09, 7.11, 8.00}
pitch_length = len(pitch_array)

start = 0
//save “pitch” for later
duration = 1.0
amplitude = 30000
squish = 1.0
decay = 1.25

for(iteration = 0; iteration < 1; iteration += 1){

    for(index = 0; index < pitch_length; index += 1){

```

---

<sup>41</sup> We’re using the STRUM2( ) instrument, which is a synthesized guitar that gives the user the ability to mimic the hardness of the plectra (“squish”) and a decay time. Because STRUM2( ) can interpret octave point pitch class data, there was no need to convert it to Hz using cpspch( ). Moreover, careful readers will see that the variables start and pitch were declared twice and changed, once in the list of p-field declarations and then in the loop. While this is redundant, it does give a good sense of the anatomy of STRUM2( )’s p-field elements before delving into it in the loop. You might at first list all of the p-field elements as variables regardless of later declarations as good practice, but then later abandon the practice as you better understand each instrument and its constituent elements.

```

    pitch = pitch_array[index]
    pan = pickrand(0.3, 0.7)
    STRUM2(start, duration, amplitude, pitch, squish, decay, pan)
    start = start + duration
  }

  for(index = index - 1; index >= 0; index -= 1){
    pitch = pitch_array[index]
    pan = pickrand(0.3, 0.7)
    STRUM2(start, duration, amplitude, pitch, squish, decay, pan)
    start = start + duration
  }
}

```

*Score file 12: Ascending and descending C major scale*

We can always use more than one loop in our score file and in the above example, we’re using a nested loop, or a loop within another loop. Here, the variable “start” isn’t being used both in the for( ) loop and as a p-field in the instrument, but instead we’re using something else. This is because we want to have one meta-loop that will initiate the events for the other loops found inside of it. Our meta-loop gives us the number of times, in total, that we want each of our inner loops to go through. Since I like to think of these as iterations, I used “iteration” as a variable for the meta-loop, which you’ll note only happens once.<sup>42</sup>

Within the meta-loop are two more for( ) loops, one to play our scale ascending, and the other, descending. Again, we don’t want to use “start” as our incremental variable in the loop, since we’ll be decrementing that value for the descending scale. Instead, we can use a variable called “index”, which determines the index value of our array, or the particular note being played. With all of this in mind, it becomes clearer how the value for index is being treated, even if we haven’t seen the second loop

---

<sup>42</sup> It’s common to encounter loops that abbreviate “iteration” for simply “i”. Thus, many programming examples that use loops will state for(i = 0; i < something;...).

structure, which will start at index value 7 and count down from there.<sup>43</sup> Here is a clearer picture of the value for index as we move from top to bottom in this loop.

in the first loop, index will equal: 0, 1, 2, 3, 4, 5, 6, 7  
through the second, it will equal: 7, 6, 5, 4, 3, 2, 1, 0

Now we can see how we're navigating our way through the loop, but what about that value for start times? Since we didn't include it as the incremental value in the loop, we need to work with it somewhere else, which we've done in the last line of each respective loop structure. Since "duration" equals one, start will equal itself plus one each time through. If we play our major scale both ascending and descending, can you figure out the corresponding values for start as we move along?<sup>44</sup>

We can also transpose our major scale script, in order to include all twelve keys.

```
rtsetparams(44100, 2)
load("STRUM2")

pitch_array = {7.00, 7.02, 7.04, 7.05, 7.07, 7.09, 7.11, 8.00}
pitch_length = len(pitch_array)

start = 0
duration = 0.125 // new duration, akin to sixteenth notes
amplitude = 20000
envelope = maketable("window", 1000, "hanning")
pitch = pitch_array[1]
squish = 1.0
decay = 1.25
```

---

<sup>43</sup> Here's another example where understanding the index value in an array is important. We needed to state `for(index = index - 1...)` in order to ensure that the starting value is 7. Check to see what happens when you say `index - 3`, or `index - 4` instead. Can you configure this script so that it doesn't repeat the C an octave above middle C?

<sup>44</sup> Here's a cool hint: Try placing `print(start)` inside each of the loops and RTcmix will print out that value for you. This can be a useful strategy for tracking values of variables in your scripts, especially if something is clipping or causing your score file to crash.

```

transpositions = {0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10, 0.11,
                  1.00}
num_transpositions = len(transpositions)

for(iteration = 0; iteration < num_transpositions; iteration += 1){

    for(index = 0; index < pitch_length; index += 1){
        pitch = pitch_array[index] + transpositions[iteration]
        pan = pickrand(0.3, 0.7)
        STRUM2(start, duration, amplitude*envelope, pitch, squish, decay,
               pan)
        start = start + duration //change durations here
    }

    // omit repeating top and bottom notes at the octave
    for(index = index - 2; index >= 0; index -= 1){
        pitch = pitch_array[index] + transpositions[iteration]
        pan = pickrand(0.3, 0.7)
        STRUM2(start, duration, amplitude*envelope, pitch, squish, decay,
               pan)
        start = start + duration
    }
}

```

### *Score file 13: Transpositions*

Data that is placed within any loop in an RTcmix script can also be subject to conditional tests, which are constructs that we find examples of in our everyday language. For example, we might commonly express to someone, “If it’s raining outside, then I’ll grab my umbrella.” In the world of RTcmix and MINC, we can represent this using an `if( )` statement.

```

if (it_is_raining){
    grab_the_umbrella
}

```

Statements such as these place the condition inside of parentheses and the

outcome of that test within curly braces. There is no limit to the number of outcomes that you might use, so these `if( )` statements can be quite powerful.

Even more exciting are logical tests, which enhance how we utilize our conditionals. For example, we might pose this question: “Choose a random integer between 0 and 10. If the random number is greater than or equal to 5, turn up my amplitude by 0.25, or else turn it down by 0.25.”

```
rand()  
  
x = rand(0,10)  
printf("the variable x equals: %f \n", x)  
  
amplitude = 0.75  
  
if(x >= 5){  
    amplitude = amplitude + 0.25  
}  
  
else{  
    amplitude = amplitude - 0.25  
}  
  
printf("amplitude now equals: %f \n", amplitude)45
```

#### *Score file 14: Conditional example<sup>46</sup>*

---

<sup>45</sup> `printf( )` is more of a C-style printing. You’ll see the results in your Terminal window.

<sup>46</sup> Important note about the use of `srand( )` in the opening line of this script. Each equation that returns a random number (think `irand( )`, `trand( )`, or `random( )`) needs to be seeded in order to generate truly new random numbers. If we didn’t include `srand( )` at the beginning of our script, our value for `x` would be the same each time we execute the score file. The same goes for seeding `srand( )` with something like `srand(12)`. When we include `srand( )` with empty parentheses, the seed value corresponds with the clock time of our computer, or how long it has been in operation since we turned it on. This ensures that all random values will be different each time we run the script. Try commenting out `srand( )` or filling it in with an integer and see what types of values you return for `x`. I find it useful, when composing sections of pieces that use random numbers, to seed my random number generator with `srand(1)`, listen to the result, then seed with `srand(2)`, listen to the result, etc. This way, I can perhaps find a best version of my script that I’d like to use in the work at hand.

Even though this seems like a rather vanilla conditional example, it has the elementary constructs of a random walk. Random walks are processes that either increment or decrement according to a given probability.

Let's use pitch as a variable we can subject to a random walk. We'll set up the random walk so that the pitch of our instrument will either increase by a half step or decrease by a half step equally. Think of it this way: A two sided coin has the equal probability of landing on either heads or tails, so if we flip a coin 100 times and take note of all the heads and all the tails, we can reasonably expect to have an outcome close to 50 heads and 50 tails. Similarly, we'll do a conditional test that reasonably ensures we increment 50% of the time and decrement 50% of the time.

```
rtsetparams(44100, 2)
load("WAVETABLE")
srand()

start = 0
duration = 0.25
amplitude = 18000
envelope = maketable("curve", 1000, 0,0,1, 0.1,1.0,-10, 0.33,0.5,-1, 0.8,0.25,-10,
1.0,0)
note = 60
pan = random()
waveform = maketable("wave", 1000, "tri")

increment = 0.0625
for(start = 0; start < 100; start += increment){
    WAVETABLE(start, duration, amplitude*envelope, cpsmidi(note), pan = random(),
waveform)

    x = random()
    if(x < 0.5){
        note += 1 //50% of the time, increase by a half step
    }

    if(x > 0.5){
        note -= 1 //50% of the time, decrease by a half step
    }
}
```



```
    }  
}
```

### *Score file 15: Random walk*

Using the conditions put forth in this score file, it's possible to change the probability of the outcomes. Because “x” is returning numbers from 0.0 to 1.0, it's useful to think of those as percentages, thus the 0.5 condition. However, if you'd rather have the melody that slightly favors and ascent, change that number to 0.52 or 0.55, remembering to correspondingly change the values for decrementing to 0.48 and 0.45, respectively.<sup>47</sup>

Conditions aren't limited to phrases such as “if x is less than” or “if x is greater than or equal to”. Remember our first example when we used “If it's raining, then...” as a conditional test? Well, what if we wanted to subject our outcome to more than one condition, such as “If it's sunny or if it's raining, then...” For this, we'll need to add some more symbols to our tool belt.

```
if(it_is_sunny || it_is_raining){  
    grab_the_umbrella  
}
```

Note how we had to use two parallel lines ( | | ) to complete our statement. This is another example of an operator that we can utilize for conditional tests in C.

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

---

<sup>47</sup> Or, create an if( ) else( ) statement as we did before.

`==`     is equal to<sup>48</sup>  
`!=`     is not equal to  
`&&`     and, such as `if(x > 10 && x <=20)`  
`||`     or, such as `if(x > 10 || x <= 20)`

We can also test to see if a number is even or odd, by checking to see if there is a remainder when a given number is divided by two. This works well for the following example (a personal favorite), which outlines the “wondrous number” algorithm laid out by Douglas Hofstadter.<sup>49</sup>

```

rtsetparams(44100, 2)
load("MMODALBAR")
srand()

/*
3n + 1 algorithm = a simple procedure for generating data
"The Wondrous Number algorithm"

Step 1: Take a random number.
Step 2: If the number is ODD, triple it and add one.
Step 3: If the number is EVEN, divide by two.

A wondrous number will divide itself all the way down to 1 and repeat over and over
and over.
*/

n = trand(100,127)
print(n)

for(i = 0; i < 5; i += 0.125){

    if(n%2 == 0){ //if n / 2 DOESN'T leave a remainder (is even...)

```

---

<sup>48</sup> When defining variables, use `=`, such as `alpha = 100`. In a conditional test, use `==`, such as asking `if(alpha == 100)`.

<sup>49</sup> The score file that follows is a realization of an algorithm found in his book *Gödel, Escher, Bach: An Eternal Golden Braid*. The book is absolutely phenomenal in every way and please, please take a chance to read it for yourself.

```

        n = trunc((n / 2)) //trunc() takes off any decimals
    }

    else if(n%2 == 1){ //if n / 2 DOES leave a remainder (is odd...)
        n = trunc(((n * 3) + 1))
    }

    print(n)

    MODALBAR(start = i, duration = 0.125, amplitude = 20000, frequency =
        cpsmidi(n), hardness = 1.0, position = 0.5, 0, pan = 0.5)
}

```

*Score file 16: Wondrous Number  $3n + 1$ <sup>50</sup>*

Understanding these principles of the MINC parser, which are so closely tied into the C programming language, will greatly enhance your experience with RTcmix. In fact, the concepts covered in this sonata form the essence of the program itself and separate it from so many proprietary software programs available to us today.

Most computer music audio workstation programs, be it Ableton Live, Reason, Pro Tools, Reaper, or Logic, contain plugin effects that modify your sounds. However, RTcmix affords us the ability to algorithmically design how we use those sounds and effects in time using a programming language, rather than the admittedly gorgeous (you have to pay for something, right?) user interface of the aforementioned programs. When we encounter those effects and sounds in syntax, we're more apt to understand and research the parameters that define them, all while sculpting and refining those sounds using unique processes. More than one student has remarked, after learning a bit of RTcmix, how much their experience with proprietary DAWs has been enhanced after realizing exactly how those GUI plugin effects are actually functioning. Brad Garton said it best and I couldn't agree more: When it comes to composing electronic and

---

<sup>50</sup> You'll want to run this script multiple times to get the full effect of the subtle differences, especially the difference between randomly starting with a wondrous number and not returning one.

computer music, "...I generally think the way RTcmix does."

Here is one last example score file that relies heavily on conditional tests. In it, I'm creating a drum machine from individual, mono drum set samples (bass drum, snare drum, hi hat, etc.) that are arranged in an array from low sounds to high sounds. From there, I'm using a variety of conditional tests to first choose from the outermost (extreme) elements of the array (so this should rely more heavily on bass drum and hihat sounds) and gradually shift toward choosing from the middle of the array (tom sounds and snare), eventually randomly selecting from any part of the array. We haven't looked at the JFUNCS( ) instrument (really a library of functions) yet, but I'm loading it in order to utilize map( ), which takes incoming numbers in a range of 0 to 1 and expands them to another range, specified by the user. In this case, I'm expanding 0 to 1 to the range of 0 to 5 to randomly select from the drum sounds in my array. Moreover, I need to use trunc( ) to take off the decimal point that will result from map( ), since array locations can't intrinsically be floating point numbers.<sup>51</sup>

```
rtsetparams(44100, 1)
load("DELAY")
load("JFUNCS")
srand()

bd = "/Users/jerod_s/Documents/Pd/drum-machine/samples/BD01.wav"
sd = "/Users/jerod_s/Documents/Pd/drum-machine/samples/SD01.wav"
tomlow = "/Users/jerod_s/Documents/Pd/drum-machine/samples/TOM05L.wav"
tomhigh = "/Users/jerod_s/Documents/Pd/drum-machine/samples/TOM09H.wav"
ride = "/Users/jerod_s/Documents/Pd/drum-machine/samples/RIDE1.wav"
hihat = "/Users/jerod_s/Documents/Pd/drum-machine/samples/CLHAT1.wav"

drums = {bd, tomlow, tomhigh, sd, ride, hihat}
numdrums = len(drums)

//drums arranged from low sounds to high sounds
```

---

<sup>51</sup> If I wanted to truly round these numbers, I'd need to do something like `index = (trunc(map(outcome, 0,1, 0,numdrums) + 0.5))`. In the example score file, I'm literally taking a number like 2.65 and making it 2, whereas if I included the 0.5, I'd be sure to get 3.

```

//begin by tending toward extremes, then middle, then all

increment = 0
for(start = 0; start < 50; start += increment) {
    x = random()
    y = random()

    if(start < 10){
        if(x > 0.75){
            outcome = x
            index = trunc(map(outcome, 0,1, 0,numdrums))
        }
        if(y < 0.25){
            outcome = y
            index = trunc(map(outcome, 0,1, 0,numdrums))
        }
    }
    else{
        index = pickrand(0,1, 4,5)
    }

    if(start > 10 && start < 20){
        outcome = ((x + y) / 2)
        index = trunc(map(outcome, 0,1, 0,numdrums))
    }

    if(start < 20){
        index = trand(numdrums)
    }

    file = drums[index]
    rtinput(file)
    DELAY(start, instart = 0, duration = DUR(), amplitude = 1.0, delaytime = DUR())
    increment = DUR()
}

```

*Score file 17: Random drum machine*

### Sonata III: Conditionals

Search the documentation for an instrument that hasn't been used or discussed thus far

and find as out as much as you can about it. Then, create a 1' score file that comments on each of the p-fields that constitute the instrument, generates values from an array, and uses at least three different conditional test to alter parameters.

## || Interlude III: RTcmix and Python ||

In the RTcmix world, MINC is sometimes referred to as a parser, front end, or “language”. Any of these terms refer to the fact that MINC code (which again, borrows from C) is the syntax used to create our score files. However, MINC isn’t the only language that we can utilize in the course of our work with RTcmix: Both the Perl and Python languages have been ported to RTcmix, so that we can compose our scripts using their constructs, rather than those of MINC. For some, this interlude is likely going to be completely useless, as they’ll be more than happy to continue working in MINC. For others who might already be familiar with either Perl or Python, this will be a fun interlude to look through. Because I enjoy using the Python programming language, this interlude will focus on some of the basics of the language and how it can be used in conjunction with RTcmix.

Are there advantages to using Perl or Python with RTcmix? Yes and no. Unlike MINC, Python is a so-called object-oriented programming language, which allows the user to define classes of “objects” that can be used throughout the program. Moreover, when using Python, we can import all of its distinct libraries to further refine our scripts.<sup>52</sup> However as of this writing, creating RTcmix scripts using a feature-rich language like Python isn’t supported when using the [rtcmix~] object in Pure Data or Max/MSP.

As noted in the installation process, it is possible to configure your build of RTcmix using Python, among other ancillary features. We’ll still save our score files with the suffix .sco — not .py, which denotes a Python file — and we can still play them in our Terminal using:

---

<sup>52</sup> The purpose of this interlude isn’t to discuss Python in full, so please don’t rely on it as a primer to the world of Python. Rather, we’re only going to be covering a very, very small amount of Python, with the mindset that we’re doing so in order to approach RTcmix from a new angle. For full on accounts of Python, I’m a fan of Zed Shaw’s *Learn Python the Hard Way* and John Zeele’s *Python Programming*. I can’t tell you the number of times I visited [stackoverflow.com](https://stackoverflow.com) to seek answers to my totally amateurish Python-related questions.

```
pycmix < /Path/to/your/file.sco
```

Our first step in a script that uses Python isn't to call on `rtsetparams( )`, but instead to write

```
from rtcmix import *
```

which will add all the elements of the “rtcmix” namespace to our script. Think of this as an RTcmix library working within Python, which ensures that Python knows what to do with program-specific tasks like `WAVETABLE( )` and `PANECHO( )` and `bus_config( )`, which obviously aren't native to Python itself. From there, we can call on `rtsetparams( )` like we did earlier, as well as `import( )` for loading instruments. Let's rewrite score file 9 in Python and compare its new syntax with MINC.

```
from rtcmix import *    # note that using “#” is the correct way to comment in Python
rtsetparams(44100, 2)
load("WAVETABLE")

amplitude = 10000
envelope = maketable("line", 1000, 0,0, 0.1,1.0, 0.8,1.0, 1.0,0)

increment = 1.0
for start in range (0, 50, increment):
    duration = trand(4,10)
    note = trand(60,72)
    pan = irand(0.0,1.0)
    WAVETABLE(start, duration, amplitude*envelope, cpsmidi(note), pan)
    increment = irand(0.25,4.0)
```

*Score file 18: Score file 9 in Python*

Because MINC intrinsically does away with the semicolons found at the end of each line of code in the C programming language and Python doesn't have any



semicolons at all, both scores look strikingly alike. I find Python to be an incredibly elegant way of writing code. In fact, the for loop structure omits the use of curly braces and instead relies on indentation to delineate processes that need to be executed in the loop itself. However, attempting to change score file 17 to increment by a floating point number will cause some headaches, as loops in Python are solely incremented by integer numbers.<sup>53</sup>

Here's a bit of code that will solve the issue for us.

```
def custom_range(start, stop, step_size):
    x = start
    while x < stop:
        yield x
        x+= step_size
```

The three constituent components of our object are a start value, stop value, and step size for incrementing. Let's say that we start at 0, have a value of 10 for stop, and increment by 0.5 each time through the loop.<sup>54</sup> Within our definition, we create an arbitrary variable, "x", that will equal start, since we need to declare start times for our

---

<sup>53</sup> This is one point where Python becomes powerful and customizable, as we can create our own custom loop that will increment by decimals. Imagine for a moment that we wanted to create a simple command that takes an arbitrary input number and always outputs the sum of that number and 15, similar to an addition object in Pure Data or Max/MSP that looks like [+ 15].

```
def sum_fifteen(input_number):
    return input_number + 15
```

We've created a definition for a class called "sum\_fifteen", which is a bit of code that we can use within our script to use at any time. We started with def, which needs a name and a set of values to process. In this case, our "sum\_fifteen" object is only in need of an input number, which we've named accordingly. Next, we use the return command to output the sum of our "input\_number" and 15.

```
y = sum_fifteen(10)
print y
```

It's safe to say that the value for "y" will be 25 and "sum\_fifteen" has done its job.

<sup>54</sup> Any guesses on how many times we'll run through this loop?

loop structures. Then, as long as “x” is less than “stop”, yield “x” and increment it by the “step\_size”. My hope is that the previous two sentences sound stunningly similar to the process for utilizing for( ) loops in MINC because they should. Note again the use of indentation and how it functions within “custom\_range”.

```
from rtcmix import *

rtsetparams(44100, 2)
load("WAVETABLE")

# here's one example of a Python library available for our use
import random
# in order to use an feature of the random library, use random.nameoffeature
# below, we'll use random.random, which works in the same way as MINCs random()

srand()

#----- WAVETABLE p-fields
start = 0
duration = 0.125
amplitude = 20000

# generate a list of MIDI notes, similar to filling an array

my_list = [] # declare the list of elements
num_elements = 100

for i in range(num_elements + 1):
    my_list.append(random.randrange(50,101,1)) # can you see how this line works?

print my_list

def custom_range(start, stop, step_size):
    x = start
    while x < stop:
        yield x
        x += step_size

increment = duration
index = 0
```

```

for start in custom_range(0, (num_elements / 8), increment):
    WAVETABLE(start, duration, amplitude, cpsmidi(my_list[index]), random.random())
    index += 1

```

*Score file 19: Python script with a list and class definition*

With MINC, we work with arrays of elements and in Python we work with lists. In score file 18, we are using a loop to fill a list of elements *my\_list* with random integers between 50 and 101, which are in turn being translated to MIDI note numbers to use as a random melody with WAVETABLE( ). Using the *append* feature allows us to add a new random element to our list each time we cycle through the loop. Moreover, because our duration values are at 0.125 and our incremental value for the loop is equal to that duration, we are hearing our melody in a sixteenth note pattern.<sup>55</sup>

Conditionals in Python are also similar in MINC, with the syntactical caveat that again we focus on indentation instead of curly braces. The next example score file doesn't make sound, but instead creates four note chords for use in a piano or pitched percussion part.

```

from rtmix import *

import random

```

---

<sup>55</sup> Regarding tempo: If we consider that an incremental value of 1 will intrinsically increment every second, it stands to reason that we're working with a metronome value of quarter note = 60. Thus, values of 1 will sound — in terms of quarter = 60 — like quarter notes, 0.5 as eighth notes, 0.125 as sixteenth notes, etc. In order to convert values in seconds to beats per minute, you'll want to set up a definition that looks something like this

```

def sec_to_bpm(input):
    return 60 / input

```

knowing that you can change the 60 to 60000 if you're more comfortable thinking of time values in milliseconds, as is the case with Pd or Max/MSP.

```

from random import randrange
random.seed()

lowest_note = 45    # In MIDI
highest_note = 96
mode = highest_note - lowest_note # Think back to mean, median, and mode. What is mode
again?

s = a = t = b = mode    # Have each voice (soprano, alto, tenor, bass) start on mode
value
# In the loop, we'll watch each voice move inward and outward

for i in range(0,10):
    x = random.randrange(10) #random number to use for conditional tests

    if x < 5:
        s += 1
    else:
        s -= 1

    if x < 2 or x > 8:
        a += 1
    else:
        a -= 1

    if x < 7:
        b -= 1
    else:
        b += 1

    if s - a > 12 or t - b > 12: # this is quite slick, writing "or" instead of ||
        s = s - 12
        t = t - 12

    gamut_in_range = [b, t, a, s]
    print "gamut in spread range: %s" % gamut_in_range

```

### *Score file 20: Making chords from conditionals*

Essentially, each respective voice is being subjected to its own random walk, going up or down by half steps according to the returned value for x. Moreover, the final

if statement ensures that the soprano voice will always be within an octave of the alto voice and that the tenor will always be within an octave above the bass voice. This ensures that future pianists or vibraphone players who play our gorgeous random chord collection won't have to spread their fingers or mallets too far to reach the notes.

One very important advantage of Python over MINC is the ability to create *callable objects*. They are routine tasks that you can define and call on at any point in your score file after you've created them. For example, let's say that we want to create a *function* that returns a number squared, such that number  $x$  will equal  $x$  times  $x$ .

```
def square(x):  
    return (x * x)
```

At any point in our score file, we can now call upon our new function, *square*, by writing it with a corresponding element to evaluate in the same way that we use commands like `abs( )`, `round( )`, `trunc( )`, or `wrap( )` in MINC.

```
element_squared = square(4)
```

We could create another function to evaluate the square root of a given element, but this would also be a pretty routine task and one that we'd expect to have available to us at any time. Thankfully, a host of exciting features await those who import Python's math library, just as we imported elements from its random library earlier.

Python is incredibly adept at treating lists, which is a powerful tool when we consider pitch transformations later on in the book. Let's fill a list with six random elements that we will use as pitches.

```
spray_table = 1  
spray_size = 6  
seed = 12
```

```

aggregate = []
spray_init(spray_table, spray_size, seed)    #spray_init is an RTcmix-intrinsic command
for i in range(0,spray_size):
    aggregate.append(get_spray(spray_table))

print(aggregate)

```

The `spray_init( )` command comes from MINC and returns unrepeated random numbers from 0 up to the range specified. In this case, we’re getting random numbers from 0 to 5 that we’re using to fill our aggregate list. We can transpose them into a meaningful range, such as MIDI, by creating our own transposing object.

```

def list_to_midi(pitches, range):
    return [x + range for x in pitches]

series = list_to_midi(aggregate, 60)
print(series)

```

Note that we are first declaring two arguments. The first is the set of pitches that we’d like to transform — in this case, our *aggregate* list — and a *range* to transpose them in to, such as 60 for the range starting above middle C. From there, we’re returning each individual *x* value plus 60 when we finally utilize our object to create a new list, called *series*.

For the majority of our work, we’ll be using MINC. However, for those excited about and interested in the world of Python itself and how it can be used in conjunction with RTcmix, I highly encourage you to write the remaining examples in both languages. To end this interlude, we’ll explore a complex Python example and it will be your job to research each of the lines in question and how they are working.<sup>56</sup>

```

from rtcmix import *

```

---

<sup>56</sup> This example will come up again when we explore twelve-tone transformations in Interlude VI, but it’s presented here as a teaser for those who are interested right now. There is an example of “list slicing” in it, which we will cover later on, but you are welcome to seek out if you’d like to.

```

import random

rtsetparams(44100, 2)
load("MMODALBAR")

#----- MMODALBAR
start = 0
duration = 1
amplitude = 30000

#---pitch stuff
spray_table = 1
spray_size = 12
seed = 12

aggregate = []
spray_init(spray_table, spray_size, seed)
for i in range(0,spray_size):
    aggregate.append(get_spray(spray_table))

def retrograde(pitches):
    return pitches[::-1] # Search "list slicing in Python" for an explanation

def invert(pitches):
    return [(x - 12) * -1 for x in pitches]

def retrograde_invert(pitches):
    return invert(retrograde(pitches))

def series_to_midi(pitches, range):
    return [x + range for x in pitches]

p_0 = series_to_midi(aggregate, 32)
r_0 = series_to_midi(retrograde(aggregate), 48)
i_0 = series_to_midi(invert(aggregate), 60)
ri_0 = series_to_midi(retrograde_invert(aggregate), 72)

hardness = 1.0
position = 1.0
instrument = 4
pan = 0.5

# series of chords

```

```

for start in range(0,spray_size):
    MMODALBAR(start, duration, amplitude, cpsmidi(p_0[start]), hardness, position,
              instrument, pan)
    MMODALBAR(start, duration, amplitude, cpsmidi(r_0[start]), hardness, position,
              instrument, pan)
    MMODALBAR(start, duration, amplitude, cpsmidi(i_0[start]), hardness, position,
              instrument, pan)
    MMODALBAR(start, duration, amplitude, cpsmidi(ri_0[start]), hardness, position,
              instrument, pan)

```

### *Score file 21: Dodecaphonic transformations in Python*

I find that in our era of unparalleled access to information, it is incredibly useful to seek out answers using known resources. From experience, I can say that learning the basics of Python by rewriting example scripts from others and using comments to help me better understand the functions of the program was a very enriching experience.

I'm also a proponent of immersive learning styles and building skills incrementally. If even twenty minutes per day are spent rewriting known RTcmix files into Python and taking the time to search concepts that you either haven't encountered before or would like to know more about, you'll soon find a sense of facility with your work and a more enlightening experience using RTcmix while composing your work.

### **Interlude III: Python and RTcmix**

Rewrite score file 20, researching each command that is foreign to you, and use comments to note how those commands and concepts are functioning in the script. This will be especially useful if you are unaware of 12-tone music (dodecaphony) and how aggregate row forms are constructed and how inversions, retrograde, and retrograde-inversion transformations relate to the aggregate.

As a second part of your interlude, rewrite a previous score in MINC — either from the



book or one that you've composed on your own — into Python.

## || Sonata IV: Adding various effects to your scripts||

Much like a chain of guitar pedals, RTcmix is capable of sending your processed sounds into any of a number of effects to further enhance and refine your musical ideas. Not only does RTcmix contain the standard set of DSP effects — such as delay, various types of filters, EQ, flange, or chorus — but it can dynamically update those effects in real-time, which again yields endless possibilities during the composition process.

We'll begin by sampling a standard audio file and passing it through an effect via the `bus_config( )` command.

```
rtsetparams(44100, 2)
load("STEREO") //simple stereo sampling, must be .aiff or .wav!

rtinput("/path/to/your/file.aiff") //remember to point to a file on your computer

start = 0
instart = 0 //input start time
duration = DUR()
amplitude = 1.0 //relative, not absolute
pan = 0.5
STEREO(start, instart, duration, amplitude, pan)
```

### *Score file 22: Simple stereo playback of a sound file*

With `STEREO( )`, there are two p-field values for start times. The first is the overall start time, which we are familiar with, and designates when to onset the `STEREO( )` instrument itself. Again, we want the instrument to start right way, so we choose a start time of 0. Now, `STEREO( )` also includes an input start time, or at what point within the audio file to begin playback. For example, if we are working with a 30" sound file and set an input start time of 15, `STEREO( )` will play the sound file starting at 15", and then play through to the end. The `DUR( )` command returns the overall

duration of the audio file, so if we are working with a 30" clip, DUR( ) would return a value of 30.<sup>57</sup> P-field values for amplitude and panning are dynamic, so those can be updated with a table for more exploration.

As a first example, we'll pass the audio from STEREO( ) to a REV( ) instrument using the bus\_config( ) command to add a little bit of reverb to our sound.<sup>58</sup>

```
rtsetparams(44100, 2)
load("STEREO")
load("REV") //reverb instrument
rtinput("/path/to/your/file.aiff") //remember to point to a file on your computer

bus_config("STEREO", "aux 0-1 out") //send audio through an auxiliary line
start = 0
inststart = 0 //input start time
duration = DUR()
amplitude = 1.0 //relative, not absolute
pan = 0.5
STEREO(start, inststart, duration, amplitude, pan)

bus_config("REV", "aux 0-1 in", "out 0-1") //receive audio from aux line 0-1 and send
out
type = 1 // 1 is Perry Cook's, 2 is John Chowning's, 3 is Michael McNabb's
rvbtime = 2.5
rvbpct = 0.5
inchan = 0
REV(start, inststart, duration + rvbtime, amplitude, type, rvbtime, rvbpct, inchan)
```

---

<sup>57</sup> If we did want to use an input start time of 15, it's good practice to use

```
duration = DUR() - inststart
```

to ensure that STEREO( ) will only play for the specified amount of time that you are utilizing within the audio file itself.

<sup>58</sup> RTcmix also includes a Schroeder reverb instrument, REVERBIT( ) and two other reverb instruments worth mentioning at this point are FREEVERB( ) and GVERB( ), which can produce very long and very smooth-sounding reverberations. RTcmix also includes some room-simulation instruments that are a bit more complicated to use, but of course the sample scorefiles in the RTcmix package will help you utilize them correctly. They are ROOM( ) and PLACE( ), which have been updated with enhancements in the form of MROOM( ) and MPLACE( ).

*Score file 23: Adding reverb with bus\_config( )*

Let's have a bit of fun with our audio file before passing it to a delay instrument. We'll first put it through a loop that will pick random start times within the file and play them for 0.25 seconds, which corresponds with our increment value. Then using DEL1( ) — just one of RTcmix's delay instruments — we'll be able to very succinctly take a copy of the sample in question and play it again at a (you guessed it) random amount of time.

```
rtsetparams(44100, 2)
load("STEREO")
load("DEL1") //delay instrument
rtinput("/path/to/your/file.aiff")

bus_config("STEREO", "aux 0-1 out")

increment = 0.25
for(start = 0; start < 100; start += increment){
    instart = irand(0,DUR())
    duration = increment
    amplitude = irand(0,1.0)
    pan = 0.5
    STEREO(start, instart, duration, amplitude, pan)
}

bus_config("DEL1", "aux 0-1 in", "out 0-1")
start = 0
instart = 0
delay_duration = 30
amplitude = 1.0
envelope = maketable("line", 1000, 0,0, 0.1,1.0, 0.9,1.0, 1.0,0)
delay_time = maketable("random", 10.0, "triangle", 1.0, 8.0)
r_ch_amplitude = 1.0 // right channel relative to left channel
DEL1(start, instart, delay_duration, amplitude*envelope, delay_time, r_ch_amplitude)
```

*Score file 24: Simple delay instrument*

The DEL1( ) instrument processes a delayed copy of the original sound file (or input audio source) at a specified delay time in p-field 6, where we've opted to randomize the process using a table of random numbers between 1.0 and 8.0 (seconds) and processes it in the right channel relative to the original single in the left channel. The result is a 30 second excerpt of randomly-derived moments of your sound source, panned across the stereo field according to the makeLFO( ) command. There are other delay instruments in RTcmix, including DELAY( ), which processes delayed copies of your signals with a specified feedback parameter. This will afford you to ability create ping-pong style delay lines (think of the phenomenon of the sound of a ping pong ball falling toward a table, having enough energy to spring back up, then down, then up, then down, ever so slightly less and less) and other types of delays according to your own algorithmic procedures.<sup>59</sup>

Let's shift focus on to some of the so-called short-term delay effects, such as echo, doubling, chorus<sup>60</sup>, and flange, all of which are available to us in the RTcmix library of instruments.<sup>61</sup> Each of these effects involve some type of delayed copy of an input source and differ in the amount of overlap or time processed. Let's first look at PANECHO( ), RTcmix's echo instrument, which takes input audio and not only provides echo, but pans it back and forth across the stereo field in the aforementioned "ping pong" style.

```
rtsetparams(44100, 2)
load("PANECHO")    // an echo instrument that includes panning

rtinput("/path/to/file.aiff")
bus_config("PANECHO", "in 0-1", "out 0-1")
```

---

<sup>59</sup> See also JDELAY( ), which is similar to DELAY( ), but provides a p-field for wet/dry mix, DC block, and a low pass filter.

<sup>60</sup> RTcmix doesn't have a standalone chorus instrument per se, but does include John Gibson's lovely JCHOR( ), which is a granulated chorus instrument that combines the older, CMIX CHOR( ) instrument by Paul Lansky, in conjunction with Doug Scott's TRANS( ).

<sup>61</sup> We could include reverb as another short-term delay effect, but we took a look at it earlier.

```
//-----ping-pong delay
start = 0
instart = 0
duration = 60
amplitude = 1.0
envelope = maketable("window", 1000, "hanning")
channel0delay = 0.25 // in sec
channel1delay = 0.50
delay_feedback = 0.8 // ALWAYS less than 1.0 or a crazy feedback loop ensues!
ringdowndur = 1.0

PANECHO(start, instart, duration, amplitude*envelope, channel0delay, channel1delay,
delay_feedback, ringdowndur)
```

*Score file 25: Echo with panning via PANECHO( )*

Echo is itself a process that delays the signal by about 40 milliseconds or longer and one super important concept to keep in mind with PANECHO( ) is the p-field for feedback, which can get out of hand quickly if it's accidentally set above 1.0. You'll note that the echo plays back in the left channel at a delay time of 0.25 seconds and in the right at 0.5 seconds, which — along with the feedback time — can be updated using dynamic tables.

If a signal is played back with an exact copy of itself that is delayed anywhere from fifteen to 40 thousandths of a second, the overall effect will be a doubling of the original signal. You can verify this phenomena by first playing a file back using STEREO( ) and seeing its resulting maximum amplitude in the Terminal window, then doubling the signal with another STEREO( ) call that is delayed by, say, 0.004 seconds and seeing its resulting maximum amplitude.

```
rtsetparams(44100, 2)
load("STEREO")
rtinput("/path/to/file.wav")

start = 0
```

```

instart1 = 0
instart2 = 0.004
duration = DUR()
amplitude = 1.0
pan = 0.5
STEREO(start, instart1, duration, amplitude, pan)
STEREO(start, instart2, duration, amplitude, pan)

```

*Score file 26: Doubling a signal using STEREO( ) and requisite delay*

While doubled signals are of course great to use, RTcmix also includes a wonderful instrument called HOLO( ), which produces a “Carver Sonic Hologram” generator that widens the stereo field. What we’re talking about here with doubling and HOLO( ) is really ensuring that — much like normalization— we’re using the most robust, cleanest, clearest audio source file in our compositions, which can in turn be processed later. Let’s first use HOLO( ) to widen our original file and then send it to the flange instrument.

```

rtsetparams(44100, 2)
load("HOL0") //phase cancellation
load("FLANGE")

rtinput("/path/to/file.wav")

bus_config("HOL0", "in 0-1", "aux 0-1 out")
start = 0
instart = 0
duration = DUR()
signal_amplitude = 0.5 //amplitude of original audio
processed_amplitude = 0.75 //amplitude of "hologram" of signal
HOLO(start, instart, duration, signal_amplitude, processed_amplitude)

bus_config("FLANGE", "aux 0-1 in", "out 0-1")
amplitude = 1.0
resonance = irand(0,1) //percentage, 0-1
maxdelay = 1 / cpspch(9.02) //usually determined as 1 / cpspch(octave.pitch)
moddepth = 25 //percentage, 0-100

```

```
modrate = makeLFO("sine", 10.25, 1.0, 5.0) //in Hz, fun to use an LFO for this
wetdrymix = 0.8 //percentage, 0-1
FLANGE(start, instart, duration, amplitude, resonance, maxdelay, moddepth, modrate,
        wetdrymix)
```

*Score file 27: Stereo hologram of sound with FLANGE( )*

The above score file works really well when processing sounds of speech or other audio sources that have a particularly noisy spectrum, as the flanger's periodic shifting causes cancellation and reinforcement of certain frequencies of your source audio.

There are many, many more instruments and effects to explore in conjunction with `bus_config( )`. While this Sonata explored short term delay effects in detail, the following interlude will focus specifically on subtractive synthesis and digital filters, all while introducing more complex types of chained connections for further sculpting and refinement of your sounds.

#### **SONATA IV: A brief RTcmix etude**

Using at least three delay effects covered in this Sonata, as well as at least two more RTcmix effects that you discover on your own via the online documentation, create a two to three minute etude using chained effects via `bus_config( )`. See how many “aux in” and “aux out” connections you can make and supplement your etude with a brief paragraph or two detailing your compositional process as you wrote your work.



## **|| Interlude IV: A detailed look at bus\_config( ) and connections ||**

In order to begin understanding the connection of instruments in RTcmix in detail, we'll need to begin by first exploring one of the oldest CMIX instruments that is still in use with the program, and little by little begin building complex chains of connections using some of the more recent additions to the suite of RTcmix tools.

As alluded to in the Prelude, Paul Lansky's work with CMIX more or less boiled down to creating a compositional tool that would afford the user the ability to mix together various sound files using the C programming language. This early instrument still exists in RTcmix and you'll be able to use it without having to first load it using load( ) as we've become accustomed to thus far. That is because MIX( ) remains a critical piece of the RTcmix, and as the online documentation states, MIX( ) is just "always there. Always."

We've already seen MIX( ) in action when we used STEREO( ) earlier. This is because while MIX( ) has existed from the beginning, STEREO( ) has more or less taken over as the main tool for audio file sampling in RTcmix. That isn't to say, however, that we can't still use MIX( ), but in order to do so, we need to get used to its channel matrix. In essence, using either of these two instruments gets to the heart of connections in RTcmix, namely, taking an audio file in (by loading it) and sending it out to your sound card.

```
rtsetparams(44100, 2)
rtinput("/Users/jerod_s/Desktop/bwv20.aif")

duration = DUR()
amplitude1 = makeLF0("sine", 0.3, 0.25,1)
MIX(start = 0, instart = 0, duration, amplitude1, 0, 0)

amplitude2 = makeLF0("sine", 0.4, 1,0.25)
MIX(start = 1, instart = 0, duration, amplitude2, 1, 1)
```

*Score file 28: MIX( ) with amplitudes in and out of phase*

The above score file takes one input file<sup>62</sup> and mixes it between the left and right channels of the stereo field, delaying one copy of the sound file by one second (the result is something of a canon or *caccia*) and introduces a phasing effect to each respective channel's amplitude via `makeLFO( )`. The last two p-fields for `MIX( )` can be confusing for multichannel diffusion<sup>63</sup>, but for stereo output is pretty straightforward: The last two p-fields following the p-field for amplitude delineate the input channel and output channel. So, in the first call to `MIX( )`, I'm stating that I'd like to have input channel 0 (from "bwv20.aif") sent to output channel 0. That's pretty simple to understand, and you can deduce what 1,1 means in the last two p-fields for the second `MIX( )` call. One particularly useful strategy for `MIX( )` is to use -1 as an output "destination" in order to mute that channel.

Each time you send a sound out of an auxiliary line to another instrument, you are losing its original constructs, which you might want to use at another point in your score file. For example, think about a call to `WAVETABLE( )` that is being sent via "aux 0-1 out" to `FLANGE( )`, which is then sent out to your sound card. Your original waveform or whatever cool processes that you were doing with `WAVETABLE( )` alone will forever be subject to flanging, and we'll cease to hear the original waveforms themselves. There may be times when you might want to hear both the original sounds and their flanged results at the same time, but we can't send a sound out to two places

---

<sup>62</sup> The "bwv20.aif" file features my two amazing sight singing classes from the Crane School of Music in the 2014-2015 academic year singing the final chorale from Bach's "O Ewigkeit, du Donnerwort" in solfege. It's a personal favorite and is included as a companion file to this text for your own processing.

<sup>63</sup> Most readers are probably going to be using `RTcmix` in stereo, however, there are a number of instruments built for surround sound diffusion. In particular, `QPAN( )` is a fantastic quadraphonic panning instrument that utilized (x,y) coordinates to diffuse your sound source across the surrounding audio field. `NPAN( )` is similar, but in addition to the Cartesian placement that `QPAN( )` uses, you can also move sounds around, circular, in degrees, which is really helpful for 8-channel works. With either of these instruments, you'll need to of course call `rtsetparams(44100, 8)` or something similar to reflect the number of speakers that you are utilizing in your mix. Your terminal readout will reflect this with peak amplitudes for channels 0, 1, 2, 3, 4, etc.

simultaneously using `bus_config( )`.

```
bus_config("WAVETABLE", "aux 0-1 out", "out 0-1") //ERROR!
```

To mitigate this dilemma, we can use `SPLITTER( )`, which affords us the ability to circumvent the problem of the “one in and one out” methodology of `bus_config( )`. In essence, `SPLITTER( )` takes inputs and sends them out to multiple outputs, with the exception that you won’t be able to output to both an auxiliary out and out to your sound card.

```
bus_config("SPLITTER", "aux 0-1 in", "aux 2-3 out", "aux 5 out", "aux 12-13 out")// OK
```

```
bus_config("SPLITTER", "aux 0-1 in", "aux 2 out", "aux 7-8 out", "out 0-1")// NOT OK
```

This is a powerful way to control a large number of effects in your score file. Here’s an example of this technique in action, which sends `FMINST( )` to both a main output and via an aux send to `SHAPE( )`.

```
rtsetparams(44100, 2)
load("FMINST")
load("SHAPE")
load("SPLITTER")
srand()

// This is how we'll route our sounds
bus_config("FMINST", "aux 0-1 out")

bus_config("SPLITTER", "aux 0-1 in", "out 0-1")
SPLITTER(start = 0, instart = 0, duration = 60, amplitude = 1.0, input_channel = 0,
          amp0 = 1.0, amp1 = 1.0)

bus_config("SPLITTER", "aux 0-1 in", "aux 2-3 out")
SPLITTER(start, instart, duration, amplitude, input_channel, amp0, amp1)

bus_config("SHAPE", "aux 2-3 in", "out 0-1")
```

```
//-----FMINST
amplitude = 3000
envelope = maketable("curve", 1000, 0,0,0, 0.2,0.5,10.0, 0.7,0.01,10.0, 0.8,0)
carrier = 7.02
modulator = maketable("random", 3, "even", 440, 550)
min_index = trand(1,10)
max_index = 20 - min_index
pan = makeLFO("sine", 10.125, 1,0)
waveform = maketable("wave", 8, "sine")
guide = maketable("cheby", "nonorm", 10, 0.9, 1.0, 1.0, 0.3, -0.2, 0.6, -0.7, 0.9,
                -0.1, 0.0, -0.25, 1.0, 0.25, -0.125)

FMINST(start, duration, amplitude*envelope, cpspch(carrier), modulator, min_index,
max_index, pan, waveform, guide)

//-----SHAPE
min_distortion = 0.5
max_distortion = 1.0
normalization = 0
input_channel = 0
pan = makeLFO("sine", 0.125, 0,1)
transfer_function = maketable("cheby", "nonorm", 1000, 0.4, -0.25, 0.7, 0.6, 0.7, 0.9)

SHAPE(start = 0, instart = 0, duration, amplitude = 1.0, min_distortion,
max_distortion, normalization, input_channel, pan, transfer_function)
```

*Score file 29: Multiple outputs with SPLITTER( )*

You'll note that both FMINST( ) and SHAPE( ) are utilizing makeLFO( ) for panning, albeit with one moving faster than the other. This is to highlight the fact that SPLITTER( ) is doing its job and not only taking the frequency modulation sounds directly to our main output, but also sending them for further sculpting. SHAPE( ) is a waveshaping instrument, which is an example of distortion synthesis that utilizes a so-called nonlinear transfer function to "distort" the incoming signal. This will reshape the incoming signal by adding the presence of new frequencies to its spectrum. One of my

favorite transfer functions uses Chebyshev polynomials<sup>64</sup>, which are defined as “cosine curves with a somewhat disturbed horizontal scale, but the vertical scale has not been touched.”<sup>65</sup>

Recall Score File 17 for a moment. When you heard it, you probably remember that because we called `rtsetparams(44100, 1)` at the top of our score file — due to the fact that we were working with mono audio files — we only heard playback in one of our speakers. Getting that mono input to output to stereo is another useful strategy for `SPLITTER( )`. At the bottom of the loop in Score File 17, simply add the following after line 49 (after, of course ensuring that we’ve called on `rtsetparams( )` for stereo output).

```
bus_config("DELAY", "in 0", "aux 0 out")
DELAY(start, instart = 0, duration = DUR(), amplitude = 1.0, delaytime = DUR())

bus_config("SPLITTER", "aux 0 in", "out 0-1")
SPLITTER(start, instart, duration, amplitude, input_channel = 0, amp0 = 1.0,
          amp1 = 1.0)
```

Now let’s go back to using an audio sound sample and we’ll then pass it to a variety of subtractive synthesis instruments, or filters.<sup>66</sup> One of the more important aspects of audio sculpting in electroacoustic music is the use of digital audio filters and equalization to better refine the amplitudes of frequencies across the audio spectrum in our compositions. RTemix boasts a large number of instruments for successfully realizing filtering in your work and we’ll start with the basics.

---

<sup>64</sup> In the case of `bus_config(“cheby”)` we’re stating that we don’t want the values of our table to be normalized, that we want 1000 points on our graph, and that we want to perform the polynomial equation at index 0.4. The remaining p-fields denote the “relative strength” of the harmonics at the given index, whose values can be negative. All of this is to say that we have a way to somewhat determine the harmonic spectrum of the signal in the waveshaping process.

<sup>65</sup> This is a direct quote from Forman S. Acton’s book *Numerical Methods that Work*.

<sup>66</sup> We’re going to spend a lot more time taking a look at subtractive synthesis in Interlude V shortly.

Of the many varieties of digital audio filters, there are two whose names are pretty self-explanatory: Low pass filters will allow frequencies to pass below a specified cutoff frequency and high pass filters will do the same, albeit passing frequencies above the cutoff. The signal that is allowed to pass is called the passband and the signal that is filtered out is called the stopband. Those of you familiar with Pure Data will probably be thinking about [lop~] and [hip~] objects and how they are used, however; RTcmix doesn't differentiate between low and high pass filters as separate instruments, per se, so instead we'll need to define specific parameters in one instrument to get the job done.

The first filter instrument that we'll explore is ELL( ), which is an elliptical filter. Elliptical filters have sharply defined passbands and stop bands (fast cutoffs), with a certain amount of "ripple" present at each band.<sup>67</sup> This means that the cutoff frequency that you declare will be pretty well defined and steep in comparison to some other filters that tend to rolloff a bit more.<sup>68</sup>

In the following scorefile, we'll take an input sound and send the left channel to a low pass filter and the right to a high pass filter with randomly derived cutoff frequencies.

```
rtsetparams(44100, 2)
load("STEREO")
load("ELL")

rtinput("/Users/jerod_s/Desktop/bwv20.aif")

//some global variables
start = 0
instart = 0
```

---

<sup>67</sup> Right now, we're at one of those points where this could easily delve more into a detailed description of filter design that again has already been explained way, way better in other introductions to computer music. I would, however, recommend the Wikipedia entry for digital filters ([https://en.wikipedia.org/wiki/Filter\\_\(signal\\_processing\)](https://en.wikipedia.org/wiki/Filter_(signal_processing))), which has some nice looking graphs for elliptical filters, as well as Butterworth filters, which we'll explore later on.

<sup>68</sup> Graphs for elliptical filters look like vertical walls while others look like gently rolling hills. There are many parameters to define the shape and contour of your filter, no matter the type that you are utilizing.

```

duration = DUR()
amplitude = 1.0

//read in our audio file and send to an aux line
bus_config("STEREO", "in 0-1", "aux 0-1 out")
STEREO(start, instart, duration, amplitude, pan = 0.5)

// lowpass filter, creating a stopband at around 100 Hz
passbandcutoff = 499 //in Hz
stopbandcutoff = 501
p2 = 0 // this p-field is always 0 for low and highpass filters
ripple = .8 // amount of ripple (0.0 - 1.0)
attenuation = 90 // in dB (higher the number, the steeper the filter)
ellset(passbandcutoff, stopbandcutoff, p2, ripple, attenuation)

ringduration = 1.0 // in seconds (a slight ringing for sound to fade out)

bus_config("ELL", "aux 0 in", "out 0")
ELL(start, instart, duration, amplitude, ringduration)

// highpass filter, creating a stopband at around 1000 Hz
passbandcutoff = 1001
stopbandcutoff = 999
p2 = 0
ripple = .8
attenuation = 90
ellset(passbandcutoff, stopbandcutoff, p2, ripple, attenuation)

bus_config("ELL", "aux 1 in", "out 1")
ELL(start, instart, duration, amplitude = 2.0, ringduration)

```

### *Score file 30: Low and high pass filters*

Careful copiers of the above code will hopefully get to the last line and notice that I've asked for an amplitude value of 2.0! This will happen from time to time in your score files that utilize filtering because we've filtered out quite a bit of the audio spectrum, so there'll be times like this where we might need to cook the amplitude to compensate for what might be a pretty quiet portion of your audio file.

The two most important aspects of ELL( ) and its design are the p-fields for the

passband and stopband, which go to a subcommand that is unique to ELL( ) called ellset( ). Note that in order to use a lowpass filter, the p-field for the stopband's cutoff frequency needs to be higher than that of the passband and vice versa for creating a highpass filter.<sup>69</sup> Moreover, these p-fields can be controlled using maketable( ) in order to sweep along different frequencies in real time.

At this point, we have two separate, pretty clearly demarcated ideas going on. During the composition process, you might be asking yourself, "But what more can I do with this? It's a good idea to split the high and low pass filtered sounds between the different stereo channels, but what if I wanted to move those around a bit? Could each separate sound pan in an interesting way, more or less interacting with the other sound?"

The answer to all of this or really any ideas you come up with during your composing time is of course, yes, but it's just a matter of identifying the problem at hand and getting a grasp on how to implement it using the program at your disposal. Since bus\_config( ) is pretty limitless<sup>70</sup>, we can continue sending our sounds out of it to more aux lines and we can introduce the PAN( ) instrument to realize our goal here.

```
rtsetparams(44100, 2)
load("STEREO")
load("PAN")
load("ELL")

rtinput("/Users/jerod_s/Desktop/bwv20.aif")

start = 0
instart = 0
duration = DUR()
amplitude = 1.0
```

---

<sup>69</sup> ELL( ) will also be able to generate a bandpass filter, which has some variation in its p-fields for ellset( ) and designates an entire band of frequencies to pass through, with stopbands both above and below the passband. Looking at a bandpass filter on a graph sort of looks like a small hill that you can control the steepness of (the bandpass, or, the hill).

<sup>70</sup> You'd have to have a pretty crazy amount of aux lines going to really test the limit of bus\_config( ).



```

bus_config("STEREO", "in 0-1", "aux 0-1 out")
STEREO(start, instart, duration, amplitude, pan = 0.5)

passbandcutoff = 499
stopbandcutoff = 501
p2 = 0
ripple = .8
attenuation = 90
ellset(passbandcutoff, stopbandcutoff, p2, ripple, attenuation)

ringduration = 1.0

bus_config("ELL", "aux 0 in", "aux 2 out")
inputchannel = 0
ELL(start, instart, duration, amplitude, ringduration)

pan1 = maketable("line", 10, 0,0, 0.5,1, 1.0,0)
bus_config("PAN", "aux 2 in", "out 0-1")
PAN(start, instart, duration, amplitude, inputchannel, panmode = 1, pan1)

passbandcutoff = 1001
stopbandcutoff = 999
p2 = 0
ripple = .8
attenuation = 90
ellset(passbandcutoff, stopbandcutoff, p2, ripple, attenuation)

bus_config("ELL", "aux 1 in", "aux 3 out")
ELL(start, instart, duration, amplitude = 2.0, ringduration)

bus_config("PAN", "aux 3 in", "out 0-1")
pan2 = maketable("line", 10, 0,1, 0.5,0, 1.0,0)
PAN(start, instart, duration, amplitude, inputchannel, panmode = 0, pan2)

```

### *Score file 31: Sweepable panning*

I've found that using PAN( ) as a final instrument call in my score files will ensure that I'm completely in control of how I'm shaping and sending my sounds in space.

It was alluded to in an earlier footnote, but there are a multitude of instruments that are capable for realizing surround sound directly in RTcmix. That isn't to say, however, that you won't be able to export your sounds as audio files using `rtoutput( )`<sup>71</sup> to bring into your digital audio workstation of choice for not only further refinement, but also routing to your surround mix.

```
rtoutput("/path/to/your/file.aiff") // save as .aiff (or .wav) file72
```

If you are committed to rerunning your score file that includes saving to an audio file via `rtoutput( )`, you'll need to call on the command `set_option( )`.

```
rtoutput("/path/to/your/file.wav")
set_option("clobber = on") // "clobber = on" will overwrite files.
```

There are a number of RTcmix options that can be turned on or off using `set_option( )` and the online documentation will walk you through all of them. Suffice it to say that there are quite a few ways to interact with RTcmix and you are free to customize that experience as much as you'd like to. For example, if I'm running a score with quite a few iterations through my loops or nested loops, I'll often use the command `print_off( )` to prevent the long sequence of commands and parameters that I'm running from printing to the Terminal. This will keep your Terminal window clean, but won't save any time for score file rendering, unfortunately.<sup>73</sup>

With a wide array of instruments at its disposal, RTcmix won't disappoint when it comes to completely customizing and designing sound. With a myriad of ways to

---

<sup>71</sup> Which will output the requisite number of channels based on `rtsetparams( )`.

<sup>72</sup> Don't forget that simply writing `rtoutput("myfile.aiff")` will save the file to your current directory. Tired of only processing audio files? Using `rtinput("AUDIO")` will allow you to prompt your internal microphone or input from your external sound card. (We're going to look at interactivity and the real-time processing of instruments when we explore RTcmix with Pd and Max/MSP.)

<sup>73</sup> But consider what it must've been like before the days of real time synthesis when you'd have to wait a day to hear what your code ended up generating in terms of sound. Quite the world we live in, really.

internally send and receive digitals via `bus_config( )` and `SPLITTER( )`, you'll be sure to find yourself incorporating subtle changes to your sounds to better enhance your compositional experience. Who knows, maybe you'll find yourself feeling more apt to algorithmically design a sweepable EQ or completely mix your work in a series of .sco files and ditch your big box, proprietary DAW all together?

#### **INTERLUDE IV: COMPLEX PANNING ETUDE**

Using `bus_config( )` and `SPLITTER( )`, create an etude that focuses nearly exclusively on panning. Think about ways that you can isolate sounds exclusively to either side of the stereo field and have them interact in a meaningful way. Could you create a panning etude that has two sounds constantly crossing over each other at different rates, while maintaining their unimpeded signal in each's respective beginning location?

## || Sonata V: Using RTcmix the “wrong” way ||

Of course there is no “right” or “wrong” way to use RTcmix and the title of this Sonata shouldn’t instill a feeling that your personal use of the program will inherently be thought of as either par for the course or unusual in any respect, rather; For the past few years, I’ve become increasingly interested in a subgenre of electronic music referred to as “glitch” and the ways that I can achieve those glitch sounds and textures into my own music using RTcmix. Thus, what follows is a creative exploration of glitch techniques using RTcmix that in no way “breaks” the program or suggests that we’ll use it in any way that is unintended. In fact, for me, half of the fun of glitch is using what are (were?) at one time or another thought of as mistakes in the digital audio domain (think clicks and pops and distortion, etc.) that can be creatively harnessed into an exciting sound world.

Glitch music itself could mean a variety of things, including the suggestion that the music itself is borne from some form of failure or destruction.<sup>74</sup> One example is the systematic marring of CDs with the intention that they will skip (codified by, among others, Oval and Yasunao Tone), providing a rhythmic backdrop of unintentional jumps and clicks. Anyone who has plugged an 1/8” jack into their laptop while connected to a mixer with the levels up will certainly understand the wonderfully egregious sounds (glitches) that emanate from your studio monitors.

Generating glitch sounds don’t always need to come from alterations or cracks in hardware however, and can be cultivated systematically from digital signal processing using RTcmix. This Sonata uncovers a few of those processes and hopefully engenders a willingness to explore the program beyond its perceived functionality. We’ll begin by looking at quantization noise and aliasing — which will inevitably lead to a brief discussion of digital audio theory — and move to manufacturing clicks and pops, finally

---

<sup>74</sup> A definitive look at the history and discussion of glitch music is undoubtedly Caleb Kelly’s *Cracked Media: The Sound of Malfunction*.

exploring ways to call on RTcmix instruments beyond their “limits.”

As mentioned in footnote 31, bit depth is the number of bits available to represent each sample in the digital domain. In 16 bit audio, there are 65,536 integer values available to represent the analog signal being sampled (with 44,100 samples/second in standard CD quality audio). However, analog signals will always be more accurate than their digital representations and quantization error is the resulting numerical difference between incoming, continuous analog signals and their discrete digital representations during the sampling process. This inherently produces noise that will become far more apparent as the bit depth is lowered.<sup>75</sup> Introducing quantization noise is a simple procedure in RTcmix using the DECIMATE() instrument.

```
rtsetparams(44100, 2)
load("DECIMATE")
rtinput("/path/to/your/file.aiff")

inputstart = 0
duration = 3.0
preamplitude = 0.5 //amplitude of original signal
postamplitude = 0.5 //amplitude of decimated signal
bitdepth = 16.0
filtercut = 0
inputchannel = 0
pan = 0.5

for(start = 0; start < 40; start += duration){
    bitdepth -= 2.0 //decrease each time through the loop
    DECIMATE(start, inputstart, duration, preamplitude, postamplitude,
    bitdepth, inputchannel, pan)
}
```

### *Score file 32: Quantization error*

---

<sup>75</sup> Although introducing a small amount of noise (called *dithering*) into the sampling process randomizes the amount of error present and will thus eliminate large patterns, or overt blocks, of noisy signal, which is a common feature of most digital audio workstations during the digital to analog conversion process.

For this score file, the bit depth is initialized at 16.0 and reduced by 2.0 each time through our loop. What results is a gradual shift from the clean sound of the original sampled file to something far more noisy. I've found that when sampling crisp sounds from something like a music box will turn its tones into something akin to an electric guitar with feedback, for example. There are many ways to refine your decimated sounds, including experimenting with the amplitude values for the incoming and decimated signals, as well as going so low as to recreate 1-bit sounds in a simple on/off scenario, which is great for those who have an affinity for harsh noise.<sup>76</sup>

According to the Nyquist theorem, frequencies that can be accurately sampled are less than or equal to one half of the sample rate. Thus, when sampling frequencies at 44.1 kHz, those frequencies greater than 22.05 kHz will be aliased, or folded over, which produces inaccurate, noisy representations in the range of frequencies that we can hear.<sup>77</sup>

The following equation is used to find the frequency of the aliased signal in Hertz. In the score file example that follows, a sine wave is sampled at 39.1 kHz, which when folded over produces a tone at 5,000 Hz.

$$a(N) = |s - (N)r|$$

a = frequency of aliased signal in Hz

s = frequency of sampled signal in Hz

r = sample rate

---

<sup>76</sup> Now, whether or not you want to cook your amplitudes above 1.0 for some extra shock value (or Merzbow remix) is up to you, but I will say that (and trust me on this one) caution should be used when experimenting with these procedures and should preferably should never be realized from the outset using headphones. Remember that the Terminal provides a readout to let you know if clipping is occurring, as well as which particular samples are being affected. Thus, turning down the main output volume on the mixer *prior to* starting the score file, checking the readout, and slowly increasing amplitude over time will prevent any surprising, and potentially damaging, aural moments.

<sup>77</sup> The general, agreed upon range of human hearing is about 20 Hz to 20 kHz, so the Nyquist frequency is pretty safe in that we can't really hear any of those frequencies above it anyway. Nonetheless, its often a good idea to utilize a low pass filter around 20 kHz to ensure that no unwanted signal makes its way to your mix, though this is of course exactly what we're after here.

$N$  = nearest common multiple (integer) of  $a$  and  $r$

$$a(1) = |39.1 - (1)44.1|$$

$$a = |-5|$$

$$a = 5 \text{ kHz}$$

```
rtsetparams(44100, 2)
load("WAVETABLE")

start = 0
duration = 10
amplitude = 5000
envelope = maketable("line", 1000, 0,0, 0.2,1, 0.4,0.5, 0.7,0.5, 1,0)
frequency = 39100 // some frequency > 22.05 kHz (Nyquist frequency)
pan = 0.5
waveform = maketable("wave", 1000, "sine") // sine wave
WAVETABLE(start, duration, amplitude*envelope, frequency, pan, waveform)
```

### *Score file 33: Aliased signal*

Outside of the glaring initialization of such an outstanding supersonic frequency, the aural result of this score file fails to provide anything unusual. However, aliased score files that utilize wave forms with more robust overtone content that shift frequencies within a loop structure provide more interesting results. The following example uses sawtooth waves with a random walk algorithm to change frequencies over time.

```
rtsetparams(44100, 2)
load("WAVETABLE")

srand()

envelope = maketable("line", 1000, 0,0, 0.5,1.0, 1.0,0)
```

```

frequency = 37100
pan = 0.5
waveform = maketable("wave", 1000, "saw")

for(start = 0; start < 40; start += 1){
    walkvalue = irand(0, 1) // generate random number between 0 and 1

    if(walkvalue <= 0.7){ // decrease by 0.5 Hz freq 70 % of time
        frequency -= 20.5
    }

    else{ // else increase by 0.5 Hz 30 % of time
        frequency += 20.5
    }

    duration = irand(2, 10)
    amplitude = irand(500, 4000)
    WAVETABLE(start, duration, amplitude*envelope, frequency, pan,
    waveform)
}

```

*Score file 34: Aliasing with random walk*

The snaps, clicks, and pops of speaker cones aided by sudden, drastic shifts of amplitude are some of the more salient features of glitch. Once considered the unwanted byproducts of the mastering and mixing process — and one of the closest notions of a “mistake” when working with digital audio that strives for clean signals — these sounds have been successfully implemented by many artists.

Using WAVETABLE() will allow us to specify a wave shape with a specific duration, amplitude, and frequency, as well as a custom waveform. As we are now well aware, maketable( ) is able to generate any of a number of graphs, and to generate clicks, we’ll use a series of line segments denoted by “line”, although “curve” and other shapes can be specified followed by the number of points in the graph. Again, I like to represent the specific values (x, y) of the graph in pairs with no spaces.



```

rtsetparams(44100, 2)
load("WAVETABLE")

duration = 1 // total duration
amplitude = 30000
frequency = 2 // initial frequency
waveform = maketable("line", 32767, 0,0, 16384,1, 16385,-1, 16386,0, 32767,0)

for(start = 0; start < 20; start += 1){
    // random #s between 0 and 0.5, avoid values > 3
    frequency = irand(0, 0.5)
    pan = irand(0, 1)

    WAVETABLE(start, duration, amplitude, frequency, pan, waveform)
}

```

*Score file 35: Generating clicks from maketable( )*

Creating a table with 32,767 points in straight line segments corresponds directly with  $2^{16}$  samples in 16-bit audio. Using values less than 32,767 table points will begin to take on a discernible pitch as the distance between each respective point becomes larger. Moreover, specifying frequency values greater than three should be avoided, as the pop will cycle through our `for( )` loop fast enough to coalesce into a discernible pitch. This particular `maketable( )` writes the pop directly in the middle of the graph, going from the amplitudes 1 to -1 and back to 0. Shifting the frequency values will change the rate at which the pops occur, though using `irand( )` also gives this score much more of a “popcorn” effect, as it returns random floating-point values between 0 and 1 each time through the loop. For a more interesting texture, panning has also been randomly distributed across the stereo field.

These particular clicks are intrinsically harsh: Their function is to send the speaker cone from full compression to complete rarefaction in a nearly indiscernible amount of time. Playing them back at loud amplitudes over long periods of time can be

damaging to the speakers and the user. However, raw click sounds can be molded into something much more subtle and beautiful using filters.

```
rtsetparams(44100, 2)
load("WAVETABLE")
load("ELL")

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("ELL", "aux 0-1 in", "out 0-1")
srand()
duration = 1
amplitude = 8000
frequency = 2
waveform = maketable("line", 32767, 0,0, 16384,1, 16385,-1, 16386,0, 32767,0)

pbcut = 9000      // passband cutoff frequency in Hz
sbcut = 900       // stopband cutoff frequency in Hz
ripple = 0.2      // amount of ripple (dB)
attenuation = 90  // attenuation at stopband (dB)

ellamp = 9        // filter amplitude
ringdur = .8      // ring-down duration (in sec.)

// passband value > stopband = highpass filter
ellset(pbcut, sbcut, 0, ripple, attenuation)

for(start = 0; start < 20; start += 1){
    frequency = irand(0, 0.5)
    pan = irand(0, 1)

    WAVETABLE(start, duration, amplitude, frequency, 0.5, waveform)
    ELL(start, 0, duration, ellamp, ringdur, 0, pan)
}
```

*Score file 36: Filtering clicks with ELL( )*

We will again call on ELL( ), our elliptical filter, which is customizable by way of its p-field commands, and can function as either a high pass, low pass, or band pass filter. To review, ELL( ) first requires the ellset( ) command to specify its parameters

before being called in the score file. This score file uses a high pass filter to accentuate the sharp, snapping quality of the clicks. To achieve this, the passband cutoff frequency (9,000 Hz) must be greater than the stop band cutoff (900 Hz). The amount of ripple, or ringing in the filter, is specified in dB. In this instance, larger values such as 50 dB will transform the clicks into a discernible pitch, thus the very low value of 0.2 dB. Finally, the amount of attenuation at the stop band is set to 90 dB, a steep filter that works well for this intended effect.<sup>78</sup>

After filling out the parameters for `ellset( )`, a few P-field commands need to be specified for `ELL( )` itself. Outside of the input and output start times, the filter's duration — set to the same value of `WAVETABLE( )` — amplitude and ring-down duration need to be declared either as numerical values or again in this instance as variables. Although panning was set to the center in `WAVETABLE( )`, it was again randomized within the loop for `ELL( )`.

Panning plays a key role in another useful approach for creating clicks and pops. Rather than setting the pan value directly in the middle of the stereo field — or even a random distribution specified with `irand( )` — it is possible to control spatialization through the `makeLFO( )` command.

```
rtsetparams(44100, 2)
load("WAVETABLE")

duration = 5
amplitude = 2000
envelope = maketable("curve", 1000, 0,0,1, 1,1,0, 3,1,1, 4,0)
frequency = 100
pan = makeLFO("sine", 5.0, 0.0, 1.0)
```

---

<sup>78</sup> As long as we're on the subject of using `ELL( )` in our score files, I'd like to point out that culminating your score files using `ELL( )` as a high pass filter set to pass high frequencies above 5 Hz will eliminate DC offset. There'll be times when your score files will include a constant amount of signal, that you'll be able to literally see when you export your score file as a .wav or .aiff file and import it using your DAW of choice. You'll see you waveform graphed out, but it'll compress and rarefact *above* the line of unity, because it is oscillating in moments of pure compression. DC offset literally pushes your speaker cones out constantly while playing back your sounds, so its best to filter that out.

```

waveform = maketable("wave", 1000, "sine")

for(start = 0; start < 20; start += 1){
    WAVETABLE(start, duration, amplitude*envelope, frequency, pan, waveform)
}

```

*Score file 37: makeLFO( ) within loops for clicks*

The frequency for makeLFO( ) is set at a 5.0 Hz sine wave, slowly oscillating between 1.0 (stereo left) and 0.0 (stereo right). By using the makeLFO( ) as a variable for panning and then putting it in the loop structure, clicks will occur as the pan values continually reset. One will notice that by simply eliminating the loop and initiating the WAVETABLE() instrument with a start time of 0 (substitute start with 0), no clicks will materialize and the score will produce a 100 Hz sine wave for five seconds. Moreover, the frequency values in makeLFO() can be altered to produce interesting effects. Frequencies ranging from 0.1 Hz to 20 Hz will simply produce clicks and pops against the frequency of the sine wave specified in the MAKETABLE() instrument, while anything greater than 20 Hz will begin to interact with the sine wave, producing a sound akin to ring modulation.

The following score file features a series of overlapping and unfolding triangle and sine waves with aliased frequencies. In its various transformations, this score file produced much of the music in in my electroacoustic work *kernel\_panic* from about 4'25" to 6'30".

```

rtsetparams(44100, 2)
load("WAVETABLE")
load("ELL")

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("ELL", "aux 0-1 in", "out 0-1")

srand()

```

```

pbcut = 80
sbcut = 24000
ripple = .8
attenuation = 90

ellamp = 60
ringdur = .1
dur = 1
amp = 9000
// envelope with curved line segments
env = maketable("curve", 1000, 0,0,1, 1,1,0, 3,1,-1, 4,0)
freq1 = 22000
pan1 = makeLFO("sine", 1, 0, 0.5)

freq2 = 20000
pan2 = makeLFO("sine", 3.5, 0.5, 1)

triwave = maketable("wave", 1000, "tri")
sinewave = maketable("wave", 1000, "sine")

for (st = 0; st < 100; st += 1){
    freq1 += irand(0, 40)
    freq2 -= irand(0, 40)
    dur = irand(5, 10)
    pbcut += 5
    sbcut -= 10
    st += irand(0, 3)

    if (st > 60){
        freq1 -= irand(0, 40)
        freq2 += irand(0, 40)
        sbcut += 100
    }

    // triangle waves produce discernible pitches, sine waves will click
    WAVETABLE(st, dur, amp * env, freq1, pan1, triwave)
    WAVETABLE(st, dur, amp * env, freq2, pan2, triwave)

    WAVETABLE(st, dur, amp * env, freq1, pan1, sinewave)
    WAVETABLE(st, dur, amp * env, freq2, pan2, sinewave)
}

ellset(pbcut, sbcut, 0, ripple, attenuation)

```

```
ELL(start = 0, instart = 0, dur = 100, ellamp, ringdur, 0, 0.5)
// keep dur (p3) as long as total script time
```

*Score file 38: Score file from kernel\_panic*

Using triangle waves in maketable(“wave”) produces discernible pitches as each subsequent harmonic frequency is folded over. Conversely, the aliased fundamental frequencies from the sine waves produce much softer pitched sounds that enhance the constantly resetting pan values. Moreover, the variables for WAVETABLE( ) and ELL( ) constantly change via the irand( ) command in the loop. Despite being a relatively simple score file, it produces a variety of results as its variables — such as those for frequencies, durations, and filter parameters — are altered. Using a variety of waveforms and a multitude of envelope shapes further adds to the effect of different waves entering in canon.

The Synthesis Tool Kit (STK) was developed by Perry Cook and Gary Scavone as a means to, among other things, design and port physical modeling instruments to a variety of programs. One famous example of physical modeling is the so-called “Karplus-Strong Plucked String Algorithm,” which sends bursts of white noise to a feedback loop in a short delay line to replicate the phenomena of plucked strings, such as harpsichords or guitars. In the STK and RTcmix, you can find these plucked string sounds in the form of the STRUM( ) instrument. In fact, we encountered one of Brad Garton’s example score files during our installation of the standalone version of RTcmix, but we’ll revisit it now in its entirety.<sup>79</sup>

```
/* START:
  p0 = start; p1 = dur; p2 = pitch (oct.pc); p3 = fundamental decay time
  p4 = nyquist decay time; p5 = amp, p6 = squish; p7 = stereo spread [optional]
  p8 = flag for deleting pluck arrays (used by FRET, BEND, etc.) [optional]
*/
```

---

<sup>79</sup> You’ll find this score file and the rest of the STK examples in /path/to/RTcmix/docs/sample\_scores. This particular example score file is called “STRUM1.sco.”

```

rtsetparams(44100, 2)
load("STRUM")
makegen(2, 2, 7, 7.00, 7.02, 7.05, 7.07, 7.10, 8.00, 8.07)

srand(0)
for (st = 0; st < 15; st = st + 0.1) {
    pind = random() * 7
    pitch = sampfunc(2, pind)80
    START(st, 1.0, pitch, 1.0, 0.1, 10000.0, 1, random())
}

```

*Score file 38: Brad Garton's STRUM1.sco example*

I made no changes to the example file to give you an example of one of RTcmix's original author's code, which is a tremendously valuable learning experience for anyone looking to use RTcmix.<sup>81</sup> You'll note that STRUM( ) is being loaded, but was called in our loop as START( ). The STRUM( ) instrument proper is really a family of plucked string instruments, which includes START( ), BEND( ), FRET( ), START1( ), BEND1( ), FRET1( ), VSTART1( ), and VFRET1( ). Each has its own unique take on the plucked string algorithm, and for lack of better words, START( ) is the most basic, though it hasn't been updated for use with maketable( ), hence the call to makegen( ) for its envelope shape.

Take a moment to scour Brad's code and come up with some educated guesses as to what exactly all of the p-fields represent. You probably came up (going from left to right) p-field 0 for a start time, then duration, then a value for pitch, and then a series

---

<sup>80</sup> While I of course always advocate for you to scour the documentation on your own, you won't find any reference to sampfunc( ) on the RTcmix page, so here's a good resource via Christopher Bailey: <http://www.music.columbia.edu/cmix/algo.html>. Once upon a time when RTcmix relied solely on makegen( ) routines for drawing tables, utilizing tables of random values couldn't (obviously) be accomplished using maketable( ), so this is one older style MINC command that is more or less no longer in use. Should you encounter legacy score file examples like this one that you'd like to explore in more detail but are unsure of the syntax, do some searching around online and if all else fails, always feel welcomed to email the RTcmix mailing list.

<sup>81</sup> Not to mention that, you know, it's just cool to look at their work.

of values that became ambiguous. P-fields 3 and 4 represent a fundamental delay time and Nyquist delay time, respectively. Following that is amplitude (absolute, of course) and a “squish” time.<sup>82</sup> Lastly, a value for pan is declared, which has been randomized.

Let’s have a little bit of fun with this score file and glitch it out just a bit.

```
rtsetparams(44100, 2)
load("STRUM")

table_number = 2
makegen(table_number, 2, 7, 3.00, 4.02, 7.05, 7.07, 7.10, 8.00, 18.07)

srand()

increment = 0.1
for (start = 0; start < 10; start += increment) {
    pitch_index = random() * 7
    pitch = sampfunc(table_number, pitch_index)
    amplitude = pickrand(500, 10000, 15000, 20000)
    decay = random()
    nyquist_decay = random()
    squish = start
    START(start, duration = increment, pitch, decay, nyquist_decay, amplitude,
          squish, random())
}
```

*Score file 39: “Glitched” version of Score File 38*

We need to first make sense of the `makegen( )`. This particular type of `makegen( )` is a type-2, which fills a table with random values, similar to what we’ve already encountered with `maketable(“random”)`. In this `makegen( )`, we have seven elements, which are octave point pitch class designators for pitches. In the loop, we’re calling on each of those elements, randomly, via `sampfunc( )` and a variable for the index of our `makegen( )` itself, called “pitch\_index.” From there, we’ve randomized a number of parameters and incremented our “squish” value to go from a super hard plectra, to the

---

<sup>82</sup> Or, how soft is the item that is “plucking the string?” 0 = super hard while 10 is akin to fleshy pads of fingers.



fattest of fingers. None of this, however, makes our score file particularly glitchy, and when you’ve played it back, you’ll notice some clicks, but where do they come from?

Take a closer look at the list of pitches in `makegen( )`. You’ll see that I’ve altered a few away from the comfortable octave range of 7 and 8 to the extremes, namely 2 and 18. These two octaves fall well outside of the range of frequencies that we could reasonably expect a more “normal” sized string to be able to play (unless we were doing something along the lines of Alvin Lucier’s music for long, thin wires), and so these parameters are essentially breaking the algorithm for pluck strings, at least in the strictures set by the code that constitutes the `START( )` version of `STRUM( )`, and we get resulting clicks in our audio, which again can be further refined or sculpted using filters or delay lines, etc.

The final example score file in this Sonata draws upon many of the aforementioned techniques, but introduces the `control_rate( )` command. Each time that you call upon a table in `RTcmix`, the program will need to essentially “sample” the table in question. This is analogous to the way that digital sampling represent waveforms at 44,100 samples/second, and the standard control sampling rate for `RTcmix` is 1000 times/second. You are able to sample your control function at the sample rate by using `control_rate(44100)`, but this can be expensive on your CPU and you probably won’t notice a tremendous difference in the end. However, *lowering* the control sample rate can have a deliciously adverse effect on your sounds and can in some instances get you to the lo-fi “8-bit” sound that you’ve been craving using `control_rate(8)`.<sup>83</sup>

```
rtsetparams(44100, 2)
load("FMINST")
load("WAVETABLE")
control_rate(4) // have fun experimenting with these values
print_off()
```

---

<sup>83</sup> Even though, for example, you may have designed a sine wave using `maketable("wave", 1000, "sine")`, which should in theory garner a pretty nice looking sine wave in the digital domain, setting the `control_rate( )` to 8 will trump the look of that wave and get you the 8-bit stair-stepped, chip tune-like sound you’ve been craving all along. Suffice it to say that I adore using `control_rate( )`. (For good and not evil, of course.)

```

srand()

//----- FM
start = 0
notedur = 0.05
amplitude = 20000
carrier_frequency = 22051
carrier_envelope = maketable("line", 100, 0,0, 1,0.5, 80,0.5, 90,0, 100,0)
modulator_frequency = (carrier_frequency * (2 / 3))
modulator_envelope = maketable("line", 100, 0,0, 1,1.0, 90,0.3, 100,0)
max_index = 1
min_index = 1
waveform = maketable("wave", 1000, "sine")
index_wave = maketable("window", 1000, "hanning")

iteration = 0.05

for(start = 0; start < 100; start = start + iteration) {
    pan = random()
    FMINST(start, notedur, amplitude,
           carrier_frequency*carrier_envelope,
           modulator_frequency*modulator_envelope,
           max_index, min_index, pan, waveform, index_wave)

    x = random()
    y = random()

    max_index = (0.166666 * ((x + y) - 20) + 0.5)
    carrier_frequency *= max_index

    if(x <= 0.5){
        modulator_frequency += irand(1, 3)
        min_index = ((x + y) / 2)
    }

    if(x > 0.9){
        carrier_frequency += 100
        modulator_frequency -= 100
    }

}

//----- WAVETABLE

```

```

start = 0
duration = 4.5
amplitude = 5000
envelope = maketable("line", 100, 0,1, 0.99,1, 1,0)
frequency = 40
pan = makeLF0("sine", 2.0, 0,1)
waveform = maketable("wave", 1000, "tri")

for(start = 0; start < 35; start += 7){
    WAVETABLE(start, duration, amplitude*envelope, frequency, pan, waveform)
    duration += 0.5
}

```

*Score file 40: Fun with control\_rate( )<sup>84</sup>*

There are any of a number of ways to creatively explore RTcmix and the best advice for realizing glitches textures in your music is to start with some of the basic tenets of digital signal processing and don't be afraid to stretch the perceived limits of the instrument or command at hand and trust your ears when creatively applying some of these examples into your own work. The Synthesis Tool Kit is one example of a set of instruments that may be exactly what you are looking for in a suite of tools to help you realize some of the more classic sounds of 90s computer music — but like any instruments in RTcmix — can be further enhanced to extend far beyond its original intended use.

## SONATA V: GLITCH ETUDE

Begin by using an example score file from the RTcmix application itself and step-by-step alter it to either distort it or glitch it in some meaningful way. Write down each of the steps that you used to realize your work and present it in a before and after demonstration for your peers.

---

<sup>84</sup> Although it is reserved specifically in the Perl language, reset( ) will do exactly the same thing as control\_rate( ).

## || Interlude V: More synthesis and modulation ||

By now, we've either briefly discussed or used the sounds of a variety of types of digital synthesis procedures. This interlude will explore three main tenets of synthesis in a bit more detail: Additive, subtractive, and granular synthesis, while also discussing ring and frequency modulation.

Our very first foray into RTcmix was by way of the innocuous sine wave, which is one of the most basic waveforms in electronic music. Sine waves are devoid of any harmonic content, meaning that they singularly sound at a fundamental pitch — calculated in cycles per second or Hertz (Hz) — and nothing more, which is why they are sometimes referred to as pure waves. However, by adding sine waves above the fundamental pitch together at specified amplitudes and frequencies, it is possible to construct more complex waveforms through additive synthesis.

The harmonic series is a naturally occurring phenomena that outlines the respective partials (or harmonics) of any fundamental pitch. For example, a string that is tuned to 110 Hz will sound at 220 Hz when it is shortened to half of its length. If you were to half the half, then half the half, etc., you'd find that each respective harmonic sounds at 330 Hz, 440 Hz, 550 Hz, etc. This harmonic series is critical in constructing waveforms from scratch. Some of our most basic waveforms in electronic music are, in addition to the sine wave, square, sawtooth, and triangle waves.

A triangle wave is built by taking a fundamental pitch and adding only the odd numbered partials to its spectrum at amplitudes of  $1/n^2$  where  $n$  is equal to the partial number. So, we can build a triangle wave using sine waves at 110 Hz (1.0 amplitude), 330 (0.33), 550 (0.2), 770 (0.14), etc. When built this way in a digital audio workstation, you'll find that the more partials that are added, the more discernible your waveform

will begin to look.<sup>85</sup>

Thankfully, RTcmix doesn't make us do all of the math ourselves when calling for various waveforms via `maketable("wave")`.<sup>86</sup> However, there may be some times when you'd like more control over the constructs of your waveform and `maketable("wave3")` will be your resource, which gives you the ability to specify not only partials and amplitudes, but also phase shifts between 0 and 360 degrees.

```
wave = maketable("wave3", 1000, 1,1,0, 3,0.33,45, 5,0.2,90, 7,0.14,120)
```

This whole business of constructing waves becomes way more fun when you use them in conjunction with `HALFWAVE( )`. A versatile instrument, `HALFWAVE( )` essentially joins two waveforms together with the ability to dynamically update their meeting point. In the following example score file, I'm taking a variety of waveforms and randomly selecting from them to construct the two waves that I'll need for each call to `HALFWAVE( )` inside of my loop. The most critical part of this process is accomplished by these two lines of code, which specify the array of waveforms that I'm using as well as a variable for the length of my array, via `len( )`:

```
wavegamut = {wave1, wave2, wave3, wave4, wave5}  
wavegamutlength = len(wavegamut)
```

What follows then, is a sort of nebulous, out of tune organ sound that could be even further refined by finding ways to more creatively design the constituent parts of each call to `maketable("wave3")`.

```
rtsetparams(44100, 2)
```

---

<sup>85</sup> This is another wholly vague and glossed over explanation of basic additive synthesis, but again, I encourage you to dig into a few introductory texts on your own or a good old fashioned Google search. Some of my favorite texts that discuss synthesis in a really approachable way are Thom Holmes' *Electronic and Experimental Music* and *An Introduction to Music Technology* by Dan Hosken.

<sup>86</sup> That isn't to say that you couldn't construct a crude triangle wave by using `maketable("wave", 1000, 1.0, 0, 0.33, 0, 0.2, 0, 0.14, 0...)`.

```

load("HALFWAVE")
load("REVERBIT")
srand()

bus_config("HALFWAVE", "aux 0-1 out")
bus_config("REVERBIT", "aux 0-1 in", "out 0-1")

totalduration = 60

//-----HALFWAVE
start = 0
duration = 1.75

octavegamut = {2, 10, 11}
octavegamut_length = len(octavegamut)

pitchgamut = {0.00, 0.01, 0.04, 0.05, 0.06, 0.08, 0.10, 0.11}
pitchgamut_length = len(pitchgamut)

amplitude = 1000
envelope = maketable("line", 100, 0,0, 0.5,1, 1.0,0)
wave1 = maketable("wave3", 1000, 3.14,1,0, 6.28,1,0.5)
wave2 = maketable("wave3", 1000, 1.00,1,0, 2.00,1,0.5)
wave3 = maketable("wave3", 1000, 1,1,0, 3,0.3,0, 5,0.2,0, 7,0.05,0, 9,0.01,0,
                    11,0.001,0)
wave4 = maketable("wave3", 1000, 1,1,0, 2,0.5,0, 3,0.3,0, 4,0.25,0, 5,0.2,0, 6,0.16,0,
                    7,0.14,0, 8,0.125,0)
wave5 = maketable("wave3", 1000, 1,1,0, 3,0.14,0, 5,0.04,0, 7,0.02,0, 9,0.012,0,
                    11,0.008,0)
wavegamut = {wave1, wave2, wave3, wave4, wave5}
wavegamutlength = len(wavegamut)

wavecrossoverpoint = 0.5
pan = 0.5

increment = 0.5

for(start = 0; start < totalduration; start += increment){
    octave = octavegamut[trand(0,octavegamut_length)]
    nextpitch = pitchgamut[trand(0,pitchgamut_length)]
    pitch = nextpitch + octave

    if(octave == 2){

```

```

        duration = 5.0
        pan = makeLF0("sine", 1.25, 0,1)
        HALFWAVE(start, duration, cpspch(pitch)+0.04, amplitude*envelope, wave1,
                wave2, wavecrossoverpoint, pan)
    }

    wave1 = wavegamut[trand(0,wavegamutlength)]
    wave2 = wavegamut[trand(0,wavegamutlength)]

    random_number_x = round(trand(0, 10))
    random_number_y = round(trand(0, 10))
    average_of_random_numbers = ((random_number_x + random_number_y) / 2)
    wavecrossoverpoint = average_of_random_numbers / 10

    pan = random()
    HALFWAVE(start, duration, cpspch(pitch), amplitude*envelope, wave1, wave2,
            wavecrossoverpoint, pan)
    increment = average_of_random_numbers / 7
}

//-----REVERBIT
start = 0
instart = 0
duration = totalduration
amplitude = 1.0
envelope = maketable("line", 1000, 0,0, 0.4,1.0, 0.9,0, 1.0,0)
revtime = maketable("line", 1000, 0,1.0, 0.5,1.0, 1.0,0.2) //keep these short
revamnt = 1.0 //0-1 (dry to wet)
chandelay = maketable("random", 100, "gaussian", 0.01,2.0)
cutoff = 2000 //low pass filter in Hz
REVERBIT(start, instart, duration, amplitude*envelope, revtime, revamnt, chandelay, c
        cutoff)

```

*Score file 41: HALFWAVE( )*

Earlier, we encountered subtractive synthesis using elliptical filters, designing them as either high or low pass, though we also know that ELL( ) can set up bandpass filters as well. If we consider, by way of an example, a high pass filter, we know that its design specifies a *cutoff frequency* (Hz) and allow frequencies above that cutoff, in the *passband*, to sound while rejecting those that fall below it in the *stopband*. Recall also

that filters have an amount of *ripple* at the the meeting point of the pass and stop bands, which is calculated in dB, and is the ratio of the highest and lowest amplitudes in the passband.<sup>87</sup> Perfectly flat ripple would have a gain of 0 dB, while increasing the ratio will start to give your filter a ripply, wavy look to it.

Another versatile filter in RTcmix's arsenal is BUTTER( ), a dynamic Butterworth filter.<sup>88</sup>

```
rtsetparams(44100, 2)
load("TRANS")
load("BUTTER")
rtinput("/path/to/file.aif")
control_rate(8)

bus_config("TRANS", "in 0-1", "aux 0-1 out")
bus_config("BUTTER", "aux 0-1 in", "out 0-1")

//-----TRANS
start = 0
instart = trand(0,DUR())
duration = 20
amplitude = 1.0
low = octpch(-0.01) //down a m2
high = octpch(0.01) // up a m2
transp = maketable("random", 1000, "gaussian", low,high)
transposition = makeconverter(transp, "pchoct") //convert to oct.pc
TRANS3(start, instart, duration, amplitude, transposition)

//-----FILTER

filt_type = "bandpass"
sharpness = 5
balance = 1
```

---

<sup>87</sup> A great explanation of this — and filters in general — is given in Miller Puckette's *The Theory and Technique of Electronic Music* starting on page 225. Those of you who are avid users of Pure Data will likely want to own a copy of this book.

<sup>88</sup> Head to this Wikipedia link: [https://en.wikipedia.org/wiki/Butterworth\\_filter](https://en.wikipedia.org/wiki/Butterworth_filter) for a really great visual comparison of Butterworth filters to the Elliptical filter that we looked at earlier. You'll see how much smoother Butterworth filters look in design.



```

input_channel = 0
pan = 0.5
bypass = 0 // 0 to filter sound, 1 to bypass
center_frequency = 1500
bandwidth = 250
BUTTER(start, instart = 0, duration, amplitude, filt_type, sharpness, balance,
input_channel, pan, bypass, center_frequency, bandwidth)

```

### *Score file 42: Butterworth bandpass filter*

After having a bit of fun with some random transpositions and grittiness via `control_rate( )`, our Butterworth filter has been initialized as a bandpass, with a center frequency of 1500 Hz and a *bandwidth* of 250 Hz. Bandwidth measures the width of the passband, again in Hz. This p-field can be dynamically updated and only needs to be included if you are using a bandpass or bandreject filter.

When it comes to total refinement of your sound, there really is no better solution than careful equalization, which isn't necessarily a function of filtering out certain frequencies, rather; Equalization gives you the ability to attenuate a number of frequencies in the audio spectrum, thus enhancing or quieting certain bands of sound that will better direct you to your desired end result and RTcmix's `MULTEQ( )` instrument will allow you to attenuate your sounds up to eight bands.

```

rtsetparams(44100, 2)
load("MULTEQ")

rtinput("/path/to/file.aif")

start = 0
instart = 0
duration = DUR()
amplitude = 0.25
bypass = 0

type1 = "lowshelf"
freq1 = makeLF0("tri", 2.25, 20,100)
Q1 = maketable("line", "nonorm", 1000, 0,10, 0.5,1, 1.0,10)

```

```

gain1 = maketable("line", "nonorm", 1000, 0,5, 0.25,-10, 1.0,-2)
bypass1 = 0

type2 = "peaknotch"
freq2 = makeLF0("sine", 0.0125, 200,12000)
Q2 = 5
gain2 = maketable("curve", 25, 0,-12,0, 0.25,9,-10, 1.0,0)
bypass2 = 0

type3 = "peaknotch"
freq3 = 15000
Q3 = 0.25
gain3 = maketable("line", "nonorm", 1000, 0,-2, 1.0,12)
bypass3 = 0
MULTEQ(start, instart, duration, amplitude, bypass,
        type1, freq1, Q1, gain1, bypass1,
        type2, freq2, Q2, gain2, bypass2,
        type3, freq3, Q3, gain3, bypass3)

```

*Score file 43: MULTEQ( ) with p-field updates*

Infinite impulse response (IIR) and finite impulse response (FIR) filters have an output that is dependent on a series of incoming impulses. IIR filters are infinite because they are recursive and have some amount of feedback in their design, which FIR filters are non-recursive. RTcmix will give you the ability to design very complex filters using its IIR( ), FIR( ), or FILTERBANK( ) instruments, but one exciting effect that falls in line with these principles is the comb filter, accessed by way of COMBIT( ) or MULTICOMB( ).

```

rtsetparams(44100, 2)
load("STEREO")
load("COMBIT")
rtinput("/path/to/file.aif")

bus_config("STEREO", "in 0-1", "aux 0-1 out")
start = 0
instart = 0

```

```

duration = DUR()
amplitude = 1.0
pan = 0.5
STEREO(start, instart, duration, amplitude, pan)

bus_config("COMBIT", "aux 0-1 in", "out 0-1")
frequency = maketable("line", "nonorm", 10, 0,0.1, 1.0,100.0) // in Hz
reverbtime = maketable("line", "nonorm", 1000, 0,2.0, 0.2,10.0, 0.75,4.25, 1.0,0.125)
inputchannel = 0
pan = 0.5
ringduration = 2.5
COMBIT(start, instart, duration, amplitude - 0.25, frequency, reverbtime,
        inputchannel, pan, ringduration)

```

*Score file 44: Comb filter with linear frequency growth*

You'll note that as this score file plays out, the frequency of the comb filter slowly increments from 1 to 100 Hz over a series of varying reverb times. The result is an ethereal cascade of sounds as the comb filter rings the incoming sound at the varying frequencies outlined in our `maketable( )` for p-field 4. When viewed on a graph, you'll see why comb filters are aptly named.

There are a number of filtering and equalizing instruments in RTcmix's rich array of tools. As with most sections in this book, it is my hope that your interest will be piqued enough to head to the documentation and for yourself check out the possibilities afforded to you by `FILTERBANK( )`, `FILTSWEEP( )`, `JFIR( )`, `MULTICOMB( )`, `MULTEQ( )`, and others.

While it has come to mean a few different things in digital music, *sampling* is an important component of our work in RTcmix. When we construct a waveform using `maketable( )` and `WAVETABLE( )` we're creating a series of samples that in sum create a *wavetable*. There are many different procedures that we can realize using wavetables and when we divide them into several microsonic subparts and play them back rapidly,

we are in the realm of *granular synthesis*.<sup>89</sup>

Many composers have explored granular synthesis in one form or another and RTcmix boasts a number of instruments that can realize this technique. Here is an introductory example using John Gibson's JGRAN( ) instrument, which defines small grains using a specified waveform.

```
rtsetparams(44100, 2)
load("JGRAN")
control_rate(512)

start = 0
duration = 10
amplitude = 5.0 //okay to go above 1.0 for this instrument
seed = srand()

//oscillator type: 0 = wavetable, 1 = FM
type = 1

//randomize oscillator phase? 0 = no, 1= yes
randphase = 1

// grain envelope
genv = maketable("window", 1000, "hanning") // common window function

// grain waveform
gwave = maketable("wave", 1000, "sine")

// modulation frequency multiplier
mfreqmult = maketable("line", "nonorm", 25, 0.0,1.25, 0.5,2.0, 1.0,0.5) // Hz

// index of modulation envelope (per grain)
modindex = maketable("line", "nonorm", 100, 0.0,0.5, 0.33,10.0, 0.66,25.0, 0.99,0.0,
1.0,0)

// grain frequency
minfreq = 100
```

---

<sup>89</sup> “Microsonic” is an intentional nod to the work of Curtis Roads, whose book *Microsound* is required reading for those interested in the sounds obtained from and the possibilities afforded by granular synthesis.

```

maxfreq = 2205

// grain speed
minspeed = 10    //number of grains/second
maxspeed = 1000

// grain intensity (decibels above 0)
mindb = 40
maxdb = 90

// grain density
density = maketable("line", "nonorm", 100, 0.0,1.0, 0.25,0.325, 1.0,0.8)

pan = maketable("random", 100, "gaussian", 0,1)

// grain stereo location randomization, 0 = none, 1 = full randomization
panrand = 1

JGRAN(start, duration, amplitude, seed, type, randphase,
      genv, gwave, mfreqmult, modindex, minfreq, maxfreq, minspeed, maxspeed,
      mindb, maxdb, density, pan, panrand)

```

*Score file 45: Adapted from John Gibson's JGRAN( ) help file*

Note that we have quite a bit of control of how these grains of sound interact with each other over time. Not only can we specify the type of waveform to cull these grains from — in this case a sine wave — but we can specify an envelope shape for them, their frequencies, their speed in grains/second, density, intensity, and stereo location.<sup>90</sup> All of these p-fields are tightly controlled and can be modified in real-time via `maketable( )`. For even greater stochastic control of these grains, Mara Helmuth's `SGRANR( )` is a fantastic instrument and we can perform granular synthesis on

---

<sup>90</sup> The so-called Hanning and Hamming window functions are pretty typical for grain shapes in granular synthesis.

incoming sound files using GRANULATE( ).<sup>91</sup>

```
rtsetparams(44100, 2)
load("GRANULATE")

source = "/path/to/file.aif"
file = maketable("soundfile", "nonorm", 0, source)
filedur = DUR()

start = 0
instart = trand(0,DUR())
duration = 45
amplitude = ampdb(40)
channels = 2
inputchannel = 0

windowstart = 0.0
windowend = random()
wraparound = 1
traverserate = 1.0
table = maketable("window", 1000, "hamming")
hoptime = 0.0625

injitter = 0.0625
outjitter = 0.0125

mindur = hoptime + random()
maxdur = mindur() + random()
minamp = 0.5
maxamp = 2.0

GRANULATE(start, instart, duration, amplitude, file, channels, inputchannel,
           windowstart, windowend, wraparound, traverserate, table, hoptime,
           injitter, outjitter, mindur, maxdur, minamp ,maxamp)
```

*Score file 46: GRANULATE( )*

---

<sup>91</sup> Granular synthesis was a thoroughly heavily explored synthesis technique as RTcmix came of age, so you'll find quite a few ways to approach it using the program. See also GRANSYNTH( ), STGRANR( ), STGRANR2( ), and JCHOR( ).

Ring modulation is a classic effect in the analog electronic music world and one of my absolute favorite works that uses the technique is Karlheinz Stockhausen's *Mantra*. Ring modulation involves the multiplication of two signals and in the case of Stockhausen's work, we take the complex sound of a piano and multiply it by the simple signal of a sine wave. This technique can be replicated using the `AM( )` instrument.

```
rtsetparams(44100, 2)
load("AM")
rtinput("/path/to/my/awesomefile.wav")

start = 0
instart = 0
duration = 30
amplitude = 1.0
frequency = makeLFO("sine", 0.1, 30, 1000)
inputchannel = 0
pan = 0.5
waveform = maketable("wave", 1000, "sine")
AM(start, instart, duration, amplitude, frequency, inputchannel, pan, waveform)
```

*Score file 47: Ring modulation using AM( )*

When applied to an incoming signal, ring modulation produces sidebands, which are the sum and difference (in Hz) of the signal with the frequency of the modulator. For example, a 400 Hz tone that is sent to a 100 Hz ring modulator will produce sidebands of 500 and 300 Hz, respectively. By sweeping the modulator up and down using `makeLFO( )`, you can hear these sidebands as they ascend and descend, sweeping across the rich audio spectrum of the incoming signal.

Frequency modulation (FM) is somewhat more complex than ring modulation, but its implementation is easy to understand. Discovered by John Chowning, FM synthesis involves three key components: A *carrier frequency*, *modulator frequency*, and *modulation index*. With ring modulation, we noted the presence of only one upper and one lower sideband, however with FM, there are many more sidebands present in the

audio spectrum, producing richer timbral content. The incoming carrier is modulated by the modulator at a given frequency — generally above the lower threshold of human hearing so that we can't perceive the change in pitch, otherwise we get the carrier with vibrato — while the index calculates the range of deviation between the carrier frequency and the modulator frequency, or carrier/modulator. Several factors go into changing the timbre of the resulting sound, such as the waveforms used as the carrier, modulator, or the index number, but with a simple implementation comes a cornucopia of sounds. In fact, John Chowning licensed his patent for FM synthesis to Yamaha, who in turn created the powerful DX-7 synthesizer, among other instruments.<sup>92</sup>

```
rtsetparams(44100, 2)
load("FMINST")

duration = 2.0
amplitude = 1000
envelope = maketable("window", 1000, "hanning")
carrier = 10
modulator = 110
minindex = 0
maxindex = 1.0
waveform = maketable("wave", 1000, "sine")
index_envelope = maketable("line", "nonorm", 1000, 0,0, 0.5,1.0, 1.0,0)

increment = 0.5
for(start = 0; start < 20; start += increment){
    FMINST(start, duration, amplitude*envelope, carrier, modulator,
           minindex,maxindex, pan = random(), waveform, index_envelope)

    modulator += 110
    maxindex += 0.25
}
```

---

<sup>92</sup> FM synthesis was highly desired, because you could obtain a rich variety of sounds — brassy, bell-like, percussive — based on a computationally lean algorithm. Those familiar with Pd or Max will of course have access to FM in a visual diagram, which I encourage you to check out via the help files if you'd like to try FM synthesis on those platforms for yourself.



*Score file 48: Frequency modulation with subsonic carrier*

It is interesting to note how in this score we are using a subsonic carrier frequency of 10 Hz, which still produces a dazzling array of sidebands given the sine wave modulator moving up the harmonic series at 110 Hz, with a low modulation index.

Here is one last FM scorefile, which establishes an interesting groove via a loop and the use of a “voltage-controlled” filter, designed after the famous Moog VCF.<sup>93</sup>

```
rtsetparams(44100, 2)
load("FMINST")
load("MOOGVCF")
load("PAN")
srand()

//----- FMINST
bus_config("FMINST", "aux 0-1 out")
start = 0
duration = 0.125
modulatorfrequency = 440
minindex = 0
waveform = maketable("wave", 1000, "sine")
guide = maketable("line", "nonorm", 8, 0,1.0, 1.0,0)

increment = duration
for(start = 0; start < 500; start += increment){
    carrier = cpspch(pickrand(6.02, 7.05, 5.07, 5.11, 5.02, 5.04, 5.09, 5.11))
    maxindex = irand(0,5)
    amplitude = trand(5000,20000)
    FMINST(start, duration, amplitude, carrier, modulatorfrequency, minindex,
           maxindex, pan = 0, waveform, guide)
    FMINST(start+1, duration, amplitude, carrier, modulatorfrequency, minindex,
           maxindex, pan = 1, waveform, guide)
}

//----- MOOGVCF
bus_config("MOOGVCF", "aux 0-1 in", "aux 2-3 out")
```

---

<sup>93</sup> Be careful of the resonance on this filter. One of the defining characteristics of MOOGVCF( ) is its sharp, resonant peaks and you'll find a threshold where the filter will self-oscillate, causing feedback.

```

start = 0
instart = 0
duration = 360
amplitude = 1.0
envelope = maketable("line", 1000, 0,0, 0.25,1.0, 0.9,1.0, 1.0,0)
loophan = 0
pan = 0.5
bypass = 0
centerfrequency = makeLFO("tri", 0.125, 500,12000)
resonance = maketable("line", 1000, 0,0.45, 0.25,0.1, 0.625,0.425, 1.0,0.25) //
careful with these values

MOOGVCF(start, instart, duration, amplitude, loophan, pan, bypass, centerfrequency,
resonance)

//----- PAN
bus_config("PAN", "aux 2-3 in", "out 0-1")
start = 0
instart = 0
duration = 360
amplitude = 1.0
inchannel = 0
pantype = 1
pan = maketable("random", 1000, "even", 0,1)
PAN(start, instart, duration, amplitude, inchannel, pantype, pan)

```

*Score file 49: FM groove with MOOGVCF( )*

The world of synthesis and modulation is far reaching and ripe for more exploration. While the basic tenets of synthesis as outlined in this Interlude aren't themselves malleable, what you do with those tenets in your score files is what makes for interesting sonic results. We've just seen how FM can produce velvety, cascading waves or be used as the backdrop for a groove-based etude. Depending on the types of waveforms that you utilize —whether of our basic shapes or something constructed algorithmically or from known data plots — you'll find that your results will be surprising, interesting, and most importantly fun to work with.

## **INTERLUDE V: SYNTHESIS ETUDE**

Create a 2-3' etude using at least one of the synthesis or modulation procedures outlined in this section, as well as one that hasn't yet been covered. Scour the online documentation for your new instruments' implementation and be prepared to present its basic tenets and use for your peers.

## || Sonata VI: Randomness and algorithms ||

RTcmix's basic methodology lies in the calling of scorefile commands and instruments using the MINC parser with real-time handling of p-field parameters. The elements of C that are found within MINC — not to mention Python — afford us as users the ability to create well designed algorithms and processes that will enhance our sound world over time.

An algorithm is a process or order of operations that brings about a desired result. We've already encountered one example when we explored and wrote out the  $3n + 1$  wondrous number game in score file 16. In it, we not only observed a set of numbers that were derived from the algorithm itself, but also explored randomness by way of `trand( )` for determining the original number that onsets the process. Algorithms are limited only by your imagination and in this Sonata, we will more deeply explore randomness and algorithmic procedures, creatively utilizing their principles in your work.<sup>94</sup>

By way of review, we are already familiar with, or have come across `srand( )`, `irand( )`, `trand( )`, `random( )`, `pickrand( )`, as well as `maketable( "random"`) and some of its weighted distributions. I'd say that we're off to a pretty good start! However, let's begin exploring randomness in more detail by focusing on `pickrand( )`, which has been introduced but not fully explained.

Recall that an array in MINC is a series of elements that we can draw from in our score files. If, for example, I'd like to create a C major pentatonic melody, but choose randomly from the set of pitches that comprise the scale itself, I could begin by

---

<sup>94</sup> Because this Sonata is pretty code-heavy and won't focus so much on making sound, you won't find examples marked Score file 50, 51, etc., but rather there'll be an ongoing series of code examples for you to try and of course save for yourself.

using `pickrand( )`.<sup>95</sup> Carefully copy the following code and run it a few times in your Terminal to compare the results.

```
 srand()
 print_off()

 notes = {}
 for(i = 0; i < 10; i += 1){
     note = pickrand(7.00, 7.02, 7.04, 7.07, 7.09)
     notes[i] = note
 }

 print_on()
 print(notes)
```

You'll see that each time you execute this score file, you don't have a great deal of control over the notes that are chosen at random as we fill our array with random pitches from `pickrand( )`. However, you can utilize another command called `pickwrand( )`, which chooses elements based on weighted distributions.

```
note = pickwrand(7.00,10, 7.02,10, 7.04,20, 7.07,20, 7.09,40)
```

Each note in our original set now has a corresponding number to go along with it, which you'll see I've written in pairs. 7.00 with a corresponding 10 literally means "choose 7.00 10% of the time." From this, you can see that I'm favoring the notes E, G4, and A a bit more than C or D, and A will be picked about 40% of the time. Go ahead and substitute this new line into score file 50 and check the results.<sup>96</sup>

While it is exciting to know that we can more carefully craft our parameters

---

<sup>95</sup> In reality, we're never really choosing at random since we're seeding our random number generator to something that can be determined. In this case — as with most — it's best to say *pseudorandom* numbers, but we can stick with random.

<sup>96</sup> The possibilities are now starting to take shape as we consider these random distributions. What if, for example, you wanted to favor C4 in your melody and also have the duration of each of those notes be twice as long as the other notes? Or, could you find a way to ensure that all A4s pan to the right?

using random numbers, let's go one level deeper and create some of these distributions from scratch.

We'll start by choosing random integers between 0 and 10 using `trand( )` and aim to favor a return of values in the middle, a so-called triangular distribution due to its high probability of middle values and low probability of low and high values. When the probability distribution is viewed on a graph, it resembles the shape of a triangle, hence the name.

```
 srand()
 print_off()

 gamut = {}
 for(i = 0; i < 10; i += 1){
     x = trand(0,11)
     y = trand(0,11)
     element = ((x + y) / 2)
     gamut[i] = element
 }

 print_on()
 print(gamut)
```

In order to get at values weighted toward the middle, we first choose two random integers and then take the average of them. Since we're finding a generic list of elements, I've gone with the variable name *gamut* to denote our array, filling it with the *element* variable. Careful readout of your Terminal will show that because we're taking the average of two integers, we'll at times end up with floating point numbers, but we can mitigate this by rounding up the values for each individual element.<sup>97</sup>

```
element = trunc((((x + y) + 0.5) / 2))
```

It's been a while since we've seen `trunc( )`, but recall that it takes off decimal

---

<sup>97</sup> I added the `print_off( )` and `print_on( )` commands to ensure that all that is printed to the Terminal is a clean looking array and nothing else.

points, which is our goal here, only after we've "rounded up" by adding 0.5 to each of our elements.

We can also weight outcomes toward the low or high values of our list.

```
 srand()
 print_off()

 gamut = {}
 for(i = 0; i < 10; i += 1){
     x = trand(0,11)
     y = trand(0,11)

     if(x > y){
         element = x
     }

     else{
         element = y
     }

     gamut[i] = element
 }

 print_on()
 print(gamut)
```

We're again using two random integers, this time ensuring that each number for *element* will be drawn from the highest value returned between *x* and *y*, rather than their average. This is accomplished by our conditional statement "if *x* is greater than *y*, then *element* equals *x*, else *element* equals *y*. Conversely, we can change the outcome for *element* to favor the lower end of the 0-10 spectrum, but I'll pause here and let you come up with the solution to that on your own.

The preceding examples will generate one single outcome, which we can of course transfer to a loop and then an array to garner a series of elements. However, we do have the ability to generate and manipulate random numbers in the `maketable( )` command.

```
gamut = maketable("random", 10, "gaussian", 0,1)
dumptable(gamut)
```

A Gaussian distribution<sup>98</sup> is sometimes referred to as a normal distribution and is one of the tried and true “bell curves” that you might be familiar with w/r/t grades. Like triangular distributions, it has a proclivity toward values in the middle of a given range, albeit with a slightly different distribution. The `dumptable( )` command will print the contents of your table to the Terminal window. Moreover, there are a few commands that will allow you to further manipulate your table data, much like what we saw earlier with `makefilter( )`.

```
gamut = maketable("random", 10, "cauchy", 30,100)
gamut_plus_one = add(gamut, 1)
gamut_minus_two = sub(gamut, 2)
gamut_times_five = mul(gamut, 5)
gamut_dividedby_two = div(gamut, 2)

dumptable(gamut)
dumptable(gamut_plus_one)
dumptable(gamut_minus_two)
dumptable(gamut_times_five)
dumptable(gamut_dividedby_two)
```

Note that we’ve selected a range of 30 to 100 using a Cauchy distribution, which also has a bell-curve shape, and in this example, we can interpret the outcomes for any of a variety of musical parameters, including frequency in Hz. If we want to utilize those values as pitch data for an instrument, we can convert it accordingly.

```
pitch_gamut = makeconverter(gamut, "cspch")
```

This will translate all of our values in Hz to octave point pitch class, so that, for example, a value of 440 will return 7.09. A check of the online documentation will show

---

<sup>98</sup> Returning individual Gaussian numbers exists in the `JFUNCS( )` library.



that there are a number of ways to convert your table data to fit your particular needs.

If an algorithm is a process or order of operations that will produce an outcome, then it is important to consider that a well-considered and strongly crafted algorithm will lead to more effective results. For example, we might implement a random walk to determine pitch, as we did in score file 15.

```
srand()
print_off()
pitches = {}
note = 60 //middle C
for(i = 0; i < 24; i += 1){
    x = random()
    if(x < 0.5){
        note += 1 //50% of the time, increase by a half step
    }

    if(x > 0.5){
        note -= 1 //50% of the time, decrease by a half step
    }

    pitches[i] = note
}

print_on()
print(pitches)
```

This particular random walk will return notes one half step up or one half step down fifty percent of the time, respectively. This is akin to a coin flip, which will vary over time, but we can reasonably expect our algorithm to have 12 ascending moments and 12 descending through the loop. We've laid a backdrop for our process, but this is ripe for further refinement. What if, for example, you wanted to favor, ever so slightly, ascending notes rather than descending? What ratio will you choose? 51/49? 60/40? Could all ascending notes move up by a whole step while those that descend move by a half step?

Moreover, let's say that we're writing this wonderful melody for an instrument

that has a limited range. We'll go with an octave, so we can only choose notes between 60 and 72. We're going to need to set up some boundaries and another rule for our random walk once it reaches a boundary point. There are any of a number of ways to treat this rule, but we'll go with returning our note to F#, in the middle of our octave range and a point where we can start this new melody.

```
 srand()
 print_off()
 pitches = {}
 note = 66 //F# above middle C
 for(i = 0; i < 24; i += 1){
     x = random()
     if(x < 0.54){
         note += 2 //50% of the time, increase by a half step
     }

     if(x >= 0.46){
         note -= 1 //50% of the time, decrease by a half step
     }

     if(note > 72 || note < 60){
         note = 66
     }
     pitches[i] = note
 }

 print_on()
 print(pitches)
```

We can continue refining and transforming this random walk as much as we'd like. In order to avoid getting caught up in the myriad of possibilities, I find it helpful to keep a composition journal that I write in plain english before crafting on the computer. For this example, I might write something like, "I want a limited range of MIDI pitches between 60 and 72 that will revolve around 66, or F#. Using a random walk that slightly favors ascending pitches in whole steps — as opposed to descending half steps — I'll create some boundary lines that force the melody back to F# if it surpasses the octave range. I'm thinking about tying durations into this as well, so that each whole

step corresponds with a longer note duration and greater amplitude each time I reach F#. I'll try this out on a MMODALBAR( ) instrument and maybe add REVERBIT( ) ...” You get the idea.

The study of chaos theory is incredibly interesting. Using some of its precepts, we are able to reasonably predict dynamic outcomes that are highly sensitive to initializations. One such algorithm that falls under the umbrella of chaos theory is the so-called logistic map, which among other uses can predict population density based on initial conditions. While the scope of this text isn't of course to take any of these sociological issues into consideration, we do find the the logistic map is incredibly easy to implement into MINC and provides some wicked fun results from minor deviations.

Here is the equation itself, which is actually quite simple.

$$x_{t+1} = Rx_t(1 - x_t)$$

We start with initializing two variables,  $R$ , which represents a number that corresponds with the combined effects of birth/death rates, and  $x$ , fun concept called the “fraction of carrying capacity,” which is related to the current population size.<sup>99</sup> In MINC, we'll use a loop to get our successive values for  $x$  and fill them in an array.

```
R = 2.0
x = 0.5

gamut = {}
for(i = 0; i < 20; i += 1){
    x = ((R * x) * (1 - x))
    gamut[i] = x
}
print(gamut)
```

---

<sup>99</sup> I'm culling these variables, their explanations, and the sample values from a fantastic book by Melanie Mitchell called *Complexity: A Guided Tour*, particularly the logistic map section starting on page 27. I love her quote, “Given this simplified model, scientists and mathematicians promptly forget all about population growth, carrying capacity, and anything else connected to the real world, and simply get lost in the astounding behavior of the equation itself.” This is exactly what we're about to do.

Our Terminal readout provides nothing to write home about at all, as you'll note a sequences of 0.5 values.

```
[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]
```

This is just a starting point for the fun involved with this equation, and speaks, algorithmically, to the beauty of chaotic systems and how they deviate based on minute changes in initial variables. Try, for example, changing  $x$  to 0.2 and check out the results.

```
[0.32, 0.4352, 0.49160192, 0.499858944505, 0.499999960207, 0.5, 0.5, 0.5, 0.5, 0.5]
```

It's interesting how we start below 0.5 and then slowly but surely make our way toward it before the string of repetition onsets. As noted by Melanie Mitchell, if we let  $R = 3.1$  and  $x = 0.2$ , we find that the returning values will eventually oscillate around 0.55 and 0.76.<sup>100</sup>

```
[0.496, 0.7749504, 0.540647060374, 0.769878231097, 0.549213795177, 0.767491807329,  
0.553189212337, 0.766229813842, 0.555277227287, 0.765527727245]
```

With values for  $R$  up to 3.4, this oscillation phenomena will continue. When  $R$  is between 3.4 and 3.5,  $x$  will oscillate between four values. With  $R$  between 3.54 and 3.55,  $x$  will oscillate between eight values, between 3.564 and 3.565, sixteen values. Thus,  $x$  is relatively predictable for these values for  $R$ , but chaos ensues once  $R = 3.569946$ , when  $x$  no longer hovers between a set of values and thus shows more chaotic behavior. While at first blush these values might seem to be completely random, they in fact aren't, but closely resemble random behavior.

---

<sup>100</sup> I sincerely hope that you're thinking, "Wow. This is cool. What if I used these for panning values or even transform them so that they oscillate around 0 so as to use them for a cool waveform?"

```

R = 3.569946
x = 0.5

gamut = {}
for(i = 0; i < 20; i += 1){
  x = ((R * x) * (1 - x))
  gamut[i] = x
}
print(gamut)

[0.8924865, 0.34255183839, 0.803987811424, 0.56259282557, 0.878499944893,
0.381048152586, 0.841973399124, 0.474996288695, 0.890254621243, 0.348788474365,
0.810859950725, 0.547508663226, 0.884428870981, 0.364900042489, 0.827327850893,
0.509989912316, 0.892130225286, 0.343549777878, 0.805106102693, 0.56016117682]

```

What an equation! Let's take our gamut of data and convert it into something musically significant, such as pitches for the sitar physical model instrument, MSITAR( ). Before we do that, however, we'll first need to get our individual pitches into a range that will reasonably fit into MIDI notes and then Hz. An example might look something like this:

```

pitches = gamut * 100
current_pitch = cpsmidi(trunc((pitches[ii]) + 0.5))

```

Now we can combine our processes — the logistic map, conversion to a fitting pitch range — into an example score file that includes all of the requisite p-field data for MSITAR( ).

```

rtsetparams(44100, 2)
load("MSITAR")

R = 3.569946
x = 0.5

pitches = {}
number_of_pitches = 20

```

```

//parameters for MSITAR
duration = 5.0
amplitude = 20000
envelope = maketable("line", 100, 0.0,0.125, 0.75,0.75, 1.0,0.0)
pluck = 0.25

for(start = 0; start < number_of_pitches; start += 1){
    x = ((R * x) * (1 - x))
    pitches[start] = x * 100
    current_pitch = cpsmidi(trunc((pitches[start]) + 0.5))
    MSITAR(start, duration, amplitude*envelope, current_pitch, pluck)
}

```

*Score file 50: MSITAR( ) pitches from logistic map*

Now if that isn't the apt soundtrack for a nightmare sequence in a horror film, I don't know what is.

Without a doubt, random parameters and algorithmic procedures play a large role in much of the music being composed today. With an abiding sense of curiosity and willingness to research and apply these processes in your own work, I hope that you will find as much joy and wonderment as I do while composing. There is no doubt that leaving some musical parameters and elements to chance greatly enhances our experience with our music and yields surprising, if ultimately fitting, results.

## **SONATA VI: ALGORITHMIC ETUDE**

Compose a one to three minute work that includes one, if not two algorithmic processes. You might spend some time searching for an interesting equation or process that can be implemented in MINC and applied to at least two different RTcmix instruments. Moreover, utilize at least three different random processes or weighted random distributions in your finished composition.

## || Interlude VI: Pitch transformations ||

Interlude III left us with a complex Python script that outlined a series of pitch transformations using the twelve tone system. In the early twentieth century, Viennese composer Arnold Schoenberg codified a new method of composition that would finally divorce Western classical music from its evolution of tonality and fully liberate chromaticism, which is referred to as twelve tone music, serialism, or dodecaphony.

No worries if that all sounds super confusing, as we'll take time to fully explain the notion of pitch and some of the meaningful ways — including Schoenberg's method — that we can manipulate pitches in our music using RTcmix.

Let's start by rehashing the beginning of Sonata III, where introduced the major scale. Recall that when looking at a piano, we see black and white keys, with the black keys laid out in alternating groups of two and three. Looking at the group of two black keys, we know that the white key adjacent to the leftmost black key is C. Playing from that particular key up to the next C produces our requisite sequence of half steps and whole steps that constitutes a major scale with the letter names C, D, E, F, G, A, B, and again, C representing each individual pitch.

The low C to the higher C spans an octave and if we include all of the black and white keys in that octave, we have a total of eleven pitches.<sup>101</sup> Play them all in succession and you've just played a chromatic scale!

If we put each of the notes with their letter name into an array, we might get something that looks like this:

```
notes = {"C4", "C#4", "D4", "D#4", "E4", "F4", "F#4", "G4", "G#4", "A4", "A#4", "B4"}
```

However, let's abstract our notion of pitch by divorcing each from its letter name

---

<sup>101</sup> Twelve, if you count the repetition of C once you get to the top.

and instead give each note in the chromatic scale a numerical designation.

```
notes = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

Now we can see that each C will equal 0, each F will equal 5, each B will equal 11, and so on. This is an important step, as we want to get used to seeing 6 and immediately thinking F#, or knowing that Bb equals 10, and we're not super concerned with octave designation right now, and want to be able to instead think of these notes as abstract entities that fit within one octave.<sup>102</sup>

There are many different ways that we can manipulate this array or list of pitches.<sup>103</sup> We can divide our pitches into subsets of the full gamut, creating chords. For example, a C major chord could be constructed out of our array like this:

```
notes = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
c_major_chord = {notes[0], notes[4], notes[7]}
```

Or, we might randomly select three notes from our array to make our chord.

```
rand()
chord = {}
notes = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
array_length = len(notes)

for(i = 0; i < 3; i += 1){
    new_note = notes[rand(0,array_length)]
    chord[i] = new_note
}
print(chord)
```

---

<sup>102</sup> For those who don't have a background in music theory, the # and b can *enharmonically* represent the same pitch, thus D# and Eb represent the same note on the piano, but we're going to think of it as 3.

<sup>103</sup> Recall that we construct arrays in MINC and lists in Python.



Our code also works well in Python.

```
from rtmix import *
import random
srand()
chord = []
notes = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
list_length = len(notes) #returns 12, so take away one to get indices 0-11
print(list_length)

for i in range(0,3):
    new_note = notes[random.randint(0,list_length - 1)]
    chord.append(new_note)

print(chord)
```

If we'd rather avoid repeated notes, we can always call upon `spray_init( )`, but this will work well for now.

One effective composition method of manipulating pitches is to play your particular succession in reverse.

```
pitch_gamut = {0, 1, 2, 3, 4, 5}
array_length = len(pitch_gamut)
gamut_reversed = {}
for(i = 0; i < array_length; i += 1){
    gamut_reversed[i] = pitch_gamut[((array_length - 1) - i)]
}

print(pitch_gamut)
print(gamut_reversed)
```

And in Python:

```
from rtmix import *
pitch_gamut = [0, 1, 2, 3, 4, 5]
```

```
def reverse(pitches):
    return pitches[::-1]

gamut_reversed = reverse(pitch_gamut)

print(pitch_gamut)
print(gamut_reversed)
```

Let’s take a pause to focus on Python for a second. See how thrifty the Python code looks? This is because we were able to create a function that reverses lists — thus precluding our need for a loop — using a handy device called *list slicing*. All arrays in MINC can be indexed, but in Python, we can uniquely index our lists both forwards *and* backwards. So for example if we wanted to access the first element in our gamut of pitches — just like in MINC — we’d use something like this:

```
pitch_gamut = [60, 64, 67, 68, 69, 70]
first_pitch = pitch_gamut[0]
```

We can access “68” in two ways, either by our familiar method of reading from left to right starting at 0, or now backwards.

```
pitch = pitch_gamut[3] //index 3 = 68
pitch = pitch_gamut[-3] //returns 68 as well
```

What if we wanted pitches 64, 67, and 68 in our gamut? For this, we will “slice” the list.

```
chord = pitch_gamut[1:4]
```

Note how we’re gathering elements one through three in the list. It’s a bit of a misnomer, since the list slice looks like it’s saying “take elements one through four,” so you might at first blush expect to return [64, 67, 68, 69]. Instead it’s best to think that we’re starting with `pitch_gamut[x]` and ending with `pitch_gamut[x - 1]`, which can be

confusing if you've been rocking out on MINC up until now.

To access our list verbatim, we can simply write:

```
repeat_gamut = pitch_gamut[:]
```

If you leave one end of the slice empty, Python will return only the first or last set of elements that you are looking for, respectively.

```
first_three_pitches = pitch_gamut[:3]
last_three_pitches = pitch_gamut[3:]
last_two_pitches = pitch_gamut[4:]
```

We'll return to some heavier Python when the time comes, but let's get back to pitch transformations in general. We've looked at denoting subsets of our gamut of pitches and now utilizing them in reverse, but we can also invert them.

Inversion of pitches is a simple notion that merely involves addition and subtraction, but with a caveat called mod-12 arithmetic. When we describe pitches in western music, you'll find that we only represent them with the letters A, B, C, D, E, F, and G. Rather than call the next note in this succession "H," we instead *wrap around* to "A" again and repeat our succession. Similarly, in mod-12, we wrap any results above 12 back into the range of 0 - 11, which of course corresponds nicely to our notion of chromaticism. For example,  $8 + 5 = 13$  in our numbering system, but in mod-12,  $8 + 5 = 1$ . To understand this, you can think of reaching 12 and then wrapping your way back down to 1, just like we do on a clock face. Similarly,  $8 + 4 = 0$  in mod-12.

Now we're ready to invert our pitches and to do so we'll need the `mod( )` function in MINC and the `math.fmod( )` function in Python.

```
gamut = {0, 5, 4, 9, 11}
gamut_length = len(gamut)
invert_gamut = {}
```

```

for(i = 0; i < gamut_length; i += 1){
    invert_gamut[i] = mod(((gamut[i] - 12) * -1), 12)
}

print(gamut)
print(invert_gamut)

```

And again in Python.

```

from rtmix import *
import math

pitch_gamut = [0, 5, 4, 8, 2]

def invert(pitches):
    return [math.fmod(((x - 12) * -1), 12) for x in pitches]

print(pitch_gamut)
print(invert(pitch_gamut))

```

Transposing pitches is another useful method for playing around with pitch and is easy to implement now that we have mod-12 in our tool belt.

```

pitch_gamut = {0, 5, 4, 8, 2}
gamut_length = len(pitch_gamut)

transpose_gamut = {}
transposition = 2 //transpose by a whole step, or Major 2nd

for(i = 0; i < gamut_length; i += 1){
    transpose_gamut[i] = mod(pitch_gamut[i] + transposition, 12)
}

print(pitch_gamut)
print(transpose_gamut)

```

Now for Python.

```
from rtcmix import *
import math

pitch_gamut = [0, 5, 4, 8, 2]
transpose_gamut = []

transposition = 7 # up a Perfect 5th

def transpose(input_list, transposition_value):
    return [math.fmod(element+transposition_value, 12) for element in input_list]

print(pitch_gamut)
print(transpose(pitch_gamut, transposition))
```

Before we consider Schoenberg's method of serialism, let's review. We can declare arrays or lists of pitches and have the ability to transpose them by a given value. We can also write them backwards, or in reverse, and can invert them. As we now consider dodecaphony, we'll codify these transformations using some specific methods and place each of our lists or arrays of pitches into a suitable range for RTcmix to play back as MIDI notes.

For Schoenberg, composition with twelve tones eliminated the need to write music that centered around a central, tonal key area. Instead, his method relies solely on each of the pitches of the chromatic scale and one is not permitted to repeat a pitch until each note in the chromatic collection has been utilized. Thus, you can see how his mode of operation totally and completely liberates chromaticism as the central focus of the music.

We begin with a twelve note *row*, sometimes referred to as an *aggregate*, which we've up until now been calling array, list, or gamut. Just as we encountered with algorithms, a well-crafted, carefully considered aggregate row form will assist greatly in

the outcome of your work. So, let's throw caution to the wind and make a row using chance procedures!<sup>104</sup>

```

srand()
spray_table = 1
spray_size = 12
seed = trand(0,100)
spray_init(spray_table, spray_size, seed)

aggregate = {}

for(i = 0; i < spray_size; i += 1){
    aggregate[i] = get_spray(spray_table)
}

p_0 = aggregate
```

Here is my aggregate row form, which is also called *prime form*. It's customary to say that an non-transposed, prime row is called p0, hence our variable *p\_0*.

Next, we'll take our familiar code from above for writing our notes backward and utilize it to get the *retrograde row form*, or r0 if it's not transposed.

```

retrograde = {}
for(i = 0; i < spray_size; i += 1){
    retrograde[i] = aggregate[((spray_size - 1) - i)]
}

r_0 = retrograde
```

From there, we can also invert our pitches to get the *inverted row form*, or i0.

```

inversion = {}
for(i = 0; i < spray_size; i += 1){
    inversion[i] = mod(((aggregate[i] - 12) * -1), 12)
```

---

<sup>104</sup> We're going to accomplish all of this in MINC, since Python is going to get to shine again soon.

```
}
```

```
i_0 = inversion
```

The notion of *retrograde inversion row form* is the last component of the twelve tone method. We'll take the inverted row form and write it backward, designated as ri0.

```
retrograde_inversion = {}  
for(i = 0; i < spray_size; i += 1){  
    retrograde_inversion[i] = inversion[((spray_size - 1) - i)]  
}  
  
ri_0 = retrograde_inversion
```

So far so good! Here's the entire score file tidied up with some print statements to verify our work.

```
srand()  
spray_table = 1  
spray_size = 12  
seed = trand(0,100)  
spray_init(spray_table, spray_size, seed)  
  
aggregate = {}  
  
for(i = 0; i < spray_size; i += 1){  
    aggregate[i] = get_spray(spray_table)  
}  
  
p_0 = aggregate  
  
retrograde = {}  
for(i = 0; i < spray_size; i += 1){  
    retrograde[i] = aggregate[((spray_size - 1) - i)]  
}  
  
r_0 = retrograde  
  
inversion = {}
```

```

for(i = 0; i < spray_size; i += 1){
    inversion[i] = mod(((aggregate[i] - 12) * -1), 12)
}

i_0 = inversion

retrograde_inversion = {}
for(i = 0; i < spray_size; i += 1){
    retrograde_inversion[i] = inversion[((spray_size - 1) - i)]
}

ri_0 = retrograde_inversion

print(p_0)
print(r_0)
print(i_0)
print(ri_0)

```

*Score file 51: Twelve tone transformations in MINC<sup>105</sup>*

Our final task is to put these into a suitable range for MIDI. In Python, we can accomplish this by creating a new function for MIDI conversion, as we did in score file 21. I like to think of these as starting in octaves spanning C to C, so that I'm transposing into middle Cs range (60) or an octave higher (72) or lower (48). In MINC, you can simply add each requisite transposition into your for( ) loop while constructing your row forms.

```

rtsetparams(44100, 2)
load("MMODALBAR")

//----- MMODALBAR
start = 0
duration = 1
amplitude = 30000

```

---

<sup>105</sup> Pretty much score file 21 in MINC. Take a moment to compare and contrast the two score files. Are you starting to find a preference for one language over the other?



```

//---pitch stuff
spray_table = 1
spray_size = 12
seed = 12

aggregate = {}
transposition = 48
spray_init(spray_table, spray_size, seed)
for(i = 0; i < spray_size; i += 1){
    aggregate[i] = get_spray(spray_table) + transposition
}

reverse = {}
invert = {}
retrograde_invert = {}
for(i = 0; i < spray_size; i += 1){
    reverse[i] = aggregate[i - 1] + 60
    invert[i] = aggregate[i - 12] + 72
}

retrograde_invert = {}
for(i = 0; i < spray_size; i += 1){
    retrograde_invert[i] = invert[i - 1] + 60
}

hardness = 1.0
position = 1.0
instrument = 4
pan = 0.5

for(start = 0; start < spray_size; start += 1){
    MMODALBAR(start, duration, amplitude, aggregate[start], hardness, position,
              instrument, pan)
    MMODALBAR(start, duration, amplitude, reverse[start], hardness, position,
              instrument, pan)
    MMODALBAR(start, duration, amplitude, invert[start], hardness, position,
              instrument, pan)
    MMODALBAR(start, duration, amplitude, retrograde_invert[start], hardness,
              position, instrument, pan)
}

```

*Score file 52: Score file 21 in MINC*

Neither score files 21 or 52 might be the most musically edifying composition right now — just a roving series of MMODALBAR( ) chords built from dodecaphonic transformations — but you can now imagine all of the possibilities that you have at your disposal for further refinement. You could certainly transpose any of these row forms, change durations, amplitudes, instruments, add effects, etc.<sup>106</sup>

We'll sum up this sections with a return to Python and some of the incredibly useful pitch manipulating methods that you can implement thanks to Python's ability to define functions. The following examples were culled from Christopher Burns' "Compositional Algorithms" course at the University of Wisconsin-Milwaukee, which I had the pleasure of taking in 2007. The course was designed around the LISP programming language, which I've adapted to Python for use with RTcmix.

We can place conditional tests within our functions to return values based on their outcomes. For example, if I wanted to return the absolute value of a given number, I could create a simple test to see if the number in question is either positive or negative, knowing that tests for absolute value return the number's distance from zero, which will intrinsically be positive.

```
def absolute_value(x):
    if(x >= 0):
        return x
    else:
        return x * -1

print absolute_value(-5)
```

This is useful because we can now generate interesting lists of numbers that we can turn into any form of musical data that we wish, such as generating lists of numbers according to the Fibonacci sequence up to a determined location.

---

<sup>106</sup> If you transpose your prime form up two half steps, you are customarily creating p2. Similarly, if you transpose your retrograde form up five half steps, you'll have r5.

```

fibonacci_list = []
def fibonacci(x):
    if(x == 0):
        return 0
    elif(x == 1):
        return 1
    else:
        return (fibonacci(x - 1) + fibonacci(x - 2))

for i in range(0,10):
    fibonacci_list.append(fibonacci(i))

print fibonacci_list #first ten elements in the fibonacci sequence

```

From here, we can return to list slicing to hone in on any portions of the fibonacci sequence that we'd like to use in our score.<sup>107</sup> Here, we'll take our list of fibonacci numbers and return them in a palindrome.

```

def palindrome(input_list):
    return input_list + input_list[::-1]

print palindrome(fibonacci_list)

```

However, note that we've repeated the top number in our list, 34, as we appended the original list with its reversed form. To mitigate this dilemma, we can elide the repeated note all together.

```

def palindrome_elided(input_list):
    first_half = input_list
    reverse_list = input_list[::-1]
    second_half = reverse_list[1:]
    return first_half + second_half

```

---

<sup>107</sup> For example, take a moment to figure out how you would isolate the numbers 21, 34, 55, and 89 in the list of fibonacci numbers and convert them to frequencies in Hz for use in a WAVETABLE( ) instrument.

```
print palindrome_elided(fibonacci_list)
```

We've failed to thus far utilize repetition of our list of numbers, useful for our minimalists at heart.

```
def repeat(input_list, iterations):  
    return input_list * iterations
```

```
print repeat(fibonacci_list, 4)
```

The following score file example sums up our newly codified transformations using an aggregate twelve note row while also defining functions that will divide the row into *hexachords* and rotate the pitches in the hexachords to the left or to the right.

```
from rtcmix import *
```

```
sample_series = (0, 11, 6, 2, 9, 5, 10, 4, 7, 3, 1, 8)
```

```
def reverse(input_list):  
    return input_list[::-1]  
print reverse(sample_series)
```

```
def get_first_hexachord(input_series):  
    return input_series[:6]  
print get_first_hexachord(sample_series)
```

```
def get_second_hexachord(input_series):  
    return input_series[6:]  
print get_second_hexachord(sample_series)
```

```
def get_cheap_second_hexachord(input_series):  
    return reverse(get_first_hexachord(input_series))  
print get_cheap_second_hexachord(sample_series)
```

```
def transpose(input_list, transposition_value):  
    return [(element + transposition_value) % 12 for element in input_list]  
print transpose(get_first_hexachord(sample_series), 10)
```

```

def rotate_right(input_list, rotation):
    return input_list[rotation:] + input_list[:rotation]

print get_second_hexachord(sample_series)
print rotate_right(get_second_hexachord(sample_series), 1)

def rotate_left(input_list, rotation):
    return input_list[-rotation:] + input_list[:-rotation]

print get_second_hexachord(sample_series)
print rotate_left(get_second_hexachord(sample_series), 2)

```

*Score file 53: Complex transformations in Python*

## INTERLUDE VI: PITCH ETUDE

Compose a brief, one minute RTcmix etude that focuses on pitch and various transformations. Use two instruments that are unfamiliar to you or that you have yet to use by searching the online documentation. Think of your etude as a small *invention* for two voices, so that the pitches between your two instruments interact in a meaningful, *contrapuntal* fashion.

## || Sonata VII: RTcmix in Pure Data and Max/MSP ||

Visual programming languages afford the ability to create complex patches — akin to our score files or scripts — using a set of patchable objects, messages, and other tools designed for real-time control and signal processing. Pure Data (Pd) and Max/MSP (Max) are incredibly popular platforms for realizing your work using the visual programming paradigm and RTcmix has been adapted for use within either of these powerful software packages.

In the mid 1980s, Miller Puckette was at IRCAM in Paris, working on developing a suite of patchable tools for realizing live, interactive computer music which was eventually dubbed *Max* — in homage to Max Mathews — and has been further refined by David Zicarelli and made available, commercially, through his company Cycling '74. Once Max was able to realize the handling of digital signals, it was then called Max/MSP and after adding libraries of video processing tools, has been known as Max/MSP/Jitter, though “Max” or “Max/MSP” are interchangeable titles and in the computer music world, saying that you work with Max is very common and most anyone will know what you are referring to.

In 1996, Miller redesigned his code and began offering the open source program *Pure Data*, which he continues to maintain under the heading “Pd Vanilla” due to its set functionality, which differentiates it from “Pd-extended,” a community-curated version that offers a large set of external libraries, including the powerful GEM library for video processing. Whether you are working in Pd Vanilla or Pd-extended, most will know what you are using if you indicate that you are working with Pd.

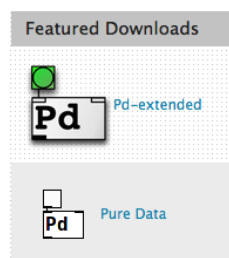
Despite their shared lineage, these two programs should really be thought of as separate entities. Due to its open source nature, Pd is — like RTcmix — completely customizable, and there is no shortage of documentation for learning how to write your own objects, add them to your library, or change the existing code that constitutes the program to personalize your own experience while using it. On the other hand, because

Max/MSP is commercially available, it is curated by a team of paid developers, and since the recent release of Max 7, boasts a powerful GUI interface that is predicated on user accessibility — especially geared toward beginning programmers — and a wide range of pre-designed patchable modules for quickly realizing work in signal processing. Please know that whatever flavor you choose between Pd and Max is of course up to you.

Personally, I enjoy working in both Pd and Max. I first learned visual programming on Pd and continue to enjoy relying on it nearly exclusively for my work in live, interactive performances. Because RTcmix was first developed for Max and has only recently realized functionality in Pd, there was a period of time where I focused my work on Max if I was going to utilize RTcmix in conjunction with playing live. Again, there is no shortage of tools available to the computer musician who is working today and after sampling freely from both Pd and Max in this Sonata, my hope is that you'll find a platform that works best for you, aids in your enjoyment while programming and patching your music, and assists in realizing your tasks quickly and efficiently.

We'll begin by downloading both programs. Because Pd is free and open source, you'll be able to enjoy it cost free for as long as you'd like. Max will offer a trial version that has full functionality. While you will eventually need to pay for Max, there are deep discounts for students and educators alike.

Head to <http://www.puredata.info> and make sure to download either Pd-extended or Pd Vanilla. The extended version is maintained by the Pd community at large and includes support for video processing and a host of custom tools. By downloading Vanilla, you'll get the version of Pd that is maintained by Miller Puckette.



By clicking on either link, you'll be prompted to link to a download for your platform and will begin the download process on the SourceForge page. Once complete, you'll receive a .dmg file that you can double-click on to run the Pd installer, which like the GUI versions of RTcmix, will do all of the work for you and won't require any work in the Terminal.<sup>108</sup>

Now, do the same for getting Max by navigating to <https://cycling74.com> and downloading Max 7.



The download process will again be straightforward and you'll now have Pd and Max ready to go in your /Applications. Feel free to place their icons in your Dock if you'd like to.

We'll also need the RTcmix *object* that runs for each program, respectively. We'll first get the version for Max by going back to the RTcmix web page and clicking on the "rtcmix~" link. The heading for Mac OS X Directories will point to a download link for the requisite .zip file.



Don't worry if that link says "max6" somewhere in its name, as it will run just fine in our Max 7 application.

When you downloaded Max, you might've missed the fact that you now have a folder in your /Documents called Max 7, or /User/Documents/Max \ 7. Within that folder are a few subfolders, one of which is called "Library." This is where you'll need to

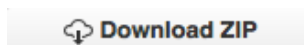
---

<sup>108</sup> For the purposes of this text, all Pd examples were completed using the Vanilla build.



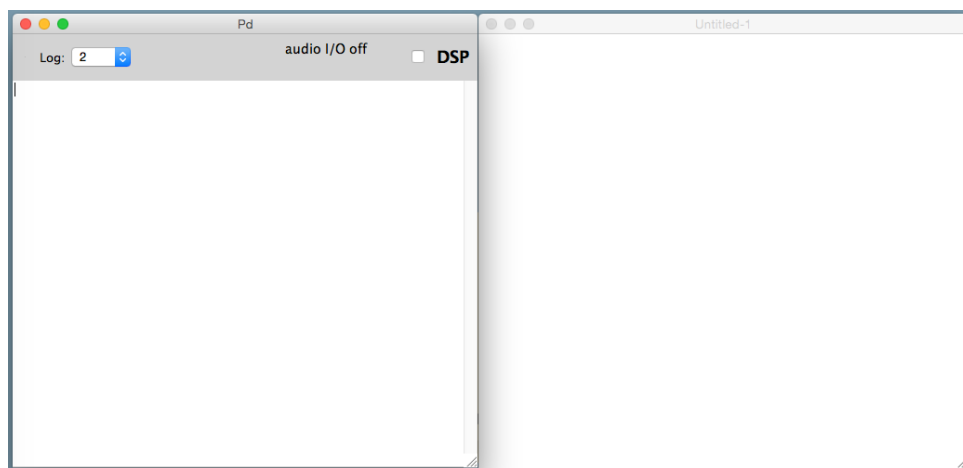
place the unzipped “rtcmix~\_1.92/osx/max6” folder that we just downloaded. The *raison d’être* of this “Library” folder is to house all of your so-called *external objects* and *libraries* so that Max can find them once we call upon them in our work.

Back to Pd. Navigate to <https://github.com/jwmatthys/rtcmix-in-pd> and click on the link in the lower right corner of your screen which denotes “Download ZIP.”

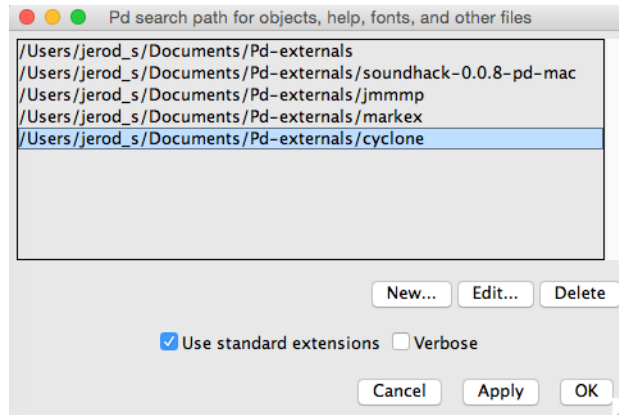


You’ll be downloading a folder called “rtcmix-in-pd-master” which will for now reside in your /Downloads. Create a folder in your /Documents called “Pd-externals” where you’ll then move the “rtcmix-in-pd-master” folder into.

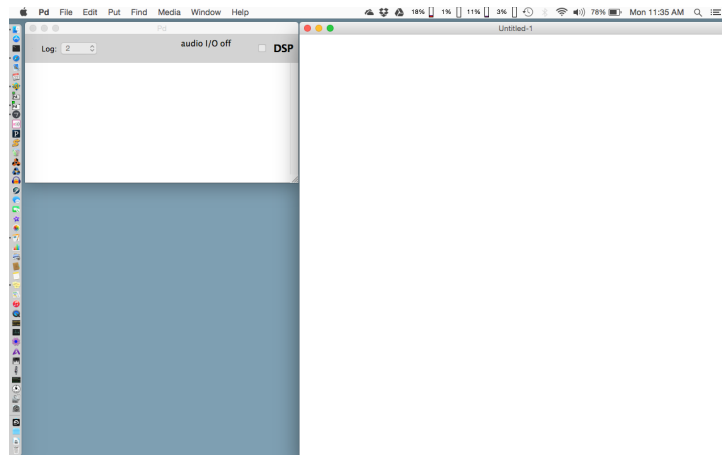
Go ahead and open Pure Data and then open a new patcher window by clicking *command-n* or head to the Menu bar and clicking File → New.



Before we can have fun with RTcmix in Pd, we have one last step. Pd needs to be able to find your “Pd-externals” folder, so click Pd-extended → Preferences → Path in the Menu bar and create a new path for Pd that points to the aforementioned externals folder, then click “Apply” and “OK” and you should be good to go.

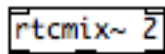


When you open Pd, you have two windows, one is the main Pd window, which depending on a variety of factors might already have some text printed to it, and the patcher window, where we will create and interact with our Pd patches. The Pd window exists to communicate with you if, for example, there is an error in your patch, but you may also print messages to it in order to verify that portions of your Pd patch are working correctly. I prefer collapsing the main Pd window a bit and maximizing the space I have on my Pd window for patching and we'll begin by creating our first object in the patcher window.<sup>109</sup>



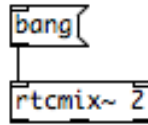
<sup>109</sup> There are way too many incredibly helpful tutorials, manuals, and YouTube videos dedicated to starting with Pd (or Max). Again, the scope of this text isn't to be able to use these programs *proficiently*, but *fundamentally* in order to use RTcmix in conjunction with them.

Type *command-1* and you'll notice that your mouse arrow has changed to a pointing hand with a rectangular box. This means that you are in *edit mode* and can create Pd objects, messages, number boxes, symbols, or comments. Pd runs in two modes, edit and *run mode*, which allows you to interact with your Pd patch once you've finished editing it. Type in "rtcmix~ 2" and click away from the rectangular box to create the object. Be sure to include a space between the tilde and 2.

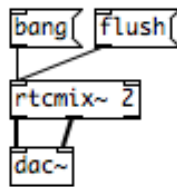


For some of you, this is your first Pd object! There are a few things to keep in mind with Pd (and Max) objects. The first is that they contain *inlets* and *outlets*. Inlets can receive a variety of things, such as audio signals or messages, and outlets will output the result of the processes that implemented within the Pd object itself. This is part of the beauty of a visual programming language: We won't be confronted with lines of text as we've been getting used to in MINC and Python, but we're now engaging with that code as it is encapsulated within a tidy, graphical object. In addition, many Pd objects have a namespace (like rtcmix~ or metro or + or moses) and an *argument*, which we've denoted as 2 for our object. We did this because we are specifying two so-called *signal* outlets that you can think of as functioning akin to the number of outlets we specify in `rtsetparams( )`.

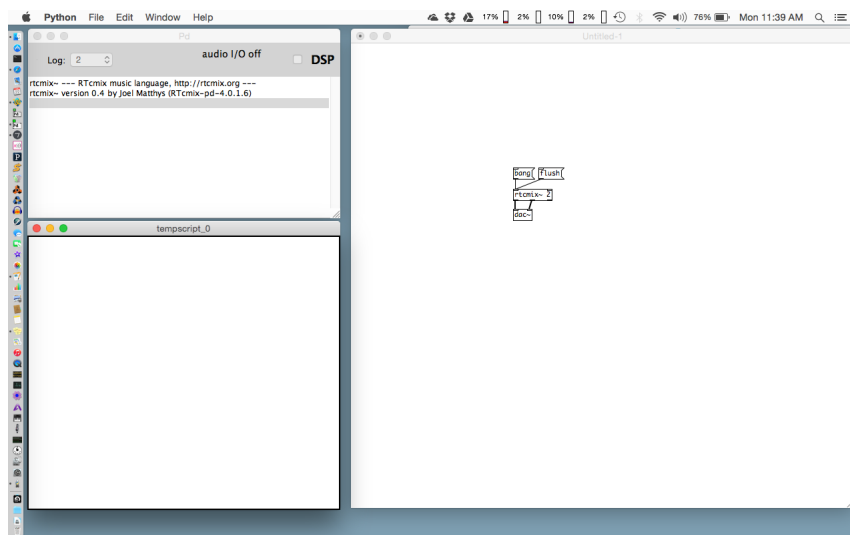
Practice getting between edit mode and run mode by typing *command-e*. As you move the cursor around, you'll notice that edit mode features our little hand icon and run mode retains the familiar arrow. In edit mode, type *command-2* to create a message and type in "bang." Click away from the message (which isn't rectangular in shape but looks like a tiny flag) to create it and hover over its outlet. You should notice a little target circle appear. Click and hold, then drag your cursor to create a patch line that can then be connected to the leftmost inlet of the [rtcmix~] object.



Create another message (*command-2*) that states “flush” and connect that to the leftmost inlet of [rtcmix~] as well. Lastly, create a [dac~] object (*command-1*) and connect the two signal outlets of the [rtcmix~] object to its corresponding inlets. The [dac~] object stands for digital-to-analog converter and translates the digital stream of data coming from our computer to acoustic energy that we can hear via our speakers or headphones.

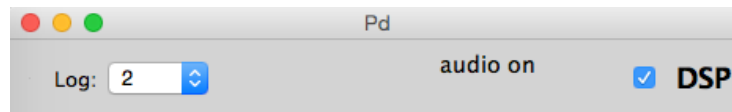


Perfect. Now we are able to go to run mode (*command-e* again if you aren’t already there) and click on our [rtcmix~] object. A new editing window should pop up where you’ll write all of your MINC code.



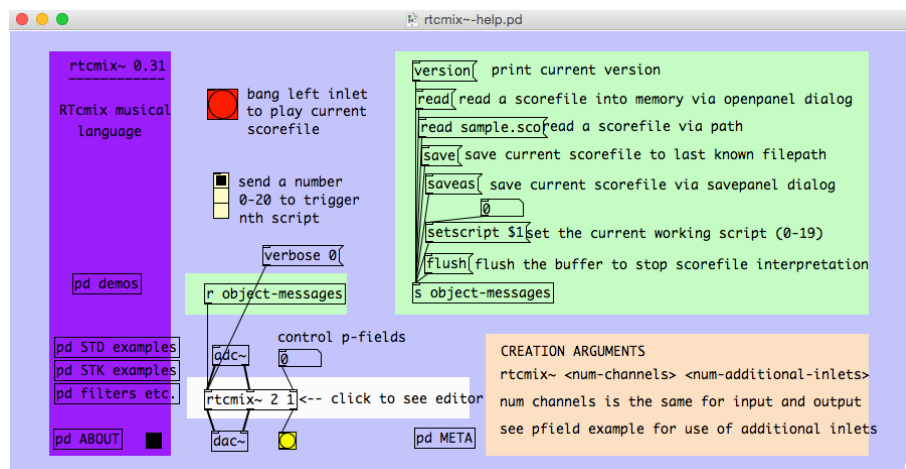
Feel free to copy and paste a completed score file here or start from scratch and build a simple oscillator like I’ve done. It’s important to remember two things at this point. First, you no longer need to write `rtsetparams( )` or use `load( )` because that functionality is built into the Pd object itself (same goes for Max, of course). Also, we need to *close* this window after we’re done editing to actually load the MINC code into the object itself.

Next, go to your main Pd window and click on the box next to “DSP” so that we can begin processing audio.



We are ready to hear our score file once we click on the “bang” message that we created earlier. If all goes well, you should hear your RTcmix script running in Pd!<sup>110</sup>

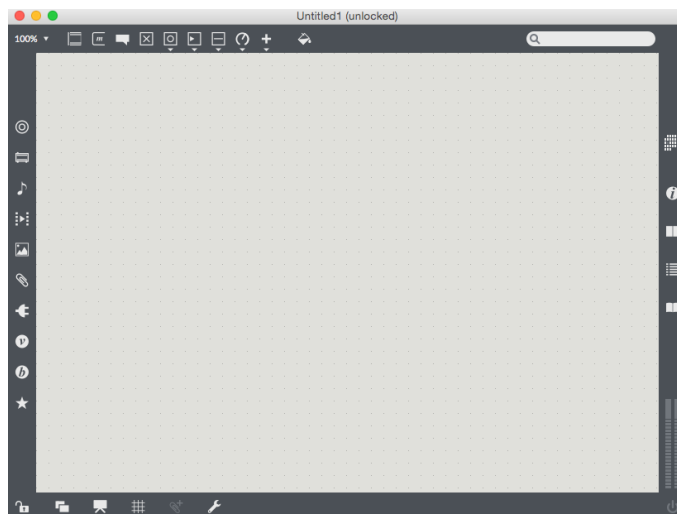
You can either let the script run its course or click on “flush” to stop rendering it. Now that we have sound working in Pd and have heard our first script, right-click on the `[rtcmix~]` object and select “Help” from the list of options.



<sup>110</sup> And such a let down, if not. The likely culprit is found in the top Menu bar under Media → Audio Settings. Make sure that these indicate your built-in input and output, or your audio interface if you are using one. Still nothing? Double check to make sure that you created “bang” with a message (*command-2*) and not an object (*command-1*).

From here, you'll be confronted with the object's *help file*, which will introduce you to a number of options for using [rtcmix~] successfully. We're not going to cover all of these options verbatim quite yet, but suffice it to say that now that we are in the visual programming paradigm, you should feel free to click away on any of the messages or objects that are posted and let the tutorial presented in the help file answer any pressing questions that you might have at this point.

We will instead shift our focus to Max/MSP, so go ahead and open Max 7.<sup>111</sup> Type *command-n* and you'll see that the presentation for Max is quite a bit different than Pd. Max opens a patcher that is full of a variety of toolbars on each side of its window.



We won't of course outline these tools in detail, but know that hovering over them will indicate, very briefly, what each of them are and for the curious, the little icon that looks like an open book on the right side of the window is a great place to start for

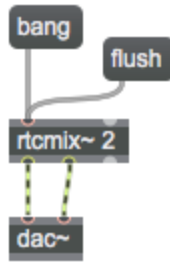
---

<sup>111</sup> Take a moment to save your Pd patch by typing *command-s* and save the .pd file anywhere that you'd like. You might find it useful to save your Pd and/or Max patches to a folder that includes all of your score files created thus far in conjunction with our work, so that it is easy to find them and call upon them when you have your Pd or Max patch running, since they'll be in the same directory.

mini-lessons about Max itself.<sup>112</sup> I might add, however, that typing *command-m* will access the Max window, which prints errors the same way that the Pd window does, which can be extremely helpful for debugging your code. You'll see that this window is also accessible by clicking on the icon above the open book that we just mentioned on the righthand side of the patcher window.

The methodology for Max is exactly the same as it is for Pd. We're going to create a variety of patchable objects and messages to realize our work in conjunction with RTcmix. Let's start by rebuilding our basic Pd patch that will play RTcmix score files.

To create an object in Max, type *n*. You'll create messages by typing *m*. By typing *command-e*, you'll toggle between edit mode and run mode, which is verified by the little lock icon on the bottom of your patcher window.<sup>113</sup>



As we did earlier, click — well, double-click this time — on the [rtcmix~] object while in run mode and a scripting window will open. Type or copy/paste some MINC code into that window and close it to load the code into the [rtcmix~] object. To get sound, click on the power button in the lower right corner of the patcher window and

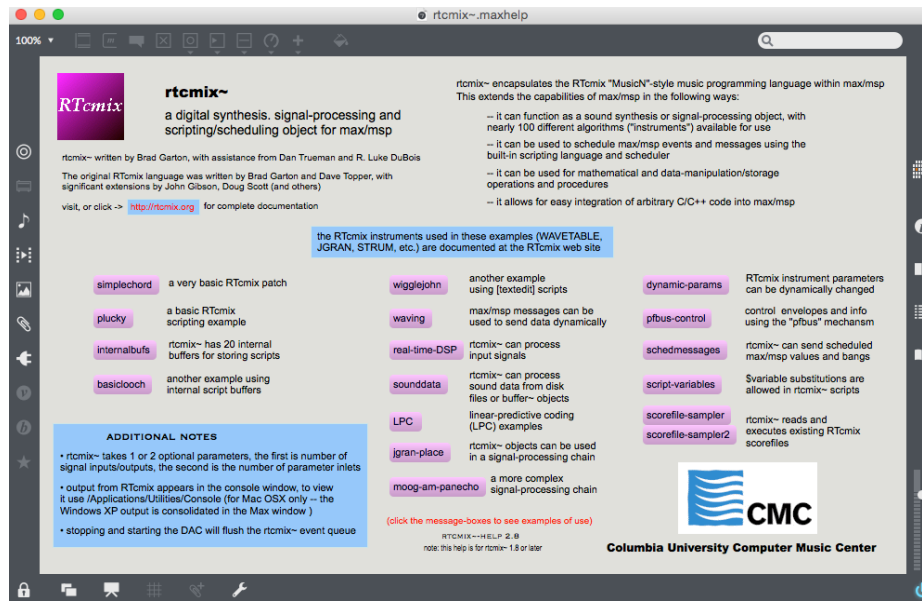
---

<sup>112</sup> Seriously, these are really fun, include step-by-step video(!) instructions and speak to the mantra of making Max approachable to the absolute beginner.

<sup>113</sup> You can also click on this lock to toggle between the two modes.

when you're ready, click on the “bang” message to hear your work!<sup>114</sup>

So far so good! Now while in edit mode, right-click on the [rtcmix~] object and open up its corresponding help file.



I highly encourage you to visit each of the modules contained within the help patch and for now, we are going to focus on the important concept of *internal buffers*.<sup>115</sup>

Until now, we've been utilizing the [rtcmix~] in a singular fashion. That is, we've only loaded one script at a time into its *buffer*, which has the ability to recall up to twenty buffers at a time. Think of it this way: You'll be able to save up to twenty score files into one [rtcmix~] object at any given time! After we understand how to implement this functionality, we'll see why it is useful for our work.

Start by creating a new message in your existing Pd or Max patch, called “setscript 0” and then connect it to the leftmost inlet of the [rtcmix~] object. This way,

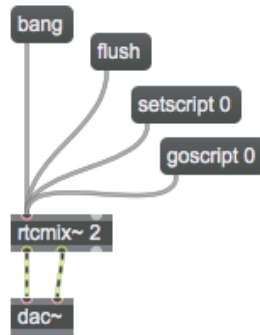
---

<sup>114</sup> No sound? Head to the Menu and navigate to Options → Audio Status and make sure you are processing sound through your internal speakers or audio interface. The VU meters in the lower right corner will verify if sound is being processed and can turn your volume up or down as well.

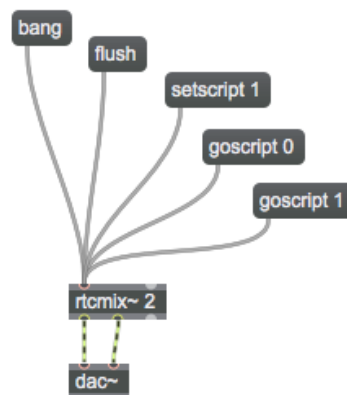
<sup>115</sup> Feel free to execute the following code in either Pd or Max.



we've stored our current score file — the innocuous sine wave or something that you copied and pasted — to buffer 0. Then, create another message, this time called “goscript 0” that will play buffer 0 in just the same way that “bang” did earlier.



In run mode, first click on “setscript 0” and then click on “goscript 0” and you should hear your score file again. Double click on the [rtmix~] object to reopen the scripting widow and create a new script within it and when you are finished, close the window. Click “bang” and you’ll hear your new script, which we are going to store into a new buffer. In edit mode, change the message for “setscript” so that it reads “setscript 1” and then click on it while back in run mode. Then, create a new message called “goscript 1” and connect it to the leftmost inlet of [rtmix~].



We're now able to click on "goscript 0" to hear our original score file and "goscript 1" to hear the newest script.<sup>116</sup>

To better answer the question of why storing multiple buffers is useful or why, even, we're bothering to play back RTcmix scripts in a new paradigm when we've been doing just fine up until now, we need to understand why Pd and Max are useful and why they are so popular in the world of computer music.

We've yet to touch on the notion that while useful as a sound generating platform, we can — and most definitely should — present our work in RTcmix in a live performance setting. While the world of electronic music and more specifically electronic music using computers is relatively young, it is today very common to see and hear live performances where laptops and speakers are just as valid an expressive medium as the piano or violin. While my undergraduate days included an incredible number of hours on my trumpet, when people invariably ask me, as a musician, "So what do you play?" I always respond with, "Laptop computer."

Pd and Max are beautiful programs to be sure, but one of their more outstanding features is their ability to assist you with transforming your laptop into the custom instrument that you might be looking to establish. For example, I've been typing and typing and typing away to create this book, but what if I used all of these keys on my keypad to trigger notes or events, just the same way that piano keys trigger various pitches. Could I create a basic instrument that will allow me to control the [rtcmix~] object in a meaningful way? Well, Pd and Max have a rich array of powerful objects to assist you with realizing your live work — or any work, for that matter — and let's get to answering the above question using a few of them.

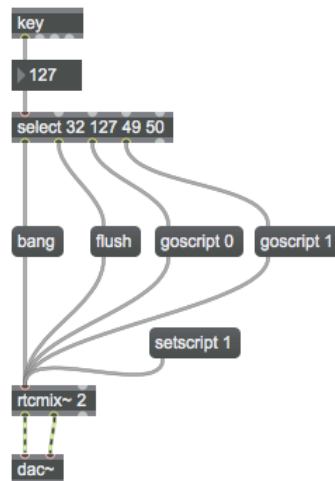
Somewhere above all of the code in your patcher window, create a new object called [key] and leave room to create a few more items below it. Now, underneath that

---

<sup>116</sup> It's customary to utilize *comments* in your patch window to help you remember what it is that you did while working on your Pd or Max patch. In Pd, type *command-5* to create a comment and place it next to the object or message in question. For Max, typing *c* will work.

[key] object, create a *number box* by clicking *command-3* in Pd or *i* in Max.<sup>117</sup> Connect the outlet of [key] to the inlet of the number box, get back into run mode, and type the numbers one through zero at the top of your keyboard. You'll notice that each key has a corresponding number coming out of the [key] object, which we're going to use to trigger the various messages we've already written into our patch.

Create a new object, [select], as well as a number of arguments to correspond with the numbers that are generated by the [key] object. When [select] receives a number in its inlet, it produces a *bang* — which we can think of as a trigger or onset that will do something useful for us — to its corresponding outlet. For example, I would like to trigger my first score using the “1” on my keyboard and the second score file using the “2.” Therefore, I'm going to need to use [select 49 50] as my object with corresponding arguments. I'll take it a step further, however, and include the space bar for playing the currently stored buffer and the delete key for stopping score file rendering using the “flush” message.<sup>118</sup>



<sup>117</sup> Quick side note. Max differentiates between floating-point numbers and integers, so always be cognizant of this fact. Type *i* for an integer number box and type *f* for a floating-point number box. The Pd number box can compute either of the two types. Be extra careful in Max if you're trying to do a little bit of math (again we're a little past the scope of this text, but it's good to know anyway) as you'll need to create a [+.] object for adding numbers that might be floats, while the [+] object will compute integers and truncate floating-point numbers in the same way that trunc( ) does in RTcmix.

<sup>118</sup> Those of you working in Pd will note that the delete key corresponds with 8 and not 127 as it does in Max.

You can imagine the versatility that we can create with this patch simply by storing enough score file buffers into our object to correspond with the top row of numbers on our keyboard. If you'd like to play with more than twenty buffers, you'll need to create a new [rtcmix~] object that can be connected to the same [dac~] object and filled with its own scripts.<sup>119</sup>

Both Pd and Max have their own set of powerful signal processing objects and you'll greatly enhance your experience with these programs when you utilize their DSP tools in conjunction with RTcmix. Earlier we discussed the notion of ring modulation, which is very easy to implement in Pd or Max and can add another layer of complexity and flexibility to our patch.

We'll transition back to Pd, but again, all of this can be done in Max using the same objects and number boxes, albeit with the caveat of specifying floating-point numbers or integers.

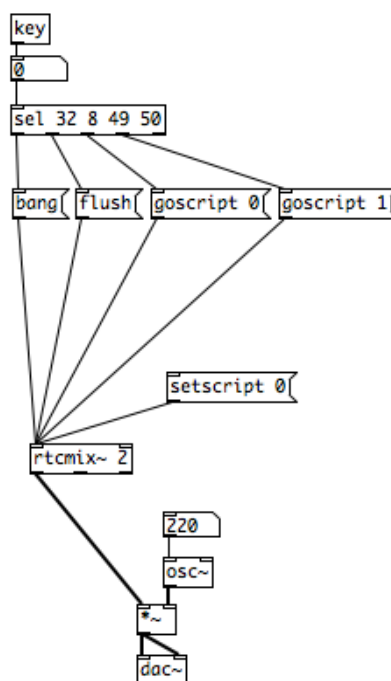
Disconnect the patch lines from the outlets of [rtcmix~] and [dac~] and create a bit of space between each of them. We're going to add the ring modulation effect after [rtcmix~] using the same methodology that we are familiar with using bus\_config( ), albeit with some enhanced control thanks to our visual programs. Recall that ring modulation multiplies a complex incoming signal with a simple modulating signal to produce sum and difference tones, or sidebands.

Both programs include a suite of tools for arithmetic and you can work with *control* data or *signal* data. Any object with the “~” following it will manipulate signals, which is evidenced by the look of their outlets and patch lines. Create a [\*~] object to multiply signal values and connect one of the outlets of [rtcmix~] into its leftmost inlet. Next, you'll create the [osc~] object in Pd or the [cycle~] object in Max to create a sine wave oscillator. Connect your sine wave to the rightmost inlet of [\*~] to complete the multiplication. Then, connect a number box to the inlet of your [osc~] or [cycle~] object.

---

<sup>119</sup> In edit mode, you can highlight entire portions of your patch by clicking, holding, and dragging to surround your selection. Then, use *command-d* to batch duplicate, but note that your internal buffers will also be copied into the new [rtcmix~] object.

Finally, connect the outlet of [\*~] to both of the inlets for the [dac~] object.



Double click on a number box, type in the desired value (in this instance, the frequency of our oscillator), and hit enter. This will store that value to our oscillator, but we can also click on the number box, hold, and drag up and down to sweep through our varying sidebands. Pretty fun!

Suffice it to say that with some research and practice, you'll quickly gain a great deal of facility on Pd and/or Max, which will be especially useful for your work with RTcmix. These two programs greatly enhance the live performance experience, as they are particularly adept at transforming your laptop into a completely customizable instrument. Moreover, they are incredibly facile at controlling MIDI data, so mapping your new USB keyboard or control surface is a cinch.<sup>120</sup>

Our next Interlude will continue our work with Pd and Max and the ways in

---

<sup>120</sup> It should be noted, however, that RTcmix contains this functionality via `makeconnection("midi")`, which we'll explore in our Postlude on customization.

which we can use their objects and messages to control p-field data in our RTcmix scripts in real time as well as ways that we can incorporate acoustic instruments into the fold for potential electroacoustic works.

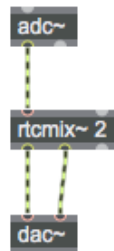
## **SONATA VII: PD OR MAX PATCH**

Using the principles outlined in this Sonata, create a Pd or Max patch that includes at least three stored buffers and is mapped to your keyboard in a way that spells out your initials or first name. Thus, it will be possible for someone to type your initials or name into the patch and hear your corresponding RTcmix sounds in a meaningful, well-sculpted musical etude.

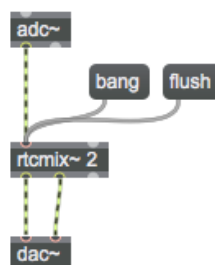
## || Interlude VII: Interactivity ||

We've yet to explore the critical world of interaction with live, acoustic instruments using RTcmix. With our newfound facility in Pd and Max, we'll be able to take live input from a microphone and process it via the [rtcmix~] object, which can be controlled or further processed, depending on our desired implementation. We're going to learn how to capture audio and update p-field parameters from Pd and Max, and finally utilize all of our tools to create a basic, performance-ready patch.

Earlier, we were introduced to the [dac~] object, which translates digital data to analog signal. The [adc~] object does the opposite: It takes real-world acoustic energy captured by your microphone and translates it to streams of digital data for processing in our patches. Start with a new Pd or Max patch and create an [rtcmix~] object that is ready for stereo output. Then, create an [adc~] object and connect its leftmost outlet to the leftmost inlet of [rtcmix~]. Finally, create a [dac~] object and connect the outlets of [rtcmix~] to [dac~] the same way we did earlier.



We'll also need “bang” and “flush” messages to play our score file once we have sound ready to process.



You are free to code your score files in your preferred text editor, or you can write them directly in the scripting window that pops up once you click on the [rtcmix~] object. I will say, however, coding in this window runs the risk of losing your work should Pd or Max crash.

Up until now, we've limited our incoming audio to sound files accessed via rtinput( ) pointing to files on our computers. However, now that we are in the Pd/Max paradigm, we can utilize rtinput("AUDIO") to capture audio that is incoming via the [adc~] object. Remember that we don't need to call upon rtsetparms( ) or load( ) while using the [rtcmix~] object, so your first line in your score file can be a call to rtinput( ).

```
rtinput("AUDIO")
```

Depending on how you've configured your system in Pd or Max to handle incoming sound, you'll now be able to process live audio. If you haven't already done so, double check your Audio Settings (Pd) or Audio Status (Max) to make sure that incoming sound is coming from your internal microphone or external, standalone audio interface with a microphone connected to it.

We'll begin with a simple processing example, so let's add a bit of delay to our voices as we speak into the microphone.

```
rtinput("AUDIO")
start = 0
instart = 0
duration = 30
amplitude = 1.0
delay_time = 1.0
r_ch_amplitude = 1.0 // right channel amplitude relative to left channel
DEL1(start, instart, duration, amplitude, delay_time, r_ch_amplitude)
```

#### *Score file 54: Simple delay of incoming signal*

Before closing the scripting window and clicking on the "bang" message to start processing sound, make sure that you are processing audio by checking the DSP button



in Pd or the power button in the lower right corner of your patching window in Max. Also, be sure to wear headphones or the proximity of your internal microphone to your internal speakers will onset some wicked feedback that we don't want.<sup>121</sup>

If all goes well, you should be able to hear a delayed copy of yourself as you speak or sing into your microphone!

RTcmix will of course stop processing audio after 30 seconds, but what if we'd like to use this budding Pd or Max instrument's functionality indefinitely? While we could amend our duration within the script itself, we can call upon MAXBANG( ), which outputs a *bang* to the rightmost outlet of [rtcmix~] at a specified time. For example, we'll amend score file 54 by indicating a duration of five seconds. We'll also use MAXBANG( ) to output a bang after five seconds, which we can then use to loop our score file.

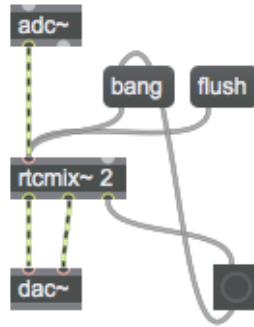
```
rtinput("AUDIO")
start = 0
instart = 0
duration = 5
amplitude = 1.0
delay_time = 1.0
r_ch_amplitude = 1.0 // right channel amplitude relative to left channel
DEL1(start, instart, duration, amplitude, delay_time, r_ch_amplitude)

MAXBANG(5)
```

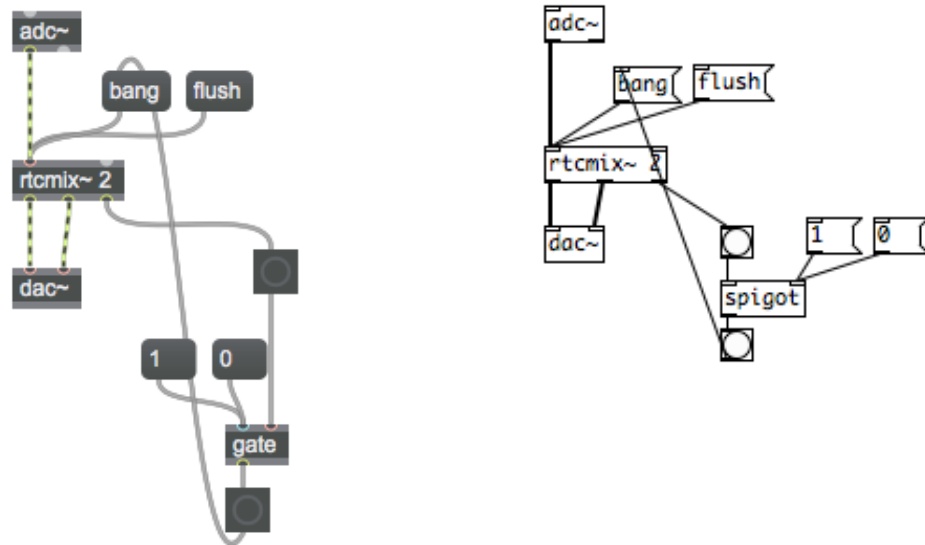
*Score file 54: Simple delay of incoming signal*

---

<sup>121</sup> Well, after working through Sonata V, maybe you do like the sound of grating feedback?



The object coming out of the right outlet of [rtcmix~] is a bang button, which you can create by typing *shift-command-b* in Pd or *b* in Max. The outlet of this bang button is connected to the inlet of the “bang” message above, which will create a loop. After clicking on the original “bang” message, our score file will loop indefinitely, which we’re going to want some control over. For now, head back to edit mode and disconnect the patch line from the bang button to and add the following to your patch.<sup>122</sup>



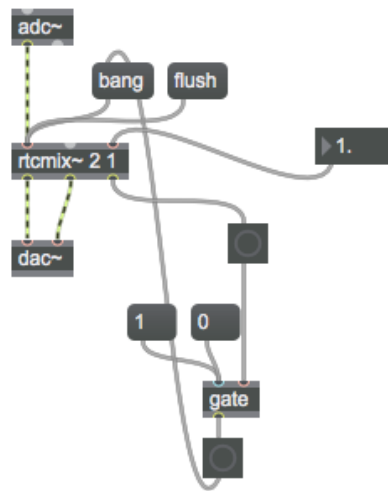
The [gate] object in Max and the [spigot] object in Pd are aptly named. Note

---

<sup>122</sup> I’ve included examples for both Pd and Max here as this is one area where the two programs will differ a bit.

how each respective object takes a bang button and the messages “0” and “1” into its inlets, though they are reversed depending on your program of choice. Functioning like a gate or spigot, these objects will allow data to pass through it as long as “1” is selected, and will stop data from passing with a “0” message. In this way, we can control whether or not our [rtcmix~] object will run indefinitely.

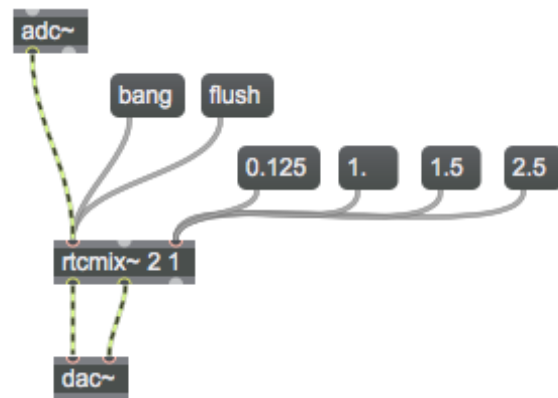
We’re familiar with updating p-field parameters using table data, however the [rtcmix~] object can be further enhanced to include a multitude of inlets that will accept control data from Pd or Max to pass to the internal p-fields. We’re going to need to create a new [rtcmix~] object with two arguments: “2” for the number of outlets that we’d like to use, and “1” for the number of inlets that we need. Moreover, let’s connect a number box to the rightmost inlet of [rtcmix~] and be sure to create it using *f* in Max for floating numbers.



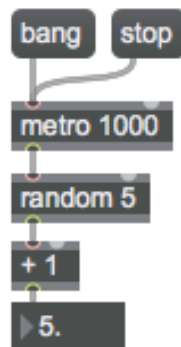
This new inlet can receive streaming numbers from Pd or Max by way of the `makeconnection("inlet")` command. For example, we’re going to control our delay time using this inlet and including the following in our score file.

```
delay_time = makeconnection("inlet", 1, 1.0)
```

We are now able to transition back to run mode and interact with our delay times.<sup>123</sup> For example, any number box can be altered by first double clicking on it, typing in your desired number, and clicking *enter*. Or, you may click on the box, hold, and scroll up and down to dynamically update values. If, on the other hand, you have a set series of numbers that you'd like to utilize as delay times, please feel free to create a set of messages that connect to the requisite inlet of the [rtcmix~] object.



We could use Pd or Max to generate random numbers to use as dynamic updates for any of our p-fields that use `makeconnection("inlet")` using the [random] object. Carefully reconstruct the following snippet of code and prepare to connect it to the rightmost inlet of your [rtcmix~] object.

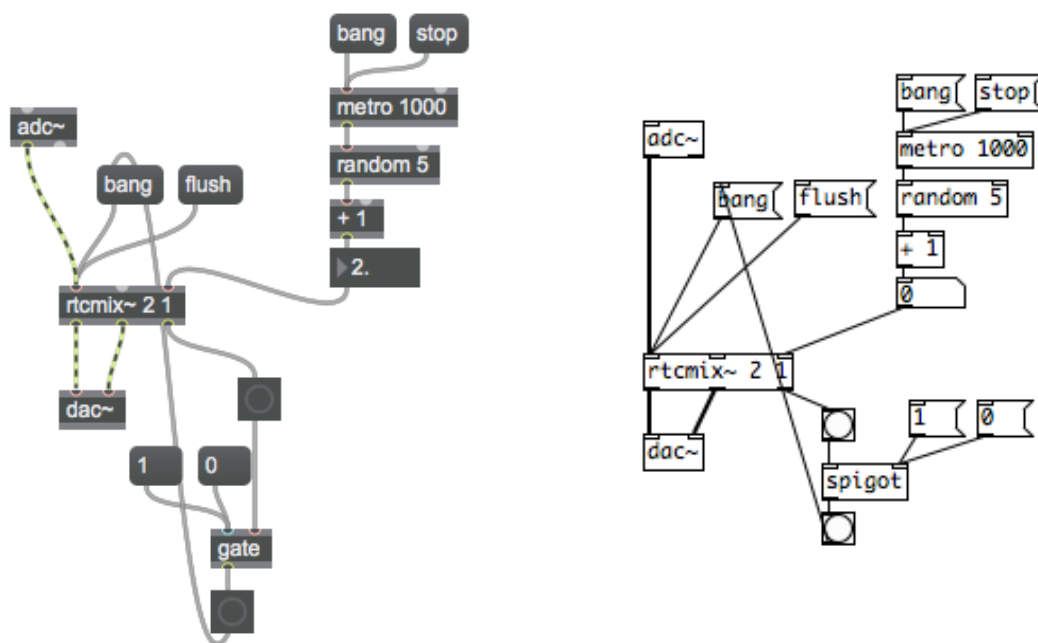



---

<sup>123</sup> Amend your MINC code to have a duration of 30, as well as `MAXBANG(30)` so that we can truly hear the changes in delay times.

Starting with two messages, “bang” and “stop,” we are able to control a metronome using the [metro] object. Time in Pd and Max is calculated in milliseconds, so initializing a [metro] object with 1000 will ensure that a *bang* is sent out of its output every second. The [random] object accepts *bangs* into its inlet and once it receives a *bang*, outputs a random number between zero and one less the number specified in its creation argument. In this case, I’m outputting numbers between one and four because I’ve added a [+] object to add one to each incoming number from [random]. Thus, my final range of values will be between one and five.

By turning this metronome on, we can speak or play into our microphones, continually loop our [rtcmix~] object to render indefinitely, and interact with random delay times.

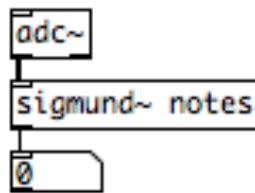


If this is your first time using Pd or Max, I’d like to propose a small challenge. Now that we have some practice with the methodology of connections in these two programs, see if you can devise a way that will start your [rtcmix~] script, open your gate or spigot, and start your metronome all at once using the *space bar*. Then, map

your *delete* key to turn all of those elements off at once.<sup>124</sup>

We're going to shift focus and concentrate instead on a powerful way to interact with the audio that is incoming via the [adc~] object.<sup>125</sup> One of the most powerful objects in both Pd and Max is [sigmund~], a real-time audio analysis tool that can return many important strings of data, including pitch.

If you'd like, go ahead and open a new patch window and save it to your folder of working Pd or Max patches. Then, code together this small snippet.<sup>126</sup>



Once you have this patched together, make sure that you are set to compute audio and sing a few pitches into your microphone. If all is working correctly, you should see some changes and know that [sigmund~] is tracking your pitches in real time and outputting them as values to your number box!

Now, we specified *notes* as an argument for [sigmund~], which outputs MIDI note values and only shifts values once a change in pitch is interpreted. Change the creation argument from *notes* to *pitch* and observe the difference in the stream of values.

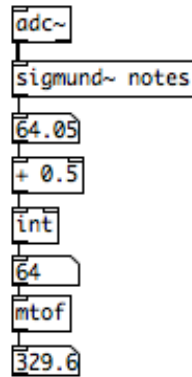
Let's revert back to *notes* and ensure that our values will be rounded up to the nearest integer value. We'll also translate those MIDI note values to Hz using the [mtof] object.

---

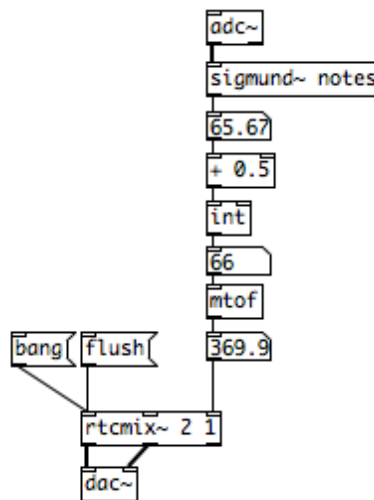
<sup>124</sup> Growing weary of the sheer number of patch lines turning into a cluttered mess of unruly spaghetti in your patching window? Take a moment to check out the [send] and [receive] objects and read their corresponding help files.

<sup>125</sup> Since most of our visual aids have been Max-centric, we'll include visuals in Pd for this string of concepts.

<sup>126</sup> Max users will want to use a floating-point number box here.



We'll also create an [rtcmix~] object and a [dac~] object and connect them accordingly. As a proof of concept, we're not going to connect the [adc~] object to [rtcmix~] just yet.



Turning our attention back to the MINC code within our [rtcmix~] object, we're going to introduce a new instrument, MBLOWBOTL( ), which we'll control using our pitch tracker.

```
start = 0
duration = 30
amplitude = 30000
pitch = makeconnection("inlet", 1, 261)
noise_gain = 0.25
```

```
pressure = 0.75
MBLOWBOTL(start, duration, amplitude, pitch, noise_gain, pressure)
```

*Score file 55: Controlling MBLOWBOTL( ) within [rtcmix~]*

MBLOWBOTL( ) is a member of the suite of STK instruments in RTcmix and replicates the sound produced by blowing over the top of a bottle. Note that sound is continuous and updates only when we sing or speak into our microphone because we are using the *notes* argument in [sigmund~].<sup>127</sup>

Connect the outlet of [adc~] to the leftmost inlet of [rtcmix~] and clear your scripting window so that we can explore live processing of our sounds via VOCODESYNTH( ), which is one of RTcmix's vocoders. The vocoder (voice encoder) was originally developed as a means of transmitting the human voice over telephone lines and is a popular synthesis technique for realizing interesting sound effects and is a common component in analog electronic music. A vocoder generates sounds based on the spectral characteristics of an input signal that is first analyzed and then synthesized using any of a number of filters.

```
rtinput("AUDIO")

start = 0
instart = 0
duration = 30
amplitude = 10

filters = 40
low_center_frequency = 220
interval_spacing = cspch(0.02) / cspch(0.0)
transposition = 0.00
```

---

<sup>127</sup> For those who might use or are exploring the *pitches* argument, take note: When no sound is produced and analyzed, a constant stream of -1500 is sent via the outlet of [sigmund~]. This can have terrible consequences (painful noise/clipping) if used as a p-field parameter. To mitigate this problem, try using the [clip] object, which constrains numbers to a set range, so something like [clip 20 500] might do the trick here.



```

bandwidth = 0.009
window_length = 0.001
smoothness = 0.98
threshold = 0.0001
attack = 0.001
release = 1.0
high_mix = 0.25
high_center_frequency = 2000

input_channel = 0
pan_frequency = makeconnection("inlet", 1, 5.0)
pan = makeLFO("sine", pan_frequency, 0,1)
carrier_wave = maketable("wave", 1000, "tri")

scalecurve = maketable("curve", "nonorm", 1000, 0.0,0.25,1.0, 1.0,1.0)

VOCODESYNTH(start, instart, duration, amplitude, filters, low_center_frequency,
            interval_spacing, transposition, bandwidth, window_length,
            smoothness, threshold, attack, release, high_mix, high_center_frequency,
            input_channel, pan, carrier_wave, scalecurve)

```

### *Score file 56: Robot voice*

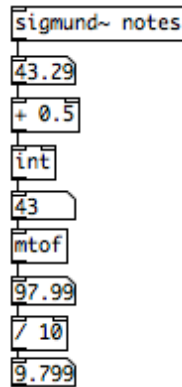
Before running this script, let's cover the constituent parts of VOCODESYNTH( ) and amend our Pd or Max patch a bit to best utilize our interaction with makeLFO( ).

We're using 40 filters in our sound, though you are of course free to play with that value and make note of the changes. After setting the center frequency of the lowest in the set of filters, we specify how the remaining center frequencies are determined, intervalically. In this instance, we are dividing our interval (cpspch(*interval*)) by cpspch(0.0), which will return stacks of the interval using equal temperament. You'll then see that we are not transposing our incoming sound and are using very minute values for our filter constructs.

The real interaction occurs in our p-field for panning. Using our makeLFO( ) command, we can dynamically update the frequency of our low frequency oscillator

using `makeconnection("inlet")`. Thus, as you sing or whistle or speak pitches that are higher into your Pd or Max patch, the frequency for our LFO will change accordingly.

In order to get pitches that are less than 20 Hz — which we want in order to truly be low frequencies — we need to divide our outgoing pitches from `[mtof]` by ten, or more depending on the range of your voice.

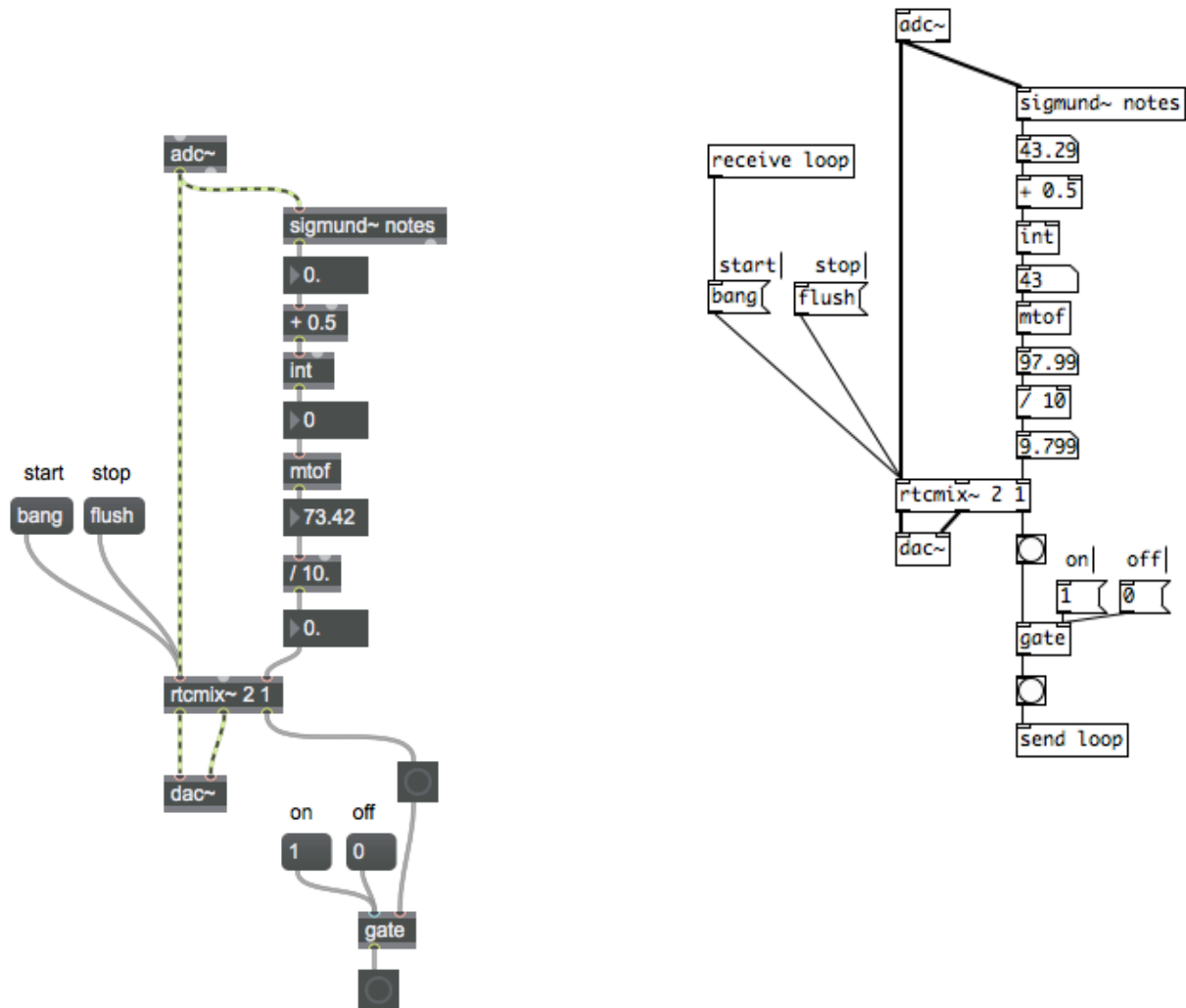


Make note that if the frequency for `makeLFO( )` reaches to or exceeds about 20 Hz, the LFO itself becomes audible and will modulate our vocoder sounds for yet another level of complexity.

The following figures sum up the concepts that we’ve covered with Pd and Max, and are included with all of the ancillary example score files to this book.<sup>128</sup> As with any examples or scripts, they are yours to rewrite, hack, utilize, or amend in any way that you’d like in order to better realize your sounds.

---

<sup>128</sup> They are .maxpat or .pd files, labeled as *-interactivity.maxpat* or *-interactivity.pd*, respectively.



We've merely scratched the surface of the multifarious possibilities afforded to anyone who is interested in composing interactive music using RTcmix in conjunction with Pure Data and/or Max/MSP. As noted earlier, Max 7 includes a wide array of learning tools built right into the program, including videos, tutorials, and of course help patches. For those who'll continue with Pd, the FLOSS (Free Libre Open Source Software) Manual is a tremendous resource, which you can find at <http://en.flossmanuals.net/pure-data/>. Moreover, while the site is about to become obsolete in April of 2016, I've found the Max Objects Database (which includes objects for Pd as well) to be very helpful in finding new objects that are written by the

community of users and might not come packaged in Pd or Max. Check out [http://  
www.maxobjects.com](http://www.maxobjects.com) for as long as the site is up and running.

## **INTERLUDE VII: INTERACTIVE PD OR MAX PATCH**

Compose an improvisatory work for your instrument or voice that uses pitch tracking to manipulate a p-field in your RTcmix script. You are free to make your piece as simple or complex as you like, though exploring graphic notation for your instrumental part is of course welcomed, as is using an array of RTcmix instruments via `bus_config( )`.

## || Postlude: Customization ||


RTcmix is a totally customizable experience and can be tailored to your individual tastes and needs. If we look all the way back to our work with installation, we noticed that there are three different ways to use the program. Moreover, some of you might have been using RTcmix in conjunction with Python for our entire journey together, rather than the standard MINC parser. Our work just took us through using RTcmix in conjunction with Pure Data and Max/MSP. In short, RTcmix shouldn't be thought of as software in the same way we view Propellerhead's Reason audio workstation, for example. Unlike proprietary software, we don't need to wait for the next X.x release to add to our enjoyment or finally include that bit of functionality we've been craving all along. Rather, RTcmix affords us the ability to use it on our own terms, using only, or adding, the features that we want to utilize. At its heart, RTcmix is a sophisticated suite of instrument libraries and commands that will assist us in realizing our music, whatever the platform may be.

We're going to wrap up our work with RTcmix by briefly introducing a few of the customizable features of the program itself. We'll begin by looking at some of its graphical possibilities via some ancillary downloads, including printing tables and using the mouse trackpad as a data source. Then, we'll explore MIDI and how we can use USB MIDI controllers in conjunction with RTcmix.

Earlier, we introduced the concept of envelopes — the ADSR envelope in particular — and included a few graphs to help us get a sense of each envelope's shape. This was especially useful when considering the difference between creating tables using `maketable("line")` and `maketable("curve")`. In order to create your own graphical tables using RTcmix's `plottable( )` command, we're going to need a few ancillary downloads.

First, we're going to need XQuartz, which is found at <http://xquartz.macosforge.org> and can be downloaded by clicking on the Quick Download link and running the installer.

## Quick Download

Download	Version	Released	Info
 <a href="#">XQuartz-2.7.7.dmg</a>	2.7.7	2014-08-18	For OS X 10.6 or later (including Mavericks)

A list of all available XQuartz releases can be found [here](#).  
(Development “beta” releases, if available, are [here](#).)

This is a version of the X Window System for Mac OS X, which allows our computers to run non-Apple software that is specifically reliant on the X Window System for its graphical user interface, which we are going to be using in our last download, AquaTerm.

Next, we’ll download AquaTerm by visiting <http://sourceforge.net/projects/aquaterm/files/?source=navbar> and clicking on the link for the .dmg image.

## AquaTerm (Mac OS X graphics terminal)

Brought to you by: [persquare](#)

Summary	<b>Files</b>	Reviews	Support	Wiki	Mailing Lists	Tickets ▾	News	Donate 
---------	--------------	---------	---------	------	---------------	-----------	------	--

Looking for the latest version? [Download AquaTerm-1.1.1.dmg \(390.2 kB\)](#)

Double click on the .dmg and run the package installer, which will place AquaTerm in your /Applications.

Lastly, head to <http://www.gnuplot.info> in order to download gnuplot, a graphics utility for the command line. Navigate to the release link for Version 5.0, which will take you to the SourceForge website.

### Version 5.0 Release

- [Download from SourceForge](#)
- [Release Notes](#)
- [User Manual \(PDF\)](#)

From there, you can click on the link for the .tar.gz package, which should

initiate your download.



Once that is complete, double click on the .tar package, which will create a folder called “gnuplot-5.0.0” in your /Downloads, which you can of course move to / Applications if you’d like to. Similar to our process for downloading RTcmix on the command line, you’ll want to open your Terminal and change directories to your / gnuplot-5.0.0 location and install the package using ./configure, then make, and make install as a three step process. Don’t forget to include the aquaterm flag in the same way that we configured RTcmix for Python earlier.

```
cd /path/to/Applications/gnuplot-5.0.0
```

```
./configure --with-aquaterm
```

```
make
```

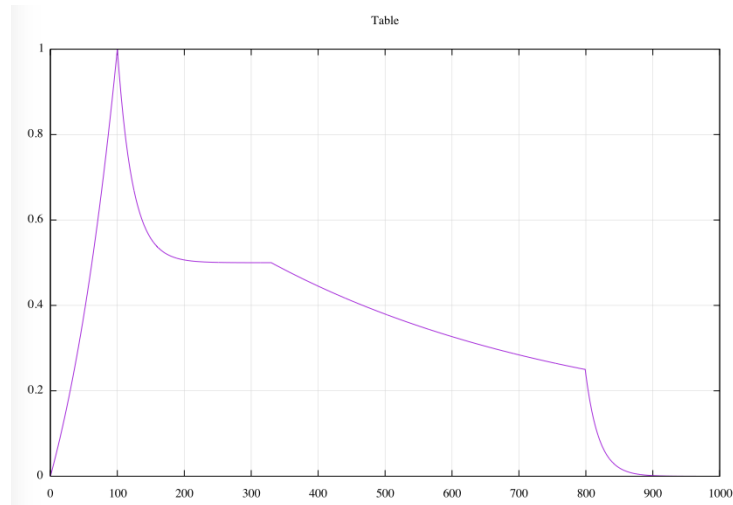
```
make install
```

We’re all set for including graphics and plotting tables with RTcmix! Check your work by opening score file 15 and adding the following to your script.<sup>129</sup>

```
plottable(envelope)
```

---

<sup>129</sup> It doesn’t really matter where you place this bit of code, but for clarity’s sake, add it after line eight where we created our envelope.



Not only can we plot tables, but the new graphics window will allow us to use our trackpad as a source for real-time data via `makeconnection("mouse")`.

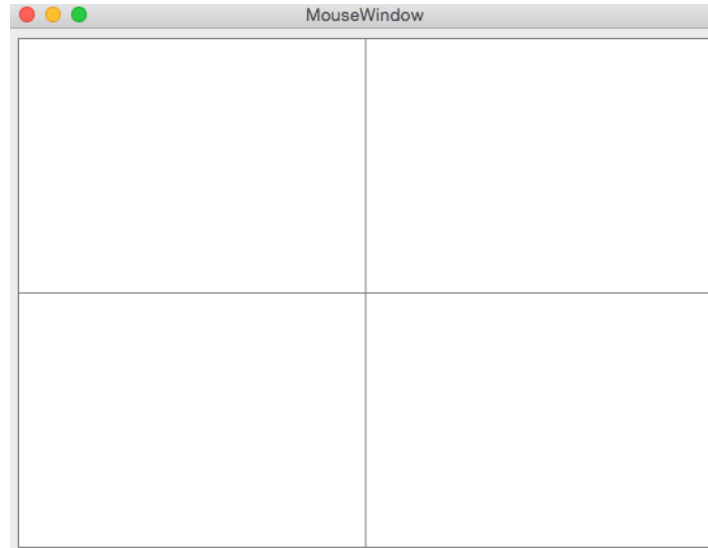
```
rtsetparams(44100, 2)
load("STEREO")
rtinput("/Users/jerod_s/Desktop/organ-test.aif")

start = 0
instart = 0
duration = DUR()
amplitude = makeconnection("mouse", "y", 0,1, default = 0.5, lag = 50)
pan = makeconnection("mouse", "x", 1,0, default = 0.5, lag = 50)
STEREO(start, instart, duration, amplitude, pan)
```

*Score file 57: Using data from trackpad*

Amplitude data is tracking on the  $y$  axis and pan data on the  $x$  axis of our mouse window, which you can see is set in the second p-field of `makeconnection("mouse")`.





From there, we set a range of values — written in pairs — as well as a default value to use when starting our script, a lag time in milliseconds to smooth out changes in values. If we didn't specify a lag time, we'd hear clicks and pops as we drag our cursor around our window.

For many, the use of USB MIDI controllers is a way to enhance the experience of playing live, due to the array of buttons, knobs, and sliders that can be utilized to alter data in their program of choice. We can use MIDI data with RTcmix in two ways and we'll first explore MIDI control in Pd and Max, then look at MIDI control natively in RTcmix itself.

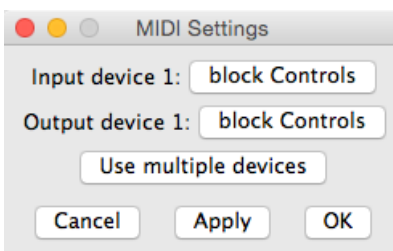
My current controller of choice is the Livid block, which is made by Livid Instruments, though, sadly, the unit is now out of production. I enjoy using this particular controller because of its sheer number of buttons, which I can use to trigger events in RTcmix, and its corresponding row of eight knobs.



*Photo courtesy of [lividinstruments.com](http://lividinstruments.com)*

Any knob, button, or slider has a corresponding value associated with it, much like we observed with the [key] object in Pd and Max. Thus, it will be possible to select any of those values for use in our patch and then passed to the [rtcmix~] object via `makeconnection("inlet")`.

Take a moment to ensure that your USB-connected MIDI controller is hooked up to your computer and then open Pd or Max. In Pd, we're going to need to visit the Menu bar and select Media → MIDI Settings. From there, you'll be able to select your input and output MIDI devices. In this case, I'm looking for my block controller and once you find yours, click on "Apply" and then "OK."

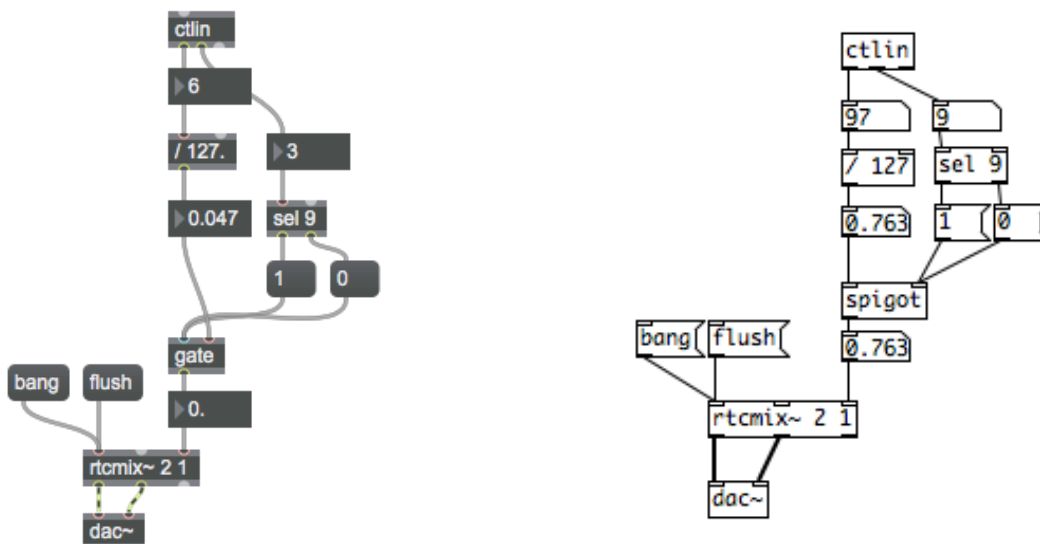


If you are using Max/MSP, head to the Menu bar for Options → MIDI Setup and select your controller.

MIDI Setup				
Type	On	Name	Abbrev	Offset
input	<input checked="" type="checkbox"/>	to Max 1	↕ _	↕ 0
input	<input checked="" type="checkbox"/>	to Max 2	↕ _	↕ 0
input	<input checked="" type="checkbox"/>	block Controls	↕ _	↕ 0
input	<input checked="" type="checkbox"/>	block Port 2	↕ _	↕ 0
output	<input checked="" type="checkbox"/>	AU DLS Synth 1	↕ _	↕ 0
output	<input checked="" type="checkbox"/>	from Max 1	↕ _	↕ 0
output	<input checked="" type="checkbox"/>	from Max 2	↕ _	↕ 0
output	<input checked="" type="checkbox"/>	block Controls	↕ _	↕ 0
output	<input checked="" type="checkbox"/>	block Port 2	↕ _	↕ 0

We aren't going to be concerned with passing MIDI messages out of Pd or Max, so it's not imperative to select your controller as an "Output" device, but it's a good habit to do straight away as Pd and Max will save your settings the next time you open the program, in case you do eventually want to pass MIDI messages out to an outboard synthesizer, for example.

Depending on your program of choice, take time to carefully code out the following in a new patching window and pay particular attention to the subtle differences in the Max patch, such as the need to specify floating point number boxes.



The [ctlin] object outputs three numbers. The leftmost outlet is the actual *control* value, which is a stream of numbers from 0-127 that corresponds with the range of numbers in the Musical Instrument Digital Interface (MIDI) protocol. The middle outlet references the control number, which in this case is the far left slider toward the bottom of my Livid block. Each time I move that slider, [ctlin] will output a “9” as that is the referential number associated with that particular slider. The far right outlet corresponds with the MIDI channel assignment, which we aren’t concerned quite yet.

The range 0-127, while useful, can be amended. In this case, I’m dividing that range of numbers by 127, in order to put them in the range of 0-1. That way, I can use them for, say, amplitude values in my script. Moreover, the [sel] object (short for “select”) will ensure that my [gate] or [spigot] is open only when I move the slider in question.

Here is a simple score file for testing this patch now that we have control over at least one aspect of our script.

```
rtinput("/path/to/file.aif")

start = 0
instart = 0
duration = DUR()
amplitude = makeconnection("inlet", 1, 0.0)
pan = 0.5
STEREO(start, instart, duration, amplitude, pan)
```

*Score file 58: Amplitude control using MIDI data from Pd or Max*

The only real advantage in constructing our MIDI mapping in Pd or Max is of course the chance to add any Pd or Max-specific elements to our work. RTcmix is equipped with the ability to make care of its own MIDI work in-house, so to speak, using set\_option( ) and makeconnection(“midi”).

```

rtsetparams(44100, 2)
load("STEREO")
rtinput("/path/to/file.aif")

set_option("midi_indevice = block")

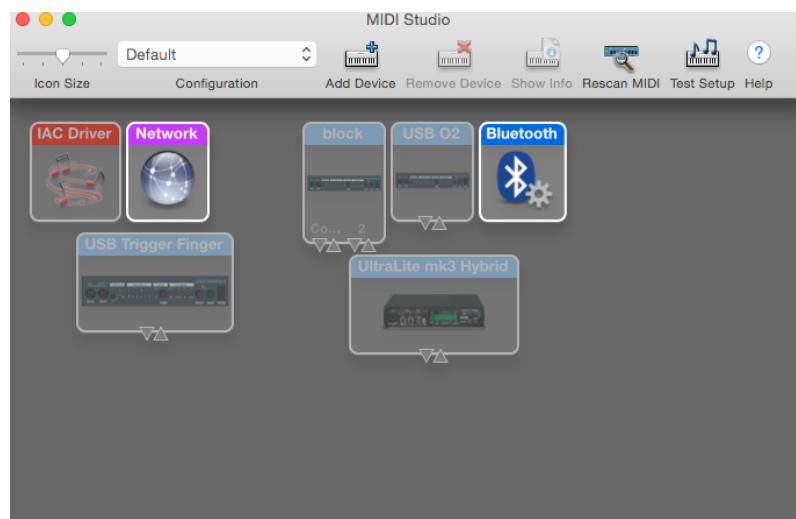
default = 0
lag = 25
channel = 1
slider1 = makeconnection("midi", 0.0, 1.0, default, lag, channel, "cntl", value = 9)

STEREO(start = 0, instart = 0, duration = DUR(), amplitude = slider1, pan = 0.5)

```

*Score file 59: MIDI input using makeconnection("midi")*

In order to correctly identify your controller for the `set_option("midi_indevice")` command, you'll need to open your Audio MIDI Setup application, which is found in your `/Applications/Utilities` folder. Once you have that opened, click *command-2* to open the MIDI setup window, which will list your controller.



In my case, my computer recognizes my controller as “block,” so I’ll change that accordingly in my call to `set_option( )`. I could use my M-Audio Trigger Finger controller by declaring `set_option(“midi_indevice = USB Trigger Finger”)`.

We will need to know which channel our controllers are sending MIDI data through, which you can find by either clicking on your controller’s icon in Audio MIDI setup or checking out the number from the rightmost outlet of `[ctlin]` in Pd or Max.

You’ll see that we’re creating a connection to the *slider1* variable, which includes a desired range of values (0.0 - 1.0), a default value, lag time, channel number, the *type* of data that we’re looking for, and the corresponding number associated with our slider.

This block controller has 64 buttons that I also use in performance. These buttons are treated similarly to piano keys on a MIDI keyboard controller, in that they correlate with pitches and are different from control data. In Pd or Max, you can get data from these buttons or keys by using the `[notein]` object and in RTcmix, you’ll need to specify “noteon” instead of “cntl” in p-field six.

It can’t be stressed enough: RTcmix is a completely customizable suite of open source tools. If there’s one overarching mantra of this book, it’s that you are free to use RTcmix in any way that you wish. It plays well with MINC, Python, Pd, Max/MSP, MIDI controllers, and even iOS and Android. Constantly evolving, it is curated by a strong community of users and continues to receive upgrades, enhanced functionality, and exciting new features that will ensure it continues to be a powerful choice for electronic and computer musicians for years to come.

## || Appendix: 100 progressive score files ||

The following examples were copied, culled, and edited from a year-long project I set forth for myself in 2013, aptly called *Script Calendar*. For the entire calendar year — in an effort to truly understand RTcmix from a variety of angles — I wrote a complete score file for each day. It was a wonderful, personal exploration and one that I am proud to have gone through and my hope is that these 100 score files serve you well as you learn the program for yourself.

If this book is being used in conjunction with a semester-long class, then 100 files will more or less provide you with the ability to rewrite one of them each day during the term, with a few days here and there for a day off. I find that like any language or skill, it is only possible to truly learn and gain facility through small, daily practice and immersive, hard work. These score files are now yours, to copy, hack, alter, and sculpt into your own compositions and I sincerely hope that you have as much fun with them as I did when I worked them out myself.

```
//Day 1
print("Hello World!")

//Day 2
x = 12.6
printf("the variable x equals: %f, %s \n", x, "C-style printing, diff from using print")

//Day 3
rtsetparams(44100, 2) //set sample rate, num channels
load("WAVETABLE") //load instrument
WAVETABLE(0, 5, 3000, 440, 0.5) //start, duration, amplitude, frequency, pan

//Day 4
rtsetparams(44100, 2)
load("WAVETABLE")
envelope = maketable("line", 1000, 0,0, 0.1,0.8, 0.8,0.5, 1.0,0) // add envelope
WAVETABLE(0, 5, 3000*envelope, 440, 0.5) //sine tone

//Day 5
rtsetparams(44100, 2)
load("WAVETABLE")
envelope = maketable("line", 1000, 0,0, 0.1,0.8, 0.8,0.5, 1.0,0)
wave = maketable("wave", 1000, "saw") // sawtooth wave
WAVETABLE(0, 15, 8000*envelope, 440, 0.5, wave)

//Day 6
rtsetparams(44100, 2)
load("DECIMATE")
rtinput("/path/to/your/file.wav")
envelope = maketable("line", 1000, 0,0, 0.1,1.0, 0.5,1.0, 1.0,0)
pan = makeLFO("sine", 0.3, 0,1)
DECIMATE(0, 0, DUR(), 0.5, 0.8*envelope, bitdepth = 2, 0, pan)

//Day 7
rtsetparams(44100, 2)
```

```

load("WAVESHAPE")
frequency = makeLFO("sine", 0.1, 400,10000)
envelope = makerandom("prob", 10, -1.0,1.0,0,0)
pan = makerandom("prob", 10, -1.0,1.0,0,0) //random panning
wave = maketable("wave", 1000, "sine")
transferfunction = maketable("cheby", 1000, 0.9, 0.3,-0.2,0.6,-0.7)
indexguide = maketable("line", 1000, 0,0, 3.5,1, 7,0)
WAVESHAPE(0, 20, frequency, 0, 1, 8000*envelope, pan, wave, transferfunction, indexguide)

//Day 8
rtsetparams(44100, 2)
load("AM")
rtinput("/Users/jerod_s/Desktop/SC9.aiff")
env1 = maketable("literal", 1000, 0,0, 2,1, 3,0.5, 7,0.5, 1,0)
env2 = makerandom("prob", 10, -1.0,1.0,0,0)
pan1 = makeLFO("saw", 8.5, 0.3, 0.7)
pan2 = makeLFO("buzz", 0.2, 0, 1.0)
wave = maketable("line", 1000, 0,-1.0, 0.5,1.0, 0.51,-1.0, 1.0,0) //ugly sawtooth wave
AM(0, 0, DUR(), 8.0*env1, 25025, 0, pan1, wave) // aliased carriers
AM(2, 0, DUR(), 8.0*env2, 25050, 0, pan2, wave)

//Day 9
rtsetparams(44100, 2)
load("MBLOWBOTL")
makegen(1, 5, 1000, 0.1, 50, 1, 50, .8, 600, .8, 300, 0.1) //old-style envelope
increment = 1
for(start = 0; start < 50; start += increment){
    amplitude = irand(40000, 80000)
    duration = irand(0.1, 4.0)
    pan = pickwrand(0.0,40, 0.5,20, 1.0,40)//target-value, probability(%)
    frequency = irand(0.1, 18) //subsonic
    MBLOWBOTL(start, duration, amplitude, frequency, noise = 0.3, pressure = 0.5, pan)
    increment = irand(0.1, 0.8)
}

//Day 10
rtsetparams(44100, 2)
load("WAVETABLE")
load("SHAPE")
bus_config("WAVETABLE", "aux 1 out")
waveform = maketable("wave", 1000, "saw124")
WAVETABLE(0, 30, 10000, 0.34, 0, waveform)

bus_config("SHAPE", "aux 1 in", "out 0-1")
amp = maketable("line", 1000, 0,0, 9,1, 10,0) * 0.25
pan = makeLFO("buzz", 0.4, 0,1)
transferfunction = maketable("random", 10, "triangle", -1.0,1.0)
indexguide = maketable("window", 1000, "hanning")
SHAPE(0, 0, 30, amp, min=0, max=3.0, 0, 0, pan, transferfunction, indexguide)

//Day 11
rtsetparams(44100, 2)
load("AMINST")
control_rate(2000) //reset update rate

duration = 1
amplitude = 8000
envelope = maketable("window", 1000, "hanning") //smooth
modulatoramplitude = maketable("line", 1000, 0,0, 1,1, 2,0)
carrierwave = maketable("wave", 1000, "buzz9")
modulatorwave = maketable("wave", 1000, "sine")
carrierfrequency = cpspch(8.00)
modulatorfrequency = cpspch(8.02)
increment = 0.5

for(start = 0; start < 20; start += increment){
    AMINST(start, duration, amplitude*envelope, carrierfrequency, modulatorfrequency,
        pan = 0.5, modulatoramplitude, carrierwave, modulatorwave)
    carrierfrequency += 2
    modulatorfrequency -= 7
    duration = irand(2, 5) * 3.14
}

//Day 12
rtsetparams(44100, 2)
load("WAVETABLE")
randomtest = irand(0, 100)
duration = 60
amplitude = 5000
frequency = cpspch(7.11)

```



```

waveform = maketable("wave", "nonorm", "nointerp", 2000, "sine")
pan = makeLFO("sine", "nointerp", 5.0, 0,1)

increment = 1.0
for(start = 0; start < 120; start += increment){
    if(randomtest < 50){
        duration = 5
    }
    else{
        duration = 10
    }
    WAVETABLE(start, duration, amplitude, frequency, pan, waveform)
    WAVETABLE(start, duration, amplitude, frequency - 10.5, pan, waveform)
}

//Day 13
rtsetparams(44100, 2)
load("WAVETABLE")
load("ELL")
bus_config("WAVETABLE", "aux 0-1 out")
bus_config("ELL", "aux 0-1 in", "out 0-1")
amplitude = 5000
wavetype = maketable("line", 32767, 0,0, 16384,1, 16385,-1, 16386,0, 32767,0)
pbcut = 1000 // passband cutoff frequency in Hz
sbcut = 90 // stopband cutoff frequency in Hz
ripple = 0.2 // amount of ripple (dB)
attenuation = 90 // attenuation at stopband (dB)
ellamp = 9 // filter amplitude
ringdur = .8 // ring-down duration (in sec.)
for(start = 0; start < 80; start += irand(0, 1)){
    frequency = irand(0, 1.0) / 7
    duration = irand(0, 1)
    pan = makeLFO("sine", 0.5, 0, 1)
    pbcut += irand(100, 400)
    WAVETABLE(start, duration, amplitude, frequency, 0.5, wavetype)
    WAVETABLE(start+1, duration+frequency, amplitude, frequency, 0.5, wavetype)
    ellset(pbcut, sbcut, 0, ripple, attenuation) // passband value > stopband = HPfilter
    ELL(start, 0, duration, ellamp, ringdur, 0, pan)
}

//Day 14
rtsetparams(44100, 2)
load("FMINST")
duration = 0.7
amplitude = 5000
envelope = maketable("line", 1000, 0, 0, 3.5,1, 7,0)
carrier = cpspch(7.00)
modulatorfrequency = 220
minindex = 0
maxindex = 10
pan = maketable("random", 5, "gaussian", 0,1)
waveform = maketable("wave", 1000, "sine")
guide = maketable("line", "nonorm", 1000, 0, 0, 5,1, 7, 0)

increment = 0.25
for(start = 0; start < 100; start += increment){
    duration = irand(0.01, 0.8)
    FMINST(start, duration, amplitude*envelope, carrier, modulatorfrequency, minindex, maxindex, pan,
        waveform, guide)
    carrier = cpspch(irand(2.07, 8.09))
    guide = maketable("line", "nonorm", 1000, 0,1, 7,0)
    FMINST(start+1, duration*4, amplitude, carrier, modulatorfrequency, minindex, maxindex, pan, waveform,
        guide)
    maxindex = irand(10, 15)
    increment = irand(0, 1)
}

//Day 15
rtsetparams(44100, 2)
load("WIGGLE")
amplitude = maketable("line", "nonorm", 1000, 0,0, 0.1,2000, 5,4000, 10,2000)
pitch = 2.03
envelope = maketable("curve", 2000, 0,0,2, 2.5,1,0, 13,1,-3, 25,0)
carrierwaveform = maketable("wave", 24051, "sine")
min = -1.00
max = 2.00
seed = srand()
gliss = maketable("random", "nonorm", "nointerp", 500, "low", min, max) //p4 controls rate
freq = makeconverter(octpch(pitch) + gliss, "cpsoct")
mod_depth_type = 2 // 0 is no mod, 1 is % of carrier, 2 is FM

```

```

mod_wavetable = maketable("wave", 1000, "sine")
mod_freq = 200
mod_depth = 20
filt_cf = maketable("curve", "nonorm", 2000, 0,1000,-4, 1,1)
pan = maketable("line", "nonorm", 1000, 0,.2, 1,.5, 2,1)
filt_type = 0 // 0 is no filt, 1 is low, 2 is high
filt_steep = 20
balance = true // balance output and input signals

WIGGLE(start = 0, 60, amplitude * envelope, freq, mod_depth_type, filt_type, filt_steep, balance,
        carrierwaveform, mod_wavetable, mod_freq, mod_depth, filt_cf, pan)

pan = maketable("line", "nonorm", 1000, 0,1, 1,0)
freq = makeconverter(octpch(pitch + 0.01) + gliss, "cpsoct")

WIGGLE(start = 1.0, 60, amplitude * envelope, freq, mod_depth_type, filt_type, filt_steep, balance,
        carrierwaveform, mod_wavetable, mod_freq, mod_depth, filt_cf, pan)

//Day 16
rtsetparams(44100, 2)
load("JGRAN")
srand()

duration = 20
amplitude = 30.0
randomseed = srand()
osconfg = 0 // wavetable, as opposed to FM (1)
oscphase = 1 // randomize osc phase? 0 is no; 1 is yes
grainenv = maketable("window", 1000, "hamming")
grainwaveclickhi = maketable("line", 32768, 0,0, 16384,0, 16385,1.0, 16387,-1.0, 16388,0, 32768,0)
grainwaveclicklow = maketable("line", 32768, 0,0, 15384,0, 16385,1.0, 17387,-1.0, 18388,0, 32768,0)
FMmult = maketable("random", 1000, "gaussian", 2.0, 100.0)
FMindex = 3.0
minfreq = 0.001 // really slow things down here to produce even faintest freq.
maxfreq = 12.0
minspeed = maketable("line", "nonorm", 1000, 0,0.2, 1,0.001) // decreasing minimum
maxspeed = maketable("line", "nonorm", 1000, 0,0.001, 1,0.2) // increasing maximum
mindb = 20
maxdb = 80
density = maketable("random", 1000, "gaussian", 0,10)
pan = makeLF0("sine", 0.5, 0,1)
panrand = maketable("random", 1000, "gaussian", 0,1)

for (start = 0; start < 80; start += 1){
    JGRAN(start, duration, amplitude, randomseed, osconfg, oscphase, grainenv, grainwaveclickhi, FMmult,

        FMindex, minfreq, maxfreq, minspeed, maxspeed, mindb, maxdb, density, pan, panrand)

    JGRAN(st, dur, amp, randomseed, osconfg, oscphase, grainenv, grainwaveclicklow, FMmult, FMindex,
        minfreq, maxfreq, minspeed, maxspeed, mindb, maxdb, density, pan, panrand)
}

//Day 17
rtsetparams(44100, 2)
load("VOCODESYNTH")
rtinput("/path/to/file.wav")

start = 0
instart = 0.0
duration = DUR()
amplitude = 150.0
envelope = maketable("line", 1000, 0,0, .1,1, 0.6,1, 1.0,0)
numbands = 25
lowcf = 300
interval = 0.025
cartransp = 0.00
bw = 0.009
winlen = 0.001
smooth = 0.98
thresh = 0.0001
atktime = 0.001
reltime = 0.01
hipassmod = 0.0
hipasscf = 2000
carwavetable = maketable("wave", 10, 20000, "sine")
scale1 = 0.5
scale2 = 1.0
scalecurve = maketable("curve", "nonorm", 100, 0,scale1,1, 1,scale2)
spacemult = cpspch(interval) / cpspch(0.0)

```

```

VOCODESYNTH(start, instart, duration, amplitude*envelope, numbands, lowcf, spacemult, cartransp, bw, winlen,
    smooth, thresh, atktime, reltime, hipassmod, hipasscf, inchan=0, pan=1, carwavetable, scalecurve)

cartransp += 0.001
spacemult += 0.002

VOCODESYNTH(start, instart, duration, amplitude*envelope, numbands, lowcf, spacemult, cartransp, bw, winlen,
    smooth, thresh, atktime, reltime, hipassmod, hipasscf, inchan=0, pan=0, carwavetable, scalecurve)

//Day 18
rtsetparams(44100, 2) //exploring 3n+1 problem from G.E.B
load("WAVETABLE")
srand()

envelope = maketable("line", 1000, 0,0, 0.1,1, 0.3,0.7, 0.7,0.7, 1.0,0)
n = trunc(irand(10, 100)) //choose random integer between 10 and 100

increment = 0.1515
for(start = 0; start < 10; start += increment){
    if(n%2 == 0){ // if n is even
        n = trunc(n * 3 + 1) // do a 3n + 1 operation
        transp = 1.0 // and transpose up and octave
        amplitude = irand(3000, 7000) // louder amplitude values
        pan = irand(0.0, 0.5) //pan L
    }
    else if (n%2 == 1){ // if n is odd
        n = trunc(n / 2) // divide n by 2
        transp = -1.0 // and transpose down an octave
        amplitude = irand(500, 2000) //quieter
        pan = irand(0.5, 1.0) //pan R
    }
    if (n == 1){ // if n equals 1
        exit() // immediately stop program
    }
    pitch = cpsmidi(n) // translate n to pitch in MIDI
    if(pitch > 128){
        pitch = pitch - 127
    }

    dB = dbamp(amp)
    pan = irand(0.0, 1.0)
    constpowpan = boost(pan) / 2
    WAVETABLE(start, increment*0.8, amplitude*envelope, pitch, constpowpan)
    print(n) // only print current n values
}

//Day 19
rtsetparams(44100, 2)
load("WAVETABLE")
reset(4)
srand(11)

envelope1 = maketable("line", 1000, 0,0, 5,1, 10,0)
envelope2 = maketable("line", 1000, 0,1.0, 5,0.5, 10,0)
waveform1 = maketable("random", 1000, "gaussian", -1,1)
waveform2 = maketable("wave3", 1000, 1,1,0, 2,0.05,0, 3,0.03,0, 4,0.02,0, 5,0.02,0, 6,0.005,1, 7,0.005,0,
    8,0.001,0, 9,0.001,0, 10,0.0001,0, 11,0.0001,0, 13,0.0002,0, 15,0.0005,1, 17,0.0001,0,
    19,0.0001,0, 21,0.00002,0, 23,0.0005,1, 25,0.0001,0, 27,0.0001,0, 29,2,0.0002,0,
    31,7,0.00005,1, 33,0.00001,0, 35,5,0.000001,0, 37,0.00005,1, 39,0.00001,0, 41,0.00001,0,
    43,0.00001,0 )
gliss1 = maketable("line", "nonorm", 1000, 0,1, 1.0,0.5)
gliss2 = maketable("line", "nonorm", 1000, 0,0, 1.0,1.0)
amp1 = 1000
amp2 = 1500
amp3 = 2000
pan1 = makeLFO("square", 0.1, 0.1,0.5)
pan2 = makeLFO("sine", 0.25, 0.5,1.0)
pan3 = makeLFO("buzz", 0.5, 0.0,1.0)
for (start = 0; start < 50; start += 1){
    WAVETABLE(start+irand(0, 1), 10, amp3*envelope1, 60*gliss1, pan1, waveform1)
    WAVETABLE(start+irand(0.5, 2), 9, amp2*envelope2, 120*gliss2, pan2, waveform2)
    WAVETABLE(start+irand(0.5, 3), 10, amp1*envelope1, 240*gliss1, pan3, waveform1)
    WAVETABLE(start+irand(0.5, 1), 9.5,amp1*envelope2, 480*gliss2, pan1, waveform2)
    WAVETABLE(start+irand(0.5, 3), 10, amp2*envelope1, 760*gliss1, pan2, waveform1)
    WAVETABLE(start+irand(0, 1), 10, amp3*envelope2, 24051*gliss2, pan3, waveform2)
    start += irand(1, 5)
    if(start <= 25){
        amp1 += 1000
        amp2 += 1000
        amp3 += 1000
    }
}

```

```

    }

//Day 20
rtsetparams(44100, 2)
load("MBANDEDWG")
srand()

duration = 0.5
amplitude = 10000
pitch = cpsmidi(70)
strikeposition = 0.3 //range 0.0-1.0
pluck = 0 // 0 no pluck, 1 pluck
maxvelocity = 0.5 //range 0.0-1.0
instrument = 1 // 0 uniform bar, 1 tuned bar, 2 glass harmonica, 3 tibetan singing bowl
pressure = 0.0 //0.0 if struck, otherwise bowed
resonance = 0.9 //range 0.0-1.0
constant = 0.8 //range 0.0-1.0
pan = makeLFO("sine", 0.2, 0.0,1.0)
n = 11700 //choose random integer between 10 and 100

increment = 0.125
for(start = 0; start < 100; start += increment){
    makegen(1, 24, 1000, 0,1, 2,0) //old style line graph for envelope
    MBANDEDWG(start, duration, amplitude, pitch, strikeposition, pluck, maxvelocity, instrument, pressure,
        resonance, constant, pan)
    if(n%2 == 1){ // if n is odd
        n = trunc(n * 3 + 1) // do a 3n + 1 operation
        constant = irand(0.0, 1.0)
        resonance = irand(0.0, 1.0)
        amp = irand(3000, 7000) // louder amplitude values
    }
    else if (n%2 == 0){ // if n is even
        n = trunc(n / 2) // divide n by 2
        maxvelocity = irand(0.0, 1.0)
        strikeposition = irand(0.0, 1.0)
        amp = irand(500, 2000) //quieter
    }
    pitch = cpsmidi(n) // translate n to pitch in MIDI
    if(pitch < 30){
        pitch = pitch + 127
    }
}

//Day 21
rtsetparams(44100, 2)
load("HALFWAVE")
rate = control_rate(16)
print_off()
srand()

start = 0
duration = 0.75
pitch = octpch(5.07) //start on C#, can also use Hz with HALFWAVE()
amplitude = 300
envelope = maketable("window", 1000, "hamming")
wave1 = maketable("wave3", 1000, 3.14,1,0, 6.28,1,0.5)
wave2 = maketable("wave3", 1000, 1.00,1,0, 2.00,1,0.5)
wave3 = maketable("wave3", 1000, 1,1,0, 3,0.3,0, 5,0.2,0, 7,0.05,0, 9,0.01,0, 11,0.001,0)
wave4 = maketable("wave3", 1000, 1,1,0, 2,0.5,0, 3,0.3,0, 4,0.25,0, 5,0.2,0, 6,0.16,0, 7,0.14,0, 8,0.125,0)
wave5 = maketable("wave3", 1000, 1,1,0, 3,0.14,0, 5,0.04,0, 7,0.02,0, 9,0.012,0, 11,0.008,0)
wavegamut = {wave1, wave2, wave3, wave4, wave5}
wavegamutlength = len(wavegamut)
wavecrossoverpoint = irand(0,1)
pan = 1

loop = 1.0
for(start = 0; start < 500; start += loop) {
    waveindex = trunc(irand(0,wavegamutlength))
    wave = wavegamut[waveindex]
    wavecrossoverpoint = irand(0,1)
    HALFWAVE(start, duration, pitch, amplitude*envelope, wave, wave, wavecrossoverpoint, pan*irand(0,1))

    waveindex = trunc(irand(0,wavegamutlength))
    wave = wavegamut[waveindex]
    wavecrossoverpoint = irand(0,1)
    pitch += (irand(-0.01,0.001) + trunc(irand(-2, 2)))
    HALFWAVE(start+2, duration+1, pitch, amplitude*envelope, wave, wave, wavecrossoverpoint,
        makeLFO("sine", 17, 0,1))
}

```

```

    waveindex = trunc(irand(0,wavegamutlength))
    wave = wavegamut[waveindex]
    wavecrossoverpoint = irand(0,1) // 0-1
    pitch += (irand(-0.01,0.001) + trunc(irand(-2, 2)))

    HALFWAVE(start+3, duration+2, pitch, amplitude*envelope, wave, wave, wavecrossoverpoint,
              makeLFO("sine", 18, 0,1))
    loop = irand(0,5)
    pitch += (irand(-0.01,0.001) + trunc(irand(-2, 2)))
    duration = irand(0, 5)
    amplitude = irand(200, 700)
    rate = round(irand(4, 128))
}

//Day 22
rtsetparams(44100, 2)
load("CLAR") //early clarinet physical model
load("MCLAR") //another
load("MBLOWHOLE") //another, with tonehole and register vent
seed = srand()
//-----//CLAR( ) parameters. Only create envelopes with makegen( )
clarduration = 2 //Original physical model by Perry Cook
clarnoiseamp = 0.001
clarlength1 = 400 // sample length1 (0-500)
clarlength2 = 99 // sample length2 (0-500)
claramplitude = 1000// absolute in 16-bit (0-32768)
clargain = 0.9 // 0-1.0, oops. d2 gain(?) pitch is determined by comb. of sample lengths/d2
clarpan = spray_init(0, 10, seed) //unrepeated random numbers (table 0, 10 elements...)
//-----//MCLAR( ) parameters. Can use maketable( ) for envelope control
mclarduration = 2
mclaramplitude = 1051
mclarfrequency = makeLFO("sine", 0.5, pchcps(440),pchcps(9000))
mclarmaxpressure = 0.5
mclarreedstiffness = 1.0
mclarpan = spray_init(1, 10, seed)
mbreathpressure = maketable("random", 25, "gaussian", 0,1.0)
//-----//MBLOWHOLE( ) parameters. Adds tonehole/vent
mblowduration = 2
mblowamplitude = 1051
mblowfrequency = makeLFO("saw", 0.55, pchcps(110),pchcps(5000))
mblownoiseamp = 0.5
mblowmaxpressure = 0.75
mblowreedstiffness = 0.25
mblowtoneholestate = 0 //0 is closed, 1 is open
mblowventstate = 1 //same
mblowpan = spray_init(2, 10, seed)
mblowbreathpressure = maketable("random", 25, "low", 0,1.0)
for(start = 0; start < 25; start += 1){
    modifier = irand(0,1.0)
    panvalue = get_spray(0)
    pan = panvalue / 10
    CLAR(start, clarduration, clarnoiseamp, clarlength1, clarlength2, claramplitude, clargain*modifier,
        pan)
    panvalue = get_spray(1)
    pan = panvalue / 10
    MCLAR(start+2, mclarduration, mclaramplitude, mclarfrequency, mclarmaxpressure, mclarreedstiffness,
        pan, mbreathpressure*modifier)
    panvalue = get_spray(2)
    pan = panvalue / 10
    MBLOWHOLE(start+3, mblowduration, mblowamplitude, mblowfrequency, mblownoiseamp,
        mblowmaxpressure,mblowreedstiffness, mblowtoneholestate, pan, mblowbreathpressure*modifier)
}

//Day 23
rtsetparams(44100, 2) // Risset bell demo, after Charles Dodge 2nd ed, p 105 -JG
load("WAVETABLE") // from John Gibon's RTcmix tutorial example, adapted JS
seed = srand()
control_rate(4)
env1 = maketable("expbrk", 1000, 1, 1000, .0000000002) // more attack
env2 = maketable("line", 1000, 0,0, 0.3,0.3, 0.6,0.3, 0.75,0.7, 1.0,0.0) //reverse ADSR
wavet = maketable("wave", 100, "sine");
durscale = 3
notes = {
// start dur amp freq
{ 0, 52, 3000, 82051 },
{ 2, 40, 4000, 91200 },
{ 5, 5, 2000, 74800 },
{ 7, 34, 1500, 50 },
{ 8, 26, 1000, 62576 },
{ 10, 11, 2000, 82709 },

```

```

{ 11, 17, 2000, 95934 },
{ 13, 20, 4000, 1348 },
{ 14, 18, 1000, 60 },
{ 16, 22, 3000, 1102 },
{ 17, 28, 1500, 55626 },
{ 20, 7, 2500, 895 },
{ 22, 8, 2500, 880 },
{ 26, 16, 3500, 220 },
{ 28, 10, 2000, 110 },
{ 34, 13, 2000, 55 },
{ 40, 14, 1050, 110 },
{ 52, 2, 1000, 220 }
}

numnotes = len(notes) // set up an array for the bell notes
paninit = spray_init(0, 10, seed)

loop = 1.0
for (i = 0; i < numnotes; i += 1) {
    note = notes[i]
    start = note[0]
    dur = note[1] * durscale
    amp = note[2] * env1
    freq = note[3]
    panvalue = get_spray(0)
    pan = panvalue / 10 //random panning between 0 and 1.0
    WAVETABLE(start, dur, amp, freq*.56, pan, wavet)
    WAVETABLE(start, dur*.9, amp*.67, freq*.56+1, pan, wavet)
    WAVETABLE(start, dur*.65, amp, freq*.92, pan, wavet)
    WAVETABLE(start, dur*.55, amp*1.8, freq*.92+1.7, pan, wavet)
    WAVETABLE(start, dur*.325, amp*2.67, freq*1.19, pan, wavet)
    WAVETABLE(start, dur*.35, amp*1.67, freq*1.7, pan, wavet)
    WAVETABLE(start, dur*.25, amp*1.46, freq*2, pan, wavet)
    WAVETABLE(start, dur*.2, amp*1.33, freq*2.74, pan, wavet)
    WAVETABLE(start, dur*.15, amp*1.33, freq*3, pan, wavet)
    WAVETABLE(start, dur*.1, amp, freq*3.76, pan, wavet)
    WAVETABLE(start, dur*.075, amp*1.33, freq*4.07, pan, wavet)
    loop = pan
}

//Day 24
rtsetparams(44100, 2)
load("MMODALBAR")
load("DELAY")
load("IIR")
print_off()
reset(8)
srand()

bus_config("MMODALBAR", "aux 0-1 out")
bus_config("DELAY", "aux 0-1 in", "out 0-1")
loop = 0.125

//-----Modal Bar parameters
barstart = 0
barduration = 3.0
baramplitude = 20000
barnote = 44051
stickhardness = 1.0 //0.0-1.0
stickposition = 0.1 //0.0-1.0
barinstrument = 1 //wood...0marimba, 1vibe, 2agogo, 3wood1, 4reso, 5wood2, 6beats, 7twofixed, 8clump
barpan = makeLF0("sine", 20.2, 0,1)

//-----Delay parameters
delayinstart = 0
totalduration = 2
delayamplitude = 0.8
delaytime = 1.0 //seconds
delayfeedback = 0.8
ringdownduration = 1.0
inputchannel = 0
for(start = 0; start < 300; start += loop){
    makegen(1, 24, 1000, 0,1, 1, 1) // old school envelope
    MMODALBAR(start, barduration, baramplitude, barnote, stickhardness, stickposition, barinstrument,
        barpan)
    barnote -= irand(10, 30)

    MMODALBAR(start+2, barduration, baramplitude, barnote, stickhardness, stickposition, barinstrument,
        barpan-0.5)
    delaypan = random()
}

```

```

        DELAY(start, delayinstart, totalduration, delayamplitude, delaytime, delayfeedback, ringdownduration,
            inputchannel, delaypan)
        loop += 0.015
    }

    filteroutstart = 0
    filterinstart = 1
    filterduration = 40
    filteramplitude = 20000
    filterenvelope = maketable("window", 1000, "hamming")
    filterpitch = 44
    filterpan = 0.5
    center1 = 300

    for(i = 0; i < 100; i += 1.0){
        setup(center1,25.0,0.8, 1200.0,15.0,0.9) //CF1,BW1,AMP1, CF2,BW2,AMP2...
        PULSE(i, filterinstart, filterduration, filteramplitude*filterenvelope, filterpitch, filterpan)
        filterpitch += irand(10.0, 40.0)
        center1 += 100.0
    }

//Day 25
rtsetparams(44100, 2, 16)
load("MULTIWAVE")//Additive synthesis instrument
control_rate(32) //Adapted from JGs 3/10/05 help file

duration = 180
masteramp = 10000
minfreq = 220
maxfreq = 6000
glide = 20
quantum = 100 // quantize freqs to this number (in Hz)
waveform = maketable("wave3", 1000, 1,1,0, 3,0.14,0, 5,0.04,0, 7,0.02,0, 9,0.012,0, 11,0.008,0)
envelope = maketable("line", 1000, 0,0, 0.2,0.4, 0.6,0.4, 0.7,1, 1.0,0)
numwaves = 12
freq = {} // freq index
pan = {} // pan index

loop = 1
for (i = 0; i < numwaves; i += loop) {
    masteramp = irand(15000, 20000)
    lfofreq = 0.007 + (i * 1.4)
    rfreq = makeLFO("sine", lfofreq, min = 0.2 + (i * 0.03), min * 3.5)
    min = minfreq + (i * 10)
    max = maxfreq - (i * 70)
    rand = makerandom("linear", rfreq, min, max, seed = i + 1)
    freq[i] = makefilter(rand, "smooth", glide)
    if (quantum){
        freq[i] = makefilter(freq[i], "quantize", quantum)
    }
    min = mod(i, 2)

    if (min == 0){
        max = 1
    }
    else{
        max = 0
    }

    pan[i] = makeLFO("sawup", 0.007 + (i * 0.026), min, max)
    loop = irand(0,1)
}

amp = makerandom("cauchy", 100, 0,1)
phase = makerandom("high", 100, 0,1)

MULTIWAVE(start = 0, duration, masteramp*envelope, waveform,
    freq[0], amp, phase, pan[0],
    freq[1], amp, phase, pan[1],
    freq[2], amp, phase, pan[2],
    freq[3], amp, phase, pan[3],
    freq[4], amp, phase, pan[4],
    freq[5], amp, phase, pan[5],
    freq[6], amp, phase, pan[6],
    freq[7], amp, phase, pan[7],
    freq[8], amp, phase, pan[8],
    freq[9], amp, phase, pan[9],
    freq[10], amp, phase, pan[10],
    freq[11], amp, phase, pan[11])

```

```

//Day 26
rtsetparams(44100, 2, 512) //Sound vaguely like really bad Nintendo
load("WAVETABLE")
load("FLANGE")
control = 128
control_rate(control)
srand()
print_offf()

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("FLANGE", "aux 0-1 in", "out 0-1")

envelope1 = maketable("line", 1000, 0,0, 5,1, 10,0)
envelope2 = maketable("line", 1000, 0,1.0, 5,0.5, 10,0)
waveform1 = maketable("random", 5, "gaussian", -1,1)
waveform2 = maketable("wave", 1000, "sawdown", 0,0, 1,0, 2,1, 3,1)
glissup = maketable("line", 1000, 0,0.1, 1.0,0)
glissdown = maketable("line", 1000, 0,0, 1.0,1)
pan1 = makeLFO("square", 0.1, 0.1,0.5)
pan2 = makeLFO("sine", 5.25,0.5,1.0)
pan3 = makeLFO("buzz", 10.5, 0.0,1.0)
amp1 = 1000
amp2 = 1500
amp3 = 2000
for (st = 0; st < 1000; st += 1){
    WAVETABLE(st+irand(0, 1), 1, amp3*envelope1, .3, pan1, waveform2)
    WAVETABLE(st+irand(0.5, 2), 5, amp2*envelope2, 3, pan2, waveform2)
    WAVETABLE(st+irand(0.5, 3), 3, amp1*envelope1, .24*glissup, pan3, waveform2)
    WAVETABLE(st+irand(0.5, 1), 9, amp1*envelope2, 8*glissdown, pan1, waveform2)
    WAVETABLE(st+irand(0.5, 3), 4, amp2*envelope1, .7, pan2, waveform1)
    WAVETABLE(st+irand(0, 1), 2, amp3*envelope2, 5*glissup, pan3, waveform2)

    if(st%2 == 0){
        amp1 += 100
        amp2 += 100
        amp3 += 100
    }

    if(st%2 == 1){
        amp1 -= 100
        amp2 -= 100
        amp3 -= 100
    }

    flangedur = irand(0, 5)
    resonance = irand(0,1)
    lowpitch = irand(0,4)
    moddepth = irand(0,5)
    modspeed = irand(0,1)
    wetdrymix = irand(0,1)
    maxdelay = 1.0 / cpspch(lowpitch)

    FLANGE(st, 0, flangedur, 0.8, resonance, maxdelay, moddepth, modspeed, wetdrymix, "IIR", 0, pan=1,
        ringdur=0, waveform1)
    FLANGE(st, 0, flangedur*0.75, 0.8, resonance, maxdelay, moddepth, modspeed, wetdrymix, "IIR", 0,
        pan=0, ringdur=0, waveform2)

    control = pickrand(2, 4, 8, 16, 32, 64, 128, 256, 512)
    control_rate(control)
    loop = pickrand(2.5, 3, 1.0, 5.0, 0.1)
}

//Day 27
rtsetparams(44100, 2, 128)
load("WAVETABLE")
load("MMODALBAR")
control = 44100
reset(control)
srand()
print_offf()
//-----cascading waveforms (weird, random, jaggedy looking things)
st = 0
dur = 10
amp = 1000
env1 = maketable("line", 1000, 0,0, 0.1,1, 0.9,1, 1.0,0)
env2 = maketable("line", 1000, 0,0.8, 1,0.0)
freq = 22051
pan1 = makeLFO("sine", 60.5, 0,0.5)
pan2 = makeLFO("sine", 50.75, 0.5,1)
waveform = maketable("random", 8, "gaussian", -1,1)

```



```

loop = 1
for(st = 0; st < 125; st += loop){
    dur = irand(5, 20)
    WAVETABLE(st, dur, amp*env1, freq, pan1, waveform)
    freq -= 100
    WAVETABLE(st+2, dur/2, amp-200*env2, freq/2, pan2, waveform)
    freq += 10
    WAVETABLE(st+3, dur/3, amp-400*env1, freq/3, pan1, waveform)
    freq -= 5
    WAVETABLE(st+4, dur/4, amp-600*env2, freq/4, pan2, waveform)
    freq += 2.5
    WAVETABLE(st+5, dur/5, amp-800*env1, freq/5, pan1, waveform)
    freq -= 100
    WAVETABLE(st+6, dur/6, amp*env2, freq/6, pan2, waveform)
    loop = irand(0, 1)
    control = pickrand(16, 32, 128, 256, 4096, 22050, 44100)
    control_rate(control)
}

//-----sort of a John Adams' Short Ride simple repeating block pattern
makegen(1, 24, 1000, 0,1, 1, 1)
start = 0
duration = 1.0
amplitude = 20000
frequency = 22051
hardness = 0.0
position = 0.4
instrument = 0 // 0marimba, 1vibe, 2agogo, 3wood1, 4reso, 5wood2, 6beats, 7fixed, 8clump
pan1 = makeLFO("square", 0.25, 0,0.6)
pan2 = makeLFO("sine", 0.3333, 0.4,1)

for (i = 0; i < 25; i +=1){
    hardness = 0.0
    for (ii = 0; ii < 50; ii += 1){
        MMODALBAR(start, duration, amplitude, frequency, hardness, position, i, pan1)
        hardness += 0.01
        start += 0.5
        frequency -= 0.5
        MMODALBAR(start*(ii/2), duration, amplitude, frequency, hardness, position, i, pan1)
    }
    MMODALBAR(start, duration, amplitude, frequency, hardness-0.4, position, i*2, pan2)
    frequency += 1.33
    MMODALBAR(start*(i/2), duration, amplitude, frequency, hardness-0.4, position, i, pan2)
}

//Day 28
rtsetparams(44100, 2)
load("COMPLIMIT")
load("MOCKBEND")
load("PANECHO")
rtinput("/path/to/file.aiff")

bus_config("COMPLIMIT", "in 0-1", "aux 0-1 out") //needs to read in from 0-1?
bus_config("MOCKBEND", "aux 0-1 in", "aux 2-3 out")
bus_config("PANECHO", "aux 2-3 in", "out 0-1")

//-----compression stuff
start = 0
instart = 0
dur = DUR()
ingain = 18
outgain = 0
attack = 0.001
release = 0.02
threshold = -20
ratio = 4
lookahead = attack
windowlen = 128
detect_type = 0
bypass = 0
inchan = 0
pan = 0.5 //no further panning, input file is already interesting enough
COMPLIMIT(start, instart, dur, ingain, outgain, attack, release, threshold, ratio, lookahead, windowlen,
detect_type, bypass, inchan, pan)

//-----MOCKBEND, uses old-style makegens
start = 0
instart = 0
dur = DUR()
amp = 0.7

```

```

pan = 0.5
setline(0,0, 1, 1, 90, 1, 100, 0) // amplitude curve
makegen(-2, 18, 512, 1,.5, 512,-.11) //transpose up 5 semitones, then down 11
MOCKBEND(start, instart, dur, amp, 2, 0, pan)

//-----ping-pong delay
start = 0
instart = 0
dur = DUR()
amp = 0.7
channel0delay = 2.12 // in sec
channel1delay = 0.55
regenerator = 0.8 // multiplier, ALWAYS less than 1!
ringdowndur = 7.2
makegen(1, 24, 1000, 0,1, 100, 1)
PANECHO(start, instart, dur, amp, channel0delay, channel1delay, regenerator, ringdowndur)

for(i = 0; i < 25; i += 1){
    channel0delay += irand(0,1)
    channel1delay = irand(0,1)
    regenerator = irand(0,1) - 0.1
}

PANECHO(start, instart, dur-4, amp, channel0delay, channel1delay, regenerator, ringdowndur-1.2)

//Day 29
rtsetparams(44100, 2) // from JG 6/3/2002
load("MIX")
load("VOCODE2")
rtinput("/path/to/file.aiff")

//----- carrier for vocoder
bus_config("MIX", "in 0", "aux 0 out") // separate between channels
instart = 0
amp = 0.8
dur = DUR() - instart
MIX(0, instart, dur, amp, 0)
//----- modulator for vocoder
bus_config("MIX", "in 0", "aux 1 out")
instart = 0
dur = DUR() - instart
amp = 0.8
MIX(0, instart, dur, amp, 0)
// ----- vocoder
bus_config("VOCODE2", "aux 0-1 in", "out 0-1")
//list of center frequencies
cftabs = maketable("literal", "nonorm", 0, 3.00, 4.07, 5.06, 6.02, 7.01, 7.04, 7.11, 8.06, 9.01, 9.03)
numpitches = tablelen(cftabs)
outstart = 0
instart = 0
dur = DUR() + 2 // allow vocoder to ring, if nec.
maxamp = 1.0
amplitude = maketable("line", "nonorm", 1000, 0,maxamp, dur-1,maxamp, dur,0)
numfilt = 0 // number of filters. this is a flag, make sure its 0
transp = 0.2
freqmult = 4.02
cartransp = -0.02
bw = 0.008
resp = 0.0001

for(ii = 0; ii < 10; ii += 1){
    resp = random() / 10
}

hipass = 0.1
hpcf = 5000
noise = 0.01
noisubsamp = 4

for(i = 0; i < 25; i += 1){
    transp += 0.01
    freqmult = random() + 3
    amp = add(amplitude, irand(0,1)) // add constant to table!
    cftab = modtable(cftabs, "shift", round(irand(1,4))) // shift contents

    VOCODE2(i, instart, 1, amp, numfilt, transp, freqmult, cartransp, bw, resp, hipass, hpcf, noise,
        noisubsamp, 0.5, cftab)
}

pan = makeLF0("sine", 1.2, 0,1)

```

```

VOCODE2(outstart , instart, dur, amp, numfilt, transp, freqmult, cartransp, bw, resp, hipass, hpcf, noise,
        noisubsamp, pan, cftab)

//Day 30
rtsetparams(44100, 2)
load("WAVETABLE")
load("JCHOR")
load("SPECTACLE2")
reset(22050)
print_off()
srand()

rtinput("/path/to/file.wav")

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("JCHOR", "in 0-1", "aux 0-1 out")
bus_config("SPECTACLE2", "aux 0-1 in", "out 0-1")

//-----Wavetables
start = 0
dur = DUR()
amplitude = 1000.0
frequency = makeconnection("midi", 33025,89000, 33041, 90, 10, "cntl",13)
waveform = maketable("wave", "nonorm", "nointerp", 1000, "buzz")
panfreq = makeconnection("midi", 22051,34011, 0.5, 10, 10, "cntl",16)
pan = makeLFO("sine", panfreq, 0,1)
WAVETABLE(start, dur, amplitude, frequency, pan, waveform)
WAVETABLE(start, dur, amplitude, frequency - 10.5, 0.5, waveform)
//-----Chorus
outskip = 0
outdur = 9
instart = 0.00
indur = 0.20
maintain_dur = 1
transposition = -0.05
nvoices = 10
minamp = 0.1
maxamp = 0.5
minwait = 0.00
maxwait = 0.60
seed = srand()
setline(0,0, .5,1, outdur/8,1, outdur,0)
makegen(2, 7, 1000, 0, 10, 1, 990, 0)
JCHOR(outskip, instart, outdur, indur, maintain_dur, transposition, nvoices, minamp, maxamp, minwait, maxwait,
seed)
//-----Spectacle
inchan = 0
instart = 0
ringdur = 15
amp = 1.0
wet = 0.8
fftlen = 1024
winlen = fftlen * 2
overlap = 2
window = 0
ienv = maketable("line", 1000, 0,0, 1,1, 19,1, 20,0)// input envelope
oenv = maketable("curve", 1000, 0,1,0, 2,1,-1, 3,0)// output envelope
mineqfreq = 0
maxeqfreq = 0
eqtablen = 1000
eq = maketable("line", "nonorm", eqtablen, 0,-90, 200,0, 8000,-3, 22050,-6, 44100, 0)
deltablen = fftlen / 2
mindelfreq = 0
maxdelfreq = 0
mindt = .4 // Delay times
maxdt = 3
seed = srand()
delttime = maketable("random", "nonorm", deltablen, "even", mindt, maxdt, seed)
minfb = .1 // Feedback times
maxfb = .8
fbtime = maketable("random", "nonorm", deltablen, "even", minfb, maxfb, seed)
print_on()
panfrequency = makeconnection("midi", 0.1, 10.0, 0.5, 10, 10, "cntl",14)
print(panfrequency)
pan = makeLFO("saw", panfrequency, 0, 1) // sine for smooth, saw for clicks (later on)
SPECTACLE2(start, instart, dur, amp*oenv, ienv, ringdur, fftlen, winlen, window, overlap, eq, deltime, fbtime,
mineqfreq, maxeqfreq, mindelfreq, maxdelfreq, 0, wet, inchan, pan)

```

```

//Day 31
rtsetparams(44100, 2)
control_rate(44100)
load("DCBLOCK")
load("TRANSBEND")
load("FIR")
load("DEL1")
load("REVERBIT")
load("WAVETABLE")
rtinput("/path/to/file.aiff")

bus_config("DCBLOCK", "in 0-1", "aux 0-1 out")
bus_config("TRANSBEND", "aux 0-1 in", "aux 2-3 out")
bus_config("FIR", "aux 2-3 in", "aux 4-5 out")
bus_config("DEL1", "aux 4-5 in", "out 0-1")

//-----DCBLOCK
start = 0
instart = 0
duration = DUR()
amplitude = 1.0
DCBLOCK(start, instart, duration, amplitude)
//-----TRANSBEND
start = 0
instart = 0
duration = DUR() // length of soundfile
amplitude = 0.8
inchan = 0
pan = 0.5
setline(0,0, 1,1, 90,1, 100,0) // draws straight line segments into array
makegen(0, 18, 512, 1,0.3, 512,-0.6) // transpose from three semitones up, to six down
TRANSBEND(start, instart, duration, amplitude, inchan, pan)
//-----FILTER
start = 0
instart = 0
duration = DUR()
amplitude = 0.8
//remaining calculations are filter coefficients
FIR(start, instart, duration, amplitude, 7,0.9,0.1,0.69,-0.49,0.314,0.2,0.09)
//-----DELAY
start = 0
instart = 0
duration = 60
amplitude = maketable("spline", 1000, "closed", curvature = 25, 0,0.1, 0.5,0.9,1.0,0)
delaytime = 3.14 // R channel delay time, L is default
rightamp = 1.0 // amplitude of R relative to L
DEL1(start, instart, duration, amplitude, delaytime, rightamp)
//-----REV
start = 0
instart = 0
duration = 60
amplitude = 0.8
revtime = maketable("line", 1000, 0,0.01, 0.5,0.1, 1.0,0.2) //keep these short
revamnt = 1 //0 is dry, 1 is wet
chandelay = 0.01 //R channel delay to L channel
cutoff = 1000 //LOP~ cutoff in Hz
REVERBIT(start, instart, duration, amplitude, revtime, revamnt, chandelay, cutoff)
//-----WAVETABLE
st = 0
dur = 10
amp = 1000
env = maketable("line", 1000, 0,0, 0.1,1, 0.9,1, 1.0,0)
freq = 22051
pan1 = makeLF0("sine", 60.5, 0,0.5)
pan2 = makeLF0("sine", 50.75, 0.5,1)
waveform = maketable("random", 8, "gaussian", -1,1)
loop = 0.25
for(st = 0; st < 50; st += loop){
    dur = irand(5, 20)
    WAVETABLE(st, dur, amp*env, freq, pan1, waveform)
    freq -= 100
    WAVETABLE(st+2, dur/2, amp-250*env, freq/2, pan2, waveform)
    freq += 10
    loop = irand(0, 1)
}

//Day 32
rtsetparams(44100, 2)
load("WAVETABLE")
load("PANECHO")

```

```

load("PAN")
control_rate(44100) // how often to update values in the score
srand() // seeded random value generator, empty? seed to clock time

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("PANECHO", "aux 0-1 in", "aux 2-3 out")
bus_config("PAN", "aux 2-3 in", "out 0-1")

waveform1 = maketable("wave3", 10, 0,1, 1,1, 2,1, 3,0, 4,0.5, 5,1)
waveform2 = maketable("wave3", 1000, 0,1, 1,1, 2,1, 3,0, 4,0.5, 5,1)
waveform3 = maketable("wave", 1000, "sine")
waveform4 = maketable("wave", 1000, "saw99")
waveform5 = maketable("wave", 1000, "tri")
waveform6 = maketable("random", 20, "even", -1,1)
waveform7 = maketable("wave", 1000, "buzz")
waveform8 = maketable("wave", 10, "saw") // "bit crushed" waves
waveform9 = maketable("wave", 10, "sine")

wavearray = {waveform1, waveform2, waveform3, waveform4, waveform5, waveform6, waveform7, waveform8, waveform9}
wavelength = len(wavearray)

start = 0
duration = 14
amplitude = 20000 // typically between 1000 - 40000
envelope1 = maketable("line", 1000, 0,0, 0.3,1, 0.5, 0.7, 0.8,0.7, 1.0,0) //ADSR
envelope2 = maketable("random", 10, "gaussian", 0,1)
frequency = irand(10,2000)
pan = 0.5

for(start = 0; start < 1000; start += 1){
    if(start < 700){
        frequency = pickrand(7.00, 7.04, 7.07, 7.09, 8.00, 8.04, 8.07, 8.09) // A-7
        pan = makeLFO("sine", 0.0125, 0,1)
        WAVETABLE(start, duration, 1000*envelope1, frequency, pan, waveform1)
    }

    if(start >=250 && start <=700){
        waveindex = trunc(irand(0, wavelength))
        wave = wavearray[waveindex]
        frequency = pickrand(7.00, 7.04, 7.05, 7.09, 8.00, 8.04, 8.05, 8.09) //F maj 7
        amplitude = pickrand(100, 200, 250)
        pan = makeLFO("saw", 2, 0,1)
        duration = pickrand(1, 1, 2, 3, 5, 8, 13)
        WAVETABLE(start, duration, amplitude*envelope2, frequency, pan, wave)
    }

    if(start >=500){
        waveindex = trunc(irand(0, wavelength))
        wave = wavearray[waveindex]
        frequency = pickrand(6.10, 7.00, 7.02, 7.05, 7.07) // Bb pentatonic
        amplitude = pickrand(50, 75, 100)
        duration = irand(1,10)
        WAVETABLE(start, duration, amplitude*envelope1, frequency, pan, wave)
    }
}

//-----ping pong delay
start = 0
instart = 0
duration = 9*60 // three minutes long
amplitude = 0.8
envelope = maketable("line", 1000, 0,0, 0.5,1, 3.5,1, 7,0)
delaytimeleft = 3.14
delaytimeright = 1.07
feedback = 0.7
ringdownduration = 3
PANECHO(start, instart, duration, amplitude*envelope, delaytimeleft, delaytimeright, feedback,
ringdownduration)

//-----panning
start = 0
instart = 0
duration = 9*60
amplitude = 2.0
envelope = maketable("line", 1000, 0,0, 1,1, duration-1,1, duration,0)
inchannel = 0
panmode = 1 //0 for constant power pan, 1 for linear pan
dynamicpan = maketable("random", 20, "gaussian", -1,1)
paninvert = makefilter(dynamicpan, "invert", 0.5)

```

```

pan = makefilter(paninvert, "fitrange", -1.0,1.0)
PAN(start, instart, duration, amplitude*envelope, inchannel, panmode, pan)

//Day 33
rtsetparams(44100, 2)
load("METAFLUTE") //flute physical model family of instruments
print_off()

//-----SFLUTE (most basic flute model)
start = 0
duration = 5
noiseamp = 0.01 //noise amplitude relative to overall
length1 = 20 //samples, between 5-200, lengths will alter pitch (documentation has a rough tuning-table)
length2 = 200
amplitude = 7000
pan = 0.5
/*
still uses the old style makegens for envelopes, one for noise amplitude and
another for the overall amplitude
*/
table = 1
gentype = 24 //
size = 1000
makegen(table, gentype, size, 0,1, 1.5,1)

table = 2
gentype = 24
size = 1000
makegen(table, gentype, size, 0,0, 0.05,1, 1.49,1, 1.5,0)

SFLUTE(start, duration, noiseamp, length1, length2, amplitude, pan)

//-----VSFLUTE (with vibrato)
start = 3
duration = 5
noiseamp = 0.1
length1lowvalue = 70 //create low and high values for vibrato depth
length1highvalue = 72
length2lowvalue = 40
length2highvalue = 43
amplitude = 7000
vibrato1freqlow = 0.5 //Hz
vibrato1freqhigh = 4.0
vibrato2freqlow = 1.9
vibrato2freqhigh = 3.2
pan = 0.5
/*
requires makegens
1) table 1 for noise amplitude envelope
2) table 2 for overall amplitude envelope
3) table 3 to build a waveform for length 1s frequencies
4) table 4 ditto, but for length 2
*/
table = 1
gentype = 7
size = 1000
makegen(table, gentype, size, 1, 1000, 1)

table = 2
gentype = 7
size = 1000
makegen(table, gentype, size, 1, 1000, 1)

table = 3
gentype = 10
size = 1000
makegen(table, gentype, size, 1)

table = 4
gentype = 10
size = 1000
makegen(table, gentype, size, 1)

VSFLUTE(start, duration, noiseamp, length1lowvalue,length1highvalue, length2lowvalue,length2highvalue,
amplitude, vibrato1freqlow,vibrato1freqhigh, vibrato2freqlow,vibrato2freqhigh, pan)

//-----BSFLUTE (with pitch bending)
start = 7.0
duration = 5
noiseamp = 0.1

```

```

length1lowvalue = 90
length1highvalue = 100
length2lowvalue = 140
length2highvalue = 150
amplitude = 5000
pan = 0.5
//makegens are the same as for vibrato
table = 1
gentype = 7
size = 1000
makegen(table, gentype, size, 1, 1000, 1)

table = 2
gentype = 7
size = 1000
makegen(table, gentype, size, 1, 1000, 1)

table = 3
gentype = 10
size = 1000
makegen(table, gentype, size, 1)

table = 4
gentype = 10
size = 1000
makegen(table, gentype, size, 1)

BSFLUTE(start, duration, noiseamp, length1lowvalue, length1highvalue, length2lowvalue, length2highvalue,
amplitude, pan)

//Day 34
rtsetparams(44100, 2)
load("WAVETABLE")
load("MMODALBAR")
load("REV")
controlrate = control_rate(44100)
srand()
print_offf()

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("MMODALBAR", "aux 0-1 out")
bus_config("REV", "aux 0-1 in", "out 0-1")

//-----start with an array of control rates
rate_array = {2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096}
rate_array_size = len(rate_array)
random_rate = trunc(trand(0, rate_array_size))
//current_rate = rate_array[random_rate]

//-----now to get a few pitch arrays
Cmaj7_array = {7.00, 7.04, 7.07, 7.11}
Gmaj7_array = {7.02, 7.06, 7.07, 7.11} //put in second inversion to fit within one octave
Amin7_array = {7.00, 7.04, 7.07, 7.09} //foist inversion
Bbmaj7_array = {7.02, 7.05, 7.09, 7.10} // "

//-----a few universals for the above
pitch_array = Cmaj7_array //or any of the others...
pitch_array_size = len(pitch_array)
random_pitch = trunc(trand(0, pitch_array_size))
current_pitch = pitch_array[random_pitch]

//-----on to the wavetables
start = 0
duration = 10
amplitude = 2000
//envelope = maketable("line", 1000, 0,0, 0.5,1.0, 1.0,0)
pitch_array = Cmaj7_array
pan = 0.5
waveform = maketable("wave3", 100, 0,1,0, 1,1,0, 2,0.5,0, 3,0.5,0, 5,1.0,0, 8,0.5,0, 13,0.25,0, 21,0.001,0)

//-----for modal bars
barstart = 0
barduration = 3.0
baramplitude = 80000
barnote = cpspch(8.09)
stickhardness = 1.0 //0.0-1.0
stickposition = 0.1 //0.0-1.0
barinstrument = 3
barpan = makeLF0("sine", 0.2, 0,1)
loop = 2.5

```

```

for(start = 0; start < 200; start += 2.5){
    random_pitch = trunc(trand(0, pitch_array_size))
    current_pitch = pitch_array[random_pitch]
    pitch = current_pitch

    WAVETABLE(start, duration, amplitude, cpspch(pitch), pan, waveform)
    current_pitch = pitch_array[random_pitch]
    pitch = current_pitch

    WAVETABLE(start+1, duration, amplitude, cpspch(pitch)+1, pan, waveform)

    barpan = random()
    MMODALBAR(start, barduration, baramplitude, barnote, stickhardness, stickposition, barinstrument,
        barpan)

    if(start >= 75 && start <= 110){
        pitch_array = Gmaj7_array
        WAVETABLE(start, duration, amplitude, cpspch(pitch), pan, waveform)
        MMODALBAR(start+0.25, barduration, baramplitude, barnote, stickhardness, stickposition,
            barinstrument, barpan)
    }

    if(start >= 90 && start <= 150){
        pitch_array = Amin7_array
        WAVETABLE(start, duration, amplitude, pitch, pan, waveform)
    }

    if(start > 140){
        pitch_array = Bbmaj7_array
        WAVETABLE(start+0.9, duration, amplitude, pitch, pan, waveform)
        MMODALBAR(start, barduration, baramplitude, barnote, stickhardness, stickposition,
            barinstrument, barpan)
    }

    random_rate = trunc(trand(0, rate_array_size))
    current_rate = rate_array[random_rate]
    loop = random() + 2
}

//-----reverb
start = 0
instart = 0
duration = 60 * 5
amplitude = 0.9
type = 1 // 1 is Perry Cook's, 2 is John Chowning's, 3 is Michael McNabb's
rvbtime = 2.5
rvbpct = 0.5
inchan = 0
REV(start, instart, duration, amplitude, type, rvbtime, rvbpct, inchan)
REV(start+1, instart, duration, amplitude, type, rvbtime, rvbpct, inchan)

//Day 34
rtsetparams(44100, 2)
load("WAVETABLE")
load("MMODALBAR")
print_off()

rate_array = {2, 4, 8, 16, 32, 64, 128, 44100}
rate_length = len(rate_array)
rate = rate_array[trand(0, rate_length)]

fibonacci_array = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233}
fibonacci_length = len(fibonacci_array)

//-----WAVETABLE
start = 0
duration = 1 //duration and amplitude taken care of by fibonacci array above
amplitude = 10000

env1 = maketable("line", 1000, 0,0, 0.4,1, 1,0.5, 10,0)
env2 = maketable("line", 1000, 0,0, 1,0.03, 2,0.1, 3,0.15, 4,0.2, 5,0.4, 6,0.6, 7,0.8, 7.5,1, 10,0)
env3 = maketable("line", 1000, 0,0, 1,0.1, 2,0.0, 3,0.4, 4,0.1, 5,0.6, 6,0.2, 7,0.9, 8,0.2, 9,1, 10,0)
env4 = maketable("line", 1000, 0,0, 5,1, 10,0)
env5 = maketable("line", 1000, 0,0, 1,0.1, 6,0.2, 9,1, 10,0)
env6 = maketable("line", 1000, 0,0, 0.4,1, 1,0.3, 2,0.1, 3,0.3, 4,0.1, 5,0.3, 6,0.1, 7,0.3, 8,0.1, 9,0.3, 10,0)

envelope_array = {env1, env2, env3, env4, env5, env6}
envelope_length = len(envelope_array)
envelope = envelope_array[trand(0, envelope_length)]

```



```

white_keys = pickrand(8.00, 8.02, 8.04, 8.05, 8.07, 8.09, 8.11)
black_keys = pickrand(8.01, 8.03, 8.06, 8.08, 8.10)
Cmaj7 = pickrand(7.00, 7.04, 7.07, 7.11, 8.00, 8.04, 8.07, 8.11)
Gmaj7 = pickrand(7.07, 7.11, 8.02, 8.06, 8.07, 8.11, 9.02, 9.06)
Amin7 = pickrand(7.09, 8.00, 8.04, 8.07, 8.09, 9.00, 9.04, 9.07)
Dmin7 = pickrand(7.02, 7.05, 7.09, 8.00, 8.02, 8.05, 8.09, 9.00)
Bmin7 = pickrand(7.11, 8.02, 8.06, 8.09, 8.11, 9.02, 9.06, 9.09)
pitch_array = {white_keys, black_keys, Cmaj7, Gmaj7, Amin7, Dmin7}
pitch_length = len(pitch_array)
pitch = pitch_array[trand(0, pitch_length)]

wave1 = maketable("wave3", 10, 0,1, 1,1, 2,1, 3,0, 4,0.5, 5,1)
wave2 = maketable("wave3", 1000, 0,1, 1,1, 2,1, 3,0, 4,0.5, 5,1)
wave3 = maketable("wave", 1000, "sine")
wave4 = maketable("wave", 1000, "saw99")
wave5 = maketable("wave", 1000, "tri")
wave6 = maketable("random", 20, "even", -1,1)
wave7 = maketable("wave", 10, "buzz")
wave8 = maketable("wave", 10, "saw")
wave9 = maketable("wave", 10, "sine")
waveform_array = {wave1, wave2, wave3, wave4, wave5, wave6, wave7, wave8, wave9}
waveform_length = len(waveform_array)
waveform = waveform_array[trand(0, waveform_length)]

panL = maketable("random", 10, "low", 0,0.6)
panR = maketable("random", 10, "high", 0.4,1)

increment = 0.25

for(start = 0; start < 25; start += increment){
    rate = rate_array[trand(0, rate_length)]
}

//left channel wavetable
for(start = 0; start < 525; start += increment){
    duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)
    envelope = envelope_array[trand(0, envelope_length)]
    pitch = pitch_array[trand(0, pitch_length)]
    waveform = waveform_array[trand(0, waveform_length)]
    panL = maketable("random", 10, "low", 0,0.6)
    WAVETABLE(start, duration, amplitude*envelope, cpspch(pitch), panL)
}

//right channel wavetable
for(start = 0; start < 525; start += increment){
    duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)
    envelope = envelope_array[trand(0, envelope_length)]
    pitch = pitch_array[trand(0, pitch_length)]
    waveform = waveform_array[trand(0, waveform_length)]
    panR = maketable("random", 10, "low", 0,0.6)
    WAVETABLE(start, duration, amplitude*envelope, cpspch(pitch), panR)
}

//-----MODALBAR
start = 0
duration = 0.02 //seconds
amplitude = 30000 //absolute btwn 0-32768
pitch = 7.00
hardness = 1.0
position = 1.0
instrument = 4 //Agogo is 2
pan = 0.5

stagger = 0.01
increment = 0.25

for(start = 0; start < 525; start += increment){
    duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)
    pitch = pitch_array[trand(0, pitch_length)]
    hardness = irand(0.2, 1.0)
    MMODALBAR(start, duration, amplitude, cpspch(pitch), hardness, position, instrument = 4, 0)
    MMODALBAR(start+stagger, duration, amplitude, cpspch(pitch), hardness, position, instrument, 1)
    stagger += 0.5
    pitch = pitch - 1
    pan = trunc(random())
    duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)

```

```

MMODALBAR(start+stagger, duration, amplitude, cpspch(pitch), hardness, position, instrument, pan)
instrument = pickrand(1, 2, 3, 4, 5, 6, 7, 8)
duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)
MMODALBAR(start, duration, amplitude, cpspch(pitch), hardness, position, instrument, pan)
duration += 0.0002
increment = (fibonacci_array[trand(0, fibonacci_length)] / 10)
duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)
MMODALBAR(start, duration, amplitude, cpspch(pitch), hardness, position, instrument, pan)
}

//Day 35
rtsetparams(44100, 2)
load("WAVETABLE")
load("FREEVERB")
load("NOISE")
load("MOOGVCF")
load("MMODALBAR")
srand()
bus_config("WAVETABLE", "aux 0-1 out")
bus_config("FREEVERB", "aux 0-1 in", "out 0-1")
bus_config("NOISE", "aux 2-3 out")
bus_config("MOOGVCF", "aux 2-3 in", "out 0-1")

fibonacci_array = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233}
fibonacci_length = len(fibonacci_array)
total_duration = 360

//-----WAVETABLE
start = 0
duration = 5 //slight overlap
amplitude = 100
envelope = maketable("line", 1000, 0,0, 0.5,1, 1.0,0)
pan = 0.5
pitch_array = {"C4", "D4", "E4", "F4", "G4", "A4", "B4", "C5"}
pitch_length = len(pitch_array)

for(start = 0; start < total_duration; start += 1){
    pitch = pitch_array[trand(0, pitch_length)]
    pan = random()
    WAVETABLE(start, duration, amplitude*envelope, cpslet(pitch), pan)
    WAVETABLE(start+1, duration, amplitude*envelope, (cpslet(pitch) - irand(0.5,5)), pan)
}

//-----Long reverb
start = 0
instart = 0
duration = total_duration
amplitude = 0.9
room = 0.9
predelay = 0.05
ringdur = 10
damp = 30
dry = 40
wet = 90
width = 100
FREEVERB(start, instart, duration, amplitude, room, predelay, ringdur, damp, dry, wet, width)

//-----NOISE
start = 0
duration = total_duration
amplitude = 10000
envelope = maketable("line", 1000, 0,0, 0.6,0.7, 1,0)
pan = makeLFO("sine", 10.5, 0,1)
NOISE(start, duration, amplitude*envelope, pan)

//-----MOOGVCF (for NOISE)
start = 0
instart = 0
duration = total_duration
amplitude = 0.3
envelope = maketable("random", 25, "gaussian", 0,1)
loopfan = 0
pan = makeLFO("sine", 0.25, 0,1)
bypass = 0
lowcf = 50
highcf = 1000
lowres = 0.7
highres = 1.0

```

```

cf = maketable("line", "nonorm", 2000, 0, lowcf, 10*.2, lowcf, 10*.5, highcf, 10, lowcf)
res = maketable("line", "nonorm", 2000, 0, lowres, 1, highres, 2, lowres)
MOOGVCF(start, instart, duration, amplitude*envelope, loopfan, pan, bypass, cf, res)

//-----Long reverb
start = 0
instart = 0
duration = total_duration
amplitude = 0.9
room = 0.9
predelay = 0.05
ringdur = 10
damp = 30
dry = 40
wet = 90
width = 100
FREEVERB(start, instart, duration, amplitude, room, predelay, ringdur, damp, dry, wet, width)

//-----MODALBAR
start = 0
duration = 0.02 //seconds
amplitude = 30000 //absolute btwn 0-32768
pitch = 7.00
hardness = 1.0
position = 1.0
instrument = 4 //Agogo is 2
pan = 0.5

stagger = 0.01
increment = 0.75

for(start = 0; start < total_duration; start += increment){
    duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 40)
    pitch = pitch_array[trand(0, pitch_length)]
    hardness = irand(0.2, 1.0)
    MMODALBAR(start, duration, amplitude, cpslet(pitch), hardness, position, instrument = 4, 0)
    MMODALBAR(start+stagger, duration, amplitude, cpslet(pitch), hardness, position, instrument, 1)
    stagger += 0.5
    pan = trunc(random())
    duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)
    MMODALBAR(start+stagger, duration, amplitude, cpslet(pitch), hardness, position, instrument, pan)
    instrument = pickrand(1, 2, 3, 4, 5, 6, 7, 8)
    duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)
    MMODALBAR(start, duration, amplitude, cpslet(pitch), hardness, position, instrument, pan)
    duration += 0.0002
    increment = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    duration = (fibonacci_array[trand(0, fibonacci_length)] / 10)
    amplitude = (fibonacci_array[trand(0, fibonacci_length)] * 10)
    MMODALBAR(start, duration, amplitude, cpslet(pitch), hardness, position, instrument, pan)
}

//Day 36
rtsetparams(44100, 2) //EXTRA thanks to Joel Matthys for looping assistance
load("STRUM2") //karplus strong algorithm
load("REVERBIT")
load("STRUM") //older kp instrument
load("FLANGE")
load("MOOGVCF")
srand()

bus_config("STRUM2", "aux 0-1 out")
bus_config("REVERBIT", "aux 0-1 in", "out 0-1")
bus_config("STRUM", "aux 2-3 out")
bus_config("FLANGE", "aux 2-3 in", "aux 4-5 out")
bus_config("MOOGVCF", "aux 4-5 in", "out 0-1")

Cmaj7 = pickrand(7.00, 7.04, 7.07, 7.11, 8.00, 8.04, 8.07, 8.11)
Gmaj7 = pickrand(7.07, 7.11, 8.02, 8.06, 8.07, 8.11, 9.02, 9.06)
Amin7 = pickrand(7.09, 8.00, 8.04, 8.07, 8.09, 9.00, 9.04, 9.07)
Bmin7 = pickrand(7.11, 8.02, 8.06, 8.09, 8.11, 9.02, 9.06, 9.09)
pitch_array = {Cmaj7, Gmaj7, Amin7, Bmin7}
pitch_length = len(pitch_array)
pitch = pitch_array[trand(0, pitch_length)]
pitch_length = len(pitch_array)

//-----STRUM2
start = 0
duration = 0.125

```

```

amplitude = 20000
pitch = pitch_array[1]
squish = 1.0 //how squishy is the plucking medium (think hard pick to finger pad)
decay = 1.0 //seconds
pan = random()

for(iteration = 0; iteration < 100; iteration += 1){
  for(index = 0; index < pitch_length; index += 1){
    pitch = pitch_array[index]
    pan = pickrand(0, 1)
    STRUM2(start, duration, amplitude, pitch, squish, decay, pan)
    start = start + 0.125 //change durations here
  }

  for(index = index-1; index >= 0; index -= 1){
    pitch = pitch_array[index]
    pan = pickrand(0, 1)
    STRUM2(start, duration, amplitude, pitch, squish, decay, pan)
    start = start + 0.125
  }
}

//-----REV
start = 0
instart = 0
duration = 60
amplitude = 0.8
envelope = maketable("line", 1000, 0,0, 0.612,1.0, 1.0,0)
revtime = maketable("line", 1000, 0,0.01, 0.5,0.1, 1.0,0.2) //keep these short
revamnt = 0 //0 is dry, 1 is wet
chandelay = 0.1 //R channel delay to L channel
cutoff = 10000 //LOP~ cutoff in Hz
REVERBIT(start, instart, duration, amplitude*envelope, revtime, revamnt, chandelay, cutoff)

//-----STRUM (older, requires makegen and comes in several flavors
start = 0
duration = 0.25
pitch = pitch_array[1]
fundamentaldecaytime = 1.0
nyquistdelaytime = 0.1
amplitude = 5000
squish = 1.0

for(iteration = 0; iteration < 50; iteration += 1){
  for(index = 0; index < pitch_length; index += 1){
    pitch = pitch_array[index]
    pan = pickrand(0, 1)
    START(start, duration, pitch, fundamentaldecaytime, nyquistdelaytime, amplitude, squish, pan)
    start = start + 0.25 //change durations here
  }

  for(index = index-1; index >= 0; index -= 1){
    pitch = pitch_array[index]
    pan = pickrand(0, 1)
    START(start, duration, pitch, fundamentaldecaytime, nyquistdelaytime, amplitude, squish, pan)
    start = start + 0.25
  }
}

//-----FLANGE
start = 0
instart = 0
flangedur = 60
amplitude = 1.0
envelope = maketable("line", 1000, 0,0, 0.305,1.0, 1.0,0)
resonance = maketable("line", 1000, 0,0.2, 0.5,0.9, 1.0,0.2)
maxdelay = 5.0
lowpitch = cpspch(9.09)
moddepth = maketable("line", 1000, 0,10, 1.50)
modrate = maketable("line", 1000, 0,20, 1,12000)
wetdrymix = maketable("line", 1000, 0,0, 1,1)
panfrequency = 0.13// makeconnection("inlet", 3, 0.5)
pan = makeLFO("sine", panfrequency, 0,1)
ringdur = 0
waveform = maketable("wave3", 1000, 1,1,45)
FLANGE(start, instart, flangedur, amplitude*envelope, resonance, maxdelay, moddepth, modrate, wetdrymix, "FIR",
0, pan, ringdur, waveform)

//-----MOOGVCF
start = 0

```

```

instart = 0
duration = 60
amplitude = 0.3
loophan = 0
pan = makeLF0("sine", 0.25, 0,1)
bypass = 0
lowcf = 50
highcf = 1000
lowres = 0.7
highres = 1.0
cf = maketable("line", "nonorm", 2000, 0,lowcf, 10*.2,lowcf, 10*.5,highcf, 10,lowcf)
res = maketable("line", "nonorm", 2000, 0,lowres, 1,highres, 2,lowres)
MOOGVCF(start, instart, duration, amplitude, loophan, pan, bypass, cf, res)

//Day 37
rtsetparams(44100, 2)
load("STRUM2") //karplus strong algorithm
load("REVERBIT")
load("STRUMFB") //karplus with feedback and distortion
load("FLANGE")
load("MOOGVCF")
srand()

bus_config("STRUM2", "aux 0-1 out")
bus_config("REVERBIT", "aux 0-1 in", "out 0-1")
bus_config("STRUMFB", "aux 2-3 out")
bus_config("FLANGE", "aux 2-3 in", "aux 4-5 out")
bus_config("MOOGVCF", "aux 4-5 in", "out 0-1")

Cmaj7 = pickrand(7.00, 7.04, 7.07, 7.11, 8.00, 8.04, 8.07, 8.11)
Gmaj7 = pickrand(7.07, 7.11, 8.02, 8.06, 8.07, 8.11, 9.02, 9.06)
Amin7 = pickrand(7.09, 8.00, 8.04, 8.07, 8.09, 9.00, 9.04, 9.07)
Dmin7 = pickrand(7.02, 7.05, 7.09, 8.00, 8.02, 8.05, 8.09, 9.00)
pitch_array = {Cmaj7, Gmaj7, Amin7, Dmin7}
pitch_length = len(pitch_array)
pitch = pitch_array[trand(0, pitch_length)]
pitch_length = len(pitch_array)
//-----STRUM2
start = 0
duration = 0.25
amplitude = 20000
pitch = pitch_array[1]
squish = 1.0 //how squishy is the plucking medium (think hard pick to finger pad)
decay = 1.0 //seconds
pan = random()

for(iteration = 0; iteration < 100; iteration += 1){
    for(index = 0; index < pitch_length; index += 1){
        pitch = pitch_array[index]
        pan = pickrand(0, 1)
        STRUM2(start, duration, amplitude, pitch, squish, decay, pan)
        start = start + 0.125 //change durations here
    }

    for(index = index-1; index >= 0; index -= 1){
        pitch = pitch_array[index]
        pan = pickrand(0, 1)
        STRUM2(start, duration, amplitude, pitch, squish, decay, pan)
        start = start + 0.125
    }
}

//-----REV
start = 0
instart = 0
duration = 60
amplitude = 0.8
envelope = maketable("line", 1000, 0,0, 0.612,1.0, 1.0,0)
revtime = maketable("line", 1000, 0,0.01, 0.5,0.1, 1.0,0.2) //keep these short
revamnt = 0 //0 is dry, 1 is wet
chandelay = 0.1 //R channel delay to L channel
cutoff = 10000 //LOP~ cutoff in Hz
REVERBIT(start, instart, duration, amplitude*envelope, revtime, revamnt, chandelay, cutoff)
//-----STRUMFB same as STRUM, but with feedback and decay
start = 0
duration = 0.25
amplitude = 5000
pitch = pitch_array[1]
feedbackpitch = pitch - 0.5
squish = 1.0
fundamentaldecaytime = 1.0

```

```

nyquistdecaytime = 0.1
distortiongain = 5.0
feedbackgain = 0.05
cleanlevel = 0
distortionlevel = 1.0

for(iteration = 0; iteration < 100; iteration += 1){

    for(index = 0; index < pitch_length; index += 1){
        pitch = pitch_array[index]
        pan = pickrand(0, 1)
        STRUMFB(start, duration, amplitude, pitch, feedbackpitch, squish, fundamentaldecaytime,
            nyquistdecaytime,distortiongain, feedbackgain, cleanlevel, distortionlevel)
        start = start + 0.25 //change durations here
    }

    for(index = index-1; index >= 0; index -= 1){
        pitch = pitch_array[index]
        pan = pickrand(0, 1)
        STRUMFB(start, duration, amplitude, pitch, feedbackpitch, squish, fundamentaldecaytime,
            nyquistdecaytime,distortiongain, feedbackgain, cleanlevel, distortionlevel)
        start = start + 0.25
    }
}

//-----FLANGE
start = 0
instart = 0
flangedur = 60
amplitude = 1.0
envelope = maketable("line", 1000, 0,0, 0.305,1.0, 1.0,0)
resonance = maketable("line", 1000, 0,0.2, 0.5,0.9, 1.0,0.2)
maxdelay = 5.0
lowpitch = cpspch(9.09)
moddepth = maketable("line", 1000, 0,10, 1.50)
modrate = maketable("line", 1000, 0,20, 1,12000)
wetdrymix = maketable("line", 1000, 0,0, 1,1)
pan = makeLFO("sine", 13.1, 0,1)
ringdur = 0
waveform = maketable("wave3", 1000, 1,1,45)
FLANGE(start, instart, flangedur, amplitude*envelope, resonance, maxdelay, moddepth, modrate, wetdrymix, "FIR",
0, pan, ringdur, waveform)
//-----MOOGVCF
start = 0
instart = 0
duration = 60
amplitude = 0.3
loopphan = 0
pan = makeLFO("sine", 0.25, 0,1)
bypass = 0
lowcf = 500
highcf = 2000
lowres = 0.2
highres = 1.0
cf = maketable("line", "nonorm", 2000, 0,lowcf, 10*.2,lowcf, 10*.5,highcf, 10,lowcf)
res = maketable("line", "nonorm", 2000, 0,lowres, 1,highres, 2,lowres)
MOOGVCF(start, instart, duration, amplitude, loopphan, pan, bypass, cf, res)

//Day 38
rtsetparams(44100, 2)
/*
Start by seeding the random number generator to CPU clock time
*/
srand()
/*
There are a few ways to generate random numbers in RTcmix
1. rand() will generate a random number between -1 and 1, useful for random waveforms
2. random() will generate a random number between 0 and 1, good for mapping later
3. irand() and trand() are one and the same, or so it seems, and allow a range to be chosen
*/
random_number_a = rand()
random_number_b = random()
random_number_x = irand(0, 10) // or trand() I suppose...
printf("1: %f \n", random_number_a)
printf("2: %f \n", random_number_b)
printf("3: %f \n", random_number_x)
/*
trand() or irand() returns a large number like 2.60620117188, so we could truncate it or round it...
*/
printf("The raw value of our random number is: %f \n", random_number_x)
x_truncated = trunc(random_number_x)
x_rounded = round(random_number_x)

```

```

printf("Random number: %f \n", random_number_x)
printf("Truncated, it equals: %f \n", x_truncated)
printf("Rounded, it equals: %f \n", x_rounded)
/*
The only weighted distribution I know for now is taking the average of two random numbers, or
a "triangle" distribution.
*/
random_number_x = round(trand(0, 10))
random_number_y = round(trand(0, 10))
average_of_random_numbers = ((random_number_x + random_number_y) / 2)
print(random_number_x)
print(random_number_y)
printf("The average is: %f \n", average_of_random_numbers)
/*
So now, I could do this in a loop to get a wide variety of numbers, which will most likely point
to numbers near the middle of 0 and 10. I think.
*/
for(i = 0; i < 10; i += 1){
    print_off()
    random_number_x = round(trand(0, 10))
    random_number_y = round(trand(0, 10))
    average_of_random_numbers = ((random_number_x + random_number_y) / 2)
    print_on()
    print(average_of_random_numbers)
}

/*
RTcmix has a built in makerandom() that will do the same thing, or a maketable() of random numbers
*/
triangular_distribution1 = makerandom("triangle", 10, 0,10) //10 values/sec
triangular_distribution2 = maketable("random", 10, "triangle", 0,10)
//plottable(triangular_distribution2)
/*
So Joel Matthys has a library of functions that will do some of the things found most commonly in
the C library of math functions. sin, cos, etc., but I'm interested in map, which works like the
[expr_scale] object in Pure Data.
*/
load("jfuncs")
print(sin(1))
print(cos(1))
print(lowrand()) //all fit in the range of 0-1
print(highrand())
print(trirand())
print(gaussrand())
print(constrain(5.3235,4.2,-98.543))
print(map(2,1,4,50,100))
print(prob(0.5,0.1))
/*
Using these, it's possible to generate random numbers from any given range
*/
srand() //these aren't reacting to srand()?
value = highrand() // random number between 0 and 1, weighted high
print(value)
value_mapped = map(value, 0,1, 0,25) //like [expr_scale], fit in range 0-25
print(value_mapped)
/*
Filling some panning values with a table of random numbers between 0 and 1, weighted toward R
*/
pan = maketable("random", 100, "high", 0,1)
//plottable(pan)
/*
Now put it to use in the PAN() instrument for more precise control
*/
load("WAVESHAPE")
load("PAN")
print_off()

bus_config("WAVESHAPE", "aux 0-1 out")
bus_config("PAN", "aux 0-1 in", "out 0-1")

//-----WAVESHAPE
start = 0
duration = 20
frequency = makeLFO("sine", 0.1, 40,22100)
mindistortionindex = 0.0
maxdistortionindex = 1.0
amplitude = 20000
envelope = makerandom("prob", 10, -1.0,1.0,0,0)
pan = 0.5 //pan later
wave = maketable("wave", 1000, "sine")
transferfunc = maketable("cheby", 1000, 0.9, 0.3,-0.2,0.6,-0.7)
indexguide = maketable("line", 1000, 0,0, 3.5,1, 7,0)

```

```

WAVESHAPE(start, duration, frequency, mindistortionindex, maxdistortionindex, amplitude*envelope, pan, wave,
transferfunc, indexguide)

//-----PAN
start = 0
instart = 0
duration = 20
amplitude = maketable("line", 1000, 0,10.0, 0.8,0.5, 1.0,0)
inputchannel = 0
pantype = 1 //0 for constant power, 1 for linear
pan = maketable("random", 10, "high", 0,1)
//plottable(pan)
PAN(start, instart, duration, amplitude, inputchannel, pantype, pan)
/*
Now, modtable(normalize) will do the same thing as map, but with elements in a table
*/
random_numbers = maketable("random", "nonorm", 10, "gaussian", 0,50) //important to include "nonorm"
plottable(random_numbers)
table_mod = modtable(random_numbers, "normalize", 4000) //constrain between 0.0 and 4000
plottable(table_mod)

//-----WAVETABLE
load("WAVETABLE")
start = 20
duration = 10
amplitude = 3000
frequency = table_mod
pan = modtable(random_numbers, "normalize", 1)
WAVETABLE(start, duration, amplitude, frequency, pan)

//Day 39
rtsetparams(44100, 2) //A drum beat
load("WAVETABLE")
load("PAN")
load("REV")
load("FLANGE")
load("JCHOR")
control_rate(4)
print_off()
srand()

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("WAVETABLE", "out 0-1")

bus_config("PAN", "aux 0-1 in", "aux 2-3 out")
bus_config("PAN", "aux 0-1 in", "aux 4-5 out")
bus_config("REV", "aux 2-3 in", "out 0-1")

bus_config("FLANGE", "aux 4-5 in", "aux 6-7 out")
bus_config("JCHOR", "aux 6-7 in", "out 0-1")

totalduration = 360
//-----Bass Drum 1
start = 0
bd_duration = 1
bd_amplitude = 20000
bd_envelope = maketable("line", 1000, 0,1.0, 0.8,1.0, 1.0,0)
bd_frequency = 70
bd_pan = 0.5
bd_wave = maketable("wave", 1000, "sine")

bd_loop = 2
for(start = 0; start < totalduration; start += bd_loop){
    WAVETABLE(start, bd_duration, bd_amplitude, bd_frequency, bd_pan, bd_wave) //BD1

    if(start >= 36){
        WAVETABLE(start, bd_duration, bd_amplitude-15000, bd_frequency*2, bd_pan, bd_wave)
    }

    if(start >= 48){
        WAVETABLE(start, bd_duration, bd_amplitude-19999, bd_frequency*24, bd_pan, bd_wave)
    }

}

//-----Hi Hat 1
start = 8
hh_duration = 0.01
hh_amplitude = 60000
hh_frequency = 3000

```



```

hh_pan = random()
hh_wave = maketable("random", 5, "high", -1,1)

hh_loop = 0.5
for(start = 8; start < totalduration; start += hh_loop){
    WAVETABLE(start, hh_duration, hh_frequency, hh_pan = random(), hh_wave) //HH1
}

//-----Snare
start = 16
sd_duration = 0.025
sd_amplitude = 10000
sd_frequency = 54052
sd_pan = makeLFO("sine", 8.0, 0,1)
sd_wave = maketable("wave", 1000, "buzz")

sd_loop = 1
for(start = 16; start < totalduration; start += sd_loop){
    if(start%2 == 0){
        WAVETABLE(start+sd_loop, sd_duration, sd_amplitude, sd_frequency, sd_pan, sd_wave) //SD
    }
}

//-----Hi Hat 2
start = 24
hh_duration = 0.001
hh_amplitude = 15000
hh_frequency = 30000
hh_pan = random()
hh_wave = maketable("random", 5, "high", -1,1)

hh_loop = 0.25
for(start = 24; start < totalduration; start += hh_loop){
    WAVETABLE(start, hh_duration, hh_frequency, hh_pan = random(), hh_wave) //HH2
}

//-----Bass Drum 2
start = 36
bd_duration = 0.75
bd_amplitude = 0.2
bd_frequency = 630
bd_pan = random()
bd_wave = maketable("wave", 1000, "sine", 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1) //lots of partials
bd_loop = 2
for(start = 36; start < totalduration; start += bd_loop){
    WAVETABLE(start, bd_duration, bd_frequency*3, bd_pan = random(), bd_wave) //BD2
}

//-----PAN
start = 36
instart = 0
duration = totalduration
amplitude = 10.0
envelope = maketable("line", 1000, 0,0, 0.4,1.0, 0.9,1.0, 1.0,0)
inchannel = 0
panmode = 1 // 0 for constant power pan, 1 for linear pan
pan = makeLFO("sine", 4.25, 0,1)
PAN(start, instart, duration, amplitude*envelope, inchannel, panmode, pan)

//-----REV
start = 36
instart = 0
duration = totalduration
amplitude = 10.0
type = 1 // 1 is Perry Cook's, 2 is John Chowning's, 3 is Michael McNabb's
rvbtime = 0.25
rvbpct = 0.05
inchan = 0
REV(start, instart, duration, amplitude, type, rvbtime, rvbpct, inchan)

//-----FLANGE
start = 44
instart = 0 //again, 0 from auxiliary bus
duration = totalduration
flamplitude = 40.0
flenvelope = maketable("random", 100, "low", 0.3,1.0)
resonance = irand(0.3,1.0)
maxdelay = 5.0
moddepth = maketable("line", "nonorm", 100, 0,1, 5,5, 10,7)

```

```

modspeed = maketable("line", 1000, 0,0.1, 1,0.9)
wetdrymix = 0.8
filtertype = "IIR"
inputchannel = 0
pan = makeLFO("sine", 4.0, 0,1)
ringdownduration = lowrand()
flangewave = maketable("wave", 2000, "sine")
FLANGE(start, instart, duration, flamplitude*flenvelope, resonance, maxdelay, moddepth, modspeed, wetdrymix,
filtertype, inputchannel, pan, ringdownduration, flangewave)

//-----CHORUS
start = 44
instart = 0.00
outdur = totalduration
indur = 0.01
maintain_dur = 1
transposition = -0.05
nvoices = 10
minamp = 0.1
maxamp = 0.5
minwait = 0.00
maxwait = 0.60
seed = srand()
setline(0,0, .5,1, outdur/8,1, outdur,0)
makegen(2, 7, 1000, 0, 10, 1, 990, 0)
JCHOR(start, instart, outdur, indur, maintain_dur, transposition, nvoices, minamp, maxamp, minwait, maxwait,
seed)

# Day 40
# RTcmix not only uses MINC as its parser, but also Python, a powerful programming language. There are
# some differences in style (take for example the commenting...), but both MINC and Python work well
# in creating new score files.

# Remember, when executing RTcmix scripts in the Terminal, you need to call the command cmix followed
# by the < symbol. (cmix < /path/to/scorefile.sco) In order to use Python, you first need to configure
# RTcmix to interpret Python scrips (done by running ./configure --with-python during installation), and
# then call pycmix < /path/to/scorefile.sco in the Terminal window.

# First, pycmix score files need the following command to run...

from rtcmix import *

# which allows Python to use the list of cmix commands. (I *think* Python comes preloaded on Mac OSX)

#Then the header of your file looks the same as it has using MINC

rtsetparams(44100, 2)
load("WAVETABLE")

#-----WAVETABLE
start = 0
duration = 5
amplitude = 90 #in dB
pitch = 7.09 #octave point pitch class
pan = 0.5
wavetype = maketable("wave", 1000, "sine")
WAVETABLE(start, duration, ampdB(amplitude), pitch, pan, wavetype)

# The above preceding code produces a five second sine wave on the note A. Now, it's possible to set a
# loop (looks a little different than MINC) and generate some random pitches.

start = 5
duration = 1
amplitude = 90 #in dB
pitch = 7.09 #octave point pitch class
pan = 0.5
wavetype = maketable("wave", 1000, "sine")

for start in range(5, 23):
    print_off()
    pitch = pickrand(7.00, 7.01, 7.02, 7.03, 7.04, 7.05, 7.06, 7.07, 7.08, 7.09, 7.10, 7.11)
    WAVETABLE(start, duration, ampdB(amplitude), pitch, pan, wavetype)
    print_on()
    print(midipch(pitch))

# This type of loop uses range( ), which is intrinsic to Python. In this case with range(5, 23) we are
# asking for a loop that STARTS at 5, counts up incrementally to 22 by 1. There are other ways to use
# range, such as range(0, 101, 2) which counts from 0-100 by 2 instead of by 1. # Conditional statements in
# Python also look a bit different.

```

```

# Day 41
from rtmix import *

rtsetparams(44100, 2)
load("WAVETABLE")

#-----WAVETABLE
start = 0
duration = 0.25
amplitude = 90 #in dB
pitch = 7.09 #octave point pitch class
pan = 0.5
wavetype = maketable("random", 16, "gaussian", -1,1)

# Getting through loops with a floating point number. (Is there an easier way?) Similar to how I've been
# using increment or loop as variables in MINC loops...
# i.e - loop = 0.25; for(start = 0; start < 10; start += loop)

def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r += step

for i in drange(0.0, 11.0, 0.25):
    pitch = pickrand(7.00, 7.01, 7.02, 7.03, 7.04, 7.05, 7.06, 7.07, 7.08, 7.09, 7.10, 7.11)
    WAVETABLE(i, duration, ampdB(amplitude), pitch, pan, wavetype)

    random_number = trunc(irand(0, 10))

    if random_number == 5:
        pan = 0

    elif random_number < 5:
        pan = 0

    else:
        pan = 1

# Day 42
# Checking out arrays in Python, help from John Gibson's 02.13.2001 help file

from rtmix import *

rtsetparams(44100, 2)
load("WAVETABLE")

#-----WAVETABLE
start = 0
duration = 0.0625
amplitude = 90 #in dB
pitch_array = [5.00, 5.001, 5.02, 5.03, 5.05, 5.07, 5.069, 5.10, 6.00]
pitch_array_length = len(pitch_array)
pan = 0.5
wavetype = maketable("random", 16, "gaussian", -1,1)

def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r += step

for i in drange(0.0, 11.0, duration):
    slot = int(random() * pitch_array_length * .999999)
    current_pitch = pchoct(pitch_array[slot])

    WAVETABLE(i, duration, ampdB(amplitude), current_pitch, pan, wavetype)

    random_number = trunc(irand(0, 10))

    if random_number == 5:
        pan = 0

    elif random_number < 5:
        pan = 0.5
        pitch = ampdB(9.01)

    else:
        pan = 1

```

```

//Day 43
/*
The [rtcmix~] object in Pure Data now includes the ability to play sound from data stored in an array. It
requires two
messages...

1. First, a "bufset nameofarray" message needs to be banged FIRST. This points the [rtcmix~] object to the
array in
question. So, if my array is called 'array1' - or the name of the default array name in Pd - my message will
thus be
"bufset array1". If my array is called 'foo', I'll say "bufset foo", etc.

2. Lastly, your rtinput( ) message needs to say "PDBUF" and "nameofarray". Example below
*/

rtinput("PDBUF", "array1") //necessary "PDBUF" and "nameofarray", in this case 'array1'.

//-----STEREO for playback (MIX is another useful instrument here)
bus_config("STEREO", "in 0-1", "out 0-1")

start = 0
instart = 0
//your duration matches the length of 'sound' in your array. Here, I'm writing to the array for ten seconds.
duration = 10
amplitude = 1.0
pan = 0.5
STEREO(start, instart, duration, amplitude, pan)

/*
...now, to play it backwards...
*/

//-----SCRUB
bus_config("SCRUB", "in 0-1", "out 0-1")
start = 10
instart = 10 //to play backwards, set instart to end of file, in this case ten seconds
duration = 10
amplitude = 1.0
speed = -1.0 //play backwards, 1.0 normal, 0.5 1/2 speed, -0.5 backwards 1/2 speed, 2.0
2X as fast...
sync_width = 16
oversampling = 40 //16 and 40 seem to be good defaults
inchannel = 0
pan = 0.5
SCRUB(start, instart, duration, amplitude, speed, sync_width, oversampling, inchannel, pan)

/*
... and have some "Meeblip-Y" fun with it!
*/

//-----STEREO again
start = 20
instart = 0
duration = 0.75
amplitude = 1.0
pan = random()

loop = 0.5 //slight overlap for each iteration
for(start = 20; start < 100; start += loop){
    instart = irand(0.25,5.0)
    pan = random()
    STEREO(start, instart, duration, amplitude, pan)
}

// Day 44
//generating rhythms to append with pitches

print_off()
srand()
rhythmgamut = {
    0.125, //eighth note
    0.25, //quarter note
    0.33, //triplet quarter note
    0.5, //half note
    1.0 //whole note
}
rhythmgamut_length = len(rhythmgamut)

```

```

octavegamut = {5, 6, 7}
octavegamut_length = len(octavegamut)

pitchgamut = {0.00, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.10, 0.11}
pitchgamut_length = len(pitchgamut)

for(start = 0; start < 10; start += 1){
    //rhythms
    random_number_x = trand(0,rhythmgamut_length)
    random_number_y = trand(0,rhythmgamut_length)

    if(random_number_x >= random_number_y){
        placeholder = random_number_x
    }

    else{
        placeholder = random_number_y
    }

    nextrhythm = rhythmgamut[placeholder]

    //octave
    random_number_x = trand(0,octavegamut_length)
    random_number_y = trand(0,octavegamut_length)

    if(random_number_x >= random_number_y){
        placeholder = random_number_x
    }

    else{
        placeholder = random_number_y
    }

    octave = octavegamut[placeholder]

    //pitches
    random_number_x = trand(0,pitchgamut_length)
    random_number_y = trand(0,pitchgamut_length)

    if(random_number_x >= random_number_y){
        placeholder = random_number_x
    }

    else{
        placeholder = random_number_y
    }

    pitch = pitchgamut[placeholder]
    nextpitch = octave+pitch

    print_on()
    print(nextpitch)
    print(nextrhythm)
    print_off()
}

//Day 45
rtsetparams(96000, 2)
load("HALFWAVE")
load("REVERBIT")
srand()
reset(8)

bus_config("HALFWAVE", "aux 0-1 out")
bus_config("REVERBIT", "aux 0-1 in", "out 0-1")

totalduration = 360

//-----HALFWAVE
start = 0
duration = 1.75

octavegamut = {2, 10, 11}
octavegamut_length = len(octavegamut)

pitchgamut = {0.00, 0.01, 0.04, 0.05, 0.06, 0.08, 0.10, 0.11}
pitchgamut_length = len(pitchgamut)

amplitude = 1000
envelope = maketable("line", 100, 0,0, 2,1, 9,0)
wave1 = maketable("wave3", 1000, 3.14,1,0, 6.28,1,0.5)
wave2 = maketable("wave3", 1000, 1.00,1,0, 2.00,1,0.5)

```

```

wave3 = maketable("wave3", 1000, 1,1,0, 3,0.3,0, 5,0.2,0, 7,0.05,0, 9,0.01,0, 11,0.001,0)
wave4 = maketable("wave3", 1000, 1,1,0, 2,0.5,0, 3,0.3,0, 4,0.25,0, 5,0.2,0, 6,0.16,0, 7,0.14,0, 8,0.125,0)
wave5 = maketable("wave3", 1000, 1,1,0, 3,0.14,0, 5,0.04,0, 7,0.02,0, 9,0.012,0, 11,0.008,0)
wavegamut = {wave1, wave2, wave3, wave4, wave5}
wavegamutlength = len(wavegamut)

wavecrossoverpoint = 0.5
pan = 0.5

increment = 0.5

for(start = 0; start < totalduration; start += increment){
    octave = octavegamut[trand(0,octavegamut_length)]
    nextpitch = pitchgamut[trand(0,pitchgamut_length)]
    pitch = nextpitch + octave

    if(octave == 2){
        duration = 5.0
        pan = makeLFO("sine", 1.25, 0,1)
        HALFWAVE(start, duration, cpspch(pitch)+0.04, amplitude*envelope, wave1, wave2,
            wavecrossoverpoint, pan)
    }

    wave1 = wavegamut[trand(0,wavegamutlength)]
    wave2 = wavegamut[trand(0,wavegamutlength)]

    random_number_x = round(trand(0, 10))
    random_number_y = round(trand(0, 10))
    average_of_random_numbers = ((random_number_x + random_number_y) / 2)
    wavecrossoverpoint = average_of_random_numbers / 10

    pan = random()
    HALFWAVE(start, duration, cpspch(pitch), amplitude*envelope, wave1, wave2, wavecrossoverpoint, pan)
    increment = average_of_random_numbers / 7
}

//-----REVERBIT
start = 0
instart = 0
duration = totalduration
amplitude = 1.0
envelope = maketable("line", 1000, 0,0, 0.4,1.0, 0.9,0, 1.0,0)
revtime = maketable("line", 1000, 0,1.0, 0.5,1.0, 1.0,0.2) //keep these short
revamnt = 1 //0 is dry, 1 is wet
chandelay = maketable("random", 100, "gaussian", 0.01,2.0)
cutoff = 2000 //LOP~ cutoff in Hz
REVERBIT(start, instart, duration, amplitude*envelope, revtime, revamnt, chandelay, cutoff)srand()

//Day 46
//building waveform from an array

amplitudes_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

for(i = 0; i < 10; i += 1){
    randomnumber_x = irand(-1,1)
    randomnumber_y = irand(-1,1)
    partial_amplitude = (randomnumber_x + randomnumber_y) / 2

    amplitudes_array[i] = partial_amplitude
}

waveform = maketable("wave3", 31, 1, amplitudes_array[1], 0,
    2, amplitudes_array[2], 0,
    3, amplitudes_array[3], 0,
    5, amplitudes_array[4], 0,
    8, amplitudes_array[5], 0,
    13,amplitudes_array[6], 0,
    21,amplitudes_array[7], 0,
    34,amplitudes_array[8], 0,
    55,amplitudes_array[9], 0,
    89,amplitudes_array[10],0)

plottable(waveform)

//Day 47
rtsetparams(44100, 2)
load("HALFWAVE")
srand()

amplitudes_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

```

```

for(i = 0; i < 10; i += 1){
    randomnumber_x = irand(-1,1)
    randomnumber_y = irand(-1,1)
    partial_amplitude = (randomnumber_x + randomnumber_y) / 2

    amplitudes_array[i] = partial_amplitude
}

//-----HALFWAVE
start = 0
duration = 10

octavegamut = {2, 10, 11}
octavegamut_length = len(octavegamut)

pitchgamut = {0.00, 0.01, 0.04, 0.05, 0.06, 0.08, 0.10, 0.11}
pitchgamut_length = len(pitchgamut)

octave = octavegamut[trand(0,octavegamut_length)]
nextpitch = pitchgamut[trand(0,pitchgamut_length)]
pitch = nextpitch + octave

amplitude = 1000
waveform1 = maketable("wave3", 31, 1, amplitudes_array[1], 0,
    3, amplitudes_array[2], 0,
    5, amplitudes_array[3], 0,
    5, amplitudes_array[4], 0,
    7, amplitudes_array[5], 0,
    8, amplitudes_array[6], 0,
    11, amplitudes_array[7], 0,
    14, amplitudes_array[8], 0,
    15, amplitudes_array[9], 0,
    20, amplitudes_array[10], 0)
waveform2 = makefilter(waveform1, "invert", 0)
wavecrossoverpoint = 0.5
pan = 0.5

HALFWAVE(start, duration, cpspch(pitch), amplitude, waveform1, waveform2, wavecrossoverpoint, pan)

//Day 48
rtsetparams(44100, 2)
load("HALFWAVE")
reset(8)
srand()

amplitudes_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

for(i = 0; i < 10; i += 1){
    randomnumber_x = irand(-1,1)
    randomnumber_y = irand(-1,1)
    partial_amplitude = (randomnumber_x + randomnumber_y) / 2

    amplitudes_array[i] = partial_amplitude
}

//-----HALFWAVE
start = 0
duration = 360

octavegamut = {1, 2, 3}
octavegamut_length = len(octavegamut)

pitchgamut = {0.00, 0.01, 0.04, 0.05, 0.06, 0.08, 0.10, 0.11}
pitchgamut_length = len(pitchgamut)

octave = octavegamut[trand(0,octavegamut_length)]
nextpitch = pitchgamut[trand(0,pitchgamut_length)]
pitch = nextpitch + octave

amplitude = 1000
waveform1 = maketable("wave3", 100, 1, amplitudes_array[1], -0.5,
    2, amplitudes_array[2], 1.0,
    6, amplitudes_array[3], 0.25,
    7, amplitudes_array[4], 0.0,
    9, amplitudes_array[5], -0.125,
    10, amplitudes_array[6], -0.33,
    11, amplitudes_array[7], 1.0,
    14, amplitudes_array[8], -1.0,
    15, amplitudes_array[9], 0.0,
    40, amplitudes_array[10], 0.25)

```

```

wavecrossoverpoint1 = maketable("line", 10, 0,0, 1.0,1.0)
pan1 = makeLFO("sine", 0.05, 0.0,0.6)

waveform2 = makefilter(waveform1, "invert", 0)
wavecrossoverpoint2 = maketable("line", 10, 0,1.0, 1.0,0)
pan2 = makeLFO("sine", 0.25, 0.4,1.0)

HALFWAVE(start, duration, cpspch(2.11),          amplitude, waveform1, waveform2, wavecrossoverpoint1, pan1)
HALFWAVE(start, duration, cpspch(2.10)+0.01, amplitude, waveform1, waveform2, wavecrossoverpoint2, pan2)

//Day 49
rtsetparams(44100, 2)
load("WAVETABLE")
load("DELAY")
load("JFUNCS")
rate = pickrand(2, 4, 8, 16, 32, 128, 512, 1024, 22051, 44051)
control_rate(8)
seed = srand()

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("DELAY", "aux 0-1 in", "out 0-1")

//-----WAVETABLE
start = 0
total_duration = 360

maxdb = 90
mindb = 62
amplitude = ampdb(maxdb-mindb)
envelope = maketable("curve", 1000, 0,1,-5, 1,0)

maxfrequency = 44052
minfrequency = 22051
waveform = maketable("wave", 1000, "sine")

maxincrement = 0.4
minincrement = 0.05
increment = 0.1

for (start = 0; start < total_duration; start += increment){
    duration = increment * 0.2
    amplitude = ampdb(irand(mindb, maxdb))
    frequency = irand(minfrequency, maxfrequency)
    glissfactor = irand(0.2, 0.7)

    if (random() < 0.5){ // 50% probability of glissing up
        glissfactor += 1
    }

    gliss = maketable("curve", "nonorm", 1000, 0,1,-2, 1,glissfactor)
    pan = random()
    WAVETABLE(start, duration, amplitude*envelope, frequency*gliss,          pan, waveform)
    WAVETABLE(start, duration, amplitude*envelope, frequency*gliss*1.01, 1-pan, waveform)

    increment = irand(minincrement, maxincrement)
    control_rate(rate)
}

//-----DELAY
start = 0
instart = 0
duration = total_duration
amplitude = 1.0
envelope = maketable("window", 1000, "hanning")
delttime = makerandom("even", 0.5, 0.1, 2.2, seed)
delaytime = makefilter(delttime, "smooth", lag = 0)
feedback = maketable("line", 1000, 0,0.18, 0.5,0.62, 1.0,0.02)
ringdur = 0.5 // seconds to ring out delay line after note is finished
inputchannel = 0
pan = 1
DELAY(start, instart, duration, amplitude*envelope, delaytime, feedback, ringdur, inputchannel, pan)

delttime = makerandom("low", 2.5, 0.1, 3.3, seed)
delaytime = makefilter(delttime, "smooth", lag = 0)
inputchannel = 1
pan = 0
DELAY(start, instart, duration, amplitude*envelope, delaytime, feedback, ringdur, inputchannel, pan)

//Day 50
rtsetparams(44100, 2)

```



```

load("MIX")
load("REV")
print_off()

rtinput("/path/to/file.wav")
bus_config("MIX", "in 0-1", "aux 0-1 out")
bus_config("REV", "aux 0-1 in", "out 0-1")

totalduration = 176 * 2

//-----MIX
start = 0
instart = 0
duration = 0.5
amplitude = 0.5
left = 0
right = 1

increment = 1.0

for(start = 0; start < totalduration; start += increment){
    instart = irand(1,176)
    left = irand(0,0.5)
    right = irand(0.5,1.0)
    MIX(start, instart, duration, amplitude, left,right)
    duration += 0.02
}

//-----REV
start = 0
instart = 0
duration = totalduration
amplitude = 1.0
type = 3
rvbtime = 0.15
rvbpct = 0.08
inchan = 0

REV(start, instart, duration, amplitude, type, rvbtime, rvbpct, inchan)

//Day 51
rtsetparams(44100, 2)
load("STRUM2")

Cmajor = {7.00, 7.02, 7.04, 7.05, 7.07, 7.09, 7.11, 8.00}
arraylength = len(Cmajor)

//-----STRUM2
start = 0
duration = 0.5
amplitude = 20000
pitch = Cmajor[0]
squish = 1.0 //how squishy is the plucking medium (think hard pick to finger pad)
decay = 1.0 //seconds
pan = random()

transposition = 0.01

for(iteration = 0; iteration < 12; iteration += 1){
    for(index = 0; index < arraylength; index += 1){
        pitch = Cmajor[index] + transposition
        pan = pickrand(0, 1)
        STRUM2(start, duration, amplitude, pitch, squish, decay, pan)
        start = start + 0.33 //change durations here
    }

    for(index = index-1; index >= 0; index -= 1){
        pitch = Cmajor[index] + transposition
        pan = pickrand(0, 1)
        STRUM2(start, duration, amplitude, pitch, squish, decay, pan)
        start = start + 0.33
    }

    transposition += 0.01
}

//Day 52
rtsetparams(44100, 2)
load("SCRUB")
load("STEREO")

```

```

rtinput("/Users/jerod_s/Desktop/movie_camera/scorefiles/nyman.wav")

//-----SCRUB meta
start = 0
instart = 0
amplitude = 10.0
width = 16
oversampling = 40
inchannel = 0

//-----tape head 1
durationL = 1.0
speedL = 1.0
panL = 1.0           //RTcmix panning is L = 1.0, center = 0.5, R = 0.0

//-----tape head 2
speedR = 0.90
panR = 0.0
durationR = durationL * speedR

for(start = 0; start < 100; start += 1.5){
    SCRUB(start, instart, durationL, amplitude, speedL, width, oversampling, inchannel, panL)
    SCRUB(start, instart, durationR, amplitude, speedR, width, oversampling, inchannel, panR)
}

//Day 53
rtsetparams(96000, 2)
load("TRANS")
load("REVERBIT")

bus_config("TRANS", "in 0-1", "aux 0-1 out")
bus_config("REVERBIT", "aux 0-1 in", "out 0-1")

rtinput("/path/to/file.wav")

//-----TRANS
start = 0
instart = 0
duration = DUR()
amplitude = 1.0
transposition = maketable("line", 1000, 0,0.0, 0.5,-0.2, 1.0,-1.0)
inchannel = 0
pan = 0.5
TRANS(start, instart, duration, amplitude, transposition, inchannel, pan)

//-----REVERBIT(so it sounds like its in my glass mantle)
start = 0
instart = 0
duration = DUR()
amplitude = 1.0
reverbtime = 5.8
reverbpct = 0.9
channeldelay = 0.1
cutoff = 500
REVERBIT(start, instart, duration, amplitude, reverbtime, reverbpct, channeldelay, cutoff)

//SC193.sco 07.12.2013
rtsetparams(44100, 2) //guitar chorale
load("STRUM2")

soprano = {7.07, 7.09, 7.11, 8.00, 8.02, 8.00, 7.11, 8.07, 8.06, 8.04, 8.02, 8.00, 7.09, 7.06, 7.07}
alto = {7.02, 7.02, 7.02, 7.04, 7.02, 7.02, 7.02, 7.07, 7.09, 7.07, 7.07, 7.07, 7.04, 7.02, 7.02}
tenor = {6.11, 6.09, 6.07, 6.07, 6.09, 6.06, 6.07, 6.07, 7.02, 6.07, 6.07, 6.07, 6.07, 7.00, 6.11}
bass = {5.07, 5.06, 5.07, 5.04, 5.06, 6.02, 5.07, 5.11, 5.09, 6.00, 5.11, 6.04, 6.00, 6.02, 5.07}

//-----STRUM2
start = 0
duration = 2
amplitude = 20000
pitch = 8.00
squish = 0.3
decay = 3.0
pan = random()

placeholder = 1

for(start = 0; start < 15*duration; start += duration){
    pan = random()

    STRUM2(start, duration, amplitude, cpspch(soprano[placeholder]), squish, decay, random())
}

```

```

        STRUM2(start, duration, amplitude, cpspch(alto[placeholder]), squish, decay, random())
        STRUM2(start, duration, amplitude, cpspch(tenor[placeholder]), squish, decay, random())
        STRUM2(start, duration, amplitude, cpspch(bass[placeholder]), squish, decay, random())

        placeholder += 1
    }

//Day 54
rtsetparams(44100, 2)
load("WAVETABLE")
load("MROOM")
control_rate(44100)

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("MROOM", "aux 0-1 in", "out 0-1")

soprano = {7.07, 7.09, 7.11, 8.00, 8.02, 8.00, 7.11, 8.07, 8.06, 8.04, 8.02, 8.00, 7.09, 7.06, 7.07}
alto = {7.02, 7.02, 7.02, 7.04, 7.02, 7.02, 7.02, 7.07, 7.09, 7.07, 7.07, 7.07, 7.04, 7.02, 7.02}
tenor = {6.11, 6.09, 6.07, 6.07, 6.09, 6.06, 6.07, 6.07, 7.02, 6.07, 6.07, 6.07, 6.07, 7.00, 6.11}
bass = {5.07, 5.06, 5.07, 5.04, 5.06, 6.02, 5.07, 5.11, 5.09, 6.00, 5.11, 6.04, 6.00, 6.02, 5.07}

//-----WAVETABLE
start = 0
duration = 2
amplitude = 10000
envelope = maketable("window", 1000, "hanning")
pitch = 8.00
pan = random()
waveform = maketable("wave", 1000, "sine")

placeholder = 1

for(start = 0; start < 15 * duration; start += duration){
    pan = random()

    WAVETABLE(start, duration, amplitude*envelope, soprano[placeholder]+1, random(), waveform)
    WAVETABLE(start, duration, amplitude*envelope, alto[placeholder]+1, random(), waveform)
    WAVETABLE(start, duration, amplitude*envelope, tenor[placeholder]+1, random(), waveform)
    WAVETABLE(start, duration, amplitude*envelope, bass[placeholder]+1, random(), waveform)

    placeholder += 1
}

//-----CHURCH HALL
start = 0
instart = 0
duration = 30
amplitude = 0.5
xdim = 500 //distance (feet) from middle of room to right wall
ydim = 800 //distance (feet) from middle of room to front
rvbtime = 25.0
reflect = 90.0 //amount reflected by walls
innerwidth = 4.0
inchan = 0
quantizationrate = 96000

timeset(0, 0-xdim, 0-ydim)
timeset(duration, xdim, ydim)

setline(0,0, duration/8,1, duration-.5,1, duration,0)
MROOM(start, instart, duration, amplitude, xdim, ydim, rvbtime, reflect, innerwidth, inchan, quantizationrate)

//Day 55
rtsetparams(44100, 2)
load("WAVETABLE")
load("FREEVERB")
srand()

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("FREEVERB", "aux 0-1 in", "out 0-1")

soprano = {7.07, 7.09, 7.11, 8.00, 8.02, 8.00, 7.11, 8.07, 8.06, 8.04, 8.02, 8.00, 7.09, 7.06, 7.07}
alto = {7.02, 7.02, 7.02, 7.04, 7.02, 7.02, 7.02, 7.07, 7.09, 7.07, 7.07, 7.07, 7.04, 7.02, 7.02}
tenor = {6.11, 6.09, 6.07, 6.07, 6.09, 6.06, 6.07, 6.07, 7.02, 6.07, 6.07, 6.07, 6.07, 7.00, 6.11}
bass = {5.07, 5.06, 5.07, 5.04, 5.06, 6.02, 5.07, 5.11, 5.09, 6.00, 5.11, 6.04, 6.00, 6.02, 5.07}

//-----WAVETABLE
start = 0
totalduration = 1000
amplitude = 10000

```

```

envelope = maketable("window", 1000, "hanning")
pitch = 8.00
pan = random()
waveform = maketable("wave", 1000, "sine")

sopranoplaceholder = 0
altoplaceholder = 0
tenorplaceholder = 0
bassplaceholder = 0

sopranoduration = 10           //tendency to speed up
altoduration = 10             //starts fast and relaxes
tenorduration = 11            //pretty even keel, cool person
bassduration = 14             //strangely slow heart rate overall

for(start = 0; start < totalduration; start += sopranoduration){
    sopranoduration += irand(0,0.6)
    WAVETABLE(start, sopranoduration, amplitude*envelope, soprano[sopranoplaceholder]+1, random(),
        waveform)
    sopranoplaceholder += 1
}

for(start = 0; start < totalduration; start += altoduration){
    altoduration -= irand(0,0.01)
    WAVETABLE(start, altoduration, amplitude*envelope, alto[altoplaceholder]+1, random(), waveform)
    altoplaceholder += 1
}

for(start = 0; start < totalduration; start += tenorduration){
    tenorduration -= irand(0,0.01)
    WAVETABLE(start, tenorduration, amplitude*envelope, tenor[tenorplaceholder]+1, random(), waveform)
    tenorplaceholder += 1
}

for(start = 0; start < totalduration; start += bassduration){
    bassduration += irand(-0.3,0.5)
    WAVETABLE(start, bassduration, amplitude*envelope, bass[bassplaceholder]+1, random(), waveform)
    bassplaceholder += 1
}

//-----FREEVERB
start = 0
instart = 0
duration = totalduration
amplitude = 0.3
room = 1.0
predelay = 0.0001
ringdur = 15
damp = 80
dry = 10
wet = 90
width = 100
FREEVERB(start, instart, duration, amplitude, room, predelay, ringdur, damp, dry, wet, width)

//Day 56
rtsetparams(44100, 2)
load("NOISE")
load("MULTICOMB")
load("PAN")
control_rate(44100)
srand()

bus_config("NOISE", "aux 0-1 out")
bus_config("MULTICOMB", "aux 0-1 in", "aux 2-3 out")
bus_config("PAN", "aux 2-3 in", "out 0-1")

//-----NOISE
start = 0
duration = 0.009
amplitude = 50000
envelope1 = maketable("line", 1000, 0,0, 5,1, 10,0)
envelope2 = maketable("line", 1000, 0,0, 7,0.5, 9,1, 10,0)
envelope3 = maketable("line", 1000, 0,0, 0.1,1, 0.5,0.05, 10,0)
pan = random()

//-----MULTICOMB
start = 0
instart = 0
combdur = duration
combamp = 0.5
combenv = maketable("line", 1000, 0,0, 0.1,0.8, 0.8,0.5, 1.0,0)

```

```

low = cpspch(13.01)
high = cpspch(13.05)
revtime = 0.1 //seconds

//-----PAN
start = 0
instart = 0
duration = duration
panamp = 1.0
inchannel = 0
panmode = 1 //0 for constant power pan, 1 for linear pan
pan = random()

loop = 0.001

for (start = 0; start < 250; start += loop){
    NOISE(start, duration, amplitude*envelope3, random())
    MULTICOMB(start, instart, duration+0.5, combamp*combenv, low, high, revtime)
    PAN(start, instart, duration, panamp, inchannel, panmode, random())

    step = irand(-0.2,0.3)
    loop += step

    if(loop >= 0.1){
        duration += 0.01
        revtime += 0.01
    }
}

//Day 57
rtsetparams(44100, 2)
load("WIGGLE")
srand()

//-----WIGGLE
start = 0
duration = 8.2
amplitude = maketable("line", "nonorm", 1000, 0,0, 0.1,2000, 5,4000, 10,2000)
envelope = maketable("curve", 2000, 0,0,2, 2.5,1,0, 13,1,-3, 25,0)

min = 2.00
max = 12.00
gliss = maketable("random", "nonorm", "nointerp", 500, "low", min, max)
pitch = 18.09
freq = makeconverter(octpch(pitch) + gliss, "cpsoct")

mod_depth_type = 1 // 0 is no mod, 1 is % of carrier, 2 is FM
filt_type = 2 // 0 is no filt, 1 is low, 2 is high
filt_steep = 2
balance = true // balance output and input signals
carrierwaveform = maketable("wave", 24051, "sine")
mod_wavetable = maketable("wave", 1000, "sine")
mod_freq = 200
mod_depth = 20
filt_cf = maketable("curve", "nonorm", 2000, 0,1000,-4, 1,1)
pan = random()

for(start = 0; start < 100; start += 5){
    pan = random()
    freq -= 10.2
    WIGGLE(start, duration, amplitude*envelope, freq, mod_depth_type, filt_type, filt_steep, balance,
        carrierwaveform, mod_wavetable, mod_freq, mod_depth, filt_cf, pan)
}

//Day 58
rtsetparams(44100, 2)
load("GRANSYNTH")
seed = srand()

//-----GRANSYNTH
start = 0
duration = 10
amplitude = 10000
waveform = maketable("wave", 1000, "tri")
genv = maketable("window", 1000, "hanning")
ghop = maketable("random", 1000, "low", 0.01, 1.0)
goutjitter = 0.01
gdurmin = 0.0001 // longer durations = "pitchy"
gdurmax = 0.009 // shorter are "clicky"
gampmin = 0.2
gampmax = 1.0

```

```

gpitches = {15.01, 19.03, 19.09, 19.11, 21.0, 23.08}
gplength = len(gpitches)
gtrans = maketable("literal", "nonorm", 0, 0.01, 0.03, 0.11, 1.0) // octpcs
gtransjitter = 1.0 // maximum random amount to shift current pitch
panmin = 0.0
panmax = 1.0

loop = 1.0 // for loop

for (start = 0; start < 25; start += loop){
    pitchindex = trand(gplength)
    gpitch = gpitches[pitchindex]
    GRANSYNTH(start, duration, amplitude, waveform, genv, ghop, goutjitter, gdurmin, gdurmax, gampmin,
        gampmax, gpitch, gtrans, gtransjitter, seed, panmin, panmax)
    goutjitter -= 0.01
    gpitch += 30
}

//Day 59
rtsetparams(44100, 2)
load("WAVETABLE")
load("REVERBIT")

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("REVERBIT", "aux 0-1 in", "out 0-1")

//-----dates
age = 30
month = 7.00
date = 19

//-----meta
amplitude = age * 1000
envelope = maketable("line", 1000, 1,1, 9,1, 10,0)
pan = makeLFO("sine", age/100, 0,1)
waveform = maketable("wave3", 1000, 1,1,0, age/10,1,0, 5,1,0, month,1,0, date,1,0, age,1,0)

//-----melody in the key of "F"
pitchgamut = {month, month+.02, month+.04, month+.05, month+.07, month+.09, month+.10, month+1.00}

//-----durations
teenth = 0.275
sixdot = 0.6
quatre = 1.0

WAVETABLE(0, sixdot, amplitude*envelope, cspch(pitchgamut[0]), pan, waveform)
WAVETABLE(0.625,   tenth, amplitude*envelope, cspch(pitchgamut[0]), pan, waveform)
WAVETABLE(1,      quatre, amplitude*envelope, cspch(pitchgamut[1]), pan, waveform)
WAVETABLE(2,      quatre, amplitude*envelope, cspch(pitchgamut[0]), pan, waveform)
WAVETABLE(3,      quatre, amplitude*envelope, cspch(pitchgamut[3]), pan, waveform)
WAVETABLE(4,      quatre, amplitude*envelope, cspch(pitchgamut[2]), pan, waveform)
WAVETABLE(6,      sixdot, amplitude*envelope, cspch(pitchgamut[0]), pan, waveform)
WAVETABLE(6.625,  tenth, amplitude*envelope, cspch(pitchgamut[0]), pan, waveform)
WAVETABLE(7,      quatre, amplitude*envelope, cspch(pitchgamut[1]), pan, waveform)
WAVETABLE(8,      quatre, amplitude*envelope, cspch(pitchgamut[0]), pan, waveform)
WAVETABLE(9,      quatre, amplitude*envelope, cspch(pitchgamut[4]), pan, waveform)
WAVETABLE(10,     quatre, amplitude*envelope, cspch(pitchgamut[3]), pan, waveform)
WAVETABLE(12,     sixdot, amplitude*envelope, cspch(pitchgamut[0]), pan, waveform)
WAVETABLE(12.625, tenth, amplitude*envelope, cspch(pitchgamut[0]), pan, waveform)
WAVETABLE(13,     quatre, amplitude*envelope, cspch(pitchgamut[7]), pan, waveform)
WAVETABLE(14,     quatre, amplitude*envelope, cspch(pitchgamut[5]), pan, waveform)
WAVETABLE(15,     sixdot, amplitude*envelope, cspch(pitchgamut[3]), pan, waveform)
WAVETABLE(15.625, tenth, amplitude*envelope, cspch(pitchgamut[3]), pan, waveform)
WAVETABLE(16,     quatre, amplitude*envelope, cspch(pitchgamut[2]), pan, waveform)
WAVETABLE(17,     quatre, amplitude*envelope, cspch(pitchgamut[1]), pan, waveform)
WAVETABLE(19,     sixdot, amplitude*envelope, cspch(pitchgamut[6]), pan, waveform)
WAVETABLE(19.625, tenth, amplitude*envelope, cspch(pitchgamut[6]), pan, waveform)
WAVETABLE(20,     quatre, amplitude*envelope, cspch(pitchgamut[5]), pan, waveform)
WAVETABLE(21,     quatre, amplitude*envelope, cspch(pitchgamut[3]), pan, waveform)
WAVETABLE(22,     quatre, amplitude*envelope, cspch(pitchgamut[4]), pan, waveform)
WAVETABLE(23,     quatre+1, amplitude*envelope, cspch(pitchgamut[3]), pan, waveform)

//-----REVERBIT
start = 0
instart = 0
duration = age
amplitude = age/100
reverbtime = month+date
reverbpct = age/100+age/100+age/100
channeldelay = age/100
cutoff = age*100

```

```

REVERBIT(start, instart, duration, amplitude, reverbtime, reverbpct, channeldelay, cutoff)

//Day 60
rtsetparams(44100, 2)
load("MIX")
load("COMBIT")
srand()

rtinput("/path/to/file.wav")

bus_config("MIX", "in 0-1", "aux 0-1 out")
bus_config("COMBIT", "aux 0-1 in", "out 0-1")

totalduration = DUR()

//-----MIX
start = 0
instart = 0
duration = 0.5
amplitude = 0.1
envelope = maketable("line", 1000, 0,0, 0.5,1, 1.0,0)
left = 0
right = 1

increment = 1.0

for(start = 0; start < totalduration; start += increment){
    instart = irand(0,30)
    MIX(start, instart, duration, amplitude*envelope, left,right)
}

//-----COMBIT
start = 0
instart = 0
duration = DUR()
amplitude = 10.0
ringtime = 1.1
inchan = 0

COMBIT(start, instart, duration, amplitude, cpspch(7.11), ringtime, inchan, random())
COMBIT(start, instart, duration, amplitude, cpspch(8.07), ringtime, inchan, random())
COMBIT(start, instart, duration, amplitude, cpspch(8.09), ringtime, inchan, makeLF0("sine", 4.075, 0,1))

//Day 61
rtsetparams(44100, 2)
load("VOCODESYNTH")
srand()

rtinput("/path/to/file.aiff")

//-----VOCODESNTH
start = 0
instart = 0
duration = DUR()
amplitude = 15.0
envelope = maketable("line", 1000, 0,0, .1,0.5, 0.6,0.5, 1.0,0)
numbands = 50
lowcf = maketable("random", 1000, "gaussian", 300,500)
interval = 0.07
cartransp = 0.03
bw = maketable("random", 100, "low", 0,1) / 100

winlen = 0.2
smooth = 0.98
thresh = 0.0001
atktime = 0.001
reltime = 0.01
hipassmod = 0.03
hipasscf = maketable("line", 1000, 0,1000, 1.0,3000)
spacemult = cpspch(interval) / cpspch(0.0)
inchannel = 0
pan = 0.5
carwavetable = maketable("wave", 1000, "sine")
scale1 = 0.5
scale2 = 1.0
scalecurve = maketable("curve", "nonorm", 1000, 0,scale1,1, 1,scale2)

VOCODESYNTH(start, instart, duration, amplitude*envelope, numbands, lowcf, spacemult, cartransp, bw, winlen,
smooth, thresh, atktime, reltime, hipassmod, hipasscf, inchannel, pan, carwavetable, scalecurve)

//Day 62

```

```

rtsetparams(44100, 2)
load("FMINST")
srand()

start = 0
duration = 0.05
amplitude = 5000
carrier = cpspch(7.00)
modulatorfrequency = 220
minindex = 10
maxindex = 30
pan = random()
waveform = maketable("wave", 1000, "sine")
guide = maketable("line", "nonorm", 1000, 0, 0, 5, 1, 7, 0)

increment = 0.125

for(start = 0; start < 100; start += increment){
    duration = irand(0.01, 0.1)

    FMINST(start, duration, amplitude, carrier, modulatorfrequency, minindex, maxindex, pan = random(),
           waveform, guide)
    modulatorfrequency = carrier * 22051
    carrier = start * 0.14
}

//Day 63
rtsetparams(44100, 2)
load("JGRAN")
srand()

//-----JGRAN
start = 0
duration = 2.0
amplitude = 1.0
randomseed = srand()
oscconfig = 0 // FM
oscphase = 0 // randomize osc phase? 0 is no; 1 is yes
grainenv = maketable("window", 1000, "hanning")
grainwaveclickhi = maketable("wave", 1000, "sine")
grainwaveclicklow = maketable("wave", 1000, "tri")
FMmult = maketable("random", 1000, "gaussian", 2.0, 100.0)
FMindex = maketable("line", 1000, 0, 1.0, 1.0, 30.0)
minfreq = 30
maxfreq = 22051
minspeed = maketable("line", "nonorm", 1000, 0, 0.2, 1, 0.001) // decreasing minimum
maxspeed = maketable("line", "nonorm", 1000, 0, 0.001, 1, 0.2) // increasing maximum
mindb = 0
maxdb = 90
density = maketable("random", 1000, "gaussian", 0, 10)
pan = random()
panrand = maketable("random", 1000, "gaussian", 0, 1)

loop = 0.0125

for (start = 0; start < 80; start += loop){
    JGRAN(start, duration, amplitude, randomseed, oscconfig, oscphase, grainenv, grainwaveclickhi, FMmult,
          FMindex, minfreq, maxfreq, minspeed, maxspeed, mindb, maxdb, density, pan = random(),
          panrand)
}

//Day 64
rtsetparams(44100, 2)
load("STEREO")
load("PVOC")
load("PAN")

rtinput("/path/to/file.aiff")
bus_config("STEREO", "in 0-1", "aux 0-1 out")
bus_config("PVOC", "aux 0-1 in", "aux 2 out")
bus_config("PAN", "aux 2 in", "out 0-1")

totalduration = DUR()

//-----STEREO
start = 0
instart = 0
duration = 1.0
amplitude = 0.5

```



```

envelope = maketable("line", 1000, 0,0.25, 0.2,1.0, 0.9,1.0, 1.0,0.25)
pan = 0.5

increment = 0.25

for(start = 0; start < totalduration; start += increment){
    instart = irand(0, totalduration)
    STEREO(start, instart, duration, amplitude*envelope, pan = random())
    duration += 0.02
}

//-----PVOC
start = 0
instart = 0
duration = totalduration
gain = 1.0
fftsize = 1024
winsize = fftsize * 2
decim = 1024
interp = 16
pitch = 0.0

PVOC(start, instart, duration, gain, 0, fftsize, winsize, decim, interp, pitch)

//-----PAN
start = 0
instart = 0
duration = 1.0
amplitude = 1.0
inchannel = 0
pantype = 1
pan = random()

for(start = 0; start < totalduration; start += increment){
    PAN(start, instart, duration, amplitude, inchannel, pantype, pan = random())
}

//Day 65

srand()
print_off()
pitchgamut = {"C", "C#", "D", "Eb", "E", "F", "F#", "G", "Ab", "A", "Bb", "B", "C"}
octavegamut = {"3", "4", "5"}

octaveplaceholder = 0
pitchplaceholder = 0

for(start = 0; start < 15; start += 1){
    /*random_number_x = irand(0,11)
    random_number_y = irand(0,11)

    if(random_number_x >= random_number_y){
        pitchplaceholder = random_number_x + 1
    }

    else{
        pitchplaceholder = random_number_y + 1
    }*/

    //pitchplaceholder = ((random_number_x + random_number_y) / 2)

    pitchplaceholder = irand(0,11)

    random_number_a = irand(0,3) + 1
    random_number_b = irand(0,3) + 1

    octaveplaceholder = ((random_number_a + random_number_b) / 2)

    //adjust pitches to stay within soprano sax range
    /*if(octaveplaceholder < 1){
        pitchplaceholder += 1
    }*/

    /*if(octaveplaceholder >= 2){
        pitchplaceholder += trunc(irand(0,3))
    }*/

    //now put them together
    octave = octavegamut[octaveplaceholder]
    pitch = pitchgamut[pitchplaceholder]

```

```

        nextpitch = pitch+octave
        print_on()
        print(nextpitch); print_off()
    }

//Day 66
rtsetparams(44100, 2)
load("AMINST")

start = 0
duration = 0.15
amplitude = 40000
carrierfrequency = 22051
modulatorfrequency = 44100
pan = random()
modulatoramplitude = maketable("line", 1000, 0,0, 1,1, 2,0)
carrierwave = maketable("wave", 1000, "sine")
modulatorwave = maketable("random", 20, "gaussian", -1,1)

increment = 0.45

for(start = 0; start < 250; start += increment){
    random_value = trand(0,10)
    if(random_value <= 6){
        carrierfrequency += 2
        modulatorfrequency += 4
    }
    if(random_value >= 6){
        carrierfrequency -= 3
        modulatorfrequency -= 2
    }
    pan = random()
    AMINST(start, duration, amplitude, carrierfrequency, modulatorfrequency,
        pan, modulatoramplitude, carrierwave, modulatorwave)
    increment = irand(0,1) / 2
}

//Day 67
/*
spray_init( ) and get_spray( ) are similar to the Pd object [urn].
It returns a series of random numbers, but doesn't repeat a number that has been chosen.
Thus, "unrepeated random number."
*/

spray_table = 1
spray_size = 10
seed = srand()

spray_init(spray_table, spray_size, seed) // initialize table 3 with 7 elements

for (i = 0; i < spray_size; i += 1) {
    current_value = get_spray(spray_table) + 1
    print(current_value)
}

//Day 68
rtsetparams(44100, 2)
load("MBRASS")

Cmajor = {7.00, 7.02, 7.04, 7.05, 7.07, 7.09, 7.11, 8.00}
arraylength = len(Cmajor)

//-----Trumpet-ish
start = 0
duration = 1
amplitude = 40000
pitch = Cmajor[0]
slide_length = 103
lip_filter = 140
max_pressure = 0.045
pan = 0.5
breath = maketable("line", 1000, 0,0, 0.05,1, 3.0,3, 3.5,0)

transposition = 0.01
loop = 0.125

for(iteration = 0; iteration < 12; iteration += 1){
    for(index = 0; index < arraylength; index += 1){

```

```

        pitch = Cmajor[index] + transposition
        slide_length -= 4
        lip_filter += 10
        max_pressure += 0.155
        pan = pickrand(0, 1)
        MBRASS(start, duration, amplitude, pitch, slide_length, lip_filter, max_pressure, pan,
            breath)
        start += loop //change durations here
    }

    for(index = index-1; index >= 0; index -= 1){
        pitch = Cmajor[index] + transposition
        slide_length += 4
        lip_filter -= 10
        max_pressure -= 0.155
        pan = pickrand(0, 1)
        MBRASS(start, duration, amplitude, pitch, slide_length, lip_filter, max_pressure, pan,
            breath)
        start += loop
    }
    transposition += 0.01
}

//Day 68 //Tweet script
rtsetparams(44100,2)load("WAVETABLE")f=140g=maketable("expbrk",10,0,1,10,9) for(i=0;i<50;i+=5){f+=i*20
WAVETABLE(i,20,1750,i*g,random())}

//Day 69 //Tweetable
rtsetparams(44100,2)
load("WAVETABLE")
load("JFUNCS")
reset(8)for(i=0;i<1000;i+=1){WAVETABLE(i,i/2,500,abs(sin(i)*220.5),sin(i))WAVETABLE(i/2,i/
3,500,abs(sin(i)*441),cos(i))}

//Day 70
rtsetparams(44100, 4) //four channels
load("WAVETABLE")
load("NPAN")

bus_config("WAVETABLE", "aux 0 out")
bus_config("NPAN", "aux 0 in", "out 0-3")

//-----START
start = 0
duration = 30
amplitude = 10000
frequency = 440
pan = 0 //send everything through channel 0
waveform = maketable("random", 50, "gaussian", -1,1) //noisy waveform to hear spatialization

for(start = 0; start < 25; start += 1){
    WAVETABLE(start, duration, amplitude, frequency, pan, waveform)
    frequency += irand(10,50)
}

//-----NPAN
NPANspeakers("polar",
    45, 1, // left front
    -45, 1, // right front
    135, 1, // left rear
    -135, 1) // right rear

start = 0
instart = 0
duration = 60
amplitude = 1.0
mode = "xy" //or "polar"
x = maketable("random", 100, "high", -1,1)
y = maketable("random", 100, "low", -1,1)
NPAN(start, instart, duration, amplitude, mode, x, y)

//Day 71
rtsetparams(44100, 4) //QPAN, similar to NPAN, but specific for quad sound
load("WAVETABLE")
load("QPAN")

bus_config("WAVETABLE", "aux 0 out")
bus_config("QPAN", "aux 0 in", "out 0-3")

//-----START

```

```

start = 0
duration = 30
amplitude = 10000
frequency = 440
pan = 0 //send everything through channel 0
waveform = maketable("random", 50, "gaussian", -1,1) //noisy waveform to hear spatialization

for(start = 0; start < 25; start += 1){
    WAVETABLE(start, duration, amplitude, frequency, pan, waveform)
    frequency += irand(10,50)
}

//-----QPAN
start = 0
instart = 0
duration = 60
amplitude = 1.0
x = maketable("random", 100, "high", -1,1)
y = maketable("random", 100, "low", -1,1)
QPAN(start, instart, duration, amplitude, x,y)

//Day 72
load("JFUNCS")

values_array = {380, 383, 660, 504, 512, 265, 212, 185, 184, 202, 204, 201, 978, 1008, 234}
num_values = len(values_array)

values_in_table = maketable("literal", "nonorm", (num_values * 2) + 1,
    0, values_array[0],
    1, values_array[1],
    2, values_array[2],
    3, values_array[3],
    4, values_array[4],
    5, values_array[5],
    6, values_array[6],
    7, values_array[7],
    8, values_array[8],
    9, values_array[9],
    10, values_array[10],
    11, values_array[11],
    12, values_array[12],
    13, values_array[13],
    14, values_array[14])

plottable(values_in_table)

values_constrained = modtable(values_in_table, "normalize", 1.0)
plottable(values_constrained)

//"center" all values between the mean value
sum = 0

for(i = 0; i < num_values; i += 1){
    sum += values_array[i]
}

mean_raw = sum / num_values
mean = map(mean_raw, 184,1008, 0,1)
print(mean)

//mean = 407.466, in this instance, scaled to 0.27 via map( )

//okay, so now have values "reflect" over the mean, creating a waveform
waveform = makefilter(values_constrained, "invert", mean)
//plottable(waveform)

//Day 73
include /path/to/a/previous/scorefile.sco

load("WAVETABLE")
srand()

//-----WAVETABLE
start = 0
duration = 3
amplitude = 10000
envelope = maketable("window", 1000, "hanning")
frequency = 0 //initialize
pan = random()
wave = waveform

for(start = 0; start < 30; start += 2){

```

```

        WAVETABLE(start, duration, amplitude*envelope, values_array[start / 2], pan = random(), wave)
    }

//Day 74
rtsetparams(44100, 2)
load("STEREO")
load("WAVESHAPE")
load("JFUNCS")
srand()

rtinput("/Users/jerod_s/Desktop/Script_Cal/input-sounds/brook-sounds.aif")

bus_config("STEREO", "in 0-1", "aux 0-1 out")
bus_config("WAVESHAPE", "aux 0-1 in", "out 0-1")

//take values and / 100 to get between 0 and 1
values_array = {.380, .383, .660, .504, .512, .265, .212, .185, .184, .202, .204, .201, .978, 1.008, .234}
num_values = len(values_array)

//-----WAVESHAPE
start = 0
duration = 10
frequency = 0 //taken from array when in for() loop
min_distortion_index = 0.0
max_distortion_index = 1.0
amplitude = 3000.0
envelope = maketable("window", 1000, "hanning")
pan = random()

waveform = maketable("wave3", (num_values * 2) + 1,
    0, values_array[0],0,
    1, values_array[1],0,
    2, values_array[2],0,
    3, values_array[3],0,
    4, values_array[4],0,
    5, values_array[5],0,
    6, values_array[6],0,
    7, values_array[7],0,
    8, values_array[8],0,
    9, values_array[9],0,
    10, values_array[10],0,
    11, values_array[11],0,
    12, values_array[12],0,
    13, values_array[13],0,
    14, values_array[14],0)

transferfunc = maketable("cheby", 1000, 0.9, 0.3, -0.2, 0.6, -0.7)
indexguide = maketable("line", 1000, 0, 0, 3.5, 1, 7, 0)

for(start = 0; start < 30; start += 2){
    WAVESHAPE(start, duration, (values_array[start / 2] * 2), min_distortion_index, max_distortion_index,
        amplitude*envelope, pan = random(), waveform, transferfunc, indexguide)
}

//Day 75
rtsetparams(44100, 2)
load("WAVETABLE")
seed = srand()

//-----WAVETABLE
start = 0
duration = 10
amplitude = 500
envelope = maketable("line", 1000, 0, 0, 0.1, 1, 0.9, 1, 1.0, 0)
frequency = 7.00
pan = 0.5

loop = 1

for(start = 0; start < 20; start += loop){
    frequency = pickrand(8.00, 8.04, 8.07, 8.11)

    phase1 = makerandom("even", 1000, 0, 1, seed)
    wavetype = maketable("wave3", 1000, 1, 1, phase1)
    pan1 = random()
    WAVETABLE(start, duration, amplitude*envelope, frequency, pan1, wavetype)

    frequency = pickrand(8.00, 8.04, 8.07, 8.11)
    phase2 = (1.0 - phase1)
    pan2 = (1.0 - pan1)
}

```

```

wavetype = maketable("wave3", 1000, 1,1,phase2)
WAVETABLE(start, duration, amplitude*envelope, frequency, pan2, wavetype)

if(pan1 <= 0.10){
    pan3 = makeLFO("sine", 1.25, 0,1)
    WAVETABLE(start, duration, amplitude*envelope, 6.00, pan3, wavetype)
}

}

//Day 76
rtsetparams(44100, 4)
load("MIX")
load("QPAN")
load("JFUNCS")
srand()

rtinput("/Users/JerodSommerfeldt/Desktop/rocks.wav")

bus_config("MIX", "in 0-1", "aux 0-1 out")
bus_config("QPAN", "aux 0-1 in", "out 0-3")

/*
Quad panning with tendency for front speakers to stay to the right
and rear speakers to stay to the left

[1]          [2]  speaker array

    me

[3]          [4]

Recall that RTcmix panning for QPAN looks like this

[-1]          [1]  front

    still me
    [0]

[-1]          [1]  rear
*/

//----- MIX
start = 0
instart = 0
duration = 0.125
amplitude = 1.0
pan = random()

//----- QPAN
start = 0
instart = 0 //reading from aux bus, must be 0
duration = 0.125
amplitude = 1.0
srcX = -1
srcY = 1

increment = 0.125

for(start = 0; start < 100; start += increment){
    instart = irand(0,DUR())
    MIX(start, instart, duration, amplitude, 0,1)

    randomtest1 = highrand()
    srcX = map(randomtest1, 0,1, -1,1)

    randomtest2 = lowrand()
    srcY = map(randomtest2, 0,1, -1,1)
    QPAN(start, instart = 0, duration, amplitude, srcX, srcY)
}

//Day 77
rtsetparams(44100, 2)
load("MIX")
load("STEREO")

rtinput("/path/to/file.wav")

/*

```

```

Going for the "doubling" effect to boost signal, -0.0015 to 0.004 sec delay (Pellman)
*/

//----- MIX
start = 0
instart = 0
duration = DUR()
amplitude = 0.9
MIX(start,          instart, duration, amplitude, 0,0)
MIX(start + 0.004, instart, duration, amplitude, 1,1)

/*
Compare with simply STEREO( )
*/
STEREO(start+DUR(), instart, duration, amplitude, 0.5)

//Day 78
rtsetparams(44100, 2)
load("DCBLOCK")
load("TRANS")
srand()

rtinput("/path/to/file.wav")

bus_config("DCBLOCK", "in 0-1", "aux 0-1 out")
bus_config("TRANS", "aux 0-1 in", "out 0-1")
//----- DCBLOCK
start = 0
instart = 0
duration = DUR()
amplitude = 1.0
DCBLOCK(start, instart, duration, amplitude)

//----- TRANS
start = 0
instart = 0
duration = DUR()
amplitude = 0.8
//transposition
x = random()
y = random()
transpositionL = (((x + y) / 2) / 10)
transpositionR = ((1 - transpositionL) / 10)
panL = 0
panR = 1
TRANS(start, instart, duration, amplitude, transpositionL, inchan = 0, panL)
TRANS(start, instart, duration, amplitude, transpositionR, inchan = 1, panR)

//Day 79
/*
This is an example .c file for a very basic RTcmix function, which can be accessed and used in any
MINC score file. It will be called squareme and after creating and compiling it in your RTcmix build,
you'll be able to use it as myfunction(pfie1)

As advertised, all squareme will actually do is take an input number (x) and multiply it by itself.
*/

//----- 1. Load header files to include in this function.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <ugens.h>
#include <math.h>

#define DEBUG //debug

//----- 2. Define your function and declare what it will do.
double myfunction (float p[], int n_args, double pp[]){
    double inputvalue;
    inputvalue = (inputvalue * pp[0]);
    return(inputvalue);
}

//----- 3. Create a profile for your function, so you can call on it in your score.
int profile(){
    UG_INTRO("squareme", squareme);
    return 0;
}

//Day 80

```

```

// Writing window functions for RTcmix, using C++. These would be used in maketable("window")

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <ugens.h>

#define DEBUG

// John Gibson's Hanning and Hamming windows from table.cpp in RTcmix bin folder (lines 1384-1427)

static int _window_table(const Arg args[], const int nargs, double *array, const int len){
    int window_type = 0;

    if (len < 2)
        return die("maketable (window)", "Table length must be at least 2.");

    if (nargs != 1)
        return die("maketable (window)", "Missing window type.");

    if (args[0].isType(StringType)) {
        if (args[0] == "hanning")
            window_type = 1;

        else if (args[0] == "hamming")
            window_type = 2;

        else if (args[0] == "rectangular")
            window_type = 3;

        else if (args[0] == "lanczos")
            window_type = 4;

        else
            return die("maketable (window)", "Unsupported window type \"%s\".", (const char *)
                args[0]);
    }

    else if (args[0].isType(DoubleType)) {
        window_type = (int) args[0];
    }

    else
        return die("maketable (window)",
            "Window type must be string or numeric code.");

    switch (window_type) {
    case 1: // hanning window
        for (int i = 0; i < len; i++){
            array[i] = -cos(2.0 * M_PI * (double) i / (double) len) * 0.5 + 0.5;
        }
        break;

    case 2: // hamming window
        for (int i = 0; i < len; i++) {
            double val = cos(2.0 * M_PI * (double) i / (double) len);
            array[i] = 0.54 - 0.46 * val;
        }
        break;

    case 3: // rectangular window
        for (int i = 0; i < len; i++) {
            array[i] = 1;
        }
        break;

    case 4: // lanczos window
        for (int i = 0; i < len; i++){
            array[i] = sinc(2.0 * (double) i / (double) len - 1.0);
        }
        break;

    default:
        return die("maketable (window)", "Unsupported window type (%d).", window_type);
    }

    return 0;
}

```



```

//Day 81
srand()

//three coins, heads = 0 and tails = 1
coin1 = 0
coin2 = 1
coin3 = 0
a = 0
b = 0
c = 0

for(i = 0; i < 3; i += 1){
  print_off()
  coin1 = pickrand(0,1)
  coin2 = pickrand(0,1)
  coin3 = pickrand(0,1)

  if(coin1 == 0){
    a = 2
  }
  else{
    a = 3
  }

  if(coin2 == 0){
    b = 2
  }
  else{
    b = 3
  }

  if(coin3 == 0){
    c = 2
  }
  else{
    c = 3
  }

  sum = a + b + c
  print_on()
  printf("%d", sum);

  if(sum%2 == 0){
    printf("-----");
  }
  else{
    printf("--- ---");
  }
}

//Day 82
// Generating "mixtures" in Stockhausen's Studie II
rtsetparams(44100, 2)
load("WAVETABLE")

frequency = 690 //fundamental frequency in mixture 67, which starts the work
frequency_rounded = frequency

//These were all rounded due to Stockhausen's rounding of frequencies for his frequency tables.
row1_ratio = 1.07
row2_ratio = 1.14
row3_ratio = 1.21
row4_ratio = 1.29
row5_ratio = 1.38

frequency_array = {0, 1, 2, 3, 4}

for(i = 0; i < 5; i += 1){
  print(frequency_rounded)
  frequency *= row4_ratio //this is where mixture 67 lies in tables of frequencies
  frequency_rounded = trunc(frequency + 0.5)
  frequency_array[i] = frequency_rounded
}

//----- WAVETABLE, generating first mixture in the score, with envelope
start = 0
duration = 1.56167 //again, roughly. Stockhausen specifies 1 sec = 76.2 cm of tape
amplitude = ampdB(70) //score is in dB and this gesture starts at -15
envelope = maketable("line", 100, 0,1.0, 1.0,0) //looks to me like the slope is linear
frequency = 1 //only initializing here

```

```

pan = 0.5
waveform = maketable("wave", 1000, "sine")

WAVETABLE(start, duration, amplitude*envelope, frequency_array[0], pan, waveform)
WAVETABLE(start, duration, amplitude*envelope, frequency_array[1], pan, waveform)
WAVETABLE(start, duration, amplitude*envelope, frequency_array[2], pan, waveform)
WAVETABLE(start, duration, amplitude*envelope, frequency_array[3], pan, waveform)
WAVETABLE(start, duration, amplitude*envelope, frequency_array[4], pan, waveform)
print(amplitude)

//Day 83
// Adding reverb to previous score, for authenticity (albeit digital...)
rtsetparams(44100, 2)
load("WAVETABLE")
load("REV")

bus_config("WAVETABLE", "aux 0-1 out")
bus_config("REV", "aux 0-1 in", "out 0-1")
rtoutput("/Users/jerod_s/Desktop/mixture.aiff")
frequency = 690 //fundamental frequency in mixture 67, which starts the work
frequency_rounded = frequency

//These were all rounded due to Stockhausen's rounding of frequencies for his frequency tables.
row1_ratio = 1.07
row2_ratio = 1.14
row3_ratio = 1.21
row4_ratio = 1.29
row5_ratio = 1.38

frequency_array = {0, 1, 2, 3, 4}

for(i = 0; i < 5; i += 1){
    print(frequency_rounded)
    frequency *= row4_ratio //this is where mixture 67 lies in tables of frequencies
    frequency_rounded = trunc(frequency + 0.5)
    frequency_array[i] = frequency_rounded
}

//----- WAVETABLE, generating first mixture in the score, with envelope
start = 0
mixture_duration= 1.56167 //again, roughly. Stockhausen specifies 1 sec = 76.2 cm of tape
amplitude = ampdB(70) //score is in dB and this gesture starts at -15
envelope = maketable("line", 100, 0,0, 0.9,1.0, 1.0,0) //trying it reversed
frequency = 1 //only initializing here
pan = 0.5
waveform = maketable("wave", 1000, "sine")

WAVETABLE(start, mixture_duration, amplitude*envelope, frequency_array[0], pan, waveform)
WAVETABLE(start, mixture_duration, amplitude*envelope, frequency_array[1], pan, waveform)
WAVETABLE(start, mixture_duration, amplitude*envelope, frequency_array[2], pan, waveform)
WAVETABLE(start, mixture_duration, amplitude*envelope, frequency_array[3], pan, waveform)
WAVETABLE(start, mixture_duration, amplitude*envelope, frequency_array[4], pan, waveform)

//----- REVERB, digital, so not super accurate
start = 0
instart = 0
rvbtime = 2.0
duration = mixture_duration * rvbtime
amplitude = 1.0
type = 1 // 1 is Perry Cook's, 2 is John Chowning's, 3 is Michael McNabb's
rvbpct = 0.5
inchan = 0
REV(start, instart, duration, amplitude, type, rvbtime, rvbpct, inchan)

//Day 84
/*
Pauline Oliveros' Sonic Rorschach truncated to nine minutes
*/
rtsetparams(44100, 2)
load("NOISE") //white noise
load("WAVETABLE") //loud, short pulse

//----- NOISE
start = 0
noiseduration = 60 * 9
amplitude = 2000
envelope = maketable("line", 1000, 0,0, 0.01,1.0, 0.9,1.0, 1.0,0)
NOISE(start, noiseduration, amplitude*envelope)

//----- WAVETABLE
start = noiseduration / 2

```

```

duration = 1.25
amplitude = 50000
frequency = 2000
pan = 0.5
waveform = maketable("wave", 1000, "buzz")
WAVETABLE(start, duration, amplitude, frequency, pan, waveform)

//Day 85
rtsetparams(44100, 2)
load("FILTSWEEP")
srand()

rtinput("/path/to/file.aiff")

start = 0
instart = 0
duration = DUR()
amplitude = 0.75
envelope = maketable("random", 10, "high", 0.1,1.0)

ringdur = 0.5
balance = 0
steepness = 1

lowcf = 30
highcf = 4000
narrowbw = -0.5
widebw = -0.0125

inputchannel = 0
pan = makeLFO("sine", 8.75, 0,1)
bypass = 0
cf = maketable("line", "nonorm", 1000, 0,lowcf, 0.5,highcf, 1.0,lowcf)
bw = maketable("random", 10, "even", narrowbw,widebw)
FILTSWEEP(start, instart, duration, amplitude*envelope, ringdur, steepness, balance, inputchannel, pan, bypass,
cf, bw)

//Day 86
rtsetparams(44100, 2)
load("IIR")
load("JFUNCS")

//----- PULSES
start = 0
duration = 0.075
amplitude = 40000
envelope = maketable("window", 1000, "hanning")
firstpitch = cpsmidi(100)
pan = random()

centerfreq = 400
bandwidth = 0.25
filteramp = 10.0

for(start = 0; start < 500; start = start + 0.1) {
    cf = highrand(start, centerfreq)
    bw = highrand(start / 10, bandwidth)
    setup(cf, bw, filteramp)
    pitch = sin(firstpitch)
    PULSE(start, duration, amplitude*envelope, abs(pitch), pan)
}

//Day 87
rtsetparams(44100, 2)
load("JFUNCS")
//more random distributions

srand()

x = random() //between 0 and 1
y = random()

//----- triangle
trianglenumber = ((x + y) / 2)

//----- gaussian
num_elements = 14
halfnum_elements = (num_elements / 2)

```

```

for(i = 0; i < num_elements; i += 1){
    randomnumber = random()
    randomnumber += random()
}

gaussiannumber = abs(((0.166666 * 1) * (randomnumber - halfnum_elements))+ 0.5)

/*
added the following to Joel's JFUNCS library...

double m_tan(float p[], int n_args, double pp[])
{
    double val;
    val = tan(pp[0]);
    return(val);
}
*/

//----- cauchy
pi = 3.1415927
x = random() * pi
randomnumber = (sin(x) / cos(y)) //tangent, or *should* be able to use tan(x)
cauchynumber = ((0.0628338 * randomnumber) + 0.5)
print(randomnumber)
print(trianglennumber, gaussiannumber, cauchynumber)

//Day 88
rtsetparams(44100, 2)
load("MSITAR")

//----- MSITAR
start = 0
duration = 1.25
amplitude = 20000
pluck_amp = 0.5
pan = 0.5
amplitude_envelope = maketable("line", 1000, 0,0, 0.5,1.0, 1.0,0)

//long hand MIDI-to-frequency conversion

midinote = 60
num_halfsteps = 24 //quarter-tones, now that 8ve is divided by 24
A4 = 440 //in Hz
frequency = (pow(2, ((midinote - 69) / num_halfsteps)) * A4)

loop = 0.125
for(iteration = 0; iteration < 50; iteration += 1){

    for(index = 0; index < num_halfsteps; index += 1){
        frequency = (pow(2, ((midinote - 69) / num_halfsteps)) * A4)
        MSITAR(start, duration, amplitude, frequency, pluck_amp, pan, amplitude_envelope)
        midinote += 1
        start += loop //change durations here
    }

    for(index = index - 1; index >= 0; index -= 1){
        frequency = (pow(2, ((midinote - 69) / num_halfsteps)) * A4)
        MSITAR(start, duration, amplitude, frequency, pluck_amp, pan, amplitude_envelope)
        midinote -= 1
        start += loop
    }

}

#Day 89
# One advantage that I'm finding with using Python in conjunction with RTcmix (instead)
# of MINC is the ability to import and use a variety of Python-specific libraries.

from rtcmix import * # Remember that this is needed in all scores that use Python

# Using cauchy distribution with Python's math library

#import libraries
import math
import random

x = random.random()
randomnumber = (x * math.pi)

cauchynumber = ((0.0628338 * math.tan(randomnumber)) + 0.5)
print cauchynumber

```

```

# Or high and low random distributions?

x = random.random()
y = random.random()

if x > y:
    low_rand = y

if x < y:
    low_rand = x

print "number weighted low is %s!" % low_rand

if x < y:
    high_rand = y

if x > y:
    high_rand = x

print "number weighted high is %s!" % high_rand

#Day 90
from rtcmix import *

rtsetparams(44100, 2)
load("WAVETABLE")

import time
import math
import random

divisor = pow(10,8)
x = time.clock()
y = time.time() / divisor

def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r+= step

#----- WAVETABLE
start = 0
duration = 2.5
amplitude = 32768
frequency = 30
pan = 0.5
waveform = maketable("wave", 1000, "sawup")

loop = 0.125
for start in range (0, 50, loop):
    duration = random.random() / 100
    pan = abs(math.sin(start))
    frequency = ((math.sin(y) * x) + y)
    WAVETABLE(start, duration, amplitude, frequency, pan, waveform)
    loop = random.random() * 2

#Day 91
# Here is the origingal 3n + 1 score, now in Python

from rtcmix import *

rtsetparams(44100, 2)
load("WAVETABLE")

import random
random.seed() # seed random numbers, like srand() in MINC

def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r+= step

#----- WAVETABLE stuff, initializing p-fields
start = 0
duration = 1
amplitude = random.randint(5000, 20000)
envelope = maketable("line", 1000, 0,0, 0.1,1, 0.3,0.7, 0.7,0.7, 1.0,0)

```

```

pitch = 60

increment = 0.1515
n = trunc(random.randint(1000, 1000000))

for start in drange(0.0, 100.0, increment):      # Boundaries and step size
    if n%2 == 0:      # if n is even...
        n = trunc(n * 3 + 1)      # do a 3n + 1 operation
        transp = 1.0      # and transpose up and octave
        amplitude = random.randrange(10000, 20000) # louder amplitude values
        pan = 0

    elif n%2 == 1:    # if n is odd
        n = trunc(n / 2)      # divide n by 2
        transp = -1.0      # and transpose down an octave
        amplitude = random.randint(5000, 10000)    # quieter, and integers only
        pan = 1

    if n == 1:      # if n equals 1
        exit()      # stop

    if pitch > 128:
        pitch = pitch - 127

    pitch = cpsmidi(n)      # translate n to pitch in MIDI
    dB = dbamp(amplitude)
    constpowpan = boost(pan) / 2
    WAVETABLE(start, increment * 0.8, amplitude * envelope, pitch, constpowpan)
    print n

#Day 92
from rtcmix import *

rtsetparams(44100, 2)
load("MBANDEDWG")

import random
srand()

#----- MBANDEDWG
start = 0
duration = 0.125
amplitude = 8000
pitch = cpsmidi(70)
strikeposition = 0.3
pluck = 0
maxvelocity = 0.5
instrument = 1
pressure = 0.0
resonance = 0.9
constant = 0.8
pan = makeLFO("sine", 0.2, 0.0, 1.0)

x = random.randint(40, 70)
cutoff = 5

def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r += step

for start in drange(0, 15, 0.125):
    rand_num = random.randint(0, 10)

    if (rand_num < cutoff):
        x += 4 # Go up by major third

    elif (rand_num >= cutoff):
        x -= 3 # Go down by minor third

    pitch = cpsmidi(x)

    if (pitch < 32):
        pitch = pitch + x * 10

    MBANDEDWG(start, duration, amplitude, pitch, strikeposition, pluck, maxvelocity, instrument, pressure,
    resonance, constant, pan)

#Day 93
from rtcmix import *

```

```

rtsetparams(44100, 2)
load("WAVETABLE")

import random
random.seed()

#----- WAVETABLE
start = 0
duration = 5
amplitude = 700
frequency = 200
pan = makeLF0("sine", "nointerp", 0.2, 0,1)
waveform = maketable("wave", "nonorm", "nointerp", 2000, "buzz")

def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r+= step

increment = 1.4

for start in drange(0, 41, increment):
    randomtest = random.randint(0, 100)

    if randomtest < 80:
        duration = 3

    elif randomtest >= 80:
        duration = 10

    WAVETABLE(start, duration, amplitude, frequency, pan, waveform)
    WAVETABLE(start, duration, amplitude, frequency - abs(randomtest/50), pan, waveform)

#Day 94
from rtmix import * # create a list of pseudo-random numbers from a for loop

rtsetparams(44100, 2)
load("WAVETABLE")

import random
srand()

my_list = [] # declare list of elements, say MIDI notes
num_elements = 10
for i in range(num_elements + 1):
    my_list.append(random.randrange(50,101,1))

print my_list

for start in range(num_elements + 1):
    WAVETABLE(start, 1.25, 20000, cpsmidi(my_list[start]), random.random())

#Day 95
from rtmix import *
import random
import math

rtsetparams(44100, 2)
load("WAVETABLE")

notes = []
num_notes = 12

for i in range(num_notes + 1):
    notes.append(random.uniform(60,62)) #use uniform for random floating pt nums

print notes[2]

def pitch_in_Hz(current_note):
    # long-hand MIDI to frequency conversion
    octave = num_notes
    raw_conversion = 440 * (math.pow(2, ((current_note-69) / octave)))
    return round(raw_conversion, 0)

#----- WAVETABLE
start = 0
duration = 0.25
amplitude = 20000
pan = random.random()

```

```

waveform = maketable("random", 18, "cauchy", -1,1)

def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r += step

loop = 1
for start in drange(0,num_notes,loop):
    x = random.random()
    WAVETABLE(start*x, duration, amplitude, pitch_in_Hz(notes[start]), pan, waveform)

#Day 96
from rtmix import *

import math
import random

lowest_note = 45
highest_note = 96

rawnotes = []
num_notes = 4 # generating four mallet chords

for i in range(num_notes):
    rawnotes.append(random.uniform(lowest_note,highest_note))

note_gamut = [round(elem, 0) for elem in rawnotes]
print note_gamut

# now, make them ascending and descending, like SATB-ish
note_gamut_ascending = sorted(note_gamut)
note_gamut_descending = sorted(note_gamut, reverse = True)
print "gamut ascending: %s" % note_gamut_ascending
print "gamut descending: %s" % note_gamut_descending

#Day 97
from rtmix import *

import math
import random
random.seed()
# ----- distributions
# random.uniform(low, high)
# random.triangular(low, high, mode)
# random.gauss(mu, sigma)
# SEVERAL more (lognormvariate, expovariate, betavariate)

lowest_note = 45 # range of instrument, but need to be cognizant of range of each hand
highest_note = 96
mode = highest_note - lowest_note

rawnotes = []
num_notes = 4

for i in range(num_notes):
    rawnotes.append(random.triangular(lowest_note,highest_note, mode)) #focus toward middle

note_gamut = [round(elem, 0) for elem in rawnotes]
print note_gamut

note_gamut_ascending = sorted(note_gamut)
print "gamut ascending: %s" % note_gamut_ascending

# to be safe, range of spread on each hand is one octave or less

s = note_gamut_ascending[3]
a = note_gamut_ascending[2]
t = note_gamut_ascending[1]
b = note_gamut_ascending[0]

if s - a > 12 or t - b > 12: # this is quite slick, just writing "or"
    s = s - 12
    t = t - 12

gamut_in_range = [b, t, a, s]
print "gamut in spread range: %s" % gamut_in_range

#Day 98

```



```

from rtmix import *
import math

trichordone = [1, 3, 7]
trichordtwo = [2, 10, 6]
hexachord = trichordone + trichordtwo

# ascending normal order
hexachord_no = sorted(hexachord)
print hexachord_no

# reversed
def reversed(input_list):
    return input_list[::-1]

print reversed(hexachord_no)

# palindrome
def palindrome(input_list):
    return input_list + input_list[::-1]

print palindrome(hexachord_no)

# palindrome elided
def palindrome_elided(input_list):
    return input_list + reversed(input_list)[1:]

print palindrome_elided(hexachord_no)

# transpose
def transpose(input_list, transposition_value):
    return [mod(element+transposition_value, 12) for element in input_list]

print transpose(hexachord_no, 5)

# rotate
def rotate(input_list, rotation):
    return input_list[rotation:] + input_list[:rotation]

print rotate(trichordone, 2)

#Day 99
# more LISP --> Python
from rtmix import *
import random

# first LISP example, although abs( ) is intrinsic to Python
def absolute_value(x):
    if x >= 0:
        x = x
    elif x <= 0:
        return x * -1

x = -1
print absolute_value(x)

# exploring conditional tests
def conditional_example(y):
    if y > 5:
        return y * 10
    elif y <= 5:
        return y * 5

y = random.uniform(0,10)
print y
print conditional_example(y)

# fibonacci number, when nth number is given (1=0, 2=1, 3=1, 4=2, etc...)
def fibonacci(nth_num):
    if nth_num == 1:
        return 1
    elif nth_num == 0:
        return 0
    else:
        return fibonacci(nth_num - 1) + fibonacci(nth_num - 2)

print fibonacci(7) #seventh number in the fibonacci series
# construct a series, up to a given iteration
def fibonacci_series(iterations):
    a, b = 0, 1

```

```

        for _ in xrange(iterations):
            yield a
            a, b = b, a + b

print list(fibonacci_series(13)) # return thirteen numbers in the fibonacci series

#Day 100
from rtmix import *
from random import randrange

rtsetparams(44100, 2)
load("WAVETABLE")

trichordone = []
trichordtwo = []
num_elements = 3
for i in range(num_elements):
    trichordone.append(randrange(10))
    trichordtwo.append(randrange(10))

    print trichordone
    print trichordtwo
    hexachord = trichordone + trichordtwo

def reversed(input_list):
    return input_list[::-1]

def palindrome_elided(input_list):
    return input_list + reversed(input_list)[1:]

def hexachord_to_midi(pitches, range):
    return [x + range for x in pitches]

pitches = hexachord_to_midi(palindrome_elided(hexachord), 60) # 0 = C, 1 = C#, etc..
print pitches

#----- WAVETABLE
start = 0
duration = 1.5
amplitude = 20000
envelope = maketable("line", 1000, 0,0, 0.3,1.0, 0.5,0.5, 0.75,0.5, 1.0,0)
pitch = cpsmidi(pitches[start])
pan = 0.5

for start in range(0,12):
    pitch = cpsmidi(pitches[start])
    WAVETABLE(start, duration, amplitude*envelope, pitch, pan)

```

## || Afterword ||

I first dreamed up the idea for this book in the spring of 2011 as I was finishing my Doctoral work at CCM. Back then, the only way to interact with RTcmix was through the command line or the [rtcmix~] object for Max 5. I remember very vividly when Brad Garton came to give a guest lecture to our electronic music studio and excitedly announced the release of his standalone application, which we encountered back at the beginning of this book.

Since that time, RTcmix has continued to evolve and change and improve and adapt to our fast-changing world of digital technology. In the roughly four years that I spent researching and writing this book, RTcmix has been adapted for use with Max 6, Max 7, and Pure Data, has been used as the primary audio engine for apps in iOS and Android, seen enhanced Python functionality via Oort, been introduced as CMixRun, moved to GitHub, and is, as of this writing, seeing exciting enhancements added to the MINC parser.

This is all to say that you shouldn't expect to always use or interact with RTcmix the exact same way every time for the rest of time. Though the program will continue to grow and adapt, its core functionality will remain the same. Whether you are designing the next great iOS app for generative music, working on an acousmatic "tape" composition, or writing a work for percussion and interactive electronics, RTcmix will always provide you with an array of powerful instruments to call upon and an amalgam of commands to execute. The sounds hidden within RTcmix are the same classic sounds that helped design some of the most indelible music of the 1990s and 2000s and will continue to enhance the landscape of new music for electronics and computers for years to come.

Those of you who've started to realize some of your own potential in the world of computer programming might want to take your work with RTcmix to the next level and begin designing your own instruments and functions. You'll find example

documentation for just such tasks in `/path/to/RTcmix/docs/sample_code`. While you might not at first grasp all of the C++ code that constitutes these instruments and functions, my guess is that at first glance you won't be completely baffled by their syntax, which speaks to how far we've all come over the course of our work together. Moreover, they are full of helpful comments to guide you in your design.

The RTcmix community is never far away should you have questions regarding your work. This is a close knit group of like-minded composers and programmers who are excited to continue curating RTcmix for use by any and all who are interested in learning more about it. It goes without saying that the book and much of my music wouldn't be possible for their time, care, and invaluable efforts in seeing RTcmix succeed. I am incredibly grateful.

## **|| Useful links in the world of RTcmix ||**

What follows are a list of helpful links that I hope you'll seek out as you continue your work with RTcmix.

### **1. RTcmix website - <http://rtcmix.org>**

If you're like me, you'll probably be referencing this site often. In fact, when I'm working in our main studio at Crane with dual monitors, I'll have the RTcmix website open on one screen with TextWrangler and my Terminal window on another. It's "Reference" section is an invaluable resource for fully understanding the ins and outs of each instrument or command at hand. Moreover, this will point you in the direction of some useful tutorials and links to music, artist pages, and examples from users in the RTcmix community.

### **2. RTcmix GitHub page - <https://github.com/RTcmix/RTcmix>**

The transition to hosting RTcmix on GitHub is a relatively recent occurrence, in that it was completed in the aforementioned four years that went into this text. A huge advantage of GitHub is the ability to test our experimental releases of the program, called "branches."

### **3. RTcmix mailing list - <http://music.columbia.edu/cmc/RTcmix/>**

This is by far the best way to keep up to date on all things RTcmix. The original authors are still highly active and are never more than an email away from answering any questions that you might have, though I of course hope that many of your questions regarding downloading, installing, implementing, etc have been covered in some way. I'm on the mailing list, too, so you'll find me in the conversation as well. There is a separate mailing list for RTcmix developers, which really should be reserved for build questions, architectural inquiries, and potential "bugs" and fixes.

#### 4. Soundcloud page - <https://soundcloud.com/groups/rtcmix>

Your one stop shop for music made with RTcmix! This is a motivated, active group that cares deeply about and is interested in *your* music. Any new works that are created with RTcmix are welcome to be hosted on the group page and we would of course love to see that community grow, so please join us!

A number of works were mentioned throughout this text and I'd like to formally list them here, along with other sources that were used for the research of this book.

Burns, Christopher. "Music 680: Special Topics in Music - Compositional Algorithms (Fall 2007)." <https://pantherfile.uwm.edu/cburns/www/680-fall-2007/>

Dodge, Charles and Thomas Jerse. *Computer Music: Synthesis, Composition, and Performance*, 2nd edition. New York, NY: Schirmer, 1997.

Garton, Brad and Dave Topper. "RTcmix - Using CMIX in Real Time." <http://www.music.columbia.edu/cmix/rtrealtime.html>

Holmes, Thom. *Electronic and Experimental Music: Technology, Music, and Culture*, 4th edition. New York, NY: Routledge, 2012.

Hosken, Dan. *An Introduction to Music Technology*, 2nd edition. New York, NY: Routledge, 2014.

McElhearn, Kirk. *The Max OS X Command Line: Unix Under the Hood*. Alameda, CA: Sybex, 2005.

Pellman, Samuel. *Introduction to the Creation of Electroacoustic Music*. New York, NY: Wadsworth, 1994.

Pope, Steven Travis. "Machine Tongues XV: Three Packages for Software Sound Synthesis." *Computer Music Journal*, Vol. 17, no. 2. Summer, 1993. pg. 23-54.

Puckette, Miller. *The Theory and Technique of Electronic Music*. Hackensack, NJ: World Scientific, 2007.

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

Roads, Curtis and Max Mathews. "Interview with Max Mathews." *Computer Music Journal*, Vol. 4, no. 4. Winter, 1980. pg. 15-22.

Shaw, Zed. *Learn Python the Hard Way*, 3rd edition. Boston, MA: Addison-Wesley Professional, 2013. Online at <http://learnpythonthehardway.org/book/>

Zelle, John. *Python Programming: An Introduction to Computer Science*, 2nd edition. Portland, OR: Franklin, Beedle, and Associates, 2010.

**|| Index of score file instruments and commands ||**