Computer Music

with examples in

SuperCollider 3

David Michael Cottle

09_01_09

Contents

What's New

Index of Examples

1 - Introduction, Music Technology

What's in the text?

This text is a compilation of materials I've used to teach courses in Music Technology, Digital Synthesis, and Computer Assisted Composition at the University of Utah[1]. It blends electro-acoustic composition exercises with SC and OOP topics. The goal is getting actual sounds as quickly as possible and enough SC language to prevent glassy eyes when reading the help files that come with SC. What this text assumes that many tutorials do not is little or no experience with code or synthesis fundamentals. The target readers are sophomore level music composition majors.

Examples of code that you can actually type and execute in the SC program appear in Monaco font (default for SC). There should be a companion folder distributed with this text that contains the lines of code only, extracted to text files. You can use these text files (open them in SC, no reformatting) to run the examples.

The text is divided into three sections. The first section contains lectures for a general music technology course. SC is not even mentioned in these chapters. The second is digital synthesis, which will teach you how to generate interesting sounds in SC. The third will describe methods of structural composition using those sounds or MIDI equipment.

I'm deeply indebted to the sc-users group who are patient with questions from terminally obtuse users. It is a wonderful resource.

For more information on SuperCollider3 connect to http://www.audiosynth.com.

What's New

Each time I teach from a chapter I read through and make small changes in wording. Those changes appear in red type. Those sections that have substantial changes are indexed in a new table of contents, "What's New," listed above.

Terms of Use

This edition of the text may be used by anyone who does not teach or study in the 84602 zip code. Feel free to share it with students or colleagues but each individual should contact me directly at d dot cottle at utah dot edu for a link to the latest edition.

Why SuperCollider 3?

It's inexpensive.

---

[1] If you're not currently a student at U of U, send your exercises to me and I'll grade them. Really.

It has a solid genealogy. It is built on Music N languages and csound, is being developed and used by collective international genius.

It's deep. I've toyed with the draw-a-sound programs. They're fun for about two days. You won't get tired of the sounds or the labyrinth of tools available in SC.

There are easier synthesis packages, but for the amount of power available in SC3 it is relatively easy to use.

Graphic user interface is optional. GUIs confine your creativity. SC, without an interface, forces students to deal with fundamentals. Software and hardware synths with a slick dial/knob/button skin, loaded up with presets may get you to interesting sounds quicker, but in order to build a patch in SC you have to understand the basics of voltage control, scale and offset, attack and decay, and so on.

It can do everything. Classic analog synthesis, AM, FM, PM, physical modeling, subtractive synthesis, additive synthesis, wave table lookup, wave shaping, Markov chains, AI, MIDI, classic concrète, real-time interactive concrète, total control, analysis, mutation, and on and on and on.

It, or something like it, is the future.

Finally, just listen. Ten years ago I launched SC and hit command-r. I was immediately hooked. It allows real time depth and complexity on any laptop computer.

1. Exercises

1.1. Why do *you* use SuperCollider?

1.2. What did *you* do for summer vacation?

Section I: Introductory Topics

2 - Macintosh OS X ("Ten"), Tiger

Why Mac?

The differences between operating systems are not as pronounced as they used to be (even less with the intel chips). There are valid arguments for using Linux, Windows, Unix, or OS X. Our labs are Mac based. Every university lab and professional studio I've worked in has used Macs. That doesn't mean you should own one, but you should know why we use them.

One reason is convention and loyalty. In the early days of personal computers Apple took the lead with software and hardware targeted for the arts and university programs. They continue this tradition.

The next reason is compatibility. Macs tend to support more file formats, including IBM. For me, and from what I've observed with students, it is much easier for me to decipher a student's Windows file than for them to use something from me. You will encounter fewer problems moving back and forth between labs and your personal machine if you use a Mac at home.

Set up is easier. Connecting to a second monitor or video projector, reading USB drives, seeing cameras, managing network connections (you just leave all of them turned on and it finds the strongest), all happen automagically. Time after time I've watched three PC techs scratch their heads over a connection while I can usually plug and play. Instructions for installation are typically 4 pages for IBM, while the Mac section is: 1) Connect the USB cable to your Mac computer. 2) You may now use your USB device.

Another reason is hardware quality and consistency. A Mac is a Mac. This makes it easier to configure and code.

Performance (speed, i/o transfer, latency) is, imho, a wash. I've heard fervent argument from both sides. The actual numbers are close enough to not matter for most of what you do.

I'm reluctant to admit it, but our devotion has an element of erudition and pride. Perhaps unjustified, but musicians are very loyal, even insulted if you so much as imply they use anything else.

0 viruses. Yup, 0 viruses. Beneath OS X is rock solid national defense grade UNIX security.

There are arguments against: they are more expensive, there is less software development, less third party support, but in most labs and studios these issues are irrelevant.

And finally, if you are used to Windows, as many of my students are, why should you learn OS X? Well, why not? This past semester I began the first day of class with a discussion of OS choices and it quickly degenerated into a student led Mac vs. IBM debate. The student

who defended IBM dropped the class a few days later, I assume because he was an "IBM" person and didn't want to be a "Mac" person. This always confuses me. It's not a religion. So let me offer one more very down to earth, rational reason for staying in my class and learning to do things with a Mac. When you sit down for an interview with your hopefully next boss and she says their lab machines are all Mac, or that their studio is Mac based, which of these responses is going to get you the job?

A) "Mac? Are you kidding? That's a dumb operating system."

B) "Thank god. I have no clue about Windows."

C) "I'm fluent with both and rusty on Unix. I can also learn Linux if you'd like. In this profession what you know is not nearly as important as how quickly you can assimilate the NeXT (wink) operating system."

D) "Would you like fries with that?"

D is always the dumb answer and C is always the correct answer. Being able to answer C has landed me several positions. As I mentioned earlier, based on my experience with a dozen or so studios and labs, 95% of music production is Mac based. Even if it were only 30%, what advantage is there to limiting yourself to one or the other?

The Finder

The Finder is an application like MS Word, Internet Explorer, or PhotoShop. It launches automatically when the computer boots and it remains open and running at all times. The Finder allows you to navigate the contents of the computer as well as connect to other storage devices. It also has menus and tools that change superficial preferences such as appearance, keyboard function, screen backgrounds, but also essential preferences such as the input and output device for audio recording.

Navigating through the finder is fairly intuitive using the mouse and menu items. But there are many faster keyboard shortcut combinations using the command key (apple key, also labeled alt on many computers). Here are some that I use often.

In many dialogs and windows you can **type a name** to select an item. Also, **tab or arrows** will scroll through a list of items.

Single click (select the item), **double click** (open the item), **Con-click** (contextual menu).

Com-N (new finder window), **Shift-Com-N** (new folder in a given window) **Shift-Com-A** (go to applications), **Com-K** (connect to remote storage), **Shift-Com-H** (home) **Com-del** (move to trash), **Com-O** (open), **Com-W** (close), **Com-Opt-F** or **Com-F** (search), **Com-space** (spotlight)**, Com-`** (rotate through windows), **Com-1, 2, 3** (change view to icons, list, columns), **Com-Shift-4** (snapshot of screen), **Com-Shift-Con-4** (snapshot to clipboard), **Con-F2** (activate menu), **Con-F3** (activate Dock), **Esc** (cancel out of any dialog)

There are many more. They can be customized, and are easy to learn. They are listed in the menus next to each item. The few seconds it takes to commit a command to memory pays off quickly.

Finder Views

Of the three view options I find icons the least useful. List view shows information such as creation date and size. You can also have them ordered by name, date, size, by clicking on the top of each column. I keep my folders ordered by date because it keeps things I'm working on at the top of the list.

The columns view I typically use for one feature: preview. The preview for documents isn't much help, but audio previews are shown as a QuickTime playback bar. It's a great tool for audio file management.

Selecting an item to open or launch can be done with a single click, or for touch typists, typing the name of the item.

The Dock

The Dock is the row of icons you see at the bottom (or side) of the screen. Each item represents an application, file, folder, or server. Click on these icons only once to launch or open them. Dragging items off or onto the dock removes or adds that item.

Control-clicking on an item in the Dock (or clicking and holding) brings up a menu that allows you to launch, quit, find the application, or switch to any open window in that application.

System Preferences

This is where you customize the computer's look and interface. Most items are self explanatory. The preference you will interact with most often is sound, setting the source for audio input and the playback device. There is no correct device, there are just choices, and you need to choose the item that matches your project.

Here are some other useful and interesting preferences: Set the background screen to change every 30 seconds, either to abstract patterns or your pictures folder. Set keyboard shortcuts for the finder or other applications. Use Con-F2, Con-F3, etc. to activate the menu or dock from the keyboard. Com-Opt-8 to turn on and off, and Com-Opt + or – to zoom the screen. Speech recognition is an amazing but still pretty useless feature (too many errors). The universal access sticky keys is very useful. It allows you to press modifiers in sequence rather than at once. This allows you to use one hand for combinations such as com-o, or shift-com-opt-p. It also shows the modifier key on the screen. Students appreciate this in a lecture. The most important aspect of having this turned on is it won't affect how you normally work. You can type along just as you did with it turned off. (I keep it on all the time.)

Other Applications

An application is what actually does the work on a computer. You give it commands and it performs those instructions. The results are edited images, typed documents, audio files, or an hour or so of diversion. Before using an application it has to be launched, or activated. Several applications can be active or running at one time and you typically switch between them while working. You can tell which program is active by the menus and windows; they will change as you switch to an application, including the Finder.

Each application has a graphic user interface (GUI). These include menu items at the top of the screen and windows that contain information about the document as well as palettes with tools for operating the application. You can tell which application you are operating by the menus and windows; they will change as you switch between applications, including the Finder.

You can launch an application by clicking once on its icon in the Dock, double clicking on its icon (which will be in the applications folder) in the finder, or double clicking on a file that belongs to that application (the Finder first launches the application then opens that file).

Once several applications are running you can switch between them using the Dock (click once), or Com-tab (press repeatedly to scroll through the list), or by clicking once on any window that belongs to that application. Try all three methods and note how the windows and menus change.

Exposé

This is a clever little device that allows you to see every window that is open, every window in an application, or the desktop. There are usually hot corners or command keys (typically f9, f10) for exposé, which can be customized in the system preferences.

Documents

Documents, or files, contain the actual information (audio data, images, text) used by you and the application. In general, the Finder manages which files belong to and are opened by which application. Double clicking on that file should send the Finder off to locate and open the appropriate application. It then opens that document in that application. You can override this automation by dragging an item over an application's icon in the Dock, control-clicking on the item and selecting "open with," or using the Open menu in the application.

Get Info

Click on a document (or application) once and choose Get Info from the File menu (or press com-I). The resulting dialog contains useful information such as file size, creation date, or comments (that you supply). You can change the default application assigned to open the document, create a stationary pad (a template that cannot be overwritten), and lock the

document. One important parameter is ownership and permission[2], which controls who has access to files, folders, and applications. For example, we have often experienced problems with permissions on assignments handed in to a drop box. Occasionally the permissions are set so that the instructor cannot open or modify them. In this case you may have to check that the "others" category is set to "read and write" before handing something in. We also regularly have difficulty with Logic when it tries to save or access a file that has incorrect permissions (usually "read only").

Storage Devices

A storage device is any type of medium that contains data. These include USB jump drives, CDs, Zip disks, internal hard drives, external hard drives, or servers. It is important when working with audio to understand the difference and distinguish between a local, internal hard drive, a removable device, and a remote server.

If you are working on one machine with one hard drive, such as your laptop, these issues are moot. But in a multi-user lab, four out of ten students (uh, that's two fifths I guess) struggle with storage issues all semester. So if you're one of those (you know who you are), read this next section carefully. If confident with storage management, skip to "we strongly encourage you."

Below are examples of several different storage options. Each icon represents a different type of storage device. Those labeled Media, Boot, and Alt are all partitions of a local drive. Their physical location is inside the computer you're using. The Data icon is a USB jump drive. The Media and 10.4.7 7.06 are firewire hard drives. These are removable devices. That is, you can take them home with you. That is the best location for your own work. The DVD (or CD) can work as a "removable" drive if it's re-writable. But even if you have a single write DVD or CD, you can burn multiple "sessions" on a single disk, storing most of your semester's work on a single $0.70 CD.

Not pictured above, because I don't own one yet, if you can believe that, is an iPod. iPods, while used by most people for playback or iTunes folders, are storage drives. You can drag items to an iPod and manage files on it just as easily as a jump drive or firewire drive.

---

[2] Our tech support disagrees, but for me, permissions has been nothing but a major annoyance in our multi-user labs.

I've never found a use for the Network icon. That doesn't mean it's not useful.

Note the icons labeled Fair_Use_Files and Music_Professors. These are servers. They are computers similar to the one you're working on, but in some remote location (maybe even another building). This is a good place to *store* work, because you can access it from any other machine[3]. But it is *not* a good idea to work from that location. What do I mean by "work from." I mean you shouldn't open the file directly from the server, particularly in the realm of audio, because the connection between the two machines is too slow, and you will also get permissions errors. Each time you come into a lab to work on a project you will copy the files over from the server to that local machine (probably the desktop), open it from there, make changes, save, then drag it back to the server or your removable drive for storage. The exception is an external firewire drive; that connection is fast enough for audio, so you can just open the files directly from that drive, edit, and save.

When you're done working on a file you can save it on any of these devices. (The CD and DVD need to be "burned," but the finder does this in a transparent way. It's not much different than copying items to a hard drive.) After it's been saved you can move the file to different locations on a single device, or move it to another (in which case it makes a copy). Saving or copying the same file to several locations (backing up) is a good habit to develop.

Why not have a copy on the server, the local machine, a USB and a CD? That's a good idea; you'll have several backups. However this raises a most recent version issue: Which of the copies has the latest changes? The one on the server, the USB drive, the CD, or the firewire drive? Suppose you open the file on the firewire drive, make a half hour's worth of changes and save it. Then the next day you open an older copy from the server, make another hour's worth of *different* additions and changes. You now have two diverging versions. How do you reconcile the two? Well, you try to avoid that situation in the first place: You can include the date in the title each time you save, or I usually order all my views by the date column so the most recent version is on the top. Ironically, this entire discussion was lost for about six months because I made this very error; working on two different versions, then forgetting about one of them. My latest strategy is to change the rename files with a date, but to also rename the file it is superseding with "old" in the title. That keeps me from opening an outdated version.

The danger in having so many storage options is that you, or the program you're working on, can lose track of files scattered around on different devices. When a class is heavily into a project nearly every day someone asks me where their files are (because they or Logic can't find them). At the risk of sounding like your mom, they are wherever you left them, and I don't know where that is. If you saved them on the local drive of lab station 192 yesterday,

---

[3] And to be thorough, any machine, including the one you're working on, can be connected to as a "server." That is, someone in another location can connect to your machine. In our labs the server is a machine dedicated to storage. No one is sitting at it, reading this text thinking about you reading this text.

but today you are on station 191, they may still be sitting on the drive 192 machine[4]. If you save them on the server and try to work on the files from there, Logic and ProTools will complain when playing back or saving (ProTools won't even open the files). If you saved them on a firewire drive that you didn't bring today, they are on that drive.

Why is this a problem? Because a Logic session (as well as most DAWs and video editors) is not self contained. It's not a book you take off the shelf, then open and read from. It is more like a working space, an empty box, where you can lay out sections from many books scattered around in your library. When you quit the session all of those little scraps go back to the books stored around the room. If you open the workspace in another library, it will not be able to find those clips.

Similarly, the session file that manages the audio recordings is just a shell, a workspace for organizing audio. The actual audio files can be spread out on that machine or even four or five external drives. This can be a valuable feature, distributing the work load over several devices, allowing greater speed and larger sessions. But it can also be your downfall.

Here is a typical error: A classmate shows you a great little sound effect you want to use. It will fit on your USB drive, so you copy it, insert the USB, and drag that file to the session. Unless you have the correct preferences selected, the session doesn't copy that file from the USB drive. It just uses it from where it sits. If you come back the next day, but don't have your USB drive, when you launch the session Logic (or ProTools) will tell you it can't find that file. You will then ask me where it is and I'll say "it's wherever you put it."

Here are some suggestions: Always *work* (and save) on the local drive. The desktop is part of a local drive. But you should never *backup* on the local drive (the desktop). You should save your work as you make changes, but when you are done copy your files to a server or some removable device (like a USB drive or Firewire drive that you own). Repeat after me "work local, backup remote." The second is to gather materials into a single folder before importing them into a session. So in the example above you would insert the USB drive, *copy* the cool sound effect to the folder that has all your materials for this project (or for all your projects), *then* drag it into a session. Finally, Logic has options when creating a new project that help mitigate these issues. Check these boxes: "create project folder," "copy external audio . . .".

Ok, those of you who are comfortable with file management, or are working on a single machine can join us. We strongly encourage you to invest in your own storage and backup media. The items you leave on these machines or even on the server do not really belong to you, but should be thought of as backups. There is no guarantee that an item you store on any devices maintained by the school will be there when you come back. The only copy you own is the one you can take home with you.

Using the Finder you can make a copy of any document on another storage device by dragging that item to the window representing a folder on that device, or by selecting the

---

[4] If you saved them on the desktop, they are gone. Desktop items are deleted when you log off. Yup, you read correctly: GONE! Poof! Erased. Weep at will.

item and use Com-c (copy) then Com-v (paste) to "paste" the item to the new storage area. You can use either of those actions on your entire folder (which would contain all your work). You can also use File/Save As . . . from inside the application you are using to edit the document to save a copy of that item in another location.

If you drag an item to a new location on the same storage device it is not copied, but rather moved to that location. To make a copy of an item on the same storage device you can hold down the option key then click and drag it to any new location.

Server Space

A server is a remote machine where space is reserved for you and that you can connect to from any machine on or off campus. This is handy when moving between machines or between labs. But don't rely on it as your master copy. It will be removed without warning at the end of the semester. The copy you own has to be on your own media.

After applying for space you are given a login name, the first letter of your first name and full last name (mine would be dcottle) and a password which you should change.

Unmounting or Ejecting a Storage Device

Once you are finished with a server, CD, Zip, USB or firewire disc, you can remove it from the desktop by clicking on the eject button or by selecting it and choosing eject, or by dragging it to the trash.

Important: unmount discs before you remove or disconnect them. You may damage the disc and in the case of firewire drives you could damage the internal firewire card.

Mac OS X Survival

The good news is OS X is very stable. For example, if one application crashes the others are usually unaffected. Even if the Finder locks up you can continue to operate other programs. Beneath the interface of icons and menus is the Unix operating system. Unix has been around for 35 years and is used in government and university systems where security and stability matter. (You can even run vintage Unix software on a Mac. You can also operate your computer using Unix commands through Terminal, a program that opens a single window with a command line. Unix is more cryptic, but once mastered, more efficient.)

Even with this stability you will encounter problems. Here is a survival guide.

Save often. Save every 15 seconds. I'm not kidding. When I'm typing I save after every sentence, after every change, even deleting a period. Use Save As… to save versions (with a different name) so that a previous version is preserved. Saving costs you nothing. Not saving could cost you hours of work, maybe hundreds of dollars.

Backup your files in multiple locations, and multiple versions. For example, save on the local hard drive, the server, and your portable device. By multiple versions I mean save the file

under different names, so you can revert back to a previous version: "myfile10_21", "myfile10_24", "myfile11_05."

Learn how to locate and read online help and manuals. There are often useful items under the Help menu of each application as well as the system (Finder). In these labs we also make it a point to provide soft copies of manuals for each application as well as help files specific to solutions for these machines and this lab. When a problem is solved we try to document it in a help file. In recent years I've found online user's groups incredibly useful. I regularly search these: http://elists.resynthesize.com/.

After trying to solve the problem on your own don't hesitate lean over and ask a classmate. I realize this is a little embarrassing. Working in music technology I have had to suppress the fear of exposing myself as a novice almost daily. If you've already tried to solve it, then it's not a dumb question.

If the operating system or an application is acting strangely, you should first try saving everything then re-launching the application or the entire system.

If an application continues flakey behavior after it has been restarted then quit the application and try discarding the preferences. The preferences for each application are in the folder [yourusername]/Library/Preferences, and/or [HardDriveName]/Library/Preferences.

If the system or an application does not respond to any commands ("locks up"), follow these steps:

1) Try typing Com-period. This is a standard key combination for "stop." (It never works on the spinning beach ball of death, but I always try it.)

2) Wish you had saved.

3) Give the computer a time out, take a coffee break, go get lunch (don't laugh, this works about 50% of the time). Maybe the solution is you being patient.

4) Com-Opt-esc (force quit). This brings up a dialog that allows you to force any application to quit. All changes will be lost. Forcing one application to quit, including the finder, should not affect other applications.

5) Wish you had saved[5].

6) Force the computer to restart. This is different for each design and model, but for most machines you can use the Com-Control-Start. Also try holding the start button down for 15 seconds or press the convex ">" button on the front near the start button. If nothing else, unplug it.

---

[5] I think I'm as good as anyone about not getting caught by a crash. Yet while writing this sentence MS Word just evaporated without explanation. One hour of work lost.

2. Exercises

2.1. Open a Finder window. Switch views from icon to list and column.

2.2. Locate and launch the applications TextEdit, Stickies, and Safari, using a different method for each.

2.3. Try four different methods for switching between TextEdit, Stickies, and the Finder.

2.4. Force quit Stickies.

2.5. Remove any item from the Dock, place any item in the Dock.

2.6. Open system preferences and change the desktop image, the alert sound effect, time until the system sleeps, and input device.

2.7. Create a new finder window, create a new folder and rename it, then delete it.

2.8. Switch to (or launch) TextEdit. In the new document enter your name, email, (optional) phone, and student id. Save the file in the new folder you created using our class naming convention (e.g. dcottle_01_hello). Make copies of this file in two other places. Hand it in by dragging it to the instructor's drop box.

2.9. In text edit, activate the menus using Ctl-F2 and navigate to Edit/Speech/Start Speaking.

2.10. Open the preferences for "Hello World" (the file, not the application). Change the permissions to include read and write for others.

2.11. Find two sources of help for TextEdit, discard the preferences for TextEdit.

2.12. Force quit TextEdit and the Finder.

2.13. Launch Adaptec Toast. Select Data, New Disc, name it back up. Drag your assignment to add it to the CD, burn it to a disc.

2.14. Open Stickies and TextEdit. Rotate through the windows of Stickies and one at a time copy the text from each window, pasting it into TextEdit. Engage the menus in text edit and navigate to the save menu item. Save the TextEdit file to the default folder (whatever is offered in the dialog), but name it something unique. Switch to the finder. Search for the file using the quick search field (Com-Opt-F). Close that window. Now locate it in the finder (this will depend on where you saved it), copy it, and open another storage drive (e.g. your folder on the server). Create a new folder, name it, open it. Paste the text file to that folder. Easy? Now do it without the mouse… in under three minutes. Go!

3 - Digital Music

Language

I'm thinking of a melody right now. What options do I have for communicating the melody? If you were sitting next to me I could sing it. Not only would you recognize the pitches, but you might also immediately remember the characters from the 1960 sitcom. Since you're not here I will have to choose some other language. Here it is:

00111100, 00111110, 00111111, 00111100 01000010

Didn't get it? How about this:

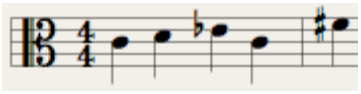
0000003C, 0000003E, 0000003F, 0000003C, 00000042

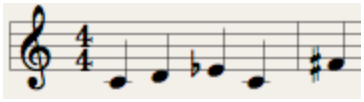
or

60, 62, 63, 60, 66

Or this: 261.62, 293.66, 311.13, 261.62, 370; or C4, D4, Eb4, C4, F#4

Still not sure? How about these:





Next to no one would recognize the first (binary). Only a handful of programmers with musical training would understand the next two (hexadecimal, and MIDI), a piano tuner would know 261.6, most educated musicians the pitch class method (C4) or alto clef, most casual musicians the treble clef, and everyone, perhaps even another species wandering through the room, would understand the sung pitch. (They might not make the association with Dragnet, but they would get that you're singing and even join in.)

A high level language is more universal, low level is less universal. In computerese low level means a language the computer understands, but is difficult for humans. That is the case with binary numbers. Higher level languages are more accessible to humans. That is the case with recent programming languages as well as decimal numbers. Music technology relies on languages. Your work will go better if you understand more low level languages. While it's true you may never have to communicate music in binary, or convert back and forth[6], you will occasionally see hex, often use MIDI and frequency, so you should understand those.

[6] See the pitch, frequency, midi, hex, binary conversion GUI in the appendix.

Hexadecimal, MIDI Numbers

In base 10 we use 10 symbols to indicate quantities; 0 through 9. After using up all of those symbols we have this clever system of starting a new column and recycling the first column. This is well illustrated on an odometer with wheels rolling around. When the right most wheel goes all the way to 9 then to 0 again, the wheel to its left moves one tick, from 0 to 1, indicating that the first wheel has made a complete revolution one time. Each column represents the number of times the wheel to its right has gone all the way around, or the number of totals. In a decimal system each wheel has ten symbols, so a 5 on the second wheel means the first wheel has moved around 5 times, so 5 sets of ten. The number 562 means five of the 100s, six of the 10s, and two of the 1s. (We all get this intuitively, but reiterating this method in detail will lead us hexadecimal numbers.)

Binary numbers use the same system, but with only two symbols: 0 and 1. A 1 in the third column from the right means the first column and second from the right have gone around once. Since we are only using two symbols, the first column represents 1s, the second 2s, the third 8s, etc. So the number 1101 would mean there are one of the 8s, one of the 4s, zero 2s and one 1 (in decimal: 13).

Hexadecimal is base 16, so 16 symbols are used. Visualize a similar odometer but instead of just 0 through 9 on each wheel it has 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The process and motion are the same. So the odometer would rotate each wheel but it would continue past 9, up to F, then 0 before the second wheel to the right clicks over to 1. In this case if the second wheel from the right says 5 it means the first wheel has gone all the way around 5 times. But the first wheel doesn't contain 10 items, rather 16, so the 5 means 5 sets of 16. The third column represents groups of 256, the fourth, 4096, and so on. Why does hex exist? You can represent higher numbers with fewer symbols in hex. The decimal equivalent for the hex number E63F1 is 943089. The other reason is that computers essentially have eight fingers on each hand, so they learned to count to 16 rather than 10 (no kidding). You will probably never have to convert hex to decimal, but you should know how to count. The number that would follow 8A6EF would be 8A6F0, then 8A6F1, 8A6F2, etc., up to 8A6F9, 8A6FA, 8A6FB, 8A6FC, 8A6FD, 8A6FE, 8A6FF, and 8A700.

Midi is the next system I used in the example above. You will encounter MIDI numbers quite often, but they are easier to understand: Each half step is a number, so each octave has 12 numbers. The lowest note musicians use is C-1, which has the MIDI number 0. The number 12 = C0, 24 = C1, 36 = C2, 48 = C3, 60 = C4 (which is middle C[7] on the piano, remember this one), and so on. A 440 is above middle C so it is 69.

Intervals are the same; one half step for each number. 0 is a unison, 2 a whole step, 4 a third, 7 a fifth, etc. If you have no musical training this system is actually easier than the major, minor, augmented, diminished interval system.

---

[7] Music theorists use C4 for middle C, though I've seen aberrations: some MIDI equipment uses C5 for MIDI pitch 60, which makes MIDI 0 a C0, and logic uses C3 for middle C.

We will cover frequency later.

File Formats

Being able to communicate in the digital realm also requires that you understand musical file formats, which are a type of language, also low and high level. Each is useful in different situations. These are the formats that I use on a regular basis: MIDI, NIFF, XML, JPG, PDF, AIFF, WAV, SDII, and MP3.

MIDI: a popular 80s standard

The first is MIDI, which stands for Musical Instrument Digital Interface. This is a standard agreed upon by synthesizer manufacturers in the 80s. It allows two musical components or applications to communicate with one another.

There are four important disadvantages to MIDI: 1) MIDI is not sound. It is only instructions for playing sounds; start, stop times, volume, instrument and channel, etc. The actual sound is generated by the synthesizer receiving the commands. MIDI is analogous to a player piano roll; it is useless without the player piano. 2) MIDI files contain no information about notation. There are no key signatures, time signatures, measure divisions, or staves. 3) MIDI files are written in a very low level, cryptic language. You can't just open a file in any text program and make sense of the data. Writing and reading MIDI files is a task for experienced programmers only. 4) It will soon be replaced.

There are also a number of advantages to MIDI. 1) It is easily edited. Because it is not actual sound, but instructions for creating sound, a single note's length, attack time, release, or pitch can be corrected. 2) It also allows for flexible orchestration. A single MIDI track can be played back as a tuba on one recording pass, or a string bass the next. 3) It is universally accepted. 4) File size. Because nearly any post-90 electronic device will understand MIDI and the actual sound is generated on the local machine (the one in the room where a real person is listening) the network does not have to transfer actual audio, and so it is a good choice for web sites, email attachments, and games.

NIFF

NIFF stands for Notation Interchange File Format which was developed as a standard interchange for notation programs. It hasn’t seen much support because of a reluctance to give up market share by developers of notation software. There are a few notable exceptions, including Lime Music Notation and Sharp Eye optical character recognition.

As with MIDI, NIFF contains no sound. It is only useful for notation. It can, however, be imported, edited, and often converted into MIDI.

XML

XML is a Music Extensible Markup Language, similar to HTML. It is the second attempt, after NIFF, to standardize music notation and has received wider support. It has all the

disadvantages and advantages as NIFF; it contains no sound, it contains no playback instructions, but it does contain information about key, clef, time signature, measures, etc. The one advantage over NIFF is that it is text based. This means that you can open an XML file using any text editor and with some practice make sense of, edit, or even write XML code. It is currently a more common standard than NIFF.

Text Converters

There are a handful of text to NIFF, text to MIDI, and text to XML converters. These converters will translate a text file filled with higher level notation information into one of the above formats.

JPG and PDF

JPG and PDF are reliable, universally accepted standards for images and portable printed documents. They contain no sound, no information about playback, and cannot be edited (in terms of music). But they are a good choice for communicating with other musicians. Attaching a PDF or JPG to email allows the receiver to see exactly what your musical ideas are, provided they read music.

The greatest advantage to PDF is that it is a standard component of OS X. They are cheap and easy to create using the print dialog.

Screen Shots

The quickest, easiest method for generating an image of anything, including music, that you see on a computer screen is to take a screen shot. Shift-Com-3 takes a shot of the entire screen. Shift-Com-4 allows you to define a rectangle using a cross hair. Both of those in combination with the control key will copy the image to the scrapbook, which can then be pasted into another document. Hitting the space bar after engaging any of these will capture an entire window.

AIFF, SDII, WAV

These three formats are digital recordings of actual audio. They contain real sounds, but that's all they contain. They have no information about note values, durations, chords, theory, notation, or instrument assignment.

Digital audio is pretty simple for applications to code and decode. It's just a stream of numbers. There is very little detail in format (compared to a text or notation document). A wave file (.wav) is an IBM standard. Most Macintosh programs can handle wave files. SDII stands for Sound Designer II, which is the pioneer digital audio editor developed by DigiDesign, the authors of ProTools products. The most reliable and interchangeable standard for digital audio is AIFF, or Audio Interchange File Format.

The disadvantages to these formats are you can't change the orchestration, you can't edit single notes from a chord, you can't extract a line or change the key (features common to

MIDI and notation programs). They are however the standard for CDs. Any of these formats can be burned to a CD (as an audio CD) and played on any CD player.

Other compression formats may soon replace CDs, but the CD will probably survive because it is based on our ability to hear. While MP3 is good, all recording studios still create masters in AIFF.

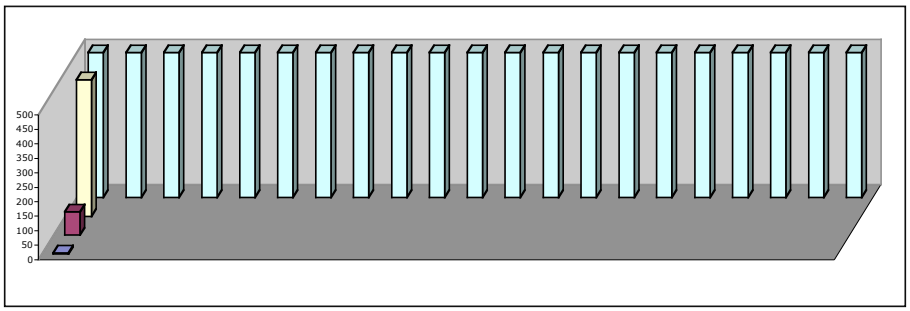
MP3 and Other Compression Formats

MP3 files are the same as AIFF, WAV, or SDII (only sound, can't change orchestration, no notation information) with one critical difference; they are NOT the standard for CDs. An MP3 file, burned to CD, will not play back on a standard CD player. They have to be played on an MP3 player.

File Size

If a colleague needs to hear a 10 minute fugue, but has a slow internet connection, then AIFF would be a poor choice. A MIDI file will communicate just as well but more efficiently. Choosing which format to use depends a great deal on file size. Here are the relative file sizes for one minute of music with estimated transfer rate using a dialup connection, followed by a chart illustrating each. The tiny dark blue bar is MIDI or XML, the purple is PDF, yellow Mp3, and all of the light blue bars combined represent a single AIFF file. I had to split them up, otherwise the first two file sizes would be invisible.

Format	File Size	Transfer Time	Ratio
MIDI, XML, NIFF	4k	.03"	1x
PDF	80k	.2"	20x
MP3	500k	26"	125x
AIFF	10,000k	3'50"	2500x



Software for MIDI, Music Notation, Optical Character Recognition

MIDI editors are common, and have been integrated into overall packages such as ProTools, Digital Performer, Cubase, etc. These are discussed later.

Just a note before I offer opinions on notation programs; they are all complex and difficult to learn because music notation is complex. A good word processor is complicated yet it only needs to represent linear components. A good notation program has several dimensions; vertical parts, horizontal notation, not to mention items that appear once in a score but in

every part, extraction of single lines or combinations of lines. On top of all the complexities of scoring we expect a notation program to play the music for us (this is as or more difficult than having a text editor read the sentences back). For these reasons notation and OCR programmers face challenges beyond word processing or purely graphic editors, and the technology is about 5 years behind both of those in terms of standardization and user interface.

I'm often asked if there is a program that will notate music correctly as you play at the piano. So far there is not. One reason is that there are so many variables. If, for example, you played a series of quarter notes around middle C, about one per second, how would the program know if it should be treble, bass, alto, tenor, movable G or F (they do exist), or percussion clef? What key is it in? which time signature? mixed meters or simple divisions? Are you playing quarter notes, halves, triplets? So no, there is no magic bullet. And frankly I'm not sure the musician who would write music by just playing at the piano would have much of interest to say, but that's just me. You need to bring more than that to the table.

Finale has long been the standard for most universities. Users often seem devoted (possibly because they have invested so much time learning) but they just as often despise its dense and complex interface. My impression is that it was designed by engineers, not musicians. When first researching the available packages I quickly passed on Finale because I found better alternatives. Other deterrents are price, poor user support, proprietary file formats (and a reluctance to support NIFF or XML).

Finale produces a free, scaled down version of its software called NotePad. This would be a good choice for assignments, text examples, and quick notes.

Sebelius is easier to use but still has a learning curve, is also expensive, proprietary (but possibly supports XML through third party sources). It would be my second choice. Our labs have a dedicated (nearly rabid) group of users.

Logic includes a notation component with it's sequencer and digital audio workstation. While I admire their attempt at a "complete" music package, as is often the case, they have spread themselves a bit too thin. Each passing month we are more impressed by it's depth and, well, "logic," but the notation functions (I've done about four projects) are flakey, inconsistent, and unpredictable. That said, it's a pretty darn good effort.

My first choice is Lime; a program that was developed before personal computers or MIDI existed. It is inexpensive, has a long trial period, easier to learn and use (designed by musicians), has unusual features such as microtone support and piano tablature, and supports NIFF and XML. I have used it for extremely complex works.

The other reason I mention Lime is that it belongs to a growing category of hobby software; packages developed by a few individuals in their spare time, free of corporate oversight. They may lack a degree of professional polish, but they make up for it in user support, price, features, and real people at the other end of an email. Other such notables: Amadeus ii, Audio Companion, LameBrain, Transcribe!, Amazing Slow Downer, etc.

Optical Character Recognition

OCR is a process where a scanned image of printed music is translated into NIFF, XML, or a proprietary format (in the case of Finale). That file can then be edited as if it had been entered as data by hand. Text OCR has reached a usable standard, but my experience with music OCR has been mixed. Complex music, especially manuscript with multiple voices and parts, is very difficult to get right. There are usually enough errors that you spend as much time correcting as it would take to enter the music by hand. But I remain hopeful.

That said, the most accurate, usable package I've worked with is developed by SharpEye.

3. Exercises

3.1. Read the getting started tutorials for Finale Notepad, Sibelius, Logic, or Lime and create a lead sheet including lyrics, title, chord symbols, and composer. It may be serious (i.e. an actual work like *Somewhere Over the Rainbow*) or whimsical (i.e. see how many notes you can put in each chord). Hand in a single folder containing the original file as well as versions in these formats: a screen snapshot, pdf, and MIDI.

3.2. Read chapter 1 of the Logic text.

3.3. Create a Logic session with three instrument tracks each routed to a different playback instrument, and record different music (perhaps a round) on each track using regular record, metronome on and off, pause record, loop record, and auto drop (punch in). It can be serious or whimsical. Import a video, add a voice over track using take folders. Add sound fx or background music using Apple Loops. Hand in the session file, a bounced audio file (aiff, 16 bit, interleaved), an exported movie, a midi file, and a pdf score.

3.4. Import a Bach fugue into a Logic MIDI (not instrument) session. Do some creative editing (enough that I can tell it's no longer the original). Copy the midi regions to several Logic tracks (three for each MIDI voice) and assign a different MIDI instrument to each. Use the Hyperdraw window in the Matrix editor to blend, mute, or switch between instruments (pro only), to create an interesting orchestration. Export it to a MIDI file and try opening it in a browser (to prove compatibility). Hand the session in and attach a copy of the MIDI file to an email to the instructor.

3.5. Do a MIDI google search for a difficult to find jazz standard (e.g. *Waltz for Debbie*) and import it into Logic. First import it into a MIDI session connected to an internal GM orchestra. Once you've heard each of the voices, reassign those tracks to internal Audio Instruments and orchestrate it using EXSP24 (or any other internal instrument). Make adjustments to the time and key signature in the notation window. Hand in a pdf of the music, a MIDI version of your orchestration, an aiff and mp3 version.

3.6. Create another Logic session with four to six Instrument tracks. Assign some interesting instruments, including a percussion collection. Compose a short work (abstract or tonal) that includes at least two loops, and at least one loop with a complex ratio, and/or a phase shifted ratio, using the transform menu.

4 - Sound

Peaks and Valleys of Density

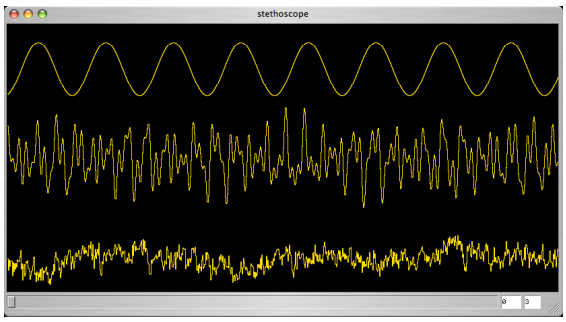
Peaks and valleys of density through a medium; this is the nature of sound (also a great title for a novel). Sound or vibration can occur in any material. Vibration is the alternating compression and decompression of the material; like ripples on a pond[8]. In digital audio editors this motion is represented as a graph with peaks and valleys. The peaks representing greater density, the valleys lesser density. When the sound is generated (for example by playing through a speaker) the forward motion of the speaker, creates the peaks and the backward motion creates valleys. The speaker's forward motion compresses the air, backward decompresses it.

Patterns, Noise, Periods

We are surrounded by patterns in our natural environment. As a matter of fact, I can't think of any natural process (even those that do not produce sound) that does not have some periodic element; birds, ocean waves, thunder, phases of the moon, and so on. Sound waves with a repeated pattern are said to be periodic. Each repetition is a period or a cycle. A wave a pattern too complex for us to recognize is aperiodic. The range of sounds from perfectly periodic to extremely aperiodic is continuous, and sounds fall anywhere along that continuum.

We can also see patterns with our eyes and interpret them as faces, trees, grass, and so on. We decipher sounds in much the same way. We look for pattern or structure and match that pattern with some real phenomenon such as a violin, airplane, rushing water, etc. Below are diagrams of three waves with different degrees of periodicity. The first has a very clear pattern, the second is complex, but still reveals patterns, and the third doesn't seem to have any pattern at all.

4.1. Waves: Clear Pattern (Periodic), Complex Pattern, No Pattern (Aperiodic)



[8] Try this experiment: fill a glass goblet with water and “play” it by rubbing your finger around the rim. Hint: clean your finger with soap or alcohol. The sound vibrations that travel through the air are also traveling through the surface of the water in the glass.

Sounds that have a periodic wave pattern are heard more or less (to the same degree as the clarity of the pattern) as pitch. The clearer the pattern, the clearer the character of the pitch. The absence of pattern, to the same degree, approaches noise. I say approach because pure noise is a theoretical abstract, like infinity. More on that later. In electronic composition and synthesis terms, noise is a wave with no apparent pattern. (To a recording engineer, noise is any unwanted sound, patterned or not.)

The number of times the wave moves through its period in one second is the frequency of the wave, expressed in hertz, or cycles.

Frequency

Frequency is the first dimension of sound. It is the number of cycles (per second) of a periodic wave. The musical equivalent to frequency is pitch. A physical example of frequency is a vibrating string or instrument body. Middle C on the piano has a frequency of about 261 cycles per second. The lowest key on the piano is about 50 Hz (Hertz = cycles per second) and the highest is about 4000 Hz. Humans can hear frequencies as high as 20 kHz (20,000 Hz). The low range of hearing capacity requires a qualification. We can *hear* sharp edged waves at frequencies as low as one pulse every minute, even one every hour or one a year. We can also feel smoother waves at frequencies as low as 1 per minute. But the point where the pulses begin to blend into a single homogenous pitch is what most texts consider the low range of hearing. So while there is theoretically no lower limit to audible frequency, pitch begins at about 20 Hz. We change frequency by shortening a string, increasing the energy of a vibrating body, or tightening our vocal cords.

Melodies are made up of consecutive frequencies. Harmonies are frequencies that sound the same time. We hear frequencies as music if they are related mathematically. We describe these relationships as intervals, using terms such as an octave, fifth, minor third, and so on. Intervals are not absolute, but relative. (Examples of absolute measurements are one foot, one liter, one ounce. Relative measures are twice as far, half as much, three fifths.) An interval, therefore, is not a specific difference in frequency, but rather calculated in relation to a given frequency; 2x higher, 1.5x higher. The actually distance in Hz changes with the starting point. An octave (2:1 ratio) above 440 is 880 (2 x 440); a difference of 440 Hz. But an octave above 1200 is 2400, a difference of 1200. Ratios are covered in depth later.

Phase

A periodic wave can also be measured in degrees of phase. Phase describes how far into the cycle the wave has progressed; like the phases of the moon. A wave begins at 0 degrees. 180 degrees is half way through the cycle, 360 degrees is one complete cycle. Changes in phase of a single wave will usually not affect how we perceive the sound. Phase does however affect how two waves interact.

Amplitude

The second dimension of sound is amplitude, which is the height of the peaks and depth of the valleys. The musical equivalent to amplitude is volume. A physical example of amplitude is the distance a speaker cone travels forward and back. The measurement of amplitude is expressed in db (decibels). We increase amplitude by driving the vibrating body harder with more energy.

We measure sound in db, or decibels. Like intervals, decibels measure relative amplitude, not absolute; nothing is just 10 db, but it can be 10 db louder than what it was before, or you can decrease amplitude by 10 db. It would be inaccurate to say that a jet engine is 120 db, but you could say it is 120 db louder than rustling leaves.

In general terms, the difference between the quietest sound and the loudest sound you would want to hear is about 120 db. One level of dynamic (e.g. mezzo forte to forte) is about 7 db.

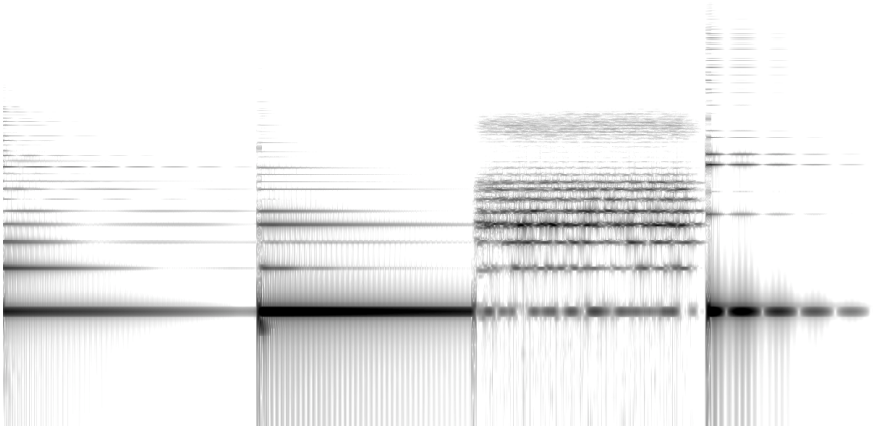
Harmonic Structure

Harmonic structure, the third dimension of sound, is the presence and strength of upper harmonics. The musical equivalent to harmonic structure is tone or timbre. The graphic representation of timbre is wave shape. In general a bright sound will have sharp edges to the peaks and valleys. A dull sound will have a smoother shape. We will cover harmonic structure as it relates to synthesis in a later chapter. For now, understand that a single sound that we hear as a certain frequency is usually a blend of that pitch and its multiples; a range of frequencies that could span 2000 Hz. A brighter sound will have more of these higher frequencies (a wider range) and a duller sound will have fewer.

There is no standard measure[9] for timbre. We change timbre with embouchure, the angle of a bow, the shape of our mouth, by changing to a different instrument, a mute, etc.

The graph below (the vertical lines are frequencies) shows an acoustic piano, nylon string guitar, choir (digital) and a bell all playing the same pitch, at the same amplitude. They only differ in timbre, which is evident in the upper harmonic structure.

[9] Each upper harmonic has a frequency and amplitude, which can be measured, but there is no method for measuring and quantifying the overall timbre as such.



Properties of sound; speed and wave length

The speed of sound depends on temperature, altitude, and air density, but for this class you can use 1000 feet per second, 100 feet in .1 seconds, 10 feet in .01 seconds. The length of a wave is calculated by dividing velocity or speed by frequency, so it also depends on weather conditions. But if we use 1000 feet per second, a wave with a frequency of 100 Hz (about a G3, where a male voice comfortably sings) is about 10 feet (1000/100). A 1000 Hz wave (about a C6, the upper range of a soprano, and many instruments) is about 1 foot (1000/1000). The pitch A 440 is about 2.2 feet (1000/440). Wave length and speed are important for understanding echo and reverb, phase cancellation, as well as proper microphone and speaker placement.

The Expressive Nature of Musical Instruments

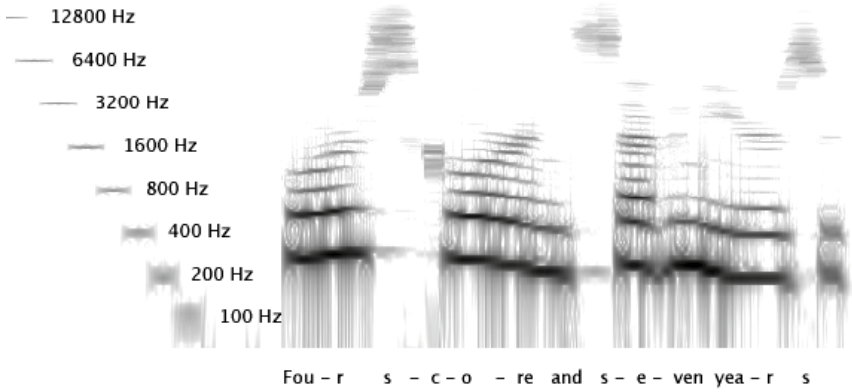
Most instruments have control over these three dimensions. Changing them is what makes an instrument expressive. The more control and potential change, the greater the expressive quality of the instrument. For example, the piano was considered a major advance over keyboard instruments of the period because it allowed the performer to control and change amplitude, and therefore increase its expressive quality. Keyboard synthesizers made a similar leap in musicality when they included weighted and velocity sensitive keys.

The human voice has precise control over amplitude and pitch. But it has more control than any instrument over timbre. It is so precise and varied that it can communicate not only musical ideas (notes, chords, melodies), but complex and precise philosophical, religious, emotional, and patriotic ideas: changing timbre is what constitutes speech.

Language is an excellent example of timbre modification. All speech comes from shaping upper harmonics with our mouth, tongue, and nasal passages. In addition to the actual words that make up speech, the character of a persons voice (e.g. your mom vs. Mr. T[10]) is defined by the presence and strength of upper harmonics.

[10] Who I'm sure would win.

4.2. Frequency Spectrum of Speech

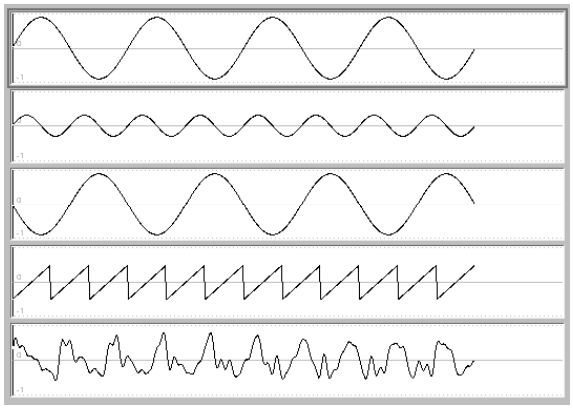


Above is a frequency spectrum graph of a woman saying "four score and seven years." The x axis is time, the y axis is frequency. The beginning of the graph has blips at 12800 Hz, 6400, 3200, 1600, 800, 400, 200, and 100, followed by the spoken words. The "s" sounds are in the 6400 to 12800 range. "T" sounds are in the 1600 to 3200. C's and K's are 800 to 1600. The vowel sounds are the fat ribs between 200 to 1600. Note that a vowel is actually made up of a band of related frequencies. The “e” in seven and years have higher bands. Note the changes in upper harmonics when the speaker closes her mouth for the “r” sound.

Adding, removing, or changing the amplitude of upper harmonics (filtering) will change the shape of the wave and therefore the timbre. More harmonics will make a wave sharper, and will result in a brighter sound.

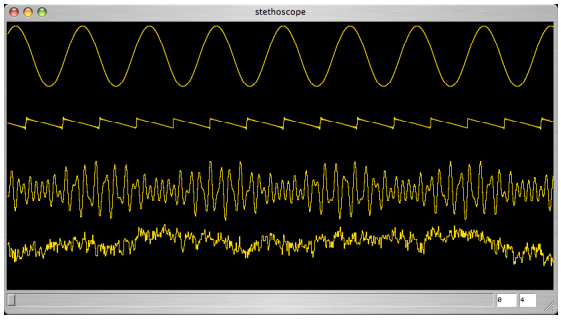
The illustration below shows graphic representations of the three different wave properties discussed above. The first is louder than the second. The second has a higher pitch than the first or third. The third is 180 degrees into its phase. The fourth will be brighter than any of the others. It has a higher pitch than the first three. The last is an example of a periodic, but complex wave with a frequency higher and a tone brighter than one, two, and three, but lower and duller than four.

4.3. Graphic Representations of Amplitude, Frequency, Timbre, Phase



4. Exercises

4.1. Of the wave examples below (each is exactly 1/100th of a second), which is the loudest? softest? highest pitch? brightest sound? is aperiodic? is periodic?



4.2. What is the length of each of these waves: 10 Hz, 250 Hz, 440 Hz, 1000 Hz?

4.3. What is the lowest pitch humans can hear? What is the highest? What is the lowest frequency humans can hear?

4.4. Given a performance hall that is 200 feet deep, if you are at one end of the hall, how long would it take sound to get to the far wall and back?

4.5. Using an oscilloscope and your own voice, demonstrate change in amplitude but not wave shape or frequency, then change in frequency but not the shape or amplitude, then shape but not frequency or amplitude.

4.6. Next record yourself making a noise sound. Copy a section of about 1 second and do a multiple paste about four times. Can you hear a pattern?

4.7. Next copy a smaller section; 1/100th of a second. Paste it several times, listen after each paste. At what point do you hear pitch? What is the pitch? Continue pasting about six times so the frequency is clear.

4.8. Increase the amplitude of the noise to be as loud as you can stand to listen (using effects/amplify). Copy and paste that section and reduce the amplitude by 10 db. Repeat this process, reducing by 10 db each time. How many db until you can barely hear it?

4.9. Use Amadeus to record your own voice changing pitch, then amplitude, then timbre. Create and hand in the sonogram.

4.10. In a Logic session, create a long drum section using loops. Add four or five filtering plug-ins. Use Automation to switch them on and off and modify their settings.

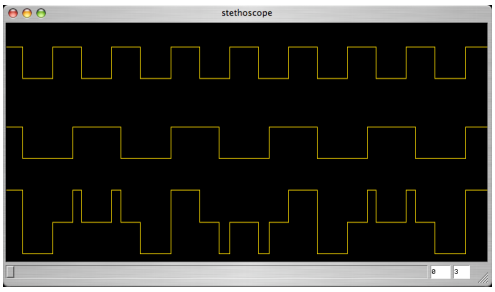
5 - The Party Conversation Chapter

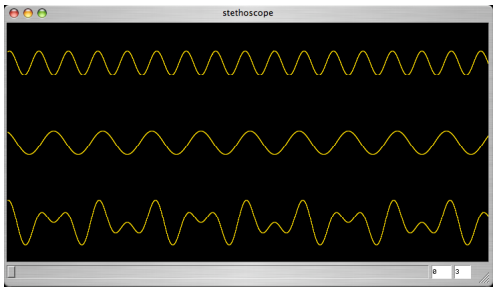
Constructive and Destructive Interference

When two sound waves interact the peaks and valleys combine to create both constructive and destructive interference. Constructive interference is when the motion and direction of two waves are the same; either both increasing or both decreasing. Destructive interference is when one is increasing and the other decreasing, so one is pushing while the other is pulling and they interfere with each other.

Example 5.1 below uses both sine and square wave forms to illustrate this phenomena. The destructive and constructive patterns are easier to see in the square wave example. There are three signals on each graph; the top is a frequency at 500 Hz, the second at 300 Hz, the third shows the results when they are combined. At the beginning of the graph both waves are in the positive part of their cycle and the two waves are added together. When the second wave moves to the negative part of its cycle the two waves (one positive, the other negative) cancel out each other to some degree. When they are both negative, the negative values are added together. The second example shows two similar sine waves, with the resulting pattern in the third channel.

```
5.1.  Constructive and Destructive Interference
```





Our music experience is made up of the patterns that result from the interference of waves. We recognize them as intervals, chords and melodies, use them to tune instruments and as clues for space and environment (reverb and echo). If they are the same frequency, but a different shape, they combine so we hear them as a single wave, since they have a common periodic cycle.

Tuning an Instrument using "Beats"

If one wave has a slightly different frequency from the other, say 440 and 444, then the two waves slowly move in and out of phase, creating full constructive interference, then full destructive interference. This phase shift is similar to the effect of two turn signals that have nearly the same blinking rate. They start out flashing together, then slowly move out of phase so that one is flashing while the other is off, then back in phase again.

When frequencies are this close the speed of the motion in and out of phase is equal to the difference of the frequencies (440 and 444 will move in and out of phase 4 times per second). The "beats" that result from this interaction are used to tune instruments. Fewer beats; more in tune. In class we have done listening tests using frequencies with varying degrees of beats. Both musicians and non-musicians concluded that a difference of 1%, or a ratio of 1:1.01 was “out of tune” and 2% was “really out of tune.” A difference greater than 7% was another pitch (but still out of tune).

Tuning aficionados use cents to describe the distance between intervals and different variations in tuning. Each half step is 100 cents. 1200 cents to the octave, and the same for every octave. 1% is about 17 cents, 2% is about 34 cents. So two pitches within 10 cents of each other could be thought of as in tune.

Phase Cancellation

There are circumstances that result in complete deconstructive interference. If two sine waves are the same frequency and amplitude, but one is exactly 180˚ out of phase, their energy cycles are exact mirrors and the two waves cancel each other out. Incorrect microphone and speaker placement can cause phase cancellation.

Here is an example: A 100 Hz sine wave has a length of 10'. If you recorded such a signal using two microphones, one placed 20' feet from the source, the other at 25' (5' farther, which is one half the length of a 100 Hz wave), then when the wave reaches the second mic it will be 180˚ into its phase. At that point it will be beginning the downward part of its cycle. When the two signals are mixed down and synchronized–one at the beginning of its upward cycle, the other at the beginning of its downward cycle–their energies are exact inversions. One will be pushing (at 0˚ phase), the other pulling (at 180˚). They will therefore cancel each other out in complete destructive interference.

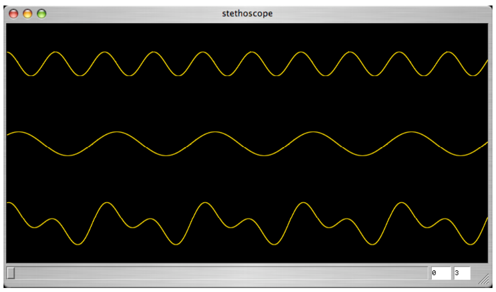
Improper speaker placement (or wiring) can also cause phase cancellation. Try this experiment: play a pure sine wave at about 100 Hz out of two speakers. Plug one ear, point the other one at the speakers, then walk around the room. Since the frequencies bounce off the walls it is difficult to achieve pure phase cancellation, but you ou will hear hollow spots, where the tone is quieter. In some cases it may disappear entirely. The effect comes about because the distance you are from one speaker is different from the other. Different by ½ the length of the wave (or a multiple thereof). You are standing in a spot where one speaker's wave is at its peak, the other at a valley.

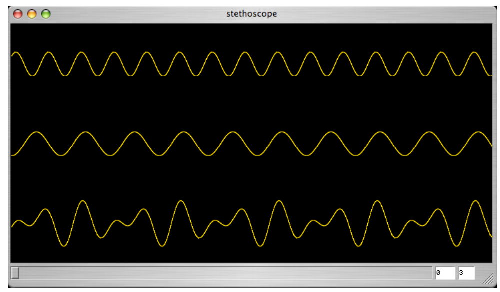
Phase cancellation can not always be avoided. As a matter of fact, it is essential for spatial clues in stereo microphone placement (read below). It is also what is responsible for the nearly unbelievable effects of sound cancellation headphones. But since it can change the character of a sound by removing frequencies you normally would want to record, we try not to invite it when placing microphones.

Musical Intervals

When two waves are related by some ratio (2:1, 3:2, 8:5), we hear the aggregate interference patterns as a musical interval. The graphs below show two sine waves with a 2:1 ratio, then a 3:2 ratio. Note the patterns as they are combined in the third signal. The frequency of the combined "cycle" of the wave is the common denominator of the two. That is when they are both in synch and beginning the next cycle. In example 5.2 it is every cycle for the top wave, but every other cycle for the bottom wave. For the second example it is every other time for the top wave, and every third time for the bottom wave.

5.2. Interference in Proportional Frequencies: 2:1 and 3:2





Another way to visualize the combined patterns of constructive interference is to mark the peaks of each wave with a small tick. If we did this with the 2:1 ratio we would see every other tick (wave peak) line up. With 3:2, every third and second tick (peak) would line up. The chart below illustrates this of constructively coincident waves using ticks.

```
2:1

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Or moving the two closer together:

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |



3:2

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

In casual conversation I'm often asked about the nature of music; why do we use the major and minor scales, why are there 12 notes in a scale, how do we account for the black and white key arrangement? People are curious about why music sounds the way it does; one style somber, the other bright, another chaotic. Do we accept the system used in Western

music because that's what we're used to (nurture), or is there a fundamental principle at work (nature)? What is the universal field theory of music?

Here it is: Music is math.

The pitches used for scales in Western diatonic (white keys) music are multiples of the base pitch. An octave is two times, a fifth three times, a third is five times and a second is nine times. When these harmonics are transposed down by octaves to one single octave above the fundamental the results would be a diatonic scale. Since notes of a scale come from harmonics, and harmonics are ratios, then the diatonic scale we use in Western styles (the white keys) can be expressed in ratios.

The third harmonic (an octave and a fifth) is a 3:1 ratio. To move it down one octave so that it is just a fifth above the fundamental, you would divide it by one half, which is a 1:2 ratio. Those two ratios combined are 3:2. A fifth then is a 3:2 ratio. The fifth harmonic, two octaves and a third, can be transposed down two octaves (1:4) for a simple third. The combined ratios of 5:1 and 1:4 are 5:4.

All of the upper harmonics can be transposed down by octaves in a similar fashion to simple intervals. The results are the intervals in our scale from unison to octave (a step is two piano keys, ½ step is one piano key):

P1	unison	C to C	1:1
M2	1 step	C to D	9:8
M3	2 steps	C to E	5:4
P4	2 ½ steps	C to F	4:3
P5	3 ½ steps	C to G	3:2
M6	4 ½ steps	C to A	5:3
M7	5 ½ steps	C to B	15:8
P8	6 steps	C to C	2:1

The black keys can be calculated using harmonics also. Or they can be derived using combinations of the white keys, for example a major third and a major second make up an augmented fourth, or C to F#. Notice these intervals have higher rations.

m2	½ step	C to Db	16:15
m3	1 ½ steps	C to Eb	6:5
+4	3 steps	C to F#	45:32
m6	4 steps	C to Ab	8:5
m7	5 steps	C to Bb	9:5

And finally, an easy way to remember 8, 5, 4, M3, m3 is that the numbers are super particular (first number is one higher than second): 2:1, 3:2, 4:3, 5:4, 6:5. Just remember the jump in the series for a 9:8 major second (what happened to 7:6, and 8:7?). Then you can easily calculate the other intervals. For example, a m6 is an inverted M3 moved up one octave (4:5 * 2:1 = 8:5), A tritone is a M3 + M2.

It is more useful, and more logical to order them from lowest ratios to highest ratios:

P1	unison	C to C	1:1
P8	6 steps	C to C	2:1
P5	3 ½ steps	C to G	3:2
P4	2 ½ steps	C to F	4:3
M6	4 ½ steps	C to A	5:3
M3	2 steps	C to E	5:4
m3	1 ½ steps	C to Eb	6:5
m6	4 steps	C to Ab	8:5
M2	1 step	C to D	9:8
m7	5 steps	C to Bb	9:5
M7	5 ½ steps	C to B	15:8
m2	½ step	C to Db	16:15
+4	3 steps	C to F#	45:32

Why more logical? Because this ranks them in order of consonant to dissonant.

Consonance and Dissonance

So music is math, but more specifically, it is the math that describes the patterns of constructive and destructive interference resulting from the ratios of the waves. Intervals, chords, melodies, and even large scale formal structures, are mathematically related. The musical experience boils down to our search for related patterns in sound, in time. We compare and match the patterns we hear with those in our memory; from the micro level of 1000^{th} of a second ago to the macro level of days and years. When we hear two or three frequencies in a row we not only recognize each cycle of a wave as a periodic pattern, and together their relationship with each other as a pattern (a set of ratios that become a melody), but we match that sequence of patterns with memories of sets of frequencies (melodies) from as far back as our childhood.

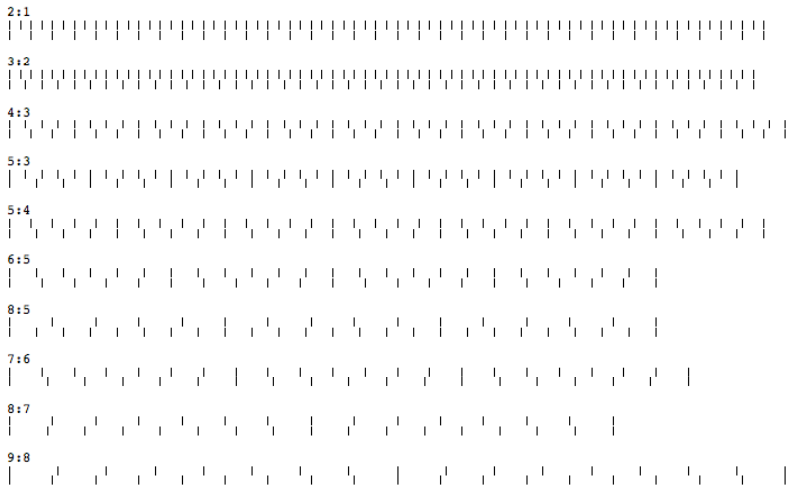
So why do different styles of music evoke such a wide range of emotional responses?

Music students spend four full semesters learning to understand and carefully control the balance of consonance and dissonance in musical styles. At the end of the first year a final exam can drop an entire grade if they don't approach and resolve an $+4^{th}$ correctly. Many books define consonance and dissonance as pitches that sound "good" together (consonance) and pitches that don't sound good together. This is wrong for two reasons. First, it implies intervals must fall into one or the other category. They don't. Secondly, it implies that you should avoid dissonance. You shouldn't and we don't.

Rather, all intervals fall somewhere along a continuing scale of relative consonance and dissonance, and musicians use intervals all along this scale to achieve a particular mood. A more appropriate definition is suggested by the Latin root *consonare*: "sounding together." Intervals with low ratios have more constructive interference, and the waves coincide more often; they "sound together." The waves of dissonant intervals coincide less often, and thus "sound apart."

Complex or remote relationships that involve difficult math make us uneasy, sound dark, even scary. Simple relationships are predictable, relaxing, maybe boring. But if the collection of ratios strike a balance that keeps our interest or matches our mood at the time (which changes depending on the context; church, movie theatre, concert hall), the music is effective. If a composer's goal is to sooth he will choose simple relationships. If she wants to unnerve us, then she may try more complex relationships. Even if a composition is not what we expect, it can still be effective, if it achieves the composer's goals through the use of relative dissonance and consonance.

Using the system of ticks described earlier to mark wave peaks, you can see the patterns for each of these ratios. Note that higher numbered ratios have a longer combined cycle, therefore more time between coincident beats, which is harder to see on the page, and more difficult to hear. Higher ratios are more dissonant and lower ratios are more consonant. Dissonant music (difficult math) makes us uneasy while consonant music (easy math) calms us down.



The pitches that are derived from these multiples (overtones, harmonics) make up musical scales. When a group of such notes are played together they seek a tonal center of gravity. The center of gravity or home base is the lowest common denominator, or lowest frequency of the multiples. When they are played sequentially or consecutively our ears accept the lower pitches on the harmonic series as stronger or more stable, the fundamental being the strongest. Western tonal music is built using motion away from stable, or consonant structures, to dissonant structures and back to consonant again. That motion, and the relative consonance/dissonance in infinite variation, is what we find satisfying in the works of Bach, Wagner, Schoenberg, Chick Corea, and Alison Kraus. All styles can be boiled down to motion to and from low ratios and high ratios, consonance and dissonance, structure and prolongation, stability and instability, low on the harmonic scale to high on the harmonic scale, tonic (1:1) to dominant (3:2) to tonic (1:1).

Tuning, Temperament, and the Pythagorean Comma

It is reassuring to know that music is founded on such rational scientific principles. So I'll follow up with this unsettling fact: Pianos (and fretted instruments) are not in tune. By their nature, they *cannot* be tuned correctly, that is to the ratios we just described. There are not enough keys or frets[11] to represent all required pitches. In addition, while non-fretted or keyed instruments such as the voice or violin *can* tune to pure ratios they often don't, because they need to match a keyed or fretted instrument in the ensemble. In the case of choir with piano accompaniment, the singers tune their notes to the piano, which is incorrect.

I can see you shaking your heads, so let's do the math. We'll start with the frequencies for the black key used for both an A-flat or G-sharp using the lowest possible ratios (most consonant or "in tune"). Starting from C4 (about 261.6) we would go up a minor sixth (8:5) to A-flat. The results (261.6 * 8 / 5) is 418.56. A G-sharp on the other hand is two major thirds above C (261.6 * 5/4 * 5/4) which comes out to 408.75; not even close to the A-flat. As a matter of fact, these two "enharmonic" pitches, A-flat and G-sharp, are nearly a quarter step (41 cents) from each other! Students are always a little suspicious of these results (or they suggest some kind of rounding error), so I'll wait while you dig through the kitchen drawer for a calculator.

Are you're eyes glazing over? We're not done. We get all kinds of results depending on the approach: 413.4 if we go two whole steps down from the octave, 418.56 using a minor third, then a fourth, 419.03 using whole tones up, 438.43 using half steps up, and 404.19 using half steps down from the octave.

Your stunned look is justifiable. But the explanation is simple; it is because while A-flat and G-sharp are represented by the same key on the piano, they are different pitches. A cellist or violinist will play them differently. But despite experimental pianos with typewriter style keyboards to accommodate each pitch, our modern piano only has one key for both. What do you do? Tune to one or the other? If you tune to the A-flat, then the key of C minor, which uses A-flat, will be in tune. But the key of E major, which uses a G-sharp, will be out of tune. Fretted instruments are similar. The precise position of each note changes depending on context. Early guitars had movable frets to accommodate such adjustments, but were difficult to manage. Fixed frets began to be used, which raises a similar dilemma. Which note do you set the fret or tune the guitar to? If you tune it to one, the other may sound off. This accounts for why you can tune your guitar to sound great on a G chord, but an E sounds wrong.

Three centuries ago the music community picked one. The practical result was you could only play in certain keys; those that included the pitch you tuned to (e.g. Ab, not G#). In this system (named "just" tuning) the farther you stray from those easy keys the more out of tune it sounded. Many musicians, notably Bach, weren't satisfied with such a restriction. The compromise they proposed was equal tempered tuning, where the strings are adjusted (the fifths are flattened by 2 cents) to be out of tune enough to represent all enharmonic pitches on the keyboard, but in tune enough to sound ok. It took nearly 100 years to be accepted, and the

[11] And the frets are fixed.

debate continues, but in rarified circles. Bach's master work "The Well-Tempered Clavier" includes stunning fugues in all 12 major and minor keys to prove the value and to encourage the migration to equal temperament.

The technique used to tune such precise frequencies, before stroboscopes or electronic tuners, was to count the number of beats per second. We saw earlier that precise tunings, such as 444 and 440, could be calculated by listening to the number of beats produced as the two moved in and out of phase (4 times per second). The "beats" that we use to tune those two frequencies would have a tempo; 240 bpm. These pre-electronic tuners would set a metronome to the tempo they calculated for the correct degree of detuning for each fifth on the piano, and tune to the beats of metronome. Sound insane? Well get this: they even adjust the equal tempered scale a little, lengthening or stretching the higher octaves.

The equal tempered pitch for the A-flat/G-sharp key is 415.3, nearly .7% or 14 cents off from the pure or “just” A-flat (418.56), and 1.3% or 27 cents off from the G-sharp (409.8); out of tune by our definition in a previous section. So the next time you are at a friends house, play a few keys on their grand piano and say, off hand, that it’s out of tune. When they counter that they just paid someone to tune it you can respond: “I know, and it’s out of tune. The G-sharp is way off.”

The acceptance of equal tempered tuning made chromatic harmonies and enharmonic interpretations more appealing. This allowed Beethoven a greater freedom in harmonic choices, which was expanded by Chopin, taken to tonal extremes by Wagner and Brahms, laying ground for the inevitable leap of logic by the second Viennese school to non-tonal systems. One could argue, therefore, that Bach, the pinnacle of tonal thought, helped plant the seeds of atonality. But I digress. (And while I'm digressing: The accordion was largely responsible for world-wide acceptance of equal temperament because it was cheap and couldn’t be re-tuned.)

Should our compositions reflect a distinction between Ab and G#? Do they already? Is the next step in synthesis a piano that can play both? Are they already out there?

Related to equal tempered tuning, and even more unsettling (even to professional musicians) is the Pythagorean Comma; a curious, almost unbelievable warp in music[12] where tuning to logically equivalent, but mathematically different intervals will result in different frequencies. That is to say, you arrive at a different place depending on the road you choose. If you start with a C4, tune to pure intervals all the way around the circle of fifths using a 3:2 ratio for calculation, then make the same journey using octaves, you arrive at a different frequency. By octaves you would go this route (using A to keep the math simple) A1, A2, A3, A4, A5, A6, A7, A8, which is 55, 110, 220, 440, 880, 1760, 3520. By fifths this would be A1, E2, B2, F#3, C#4, G#4 (Ab), Eb5, Bb5, F6, C7, G7, D8, A8, but in this case you will arrive at 3171.58. Likewise, don’t take my word for it. Get out a calculator. You can also

[12] That's a poor characterization. Music isn't warped. Our brains are not warped enough to wrap around this idea.

illustrate this using 6 major second intervals (9:8) to span an octave, then make the same calculation using a single octave (2:1).

5. Exercises

5.1. Draw a graph that illustrates constructive/destructive waves similar to the first example in this chapter. Draw square waves with 4 cycles in the first, 6 cycles in the second. In the third, draw the combined pattern of constructive and destructive interference. (Hint: use graph paper. Alternatively, you can generate the frequencies in Amadeus and hand in a pdf of that graph.)

5.2. Starting with C (261.6 Hz), and using the ratios described above, calculate the frequencies for the following pitches, using the ratios for the number of steps. Remember to invert the ratio to go down:
C4 to F#4 (3 steps)
C4 *down* to G3 (3 ½ steps down)
C4 to B4 (5 ½ steps)
C4 *down* to A3 (1 ½ steps down)
C4 to E4 (2 steps)

5.3. Prove the Pythagorean comma (show your work).

5.4. Story problem! Two trains leave the station, . . . No, wait, two microphones are set up to record a piano. One is 10' away, the other is 11 feet away (a difference of one foot). What frequency will be cancelled out?

5.5. In a stereo file use the tone generator functions of Amadeus to generate a 440 Hz tone in both channels of a stereo file. Zoom in so that you can see each wave. Carefully select ½ of a wave in the right channel only, and delete, so that the two waves are out of phase 180°. Use the characteristics menu to make it a mono file.

5.6. Using the same bass frequency (e.g. 440), calculate the frequencies for a 6:5 minor third, then a 7:6, a 8:7 and 9:8 major second, a just major third (5:4) and an equal tempered major third (multiply by 1.26). Use the tone generator feature in Amadeus to generate the bass frequency (e.g. 440) as a drone in one channel and the frequencies resulting from each of the above ratios in the other channel.

6 - Editing Techniques, Sound Quality, Recording, DSP

The current standard for CD quality audio is stereo, 44.1k sampling rate, 16 bit depth. This standard is not arbitrary, but rather based on our hearing capacity as it relates to pitch, amplitude, and timbre. We have two ears, we can hear up to 22k, and we can distinguish amplitudes ranging 90db. Stereo tracks accommodate our two ears, 44.1k can capture a 22k wave, and 16 bit produces a 144 db dynamic range.

Two Track Editors

Two track editors have similar features to a cassette deck. Sound Designer II was the first digital editor. It was followed by a handful of professional programs such as Peak and CoolEdit. So many quality editors have since appeared that they are hardly worth listing. Most of them offer a variety of sampling rates, bit rates, and compression formats. They also have basic editing tools such as zoom in and out, copy, cut, paste, markers, regions, multiple windows, and keyboard shortcuts. Digital processors such as noise reduction, reverb and echo, synthesis and modulation, and spectral analysis are also more common.

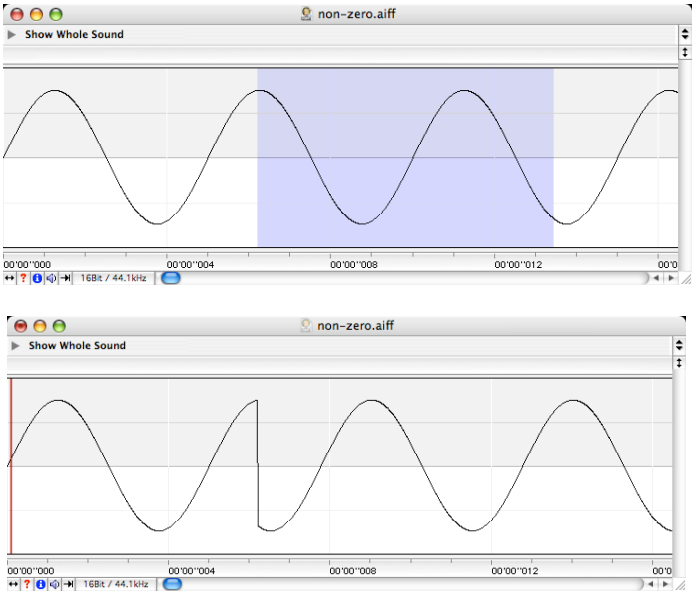
Multi-track programs are designed to handle more than two tracks and are for engineering complex audio layers that will be mixed down to a single two track CD. Multi-track editors are not a good choice for simple stereo concert recordings, recitals, or mastering (optimization, cleaning up and fading beginning and end, noise reduction, digitizing albums, etc). A good studio should have both, and the engineer should be comfortable with both.

Clean Edits

The first step to quality audio is clean edits. Years ago we edited reel to reel magnetic tape with razor blades, splicing tape, a splicing block and lots of time and practice. The next generation of recording media (after tape) were DATs, and CDs, which could not be edited. Digital audio on hard disk is revolutionary in its flexibility, ease, and accessible editing tools. It is easy to piece together complete works with sections of several takes, or even complex programs such as full orchestra. My first such project involved replacing a single note in the recapitulation of a cello sonata with one taken from the exposition, 20 minutes earlier. The graphic display makes it easy to see where to do the edits and can show waves with a resolution down to the sample level. It is easy to be precise and accurate, so it is unforgivable to make this one error: non-zero crossing edits.

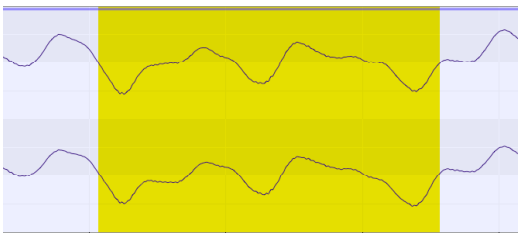
A non-zero edit is when you select, copy, or paste at a point where the wave is not at 0, or the center of the graph. Such cuts result in a sharp change from positive to negative energy in only one sample. Below is an example (in mono) of a non-zero cut. (This example is very short. The edits you actually do may be several minutes, but the principle is the same.) The first image shows the selection from an audio file. Note that the beginning and end of the selection are at positive and negative points of the wave. Both are non-zero, and therefore incorrect. In the next image it has been deleted. Note the sharp edges as the wave moves positive to negative in a single sample.

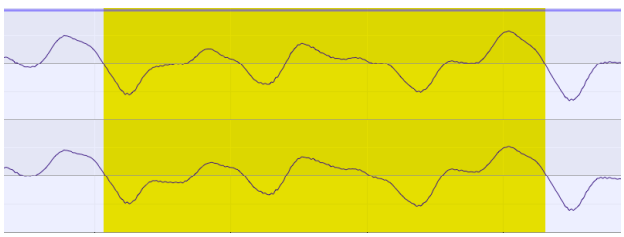
6.1. Non-Zero Crossing Edit



These incorrect edits result in clicks or pops. The error is so common that most editors provide some simple solution such as a cross-fade, interpolation of the wave, “snap to zero” menu item, or even an automatic "repair clicks." But a skilled engineer should avoid them in the first place. This is accomplished by either carefully choosing edit points at silence (and planning sessions to accommodate such edits) or by zooming in to each selection before copying or pasting and choosing non-zero edit points. This goal is often complicated by a stereo file where the left channel may be zero, but the right is in a positive position. In this case cross fades, or rapid fading in and/or out (e.g. a tenth of a second), or interpolating the wave can be used to correct the edit point.

Zero-crossing edits are considered good form, but a real pro goes one step further and chooses zero-crossing descending edit points. A descending edit point is where the wave is moving from positive energy to negative energy. If you choose the beginning of an edit at a zero crossing, descending, but at the end select a zero crossing *ascending*, then the edit will indeed be at a zero point, but will have a quick change of direction. The first example below shows a pair of non-zero edits, but one is descending and the other ascending (bad) followed by the more correct pair where both are descending, non-zero edits.





Digital audio work stations have strategies for making these transitions and edit points even easier using cross fades: millisecond ramping between two audio files, or two sections of audio files. But since most of the work I do is with two track editors, I still need to choose my edit points carefully.

Stereo

We use our two ears to decipher spatial orientation. If something is louder in one ear we conclude it is on that side, if we hear it equally in both we assume it is in front of us. (Read below for a more detailed discussion of spatial orientation.) The two sides of a headset, or two speakers in a room reproduce this phenomenon. Stereo programs are also more efficient to produce and perceive. The speakers work more efficiently if they are producing different signals and I believe (with no scientific backing) that our brains can process sound more efficiently when different instruments dominate different ears (and sides of our brains).

This duality extends throughout the entire recording and playback process. We often use two microphones to record live music in a stereo pattern, two channels on a mixer, record to two tracks on the tape deck or digital recorder, play out of two line outputs, to two amplifiers (usually housed in one box), two speakers[13], which finally reaches our ears.

Mono

Mono programs lack depth and image, but they are more economical and easier to produce. Mono signals are used in most PA systems, AM radio, and TVs.

In a multi-track session, which will eventually be mixed to stereo, most of the tracks are recorded in mono. On the final mix-down the position between the two stereo speakers is set through panning. Some tracks, such as pianos, drum sets, and backup vocals, where the stereo image needs to be preserved, are recorded to the session in stereo.

Sample Rate

There are three issues that affect choice of sampling rate: Frequency range, aliasing, and file size. The sampling rate for CDs is based on the Nyquest theorem which states that the number of samples must be twice the highest frequency being sampled. A 22k wave has a peak and a valley in each cycle, so to reproduce it you need at least two numbers; one for the peak, one for the valley, hence 44k for a 22k wave. Most digital converters now offer rates up to 64 and 96k, and down to 22, 11, or even 5k. Recording at lower sample rates will reduce the upper range of frequencies being recorded, essentially filtering higher pitches and harmonics.

---

[13] In some cases, often with PA systems, there are two speakers but not stereo. Rather, they are a duplication of mono and are more for distribution. PAs rarely mix stereo programs.

Dog whistles produce frequencies beyond our hearing range, and so do most instruments. So do we need to worry about frequencies we can't hear? Yes, for two reasons.

The first reason is to avoid aliasing. Frequencies that are normally beyond our hearing range can interact with the sampling rate to produce unwanted frequencies that are in our hearing range. Aliasing is analogous to the "wagon wheel" affect you see in movies. Movie frames are like samples, taking snap shots of real phenomena. In a film the frame rate and wheel speed can come into synch in such a way as to create the affect of a slower moving wheel, or a backward moving wheel. The same thing can happen with waves. The sample is a snapshot of a wave in a given position. The rate of the snapshots and rate of the wave can combine to produce a slower "ghost" wave that doesn't actually exist, but is simply a byproduct of the sampling rate and frequency being sampled.

The unwanted frequency that will appear is the sum or difference of the sampled frequency and the sampling rate. For example, if you sampled a frequency of 44101 Hz with a 44100, you would actually get a wave of 1 Hz. If you sample a frequency of 44500 Hz at 44.1 k you will end up with 400 Hz.

As an illustration, take a look at the second hand[14] of your watch, opening and closing your eyes once every second (i.e. "sampling" the image). The second hand sweeps over the face at a rate of 1 time per minute. If you open and close your eyes once every second, you would get a pretty good idea of the second hand's motion and frequency around the face. That's because your sample rate (60 times a minute) is much higher than the second hand's cycle (once per minute). Now open your eyes at a rate that matches the second hand's rate; only once every minute, exactly. What do you see? The second hand will appear to stand still even though it's moving at a rate of once per minute. Now open your eyes at a rate slightly slower than the second hand; every 59 seconds. The first "sample" would show the hand at 59, the next at 58, the next at 57. Based on your samples the clock is going backwards! at a rate of once every hour (it will take 60 samples, one about every minute, to get back to 0). If your sample rate were 61 seconds, it would still appear to move forward, but at 1 time every hour, because each time you open your eyes though it looks like it has moved only one second, it has moved all the way around and 1 second. You've just made a frequency of 1 time per minute look like 1 time per hour. That is aliasing.

Recording at a higher sampling rate, much higher than 30k, reduces the risk of aliasing.

The second argument for a higher sampling rate is the theory that frequencies beyond human perception interact to produce important artifacts in lower frequencies, the range that we do hear. (In other words, natural aliasing.) In a concert hall those frequencies have a chance to interact and combine. But they cannot in a session where the instruments are isolated. In this case the instruments can only blend digitally inside the computer during mix down. For that reason, the argument goes, the masters should be recorded at 96k to include frequencies in

[14] A time keeping device popular back in the 20th century.

the 30k and above range, which can then blend during the mix. I've seen no solid proof of this theory.

Finally, while you can down sample, you can never up sample. Something that was originally recorded in 96 can be reduced to 11 to save space. But if you record in 11k you can never retrieve the information that would have been recorded at 96k.

The criteria for choosing sampling rates are files size vs. the importance of upper frequency. If you are recording a spoken lecture then a frequency range up to 5k may be plenty for intelligible speech. In that case an 11k sampling rate could be used.

44.1, 48, 88.2, and 96 K

I'll dodge the complicated history of sampling rates and simply point out that in the video world we work in 48k and 96k. In the audio world we use 44.1 and 88.2. When converting from one sampling rate to another you should always double or half the values. 88.2 to 44.1 is ok, 96 to 48 is fine. But 96 to 44.1 or 48 to 44.1 will introduce noise. If you have no choice, dithering can compensate for this noise, but it is best to do the original files based on the destination of your final project. If you are working in video, use 48 or 96. If the final product is a CD, use 44.1 or 88.2.

Noise

What happened to bit depth? After seven or so editions of this text I decided to move it down below the discussion of noise because I always think of it as a noise source. So be patient, we'll get to bit depth.

The next step in quality recording is to limit noise. Noise is any unwanted sound. It can be background noise such as air conditioning fans, computer fans, buzzing lights, or system noise from mixers, fx units, or amplifiers, or DA conversion noise such as low sample rate or bit depth. You will never eliminate noise, so the next best thing is to increase the divide between signal (what you want to hear) and noise (what you don't want to hear). The term for this division is signal to noise ratio. Signal to noise ratio is expressed in db. An electronic component such as an amplifier may list their signal to noise as –80db. That means if you had been listening to a normal signal like a CD, then stopped the CD and increased the gain by 80db the background noise produced by the amp itself will be as loud as the signal you were just listening to.

One simple strategy for reducing noise is to use quality equipment. For example a Firewire mixer, an Mbox or digi002 (though I increasing hear criticism of their converters) interface will have less noise than using the 1/8 inch jack of your computer[15].

---

[15] But I have to say the difference is very small and often related to external hard drives or video connections which can be turned off or disconnected. So depending on the machine, you can often get very good quality using the mini jack of a computer. I record using condensers, a Mackie mixer, and the 1/8" jack all the time with excellent results.

Another is to reduce or eliminate the noise source (in the case of background noise). Move to a quieter location, turn refrigerators, lights, fans, and phones off. While neither of these first two techniques require much technical knowledge they are often overlooked.

The next method does require some engineering skills: setting optimum levels. The lower the recording level, the closer it is to the bed of noise that you want to be above. Before recording a source you should set the input gain as high as possible without causing distortion (see below).

Microphone proximity is one method for achieving optimum levels. A close mic position will increase the amount of signal in relation to the background noise. That is not to say you should always mic close to the source. When recording in a hall with good acoustics you may move the mic back to blend the source and the hall reverb. In this case the hall sound is not noise, but signal that you do want to include on the recording.

The last resort, for me at least, is digital noise reduction, and it should be used last (if ever). When removing unwanted sound digitally you will always compromise the quality of what you want to keep. Signal processors make up for poor engineering and poor quality equipment. It is sometimes difficult to resist fiddling with all the knobs, flashing lights and technical graphs and readouts, but a skilled engineer will never reach that point. Processing *always* compromises the signal.

Distortion

Distortion is any misrepresentation of a wave. The jagged edges created by low sample or bit rate are an example of distortion. Distortion is often used creatively, to transform the source into something new. But if your goal is accurate representation (which should be sought for masters–you can apply creative distortion later) then it should be avoided.

The most common source of distortion comes from pushing an audio device beyond its physical capacity to respond. The medium is then "saturated" or overdriven. Anything–mics, preamps, mixers, sub-masters, fx units, instruments, speakers, electronic tape, digital bit depth, even your ears–can be saturated. When the source goes beyond that physical barrier it is not reproduced and therefore lost, or clipped. The result is loss of upper frequency and a fuzzy or gravely sound[16].

It is possible to overdrive a microphone by placing it too close to the source. Most mics have a "pad" selector which cuts the input gain by 20 or 15 db. The most common source of distortion is saturating electrical or digital components in the signal path. A volume gain knob is usually supplied with an audio component along with either an led meter, showing input level, or a red and/or green single led to indicate if there is signal present and if the signal is over the units capacity.

[16] For example, sub-woofers in cars. Beauty is in the eye of the beholder.

Setting Levels

When setting levels you should first ask the performer to play a passage that represents the loudest point in the work, or even the loudest that the instrument can play. During that time set levels so that you have a little headroom (about 5 db) before clipping. It is a good idea to set levels while the artist is playing an actual excerpt of music, maybe even with other tracks playing along. I find that real performance is more aggressive, and a better representation than a "sound check."

Keep in mind that any component in the path can be overdriven and should be monitored. Many studios make a single adjustment so that all level meters register the same. This way a single meter can serve to monitor all meters. The term for this is "unity gain."

The general idea when setting levels is to be as far above the bed of noise without overdriving any component. Analog tape is much noisier than digital, and overdriving analog is a softer, more forgiving sound. So with that equipment the levels were daringly hot. Digital distortion is harsh, and the bed of noise is much lower. So if I'm not sure about a signal I err on the low side. It can be optimized later without adding as much noise.

Bit Depth

Bit depth, to me, is a source of noise, which I want to be above. The higher the bit depth, the more headroom I have to work with.

A bit is a unit of memory that can represent 0 or 1. With one bit you can only count to 1 (0 and 1). With two bits, each can be on or off (0 or 1), so you can count to 3 (00, 01, 10, 11). The more bits, the higher you can count[17]. With 4 there are 16 possible combinations, so you could count up to 16. 16 bits have a little more than 65,000 possible combinations. So you can have numbers roughly between –32,000 to +32,000.

Sample rate is how many samples are taken per second. Bit depth is the size of the numbers used for the samples. Bit depth represents amplitude, since larger numbers can accurately describe wider ranges of amplitude. Each additional bit translates into an additional 7db range. The difference between 8 and 16 bit is about 50 dB. So recording at 8 bit is analogous to recording at –50 dB. 24 bit adds another 50 dB. 8 bit gives 50 dB of headroom, 16 yields 100, 24 about 150 dB. Headroom above what? The bit depth itself (the bits themselves?).

If you record a sine wave using a low bit rate, one that allowed, let's say, 10 numbers (-5 to +5), a pure sine wave would have a ragged look, because each sample taken would have to rounded up or down to one of the whole numbers. A similar effect can be seen if you generate a sine wave at a very low dB, say 5 dB, 8 bit, then increase the amplitude around 40 dB, you would not see a clean, pure sine wave, but this:

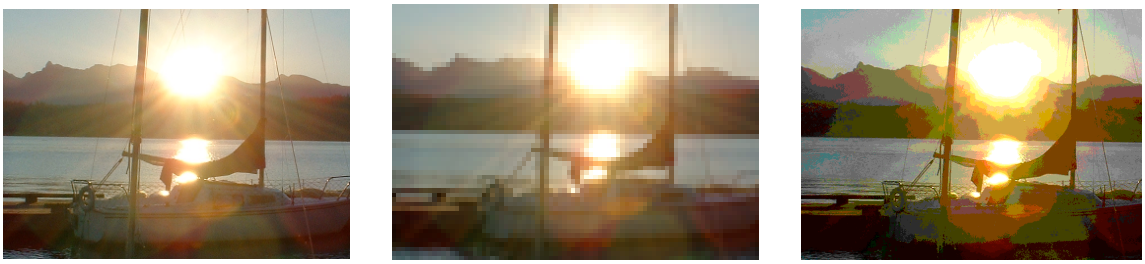
---

[17] If you want to use negative numbers you have to give up one bit for the sign. So 2 bits would really only allow –1, 0, and 1.

The terracing you see comes from the truncation of a smooth wave up or down to the numbers we have; -5, -4, -3, -2, -1, -, 1, 2, 3, 4, or 5. If, for example, the actual sound wave fluctuated between the numbers 4 and 5 the samples would all have to be rounded to 4, and that information (activity between 4 and 5) would be lost. Recording at lower bit depths adds noise, or a grainy sound.

Sample rate and bit depth can be illustrated using similar parameters for digital images (resolution and bit depth). Below is an example. The first picture is normal, the second is low resolution (notice the jagged mountain range), but high bit depth (the color range is accurate). The last is high resolution (smooth mountain silhouette) but low bit depth (terraced colors).

6.2. Sample Rate (Resolution) and Bit Depth



File Size

So why not do everything in stereo, 96k, 24 bit? I recently consulted with a student whose previous instructor made that argument. Her three minute class project was 4 gig. (Yet many of her recordings had no relevant stereo information; single instruments or spoken voice.) By comparison, the average student project is about 80M at most.

Peanuts you say? Just a DVD? I'll admit it's less and less of an issue. But if you are working towards a complete CD with 10 or so tunes, each one 3-5 minutes, that's 40 gig. Your next CD; 80 gig. With thirty other students it's certainly an issue in our labs. If a professional studio worked that way for two months they could be ten or twenty times that. Every time you open the project, back it up, or transfer it from the server, you will have to wait. If you have the time and the room, go ahead.

Maybe I'm being too concerned for economy. It comes from working on 60 minute 4 track tapes where every inch of recordable surface was precious. But there's a more subtle issue: It's a reflection on your skills as an engineer. Always pushing the limits of file size, never throwing away bad takes, duplicate files, etc. is just sloppy. Such bad habits will come back to haunt you. A talented engineer is economical and efficient.

After converting this students project to 44k, 16 bit, stereo files only when necessary her project was 100M. Much easier to manage. So that's the format I always suggest for this class. When you get your own hard drive, knock yourself out.

So here are the numbers. File size relates directly to number of tracks, sampling rate, and bit depth. If you reduce any of them by half or a fourth, the file size changes by the same degree. Here is what you should remember: 1 minute of CD quality audio (44.1k, 16 bit, stereo) is 10M. Change that to mono; 5M. Resample to 22k; 2.5M, 11k 1.25M. Record at 8 bit; .6M per minute. One hour of CD quality audio is 600M (about the size of a CD), but one hour of 11k, 8 bit, mono is about 30M. Perhaps a better choice for recording a lecture. (Then again, just do MP3.)

The "Correct" Recording Method

I apologize if this gets a little preachy, or should I say secular, since I hope to dispel the myth that audio engineering is a religion.

So far I've just given you choices. I haven't made any specific recommendations such as record with a close mic, or a distant placement, but rather to use close mic placement if you have a noisy room, distant if you want to include ambient echoes. I didn't say record at 44.1k or 96k. I gave a list of sampling rates to choose from. So what is the "correct" method to use? You should use the method that achieves your goals using the least amount of time, money, space, and resources.

I've never understood the emotional investment some have in audio technique. I usually avoid the subject, as one avoids discussing religion at dinner. I hardly dare suggest to an engineer that he might consider not using monitors. They treat it as blasphemy. So at the risk of offending I will categorically deny the existence of an audio god. There is no true sample rate. There is no one shining path to mic placement. There are just intelligent choices.

Here is the incident that inspired this rant. Now and again I need to digitize something that is playing back on my Mac; maybe the warning beeps from the system or an audio file that I can play using a browser but can't open or download for whatever reason. There are several tricks to do this; you can plug the output of one machine to the input of another, or even route the output of a single machine to its own input. There are also a number of software solutions that can record any sound your computer makes. I've used most of these methods, but they are all a little involved or expensive. One day I was in a hurry and needed to digitize spoken dialog from a website. I launched a digital recorder and started digging around for a patch cord to connect the input and output. I then noticed, I have to admit by accident, that the recorder was already seeing the signal from the website audio. This is because the system input was set to the external mic, which on my laptop is right next to the speaker. The mic was hearing what I was hearing without any patching. I hit record and was on my way. The quality wasn't great, but it was surprisingly good and good enough for what I needed.

I thought this was a pretty good trick and posted it to a Mac hints website. I'm always prepared for alternate opinions by other authors, but I was surprised by the flaming this post received. Other audio buffs jumped at the chance to make fun of my "ghetto" technique.

Surely, they said, a "real" engineer wouldn't make such a silly mistake as allowing air into the signal path. They suggested my students get a refund.

I often record the minutes of committee meetings. I guess if I didn't want to embarrass myself among peers I would set up an AKG 414 on a boom stand with wind screens for each person sitting around the table. I would route that to a PT TDM (maybe two if it's a large meeting) digitizing at 96k, then mix down using compression, reverb, a little echo, and maybe some chorus. I would then hand the director a firewire hard drive with the 75 gig of files required to play back our jokes about how we never get any funding.

But since there are no high priests watching over my shoulder I record using the cheap little mic on my laptop at 11k 8 bit mono (uncompressed; takes less time to save or open) and give the director a jump drive containing a file one thousandth the "real engineer" size for the two hour meeting.

You've probably heard that a gentleman is someone who knows how to play the accordion but doesn't, and a talented engineer is one who *has* the equipment, but doesn't always use it. He only uses what he needs. And a polite engineer will shut up and let the other guy butcher the sound with unnecessary monitors. End of soap box – I feel much better.

Music Concrète

The more universally accepted definition of musique concrète is composition using live recordings as source material which is manipulated, altered, or distorted in novel and unusual ways. Though typically used to describe art music, this technique is common in movie soundtracks and popular styles.

The lesser known original, but perhaps more accurate, definition refers to any recorded sound. It seems common to us, but the ability to reproduce a performance exactly over and over was, of course, an extraordinary idea at the time. A recording can never change, so you might say, and they did say, it is set in concrete. Using that definition we can say that a very high percentage of music we hear is concrète.

Early concrète composers were not looking for new sounds as much as they were trying to expand the range of existing instruments and abstract new sounds from that context. These innovators were limited to the editing features available on analog tape. They were: looping, abrupt edits (and therefore radical shifts in material), speed change, and reversal. These tape deck features were not originally intended for composition, but that didn't stop them, and the tradition of extending audio tools beyond their original intent continues today.

6. Exercises

6.1. Calculate the sizes of a 10 minute recording in these formats: mono, 22k sampling rate, 8 bit; stereo, 88.2k, 24 bit; mono, 44.1k, 16 bit; and mono, 11k, 8 bit.

6.2. Record any signal using an external mic and the input gain turned down as low as possible with distant mic placement. Normalize that section. Next record at optimum levels. Compare the two examples. Also record a signal with levels too hot. Zoom into the wave to observe clipping. Try to fix it by changing the amplitude (digitally). Finally, perform several non-zero edits and listen to the results. Can you hear clicks? Repair the clicks.

6.3. In Amadeus, generate a sine tone at 0.1 % amplitude (about -60 db). Increase the amplitude by 55 db. Notice the rough edges and grainy sound.

6.4. Open any audio file that contains plenty of high frequencies (e.g. cymbals) in a two track editor such as Amadeus. Repeatedly open the original quality file and save the file changing to these sampling rates, bit depths, and tracks: 44.1/16/Stereo, 44.1/16/Mono, 22/16/M, 11/16/M, 44/8/M, 22/8/M, and finally 11/8/M. Note the file sizes. Next, again working with the original file, reduce the amplitude by -30 db, then increase it by 30 db and save. Open the original again and reduce by -60 then increase by 60. (This produces the same effect as if you had recorded at -30 and -60). Import all of the files into Logic, create a take folder, then *at measures* switch between each quality. Bounce, open in Amadeus, create a sonogram.

6.5. Practice saying a short phrase such as "this is a satanic message" backwards. Record yourself saying this phrase in a two track editor. Use DSP to reverse the track. It doesn't have to sound correct, just interesting. Using correct copy and paste techniques, collect words from several recordings of news casts (provided by the instructor), then recombine these sections to construct new sentences. Then collect vowels and consonants and recombine them to say "Editing provided by <your name>" in a new file. It may not be perfect, just try to get close. Focus on clean edits.

6.6. Record the phrase "I am sitting in a room. I am recording the sound of my speaking voice, and I am going to play it back into the room again and again until the resonant frequencies in the room reinforce themselves, so that any semblance of my speech, with perhaps the exception of rhythm, is destroyed." (Alvin Lucier) Play that back into a room (you are not allowed to patch directly from one device to another) and record it on another device doing your best to maintain optimum levels. Repeat back and forth until you can no longer understand the text. How many repetitions does it take? Hand in the last audio file with the number of repetitions in the file name.

7 - Microphones, Cords, Placement Patterns

Connectors and Cords

There are two types of audio cables; balanced and unbalanced. Unbalanced connectors use two wires; one for the positive, the other for the ground or shield. They are less expensive, but more prone to noise and interference (e.g. RF noise). A rule of thumb is to use balanced lines for distances greater than six feet. In many studios all lines, even short ones, are balanced.

A balanced line requires three wires: one for the original signal, one for an inverted version of the signal, and the third as a shield or ground. The balanced line gets its name from the two mirrored signals. It uses these two signals in an ingenious method of noise elimination.

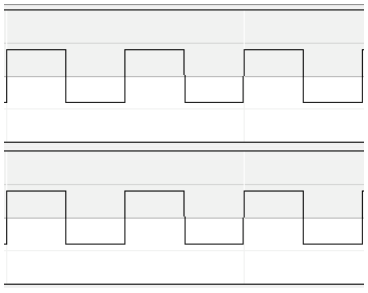
The two signals are inverted just before they travel down the cable. Think of the inverted signal as anti-matter. And as every sci-fi fan knows, when you mix anti-matter with matter, they both disappear. If, at the other end, these mirrored signals were just mixed together they would cancel each other out, being exact opposites. To prevent this from happening the receiving component re-inverts the second signal, then when they are mixed together they are just summed. The signal is not changed.

So why invert them? To eliminate noise captured along the way. The clever part of balancing a line is that it *assumes* there might be some additional noise, and that it will be present in both lines, the one carrying the original and inverted signal.

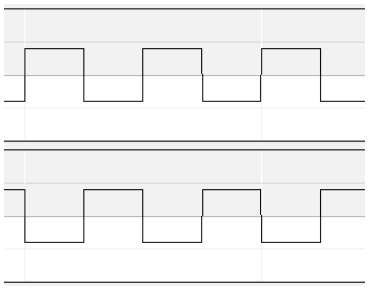
Since the noise was gained after the originating component inverted the two signals, then the noise will not be inverted (matter, not anti-matter) in the second line. If you just mixed the two lines they would be summed, and twice as loud. But when it reaches its destination, the receiving component will invert the second line, turning the original signal back to normal, but the noise in the second line, which, will be inverted, anti-matter so to speak, and will vanish when mixed with the "matter" of the first line.

As an illustration with a square wave to represent signal, triangle waves for noise.

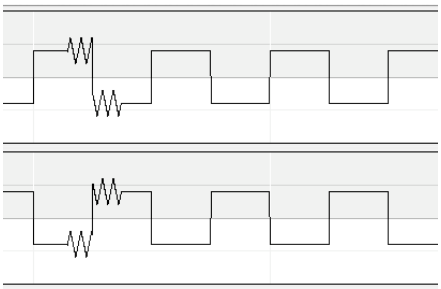
Original signal, split to two audio lines:



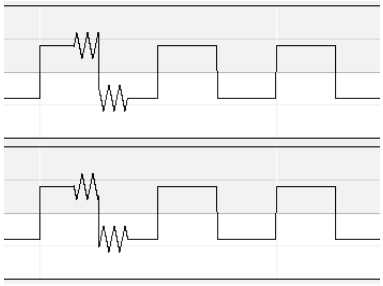
Second channel is inverted. If these two were summed they would cancel out:



This signal is sent down the cable and picks up noise in both channels. Look carefully at the peaks and valleys of the noise to confirm they are exactly the same:



At the other end of the cable, line two is re-inverted. This inverts the signal back to normal, but also inverts the noise. Look carefully at the peaks and valleys again to confirm that the noise is a mirror image:



When these two lines are mixed the signal will be summed. But the noise, which is now a mirror image, will be canceled out.

Since cords can also be stereo or mono, that complicates the number of wires I need for balanced and unbalanced lines. In stereo cords separate wires are needed for each positive signal, but the shield and negative connection can be shared. So a cord with five contacts can carry a stereo balanced signal (left original, left inverted, right original, right inverted). A more common cord has three contacts and can either be unbalanced stereo or balanced mono. Two contacts can only be unbalanced mono.

The most common types of connectors are RCA, XLR, TRS (tip, ring, sleeve), TS (tip, sleeve) as shown below. RCA connectors are always mono unbalanced, but often come in pairs for stereo patching. While XLR could carry unbalanced stereo signals, they are nearly always used for balanced mono. TRS are common as both balanced mono or unbalanced

stereo. TS can only be unbalanced mono. The TRS and TS come in 1/4" and 1/8" sizes. I usually carry a collection of adapters that include RCA female/female, RCA to quarter inch, RCA to mini TRS, and a stereo TRS to mono splitter. The TRS is also often used as an fx "insert." In this case one end has a TRS connector, tip is the send, the other for the receive, the last for the ground. The other end of the cord is split into two TS connectors for the input and output of the fx unit.

When adapting an unbalanced signal (such as a TS guitar or instrument feed) to balanced (such as XLR), it is a good idea to use a converter box.

7.1. Connectors: RCA, XLR, TRS, TS

Engineers are very particular about cord care. One of my biggest peeves is tape. Never put tape of any kind on a cord. Use other methods of labeling and use rugs to anchor them to the floor. There are several good techniques for wrapping cords. If you practice you can master the alternate reverse loop used by mountain climbers and cowboys to keep ropes from twisting. I used that method for years but now use a simpler, faster method of "gathering" the cord and letting it fall where it wants (usually in a figure 8). The figure 8 looks a little unkempt, and though counter intuitive, it actually prevents tangling. Whatever method you use DO NOT wrap them around your arm (hand to elbow). This causes twists and premature wear.

Condenser Mics

The diaphragm of a condenser mic is made of two thing metal plates. An electrical charge is placed on each disk, one positive, the other negative. The change in electrical voltage between the two is measured when the pressure of sound waves moves one. That voltage is transmitted to the recording device.

Condensers are generally more expensive, more delicate, and do require a power source. This comes either from a battery inside the microphone or from the mixing board ("phantom power") traveling down the microphone cable.

Because condensers are more sensitive and have a greater effective range they are more common in studio recording applications and less in live PA or sound reinforcement, where they are more prone to feedback. Condensers are fine when isolation is not an issue, and are the best choice for ambient stereo recordings, where a single microphone pair is used for an entire orchestra.

Dynamic Mics

The diaphragm of a dynamic microphone (like a small drum head) is connected to magnets inside a coil of wires. The sound waves move the diaphragm and magnets between the wires

which pick up the electrical field. Most speakers work the same way. You can even use headphones as a microphone in a pinch.

Dynamic mics are inexpensive, rugged, and do not require a power supply (and are not damaged by a power supply if using balanced lines). They are ubiquitous, especially in PA applications. They have an effective range of about a foot, so are a good choice for spot placement and isolation, but not for large groups. They are usually unidirectional, meaning they pick up sound mostly from the front. This also makes them a good choice for isolation.

Stereo Pairs

Dynamic microphones are mono. Condensers are usually mono, but also come as a stereo pair, where the microphone housing contains two capsules for left and right channels. Stereo mics require a cable with at least four connectors. I find stereo microphones less flexible, but they are a good choice if you will always be using them for ambient stereo recordings (such as a hanging mic in a concert hall).

Handling and Care

I use a few simple handling techniques to reduce the risk of damage. First I keep microphones in their clips. This also keeps mics and clips paired correctly. Store mics in a box, closed closet, or bag. This will protect them from dust. Next I twist the mic stand or boom onto the mic holder rather than twist the mic to attach it to the stand. To reduce possible damage to the mic and speakers it is a good idea to turn phantom power off before connecting or disconnecting a mic. You should also power down speakers and amps to avoid the mostly annoying, but possibly damaging pop.

Placement, Proximity

Mic placement is very personal. Engineers and even text books will rarely give a definitive placement position. That's because there are lots of ways to mic an instrument. Here are some guidelines.

When choosing a spot (and a microphone), first I ask the performer if they have a preference. It's their instrument, they live with it, they've probably recorded it a lot and know what works. If they don't have a preference, then I listen, first apologizing for getting in their space, to find a spot (or spots) that I like. I go for a simple and natural sound so usually one mic is enough, two (in a stereo configuration) for a piano. There's nothing wrong with using more microphones, giving you more choices when mixing, but they also may complicate unnecessarily and invite phase cancellation if used together.

You next have to decide how close to place the microphone. For a natural sound I usually mic farther back than many engineers. Close mics on a piano may produce the rich and full sound familiar to pop recordings, but a distance of two and up to six feet or more for a piano allows the instrument to do it's job of blending the overtones into a true piano, guitar, trumpet, etc. Distant mic placement also reduces mechanical noise such as dampers, flute keys, breaths, fingers on strings, etc.

Another advantage to greater distances is more uniform amplitude. With close mic placement, one inch difference in proximity can translate into 10 db. It's hard to get a performer disciplined enough to stay exactly two inches from the mic. The difference in amplitude decreases with distance, and the performer doesn't need to be as careful about their proximity. The results are very similar to a mild compression.

More distant mic placement assumes you don't want the amplitude to change when the performer moves closer or farther away. On the other hand, it is important to realize (in the case of PA reinforcement) that the performer has as much as 20 db change at their disposal through proximity. I've always admired groups that ask me to set levels at a uniform gain and then do nothing. They mix themselves by stepping into a mic for a solo. They consider the mic and the PA part of their instrument.

It has become increasingly popular for small folk groups to use a single condenser mic for the entire group.

The last consideration regarding proximity is room mix. If you want to capture the natural reverb of a performance hall, the mix (wet vs. dry) can be controlled by proximity. Some engineers use two stereo pairs in a hall; one close to the instrument, the other near the back, often even facing the back walls to pick up the reverb, which is then mixed later. Again, I try not to fiddle with nature. If it sounds great with me standing 10' away why mess with it? If the room doesn't provide the reflections you want, then mic close and put reverb in during post production.

Axis

Axis describes the angle of the microphone diaphragm in relation to the source. If the face of the diaphragm is toward the source, it is on axis. Mono mics should always be placed on axis. This is the single most common error in PA use. Most people lack the proper experience with PAs and hold a microphone so that it is pointing toward the ceiling. This places it 90˚ off axis from their voice. The signal will be weaker and thinner.

Axis can be used as a strategy for isolation. Unidirectional mics can be used to isolate instruments facing each other or a monitor source (to avoid feedback). In the case of stereo patterns (described below), axis is used to isolate left and right images.

Patterns

Each mono microphone may offer a variety of patterns; cardioid (shaped like a heart), omni (all directions equally), figure 8, etc.

Two microphones are often combined in a stereo pair. One popular pattern is an x/y figure 8 where the microphone capsules are placed directly over one another, each facing 45˚ to the left and right of the source (90˚ from each other), both in a figure 8 pattern. The figure 8 preserves the stereo image in front and adds a nice hall mix, also in stereo.

One very clever, and slightly mysterious pattern is called mid-side. One capsule is facing the source, the other faces the sides (90˚ off axis) in a figure 8 pattern. The center is mixed to the center and the sides are mixed hard left and right, with the right channel's phase reversed. This produces a less convincing stereo image, but the advantage is a clear mono reduction. Since the left and right channels are inverted, if mixed to mono they will cancel each other out and you are left with the signal from the capsule facing on axis to the source. This pattern is used extensively in television programs, where mono and stereo playback are practically equal possibilities.

My favorite pattern, not only for the results, but because it makes so much sense, is an ORTF (used by French radio stations). There are many variations, but basically it uses two microphones about 17 centimeters apart at a 110˚ angle from each other (each 55˚ off axis from the source), cardioid or figure 8 pattern. In addition to accurately capturing the relative amplitude between left and right sources, it reproduces time delays, phase cancellation, and filtering of frequencies that would occur naturally with our ears, which are also about 15 centimeters apart and at a 110˚ angle from a source. And that is a good way to remember this pattern; about as far apart as your head, and at the same angle as your ears.

In the past three years I've become a proponent of the Crown SASS, which places two capsules at roughly an ORTF configuration, but comes in a single housing that imitates the shape of the human head. Our class has done several blind studies with a number of patterns and microphones and this single stereo pair wins hands down.

7. Exercises

7.1. Place a microphone 2' from the source. Set levels and record the source while playing and while not playing (to sample the ambient noise). Repeat at 10'. Compare the two.

7.2. Set up a microphone about 1' from a source and adjust levels. Move farther away, 1' at a time, but do not readjust levels. Record each time and observe the difference in db.

7.3. Using a condenser microphone to record a source on axis, 90° off axis, then 180° off axis. Compare the results.

7.4. Place two condenser mics on axis, but at different distances. Calculate the correct distance to cancel the frequency 220, then repeat for 880 (or whichever frequencies is tonic in the example you play).

7.5. Place a condenser so close to a source that it distorts the microphone. Be careful to set all levels properly, so that it is clear you are saturating the mic capsule.

7.6. Set up two microphones, one routed through a mixer such as a Mackie 1202, then through the line out to the mini jack of a computer, the other directly to the mic input of a digi 002. Unplug both mics, then record the system noise of each. Increase the amplitude of each until they are about the same. What is the difference in db?

7.7. Place six or seven microphones at different positions around a piano. Include a treble/bass, a stereo pair close and at a distance. Record the same program with each set of mics. Using clean editing techniques, patch together a complete take where sections, maybe even single chords, come from one mic placement or another. Try to maintain consistent overall amplitude.

7.8. Do the above experiments and record each change using a video camera. Synch the audio to video (be sure to record at 48k).

(The previous chapters are used for an introductory course in computer music. The following chapters move on to digital synthesis using SuperCollider 3.)

Section II: Digital Synthesis Using SuperCollider 3

8 - SC3 Introduction, Language, Programming Basics

Basics

When you first launch SC a window appears that prints information about the program during startup. You can ignore the first barrage of cryptic lines, but move the window over to another part of the screen where you can see it, as it will continue to print useful information such as errors or results of your code. I'll call it the post window. It is possible to use this window for editing and entering code, but I prefer to open a new window.

SuperCollider is a text editor, programming language, compiler, and digital synthesizer all in one. There are two sides to the program. The *client* is essentially the windows where you type and develop code. That eventually results in commands that are sent to the *server* side, which actually makes the sound. The text editing functions (select, copy, replace, etc.) are similar to any basic editor. But there are a few handy editing features that are unique to, and useful when editing code. If you've never written code their value is not immediately apparent. So I suggest you come back and review these often.

```
Com-,                       Go to line number
Com-/                       Make selected lines a comment
Opt-Com-/                   Remove comment marks on selected lines
Shift-Com-B                 Balance enclosures
Com-]                       Shift code right
Com-[                       Shift code left
Com-.                       Stop all playback
Shift-Com-/ (Com-?)         Open help file for selected item
Com-'                       Syntax colorize
Double click enclosure      Balance enclosures
Com-\                       Bring post window to front
Shift-Com-K                 Clear post window
Shift-Com-\                 Bring all to front
```

To evaluate selected code you press the *enter* key (not *return*, but *enter*). If you want to evaluate a single line you don't need to select it, just position the cursor anywhere in the line. To run the examples in this text, type the line(s) in SC, position the cursor anywhere within the line or select all the lines and press enter.

All programming texts are bound by geek law to use "hello world" as the first example. Type in these lines precisely, position the cursor in one line (or select it), and press enter, then the other and press enter. For the third example, select both lines.

```
   8.1.   Hello World

"Hello World";

"Hello World".speak;
```

```
"I've just picked up a fault in the AE35 unit. It's going to go a hundred percent
failure within seventy-two hours.".speak;
```

Note that each line is terminated with a semi-colon. This indicates the end of a line, or expression in computerese. Example 8.2 shows more detailed examples.

8.2. Musical Math

```
5 + 4;

(5 + 4).asString.speak;

(440 * 3/2).asString.speak;

("The frequency for E5 is" ++ (440 * 3/2).asString).speak;

("The frequency for middle C is " ++ (60.midicps.round(0.1).asString)).speak;
```

There is a good reason for calling this "code." Beginners find it cryptic. It is indeed a new language, just like French or, maybe more apt, Chinese. But as languages go, smalltalk is easy, and often reads like English. The parentheses mean either "here is a list of values to use" or "do this first", and the quotations mean "this is a string, or text." The "dots" mean "send this information to that object." And the ++ means "concatenate this text with the next." Here is a translation of the last line in the example above: Convert the MIDI number 60 into cycles per second (midicps), then round that value to 0.1, convert it to a string. Concatenate that with "The frequency…" and speak the results.

The examples above don't produce any synthesized sound, which is our goal. But first you have to start the synth server, which actually makes the sound based on commands written in code. There are two servers (the two small windows in the lower left corner), internal and local. Don't worry about the differences for now.

You only need to boot (i.e. power on) the server once for each session and leave it running. This is done by either clicking the *boot* button[18] on the user interface or by running the line of code below. I prefer running the code because it names the server "s." Unless otherwise indicated all the examples in this book will assume the server is running, that it is the default server, and that it is called "s." Run, or "evaluate" the line in 8.3.

8.3. Booting the server

```
Server.default = s = Server.internal.boot;
```

[18] In this case you should also click the "default" button.

Before you run your first patch you need to know how to stop a sound. Command-period stops all playback (but leaves the server running). This text assumes you will stop playback after each example. Type and execute the following line.

8.4. First Patch

```
{SinOsc.ar(LFNoise0.ar([10, 15], 400, 800), 0, 0.3)}.play;
```

This example is more complicated, with several lines of code. Select everything from *{* to and including *play* and press *enter*.

8.5. Second Patch

```
{
   RLPF.ar(
      LFSaw.ar([8, 12], 0, 0.2),
      LFNoise1.ar([2, 3].choose, 1500, 1600),
      0.05,
      mul: 0.4
   )
}.play
```

Error messages

One thing you will notice right away is that the compiler is very unforgiving about syntax, spelling, and punctuation. Every comma, period, upper and lower case, and semicolon needs to be correct. If something is incorrect the program will print an error message in the data window. Sometimes the error messages are hard to decipher. But one piece of information is usually quite clear; it will indicate precisely where it broke down. Here is a typical error message.

```
• ERROR: Parse error
   in file 'selected text'
   line 1 char 45 :
   {SinOsc.ar(LFNoise0.ar([10, 15], 400, 800), 0 0.3•)}.play
----------------------------------
• ERROR: Command line parse failed
```

"Parse error" usually means a typo. The third line of the error message tells you where in the file the error was encountered. In this case, line 1 char (character) 45 indicated by a "•." This is where the parsing failed. However, you usually have to look carefully at the characters or lines just *before* the "•" to find the problem. Compare the line in the error message to the original in your first patch to see if you can find the error.

There is a missing comma. Don't be discouraged with error messages. They are normal. While I was writing one of the examples below (about 12 lines of code) I had to run and correct errors six times (not just six errors, but running it, correcting it and running it again six times). I should probably be more careful, but my point is even people who have been doing this a while make errors. It's so easy to run the code then correct the errors I don't really think of it as a mistake, but rather using the program to spell check my code.

Objects, messages, arguments, variables, functions, and arrays

It is essential that you learn to identify these elements in code examples.

Messages	These are lower case words, sometimes on their own, but usually connected to objects separated by a dot (.): play, scope, ar, max, rand, midicps.
Arguments	A list of items separated by commas, enclosed in parentheses, following a message: (1, 2, 4), ("string", 45, myVar), ({function}, 0.1, [1, 2, 3])
Variables	User defined names of memory locations, beginning with lower case letters (not a number), contiguous (no spaces), and often words strung together with caps in the middle: pitchClass, nextCount, freqArray, myVar, a1, start4, etc.
Enclosures	Matching parentheses, brackets, and braces, that define functions, arguments lists, and arrays: (0, 100, 555), {arg next; next = 100}, [100, 200, 500]
Functions	Anything enclosed in braces: {arg next; next = 100}, {possibly many pages of code}, {SinOsc.ar(60.midicps)}
Arrays	A list of items separated by commas, enclosed in brackets: [1, 2, 3], ["C", "D"], [a, b, c]
Objects	Actually, everything in SC is an object, but for the next few chapters we will focus on those that are upper case words, numbers, bracketed or braced items, and quoted text: Synth, SinOsc, LFNoise, EnvGen, [1, 2, 4], 5, 6.7, "string", etc.
Expression	A line of code punctuated by a semicolon.
Ugens	Unit generators are a combination of object, message, and argument list that result in some type of output. Ugens are connected together to create a patch: LFNoise0.ar(0.5), SinOsc.ar(440, 0, 0.4), Sequencer.ar(s, r, j)

Enclosures (parentheses, braces, brackets)

Enclosures are used to group things together and indicate the order of execution. Each opening parenthesis, bracket, and brace has a matched closing parenthesis, bracket, and brace. After you've programmed for a while you get used to matching up enclosures in your head. The compiler runs or evaluates the code from the inner most matching enclosures, then works outward. If you don't have matching enclosures you will get an error message. When items are nested you often will see six or seven open enclosures as you read left to right, then all of the closing enclosures later in that line, as illustrated below.

8.6. Balancing enclosures

```
{Saw.ar(Pulse.ar(max(rrand(rrand([12,15],5))),400,800),0,0.3)}.play
                                  [     ]
{Saw.ar(Pulse.ar(max(rrand(rrand([12,15],5))),400,800),0,0.3)}.play
                                 (         ) etc.
{Saw.ar(Pulse.ar(max(rrand(rrand([12,15],5))),400,800),0,0.3)}.play
{Saw.ar(Pulse.ar(max(rrand(rrand([12,15],5))),400,800),0,0.3)}.play
{Saw.ar(Pulse.ar(max(rrand(rrand([12,15],5))),400,800),0,0.3)}.play
{Saw.ar(Pulse.ar(max(rrand(rrand([12,15],5))),400,800),0,0.3)}.play
{Saw.ar(Pulse.ar(max(rrand(rrand([12,15],5))),400,800),0,0.3)}.play
```

One method for clarifying the matching enclosures is to indent each new level, as shown in example 8.7. Matching enclosures should be on the same level of indentation. Often the opening enclosure is included on the same line as its associated object message combination.

8.7. Balancing enclosures with indents

```
{ // 1
   SinOsc.ar
   ( // 2
      LFNoise0.ar
      ( // 3
         100, 400, 800
      ), // 3
      0, 0.3
   ) // 2
} // 1
.play

// Or

{ // 1
   SinOsc.ar( // 2
      LFNoise0.ar( // 3
         100, 400, 800 // 4
      ), // 3
      0, 0.3
   ) // 2
} // 1
.play
```

Pressing Shift-Com-B or double clicking on any enclosure is another way to quickly see matched sets. SC will shade everything within matching enclosures. To see how this works, place the cursor in any example and repeatedly press Shift-Com-B.

Look again at the examples above and practice identifying each of the items we've discussed. I'll do the first one: The objects are *SinOsc*, *LFNoise0,* the function, and all the numbers. The messages are *play* and *ar*. Functions and arguments are a little harder to spot. All of the text *{SinOsc … 0.3)}* is a function (everything between { and }). Arguments can be hard to see if they are nested. The way you spot them is to look for a message such as *.ar* followed by an opening parenthesis. All the items between that opened parenthesis and the matching closing

one are the arguments. In the code *LFNoise0.ar(10, 400, 800)* the *.ar* is the message, so *(10, 400, 800)* is a list of three arguments for the ar message.

Arguments

Arguments represent control parameters. They are similar in function to the knobs on analog synthesizer that change frequency range, filter cutoff, amplitude, sequencer speed, etc. You can think of arguments as the value a knob would point to.

Take the following patch as an example. The arguments for *LFNoise0.ar* are *(10, 400, 800)*. Change these three values and run the code to see how they change the sound. Try 15, 25, 30, or 40 for the first argument. Try 100 and 700, 30 and 100, 1600 and 1700 for the second two arguments.

```
   8.8.  Arguments

{SinOsc.ar(abs(LFNoise0.ar(10, 400, 800)), 0, 0.3)}.play
```

How do you know what the arguments represent, or how they affect the sound? The first step is to look up the help file; highlight the item you want help with and press *Shift-Com-/* (i.e. *Com-?*)[19].

Now that we've identified the arguments for *LFNoise0.ar*, what are the arguments for the *ar* message that follows *SinOsc*? They are all enclosed in the parentheses following the *SinOsc.ar*. Notice that the first argument for this list is all the code we were just looking at in the previous example; *LFNoise0.ar(10, 400, 800)*. The second argument is 0, and the third argument is 0.3[20]. So the arguments for *LFNoise0.ar* are combined with *LFNoise0.ar* as the first argument for *SinOsc.ar*. This is called nesting. *LFNoise0.ar(10, 400, 800)* is nested in the argument list for *SinOsc.ar*.

There is one more object and message to identify. The function is an object and *play* is the message.

Sandwich.make

The terminology of object oriented languages is more difficult because the object names and messages are often cryptic acronyms (*Synth*, *PMOsc, LFNoise0*, *midicps, rrand,* etc.). So I'll use familiar fictitious objects and messages to explain how they work. (These examples won't work in SC!) If you are comfortable with these terms, you can skip this section.

---

[19] See also Com-j. Com-y, and the "More on Getting Help" help file. Once a help file is open I also use Com-click on the title bar to open the folder it is in to see related help topics.

[20] Note that you have to use a leading 0 when writing decimal fractions: 0.3, not .3.

Suppose we had a virtual sandwich maker and a virtual tofu based meat substitute both of which understood smalltalk commands. I'll call these fictitious objects *Sandwich* and *Tofu*.

Every object understands a collection of messages. The messages tell the object what to do. Likewise, there are many objects that understand any given message. The power of object-oriented languages lies in the way you can mix and match messages and objects.

For example, let's assume that Sandwich understands three messages: make, cut, and bake. And that *Tofu* understands three messages: bake, fry, and marinate. The syntax for sending the make message to the *Sandwich* might be this:

```
Sandwich.make;
```

If you wanted the *Tofu* to be baked you might write:

```
Tofu.bake;
```

You may be wondering if we need to give the *make* message and the *bake* message some arguments to describe how the sandwich is made and the tofu is baked. Actually we don't. Most messages have default values built into the code so you can leave them off provided the defaults are appropriate for your project. Try running the lines in example 8.9, which uses no arguments in the .ar or .play messages, in SC. Next run the example with an argument list for both *ar* and *play*. (Press command-period to stop.)

```
   8.9.  SinOsc using defaults, and arguments

{SinOsc.ar}.play

{SinOsc.ar(800, 0, 0.1)}.play(s, 0, 10)
```

The first result is a sine tone at 440 Hz, 1.0 amplitude, at 0 phase, played on server "s", output bus 0, with a 10 second fade time. Those are the defaults for *SinOsc* and *play*. Often you are able to use one or two of the defaults, but rarely will you use a message with defaults only.

Before adding arguments we need to know what each of them means. To find out you use the help files.

In each of the help files are prototypes of all the messages understood by that object, with the list of arguments the message uses. *Sandwich* and *Tofu* might be documented this way:

```
Sandwich

   *make(vegArray, bread, meat)
   *cut(angle, number)
   *bake(temp, rackLevel)

Tofu

   *bake(temp, baste, rackLevel)
```

```
    *fry(temp, length, pan, oil)
    *marinate(sauce, time)
```

It is important to understand that the argument list changes for different objects. That is to say the bake message used with *Sandwich* has two arguments, while when used with *Tofu* it has three. Not understanding this, and using the same arguments with a message to different objects is a common beginner error and will have unexpected results. When *bake* is used with *Sandwich*, as in *Sandwich.bake(20, 2)* the 2 is rack level, while in *Tofu.bake(20, 2)* the 2 is baste time. Remember; argument lists are not interchangeable between objects.

Now that we understand what the arguments for Sandwich.make are, we could put together a *Sandwich* with this mock code.

```
Sandwich.make([lettuce, tomato, pickle], rye, chicken)
```

or

```
Sandwich.cut(90, 1)
```

and

```
Tofu.marinate(peanut, 160)
```

The first line will make the *Sandwich* using an array (list) of vegetables[21], bread, and chicken. The second line will make one cut of the Sandwich at an angle of 90 degrees. The *Tofu* will be marinated with peanut sauce for 160 minutes.

Another powerful aspect (the whole point, really) of SC and object oriented languages is that everything is an object, and you can substitute them freely. Instead of chicken as the third argument I could use the nest the entire section of *Tofu* code.

```
Sandwich.make([lettuce, tomato, pickle], rye, Tofu.marinate(peanut, 160))
```

The second argument in marinate could be replaced with rrand(20, 100), which chooses a value between 20 and 100.

```
Sandwich.make(
    [lettuce, tomato, pickle],
    rye,
    Tofu.marinate(peanut, rrand(20, 100))
)
```

When a program evaluates the code it begins with the inner most parts, and uses the results of those values to run the subsequent outer layers. In English, the example above might read like this: Marinate tofu in peanut sauce for some value between 20 and 100. After marinating

[21] In SC an array often represents multi-channel expansion. Here it does not.

the tofu, use it as the meat (third argument) for a sandwich with lettuce, tomato, and pickle, on rye bread.

It is possible to link messages. For example *Sandwich.make.bake.cut* would first make the sandwich (in this case using defaults), then bake it (using defaults), then cut it (with defaults). One object can be used as an argument for another instance of the same object. For example, you could write *Tofu.marinate(Tofu.marinate(peanut, 60), 60)*. In this case, a batch of tofu will be marinated in peanut sauce for 60 minutes, then another batch of tofu will be marinated in that batch of marinated tofu (ick!).

Experimenting With a Patch (Practice)

Below are two patches. Read the help files associated with each object. You may not completely understand all the terms, such as mul and add, and how they affect the patch, but it's ok to suppress some questions and be satisfied with what you do know. For now let's say the LFNoise0 is generating values between 400 and 1200 10 times per second. How does this fit into the entire patch? Look at the documentation for SinOsc and you will see that the first argument is *freq*. The entire *LFNoise0.ar(etc.)* is being used as the freq argument in the SinOsc. To confirm this, try replacing the *LFNoise0.ar(10, 400, 800)*. with a static value such as 300. In this case you will hear a single pitch, 300 Hz.

Look at the help documentation for the second patch. See if you can make sense of the arguments in relation to the actual sound. Try to predict how the changes you make will change the sound before you run the code. I've italicized those arguments that you might want change.

```
   8.10.  Experimenting with a patch

{SinOsc.ar(abs(LFNoise0.ar(10, 400, 800)), 0, 0.3)}.play

(
// Select everything between the two parentheses
{
   RLPF.ar(
      LFSaw.ar([8, 12], 0, 0.2),
      abs(LFNoise1.ar([2, 3].choose, 1500, 1600)),
      0.05
   )
}.play
)
```

The *LFSaw* in the second patch uses an array ([8, 12]) as the freq argument. The first value is used for the left channel, the other for the right (more on this later). The *LFNoise1* is generating values between 100 and 3100 and is used as a cutoff in the *RLPF* (resonant low pass filter). The *LFSaw* is the input frequency.

Feel free to experiment. Here is one of the reasons I like to teach with SC. When working with the old analog synthesizers, entering incorrect values or making patching errors (e.g.

outs to outs) could actually damage the equipment. But in SC you can try just about anything. There is indeed a danger of crashing the machine or getting error messages or possibly damaging your ears if the sound is too loud. But otherwise you can experiment at will.

Here is another nice thing about working with code: It is self-documenting. If you make changes that result in an interesting sound you can easily save it to a file. To do this choose *save as* and name the file or copy and paste the patch into a new file. Experiment with the patch below.

Practice

```
   8.11.  Rising Bubbles

(
// select everything between the two parentheses
{
CombN.ar(
   SinOsc.ar(
      LFNoise1.kr(
         4, // LFO
         24, // range in MIDI
         LFSaw.kr(
            [8,7.23],//second LFO
            0,
            3, // range in MIDI
            80 // offset in MIDI
         )
      ).midicps,
      0,
      0.04
   ),
   0.2, // max delay
   0.2, // actual delay
   4 // decay
)
}.play
)
```

8. Exercises

8.1. Identify all objects, functions, messages, arrays, and argument lists.

```
{RLPF.ar(
    LFSaw.ar([8, 12], 0, 0.2),
    LFNoise1.ar([2, 3].choose, 1500, 1600), 0.05, 0.4
)}.play
```

8.2. Identify the two errors in this example.

```
{SinOsc.ar(LFNoise0.ar([10, 15], 400 800), 0, 0.3}.play
```

8.3. Explain what each of the numbers in the argument list mean (using help).

```
MIDIOut.noteOn(0, 60, 100)
```

8.4. Modify the CombN code below so that the contents of each enclosure are indented successive levels. For example, SinOsc.ar(rrand(100, 200)) would be:

```
SinOsc.ar(
    rrand(
            100,
            200
    )
)
```

```
CombN.ar(WhiteNoise.ar(0.01), [1, 2, 3, 4], XLine.kr(0.0001, 0.01, 20), -0.2)
```

8.5. In the above example, which Ugens are nested?

8.6. Which of these are not legal variable names?
lfNoise0ar, 6out, array6, InternalBus, next pitch, midi001, midi.01, attOne

8.7. Copy the patch "rising bubbles" and paste it into that same document. Modify this copy by changing parameters. When you hit on something interesting, save the file, paste another copy of the original below that and modify it. Repeat this four times and hand in the file.

9 - Controlling the Elements of Sound, Writing Audio to a File

When discussing visual arts we use terms such as color, shape, texture, and brightness to describe the nature of a work. The nature of sound can also be broken down into frequency, amplitude, and timbre, as described in a previous chapter. The following illustrates how to control these three characteristics using SC.

Frequency

Type the code below into and open SC window and run it (after starting the server). The *SinOsc.ar* generates a sine wave at 500 Hz. The scope message reads the results of that function and plays it. It also shows a graphic representation of the sound in a new window[22]. The peaks represent the speaker moving out and the valleys are when the speaker is moving in. The graph you see in the scope is an actual representation of how the speakers move. As the speakers move they compress and rarify the air, creating sound waves.

```
   9.1.   SinOsc

{SinOsc.ar([500, 500], 0, 0.4, 0)}.scope(2)
```

The first argument for *SinOsc.ar* is *freq*, set to [500, 500], an array with two values for left and right channels. If you change the frequency of one or the other to 1000 you should see more waves in that channel in the scope. You will also hear a higher pitch.

The musical term for frequency is pitch. Frequency is what makes a C4 different from a G6. More waves in the scope mean a higher frequency.

Try changing the frequency (first argument) to values between 10 and 20000. How does the sound change? What range of frequencies, low and high, can you hear?

What happens if you try values below 10? You may not hear a pitch, but you might be able to see the speaker cones move back and forth. When an oscillator generates a wave below 25 Hz, we use the term low frequency. These LFOs can't be heard on their own, but are often used in patches as controls. We will discuss that later.

Amplitude

We can use the same example to examine amplitude. The musical term for amplitude is volume, or dynamics. Frequency was represented in the scope by the number of waves.

---

[22] The scope window can be moved and resized. It also becomes the "focus" or front window when a line of code is run. You can press com-` to switch back to the code file. When it is the front window also try pressing period to stop, space to start, m to toggle size, and s to toggle overlay.

Amplitude will be seen as the height or size of the wave. A larger wave means the speakers will move farther forward and back, compressing the air more, which means greater volume.

Look at the help file again for *SinOsc*. It shows that the arguments are: *freq, phase, mul, add*. We used the freq argument to change frequency. We will now use the mul argument to change amplitude. To understand how this argument works, first realize the *SinOsc* object creates a sine wave by generating a stream of numbers in the shape of a wave. The numbers deviate from 0 up to 1, then down to -1, oscillating around the center value of 0 (i.e., *bipolar*). Occasionally students confuse range of oscillation (1 to -1) with the frequency of oscillation. They are different. 1 and -1 are the numbers between which the oscillation occurs. 500 is how fast it moves between those values. If we took a snapshots of the position of the wave (and hence the speakers), we would see these numbers: [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0, -0.1, -0.2, -0.3, -0.4, -0.5, -0.6, -0.7, -0.8, -0.9, -1.0, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0] and so on. These numbers can be manipulated using any kind of mathematical function: division, multiplication, addition, subtraction, etc. The mul argument scales or multiplies these numbers by the mul value. If we change mul to, say, .5, then all of those numbers will be multiplied by .5, or reduced by ½. So [0, .1, .2, .3, .4,] etc. will become [0, .05, .1, .15, .2,] and so on up to .5, then back down. In the example below we will start with 1, which is the maximum amplitude the system can manage. Before you start this example you might want to turn the audio on the computer down, or remove your headsets.

```
   9.2.   Amplitude using mul

{SinOsc.ar(500, 0, 1.0)}.scope(1)
```

Stop the sound (command-period) and try changing the mul to values between 0.1 and 0.9 and run it again. The size of the wave will change, and the amplitude corresponds. It is possible to enter values greater than 1, but they are beyond the capacity of the bit depth, and will be rounded off, clipping the wave and distorting it. Values greater than 1 are too high for volume, but they are useful in other contexts, as we will see later. Below is an SC example illustrating output saturation, clipping (the tops of the waves are clipped), or distortion.

```
   9.3.   Distortion

{SinOsc.ar(500, 0, 2.0)}.scope(1)
```

There are circumstances where you *want* a wave to be clipped, but there are better ways than using excessive values for amplitude. In analog studios you might damage equipment. In digital realms it is not as dangerous, (you won't damage the program), but you compromise the sound. More importantly, clipping from excessive values is poor technique, and you give away your lack of experience. A clipped signal is telling.

Periods, Shapes and Timbre

The example below shows a sound source that we do not perceive as pitch.

9.4. Noise

```
{WhiteNoise.ar(0.7)}.scope(1)
```

The reason we do not hear a pitch is that there is no pattern (at least in a time frame that we can perceive). The sine wave above moved smoothly from 0, to 1, 0, -1, 0 then *repeated* that motion. A sound wave (of any shape or contour) that has any type of repeated pattern is periodic. Periodic waves are heard as pitch. Each revolution through the pattern is called a period, wave or cycle. A sound with no pattern is said to be aperiodic and will not be heard as pitch. These terms are not absolute, but rather two hypothetical extremes of a continuous scale between periodic and aperiodic. That is to say a wave can be more or less periodic. The clearer the pattern, or periods, the clearer the pitch.

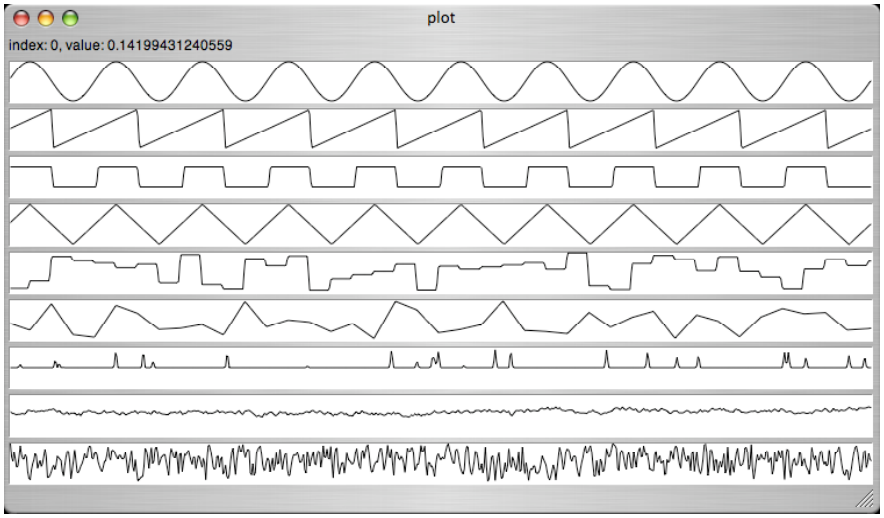
Frequency is pitch, wave size is amplitude or volume, and wave *shape* is timbre. We will discuss how different shapes come about later, but for now understand that different shapes of waves generate different timbres. In general, waves with sharp changes in direction are brighter timbres.

Below are the audio equivalents of these waves. Use the mouse (top to bottom) to hear a different wave. They are, in order, a sine wave, a saw wave, a pulse, triangle, low frequency noise, "dust", pink noise, and white noise. You can resize the scope window by dragging the corner.

9.5. wave shapes

```
 (
{Out.ar(0, In.ar(MouseY.kr(15, 23).div(1), 1)*0.8)}.scope;
{Out.ar(16, [SinOsc.ar, Saw.ar, Pulse.ar,
   LFTri.ar, LFNoise0.ar(200), LFNoise1.ar(200), Dust.ar(100),
   PinkNoise.ar, WhiteNoise.ar])}.play
)
```

Here is a graph of each all of them:



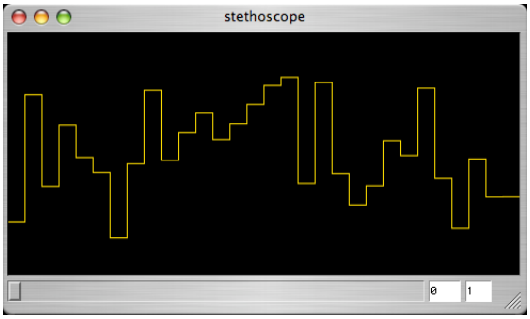
While you can clearly hear the difference between the first four, the noise examples are less distinct. That is because the patterns are less clear. The different wave types change the timbre, but they can also be used as LFO controls, that is, to shape some other parameter. In these cases it is important to understand the shape of each wave, since the parameter being controlled (e.g. frequency) inherits that shape.

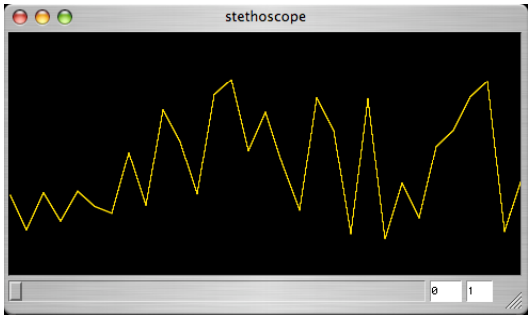
Here is the same example, except each wave form is used as a control. Again, move the mouse up and down to hear each wave. But in this case, the first SinOsc generates the pitch we hear. The others are controlling the pitch of that SinOsc, so we hear the pitch follow the shape of each wave. We will delve into controls more later.

9.6. Wave shapes evident as control

```
(
{Out.ar(0, In.ar(MouseY.kr(15, 20).div(1), 1)*0.8)}.play;
{Out.ar(16, SinOsc.ar([SinOsc.ar(2), LFSaw.kr(2), LFPulse.kr(2),
   LFTri.ar(2), LFNoise0.ar(2), LFNoise1.ar(2)] * 100 + 500))}.play
)
```

The shape of the sine, triangle, saw, and pulse waves are clear. The LFNoise0 and LFNoise1 are random (aperiodic) waves. LFNoise0 generates discrete step values. LFNoise1 generates interpolated, or ramped values. Here are diagrams of each.





Phase

In most cases the quality of sound is not affected by changes in phase, but it's an important concept in understanding how timbre, phase cancellation, and constructive or destructive interference work. Here is the line of code we started with.

```
   9.7.   Phase

{SinOsc.ar(500, 0, 0.7)}.plot
```

In the example above we use the plot message. This will not play the sound, but rather print out a plot of 1/100th of a second of the wave. The second argument (0) is phase. Try changing this argument to the values 0.785, then 1.57, then 2.355 and finally 3.14. Why are these numbers significant? They are fractions (1/4, 1/3, 3/4) of pi. The plots of the wave will begin at the corresponding point of its cycle. 3.14 (pi) is halfway through the cycle, 2*pi is the full cycle. SC3 understands pi, so it is used in the code below to illustrate progressive phases. Phase is also often expressed as degrees. 0 is 0˚, 0.5pi is 90˚, pi is 180˚, 1.5pi is 270˚, and 2pi would be 360˚, or full cycle.

```
   9.8.   Phases

{SinOsc.ar(100, [0, 0.5pi, pi, 1.5pi, 2pi])}.plot
```

Phase does affect two aspects of synthesis: the way two waves interact (when they are different phases) and the way a control wave interacts with its target, as we will see later.

To illustrate the second way phase affects sound I'm going to jump ahead to an example that uses the sine wave as a control (covered in depth later). The second *SinOsc* in the next example is controlling the pitch (rounded and converted from MIDI) of the first *SinOsc*. In simple terms, the shape of the sine wave is being applied to a virtual piano keyboard from F4 to G5; sweeping across the keys in the same way the wave moves up and down. But as you will hear, the first example doesn't begin with F4, but rather C4, the middle pitch. That's because the sine wave begins at 0, then moves up to 1, down to -1, and so on. When the phase argument is changed to 0.5pi (or ¼ through its phase), it will begin at its peak, and therefore the top of the F4, G5 range. If set to 1pi, it will begin half way through the cycle, at 0, but continue moving down. 1.5 pi will begin at its valley. Run the first line for plots of all four examples, then each example and listen carefully to where the pitches begin.

```
   9.9.   Phase you can hear (as control)
// all phases: 0, .5, 1, 1.5
{SinOsc.ar(400, [0, 0.5pi, pi, 1.5pi])}.plot;

// 0 phase
{SinOsc.ar(SinOsc.ar(0.3, 0, 7, 72).round(1).midicps, 0, 0.7)}.play

// 0.5pi (1/4) phase
{SinOsc.ar(SinOsc.ar(0.3, 0.5pi, 7, 72).round(1).midicps, 0, 0.7)}.play
```

```
// 1pi (1/2) phase
{SinOsc.ar(SinOsc.ar(0.3, 1pi, 7, 72).round(1).midicps, 0, 0.7)}.play

// 1.5pi (3/4) phase
{SinOsc.ar(SinOsc.ar(0.3, 1.5pi, 7, 72).round(1).midicps, 0, 0.7)}.play
```

Recording (Writing to a File)

The first half of this text focuses on synthesis, using SC as a slave for experiments. SC can organize and orchestrate entire works using collections of patches or MIDI. It does this by sending commands to the servers. In fact these tools are revolutionary and will change (imho) the way we compose electronic music. But for now we will take a classic approach to composition where sounds created on analog synthesizers are recorded then used as raw material, a palette of sorts, for compositional collage. In this model the formal composition takes place later, after collecting sounds, on the digital audio work station. So our goal for the first section is to generate interesting sounds with patches, record them, then import them into a digital editor such as ProTools, Amadeus, or Logic, and organize the sounds there.

On each of the servers there is a button that toggles prepare rec, rec, and stop. Keying these in sequence opens a file to record to, then records to that file, then stops the recording. The defaults for recording are 44.1k, 32 bit, stereo, with a unique auto-generated name in the recordings folder in the SC folder. That file can be imported into a digital editor and used for compositions.

I find 32 bit a problem for importing into other applications, so I change this default to 16. You can make this change in the Server.sc source file, but I move around between multi-user machines, so a command line[23] is easiest for me. I actually include it with the server boot line that we used in the beginning: *Server.default = s = Server.internal.boot; s.recSampleFormat = "int16";*

I also like to name my files before recording, so I use command lines for that too. Below are examples of command lines for setting the defaults, number of channels, file name, and starting and stopping the recording. These command lines can be used *instead* of the rec button on the server.

```
   9.10.  Record bit depth, channels, filename

Server.internal.recSampleFormat = "int16"; // or Server.local
Server.internal.recChannels = 1; // for mono

// saves file in the SC folder, will be overwritten if repeated
Server.internal.prepareForRecord("audio1.aif");
```

[23] A command line is what we've been using all along; lines of code typed, then run by the compiler. I use this term as opposed to a GUI button on the server.

```
// or saved in recordings folder of SC folder
Server.internal.prepareForRecord("recordings/audio1.aif");

// or saved in Music folder (replace students with your user name)
Server.internal.prepareForRecord("/Users/students/Music/audio1.aif");

// Then to record, don't use the button, but the following code.
// Before running these lines go back and start a previous example
// so you have something to record.

Server.internal.record;

// Then
Server.internal.stopRecording;
```

On the other hand the record button is handy for those situations when you've been experimenting with a patch and suddenly hit on something interesting; just toggle through the buttons and your cool sound is saved. You can start the record function before or after you run a patch. You can add more patches after the record is running.

Practice/Fun

Electronic music has always been fun for me. I want this text to be informative and useful, but I also hope to tap into your sense of adventure and discovery. To this end, I close each chapter with an example of practical application. The code will be more complicated, maybe out of reach, but my intention is not directly pedagogical. The examples are more of what digital synthesis should be; discovering new sounds. You can ignore the parts we haven't discussed or look up the help file.

When you hit on something interesting there are two things you can do to preserve the sound. The first is to save that patch as a separate file. You can either select all the code, copy, then paste it into a new document and save that file with an appropriate name, or you can choose "Save as" from the File menu and save that entire file with a new name.

The patch below uses three ugens to generate sound: an *LFNoise1*, a *SinOsc* and an *LFSaw*. They are linked in a rather complicated way. You can experiment as you like, changing the values for each argument, but the range of sweep should be greater than the range of overall wandering. The max delay should always be greater than the actual delay. The decay time is how long the echo resonates. I've italicized items you might want to change and indicated ranges of values you could try. The second patch sets off 5 sine waves that differ only in pan position and phase (to illustrate phase). The last one illustrates changes in frequency, amplitude, and timbre, since that's what we just covered.

Have fun.

```
   9.11.  Wandering Sines, Random Sines, Three Dimensions

(
```

```
   // Double click the parenthesis above to quickly select
    // the entire patch
    // Or use com-shift-b
{
var out, delay;
out = SinOsc.ar( //Sine wave osc
       abs( //This protects against negative values
          LFNoise1.kr(
             0.5, //frequency overall wandering
             600, //range of overall wandering
             LFSaw.kr(
                1.5, //speed of the individual sweeps
                mul: 50, //depth of sweep
                add: 500 //range of sweep
             )
          )
       ),
       0,
       0.1 //volume, stay below 0.2
    );
//delay
delay = CombN.ar(out,
       3.0, //max delay
       [1.35, 0.7], //actual delay, stay below max delay
       6 //delay decay
);
Pan2.ar(out, 0) + delay
}.play

)

// Second patch. You can run them at the same time, or several
// instances of the same patch.

( // <- double click the parenthesis
{
Mix.ar(
    Array.fill(5, // not too high, could crash
       {Pan2.ar(
          SinOsc.ar(SinOsc.ar(1/10, rrand(0, 6.0), 200, 500)),
          1.0.rand)}
    )
)*0.02
}.play
)

// This one controls the three dimensions of sound: pitch, amp, and timbre.

(
{
var trig, out, delay;
trig = Impulse.kr(6); // trigger rate
out = Blip.ar(
```

```
    TRand.kr(48, 72, trig).midicps, // range, in midi, of pitches
    TRand.kr(1, 12, trig), // range of timbre
    max(0, TRand.kr(-0.5, 0.4, trig)) // amplitudes
);
out = Pan2.ar(out, TRand.kr(-1.0, 1.0, trig));
out = out*EnvGen.kr(Env.perc(0, 1), trig);
out = Mix.ar({out}.dup(6))*0.2;
delay = CombL.ar(out, 2.0, 4/6, 6);
out + delay
}.play
)

// OK, two more; the first uses a Saw as a control (note the rising sound).
// The second illustrates how Dust can be used to trigger events.

(
{
Mix.ar(
    Array.fill(5, // number of oscillators
        {arg c;
            Pan2.ar(SinOsc.ar(  // be sure the add is greater than the mul
                LFSaw.ar((c*0.2 + 1)/3, mul: 500, add: 700)
            ), LFNoise0.kr(1)) // pan speed
        }
    )
)*0.1
}.play
)


(
{
Mix.ar(
    {Pan2.ar(
        Klank.ar(
            `[Array.fill(3, {exprand(1000, 10000)}),
              1, Array.fill(3, {rrand(1.0, 4.0)})],
              Dust.ar(1/3, 0.1)),
        LFTri.kr(rrand(3.0, 10.0)))}.dup(20)
) }.play;
)
```

9. Exercises

9.1. Draw four triangle waves starting at 0˚ phase, 90˚, 180˚, and 270˚. Do the same for a square wave.

9.2. Draw two waves, with the second a higher pitch, louder amplitude, and brighter sound.

9.3. Begin with this patch:

```
({Blip.ar(
    SinOsc.ar(2, 0, 200, 600), // control Ugen 1
    SinOsc.ar(5, 0, 5, 10), // control Ugen 2
    max(0, SinOsc.ar(5)) // control Ugen 3
)}.play)
```

Try changing the frequency of all Ugens to values between 1 and 50. Try changing the mul of the first control Ugen to values between 50 and 600. Change the mul of the second control Ugen to values between 1 and 5. Can you tell which Ugen is controlling which properties of sound?

Write new patches replacing each SinOsc with combinations of these Ugens: Saw, Pulse, LFNoise0 and LFNoise1. Adjust the arguments so that they match the new objects. (For example, the second argument in SinOsc is phase, the third is mul, then add. LFNoise has no phase argument, but it does have a mul and add.) Remember that argument lists are not interchangeable, so look up the help file.

Hand in the SC file.

9.4. Experiment with the patches in the practice section. When you find something interesting, save it as a separate file. Continue until you have several variations of each patch.

9.5. Run each of your new patches again and record it to an audio file. Collect these recordings in a single file (using Amadeus, Logic, or ProTools), correct the sample rate and bit depth, if necessary (16 bit, 44.1k) and hand that file in.

10 - Keyword Assignment, MouseX.kr, MouseY.kr, Linear and Exponential values

Keyword Assignment

In the examples above we included most of the arguments for each message. If any argument is not supplied, a default is used. Below there are no arguments for the *SinOsc*, but it still works. It uses default values of 440, 0, 1, and 0 for freq, phase, mul, and add.

```
10.1.  Defaults

{SinOsc.ar}.play
```

You might have also noticed that in previous patches we entered a 0 for the phase argument even though the default value for phase is 0. This is because the arguments have to be entered in the correct order. Phase is the second argument and mul is the third. We had to enter the 0 as a sort of place marker so that the mul would be in the correct position.

This becomes a problem if there are 10 arguments and you want to change only the sixth value. You would have to enter arguments 1-5 just so the sixth argument is in the correct position. This is not only cumbersome, but invites error.

The alternative solution is keyword assignment. Using keywords you can specify the name of the argument you are changing followed by a colon and the value. Keywords can be found in the documentation files. The documentation for *SinOsc.ar* shows: SinOsc.ar(freq, phase, mul, add). The keywords are *freq*, *phase*, *mul*, and *add*. Using keywords not only allows you to enter a single argument out of order, but to mix the order of the arguments. Here are several versions of the *SinOsc* example written using keywords. All of them have precisely the same meaning.

```
10.2.  Keywords

{SinOsc.ar(freq: 440, phase: 0, mul: 0.4, add: 0)}.play;

{SinOsc.ar(phase: 0, freq: 440, add: 0, mul: 0.4)}.play;

{SinOsc.ar(freq: 440, mul: 0.4)}.play;
```

Another good reason for using keywords is clarity. The keyword provides a explanation of what each value means, like a short comment.

The last reason for using keywords is portability. Earlier I said it is important to understand that argument lists are not interchangeable. Consider these two lines: *Saw.ar(100, 500, 600)* and *SinOsc.ar(100, 500, 600)*. The arguments for *Saw* are freq, mul, and add. But for *SinOsc* they are freq, phase, mul, and add. So the two lists are not interchangeable. I can certainly

use 100, 500, and 600 as arguments for the SinOsc, but they don't mean the same thing as they did with the *Saw*. A phase of 500 makes no sense, and a mul of 600 without an add will produce negative values, probably not what I want.

But if I use keywords (provided the two Ugens have the same keywords), then I can swap the objects: *SinOsc.ar(freq: 100, mul: 500, add: 1000)* and *Saw.ar(freq: 100, mul: 500, add: 1000)* will have a similar effect in a patch. Warning: this can be either dangerous or a source of serendipity. You should always double check the keywords[24] and understand what they mean in a patch.

Here is the first patch using keywords followed by dangerous swapping.

```
   10.3.  First patch using keywords

{SinOsc.ar(freq: LFNoise0.ar(freq: [10, 15], mul: 400, add: 800), mul: 0.3)}.play

{Saw.ar(freq: LFNoise0.ar(freq: [10, 15], mul: 400, add: 800), mul: 0.3)}.play

{SinOsc.ar(freq: LFNoise1.ar(freq: [10, 15], mul: 400, add: 800), mul: 0.3)}.play

{Pulse.ar(freq: LFNoise1.ar(freq: [10, 15], mul: 400, add: 800), mul: 0.3)}.play

{Pulse.ar(freq: LFSaw.ar(freq: [10, 15], mul: 400, add: 800), mul: 0.3)}.play

{LFTri.ar(freq: LFPulse.ar(freq: [10, 15], mul: 400, add: 800), mul: 0.3)}.play

{LFTri.ar(freq: LFTri.ar(freq: [10, 15], mul: 400, add: 800), mul: 0.3)}.play
```

MouseX.kr and MouseY.kr

In the examples above we changed each value, then ran the code, then changed the value and ran the example again. You may have wished you could attach some type of knob to the patch so that you could try a range of values at once. This is what *MouseX.kr* and *MouseY.kr* will do. They link mouse movement and position to values that can be used in the patch. The first three arguments are: *minval*, *maxval*, *warp*.

These ugens can replace any static value with a range of values that change in real time in relation to the position of the mouse on the screen. As an illustration, try the first patch reproduced below with a *MouseX* in place of the first argument for *LFNoise0*.

```
   10.4.  MouseX

{SinOsc.ar(LFNoise0.ar(MouseX.kr(1, 50), 500, 600), mul: 0.5)}.play;
```

[24] If a keyword is incorrect it is ignored, but you do get a warning posted to the data window.

Much easier than changing, trying, changing, trying, etc.

In the example below *MouseY* (Y axis is top to bottom) is used to control amplitude. The minval is 0.9 and maxval is 0.0. These may seem backwards, but the minimum position for the mouse is actually the top of the screen, the maximum is the bottom, and it makes more sense to me to have the top of the screen 0.9. The next example adds a *MouseX* to control frequency. The minimum value is A 220 and the maximum is two octaves higher, or 880. Since the motion of the mouse spans two octaves you might be able to play a tune using the mouse to find each pitch. First try an octave, then a fifth, then a scale. See if you can play a simple tune, such as Mary Had a Little Lamb.

```
   10.5.  MouseY controlling amp and freq

{SinOsc.ar(440, mul: MouseY.kr(0.9, 0))}.play;


{SinOsc.ar(MouseX.kr(220, 880), mul: 0.3)}.play;
```

One reason it may be difficult to pick out a melody is that the distance between pitches does not seem consistent across the screen. Since this is two octaves you might expect the first octave to be in the middle of the screen, but you actually encounter it about a third of the way, then the second octave spans the remaining two thirds. (Note that with violins, violas, trombones, etc., there is a similar distribution, but the opposite direction. That is, the distance between similar intervals decreases as you go higher on the fingerboard.)

The truth is, the distances *are* consistent; they are linear. The problem lies in our perception of pitch. We hear music in octaves, and octaves are exponential, not linear. The amount of change between one of the low octaves (such as 110 to 220) is smaller (110) in comparison to a higher octave (1760 to 3520; a change of 1760). With a linear range of 220 to 880 the middle of the screen will be 550. But if we want the screen to visually represent a musical scale the middle should be 440, the left half for pitches between 220 and 440 (a difference of 220) and the right half should be 440 to 880 (a difference of 440). This more logical tracking can be implemented using the *warp* parameter. The *warp* value is set using a symbol (a word in single quotes), either linear or exponential. You can also enter the number 0 for linear or 1 for exponential. Try playing a melody or major scale with the following adjustments. You will notice that with an exponential warp the whole and half-steps seem to be the same distance all across the screen.

```
   10.6.  Exponential change

{SinOsc.ar(MouseX.kr(220, 880, 'exponential'), mul: 0.3)}.play;

// or

{SinOsc.ar(MouseX.kr(220, 880, 1), mul: 0.3)}.play
```

As a general rule you will want to use exponential values when dealing with frequency.

Exponential values are more appropriate for mapping a control for pitch, but they are also important when choosing frequencies across a spectrum. Likewise, we hear music (and therefore pitch) in each octave as an "even" distribution of pitches. Here are six octaves of middle C with corresponding frequencies. Notice that the linear distance between C1 and C5 (32 to 523, about 500) is nearly equal to the distance between C5 and C6 (523 to 1046, about 500).

C1	C2	C3	C4	C5	C6
32.7	65.4	130.8	261.6	523.3	1046

In short, as pitches get higher we hear greater distances as being equal. Each octave is exponentially larger.

```
   10.7.  Exponential choices

{rrand(30, 4000.0)}.dup(20).sort.round(0.1)

{exprand(30, 4000.0)}.dup(20).sort.round(0.1)
```

I introduce *MouseX* and *Y* as a tool for trying values in patches and experimenting in real time, but it is reminiscent of one of the earliest (and very successful) electronic instruments; the Theremin. The Theremin controlled pitch and amplitude by proximity of the performer's hand to an antenna and protruding loop. Performers "played" the Theremin by moving their hands closer or farther away from the antenna or loop.

Use the components we have discussed to create a virtual Theremin where *MouseY* controls amplitude and *MouseX* controls frequency. To imitate that classic sci-fi sound you should try to play it with a heavy vibrato.

Discrete Pitch Control, MIDI [New]

The second reason you may have found it difficult to play a simple tune using the *MouseX* patch is that the pitch change was continuous. We are accustomed to hearing and playing instruments with discrete pitches. As a matter of fact, with instruments that allow continuous pitch change (e.g. violin, voice, trombone) it is usually considered good technique to perform pitches with precise discretion; no scooping or sliding.

One method for generating discrete pitches would be to use a *round* message. Try the example below to see if it is easier to play a tune.

```
   10.8.  Discrete values

{SinOsc.ar(MouseX.kr(220, 880, 1).round(10), mul: 0.3)}.play

{SinOsc.ar(MouseX.kr(220, 880, 1).round(100), mul: 0.3)}.play
```

That gives us discrete notes, but they don't seem right, and for the reason we discussed earlier. Musical scales do not track to linear values. The *round(100)* gave us values such as

400, 500, 600, and so on. But a chromatic scale, beginning at 220, is 220, 233.1, 246.9, 261.6, 277.2, 293.7, 311.1, 329.6, 349.2. Do all the math you want, the pattern between each pitch is not going to readily emerge. That's because these are not only exponential values, but based on the equal tempered scale (which we discuss in a later chapter). We have bumped up against the pervasive problem of interface; we think of pitches and scales as numbers (if C4 is 0, then E4 is 4 piano keys higher), but the *SinOsc* only understands frequency.

Fortunately there is a method for generating the frequency of an equal tempered scale by way of MIDI. SC understands MIDI numbers, and can convert them to frequency. Remember the MIDI number for C4 is 60, and each half-step is 1. So C5 is 72, A4 is 69, and so on. All Cs are multiples of 12: C0 = 12, C1 = 24, C2 = 36, C3 = 48, C4 = 60 (the octave number + 1 x 2). To convert these to frequency, use the *midicps* message.

```
10.9.  MIDI conversion

[57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71].midicps.round(0.1)
```

Now back to the original *MouseX* control patch, with even more confusion. We can now express the two octave range using the MIDI numbers 57 (A3, or 220 Hz) to 81 (A5, or 880 Hz). But wait, are you a step ahead of me? Do you use a linear warp, or exponential warp? In this case, linear, because MIDI values are linear. That is to say the difference between the first octave (57 to 59) is the same as the second (59 to 81): 12.

The first version of this improved patch appears to have correct tracking with pitches, but the values are not discrete. That is because the *MouseX* will still return fractions, that is values between the MIDI pitches, such as 62.1453 or 74.874. To force them to be discrete, we add the *round* message, which rounds them to 1. (Try 0.5 and 2.0 for quarter tone and whole tone scales).

```
10.10. MIDI MouseX.kr

{SinOsc.ar(MouseX.kr(59.0, 81.0).midicps, mul: 0.3)}.play

{SinOsc.ar(MouseX.kr(59.0, 81.0).round(1.0).midicps, mul: 0.3)}.play
```

Still not happy? Is it because this is a chromatic scale and you want a diatonic scale? For that I'll refer you to *DegreeToKey*, a discussion of which is beyond the scope of this chapter. I know the last page or two was a bit to chew on so this is a good time to point out that music is complicated. It is one of the most difficult subjects you'll ever tackle. That is, for me, a lot of the appeal. It is pervasive and accessible (everyone has a style of music that they enjoy, and nearly everyone can sing), yet the details of how it works can short circuit even the sharpest intellect.

Other External Controls

There are other methods for linking an external control to a value in a patch. You can use *TabletX*, *TabletY*, and *TabletZ* if you have a Wacom input device. You can also link controls to an external MIDI control surface or keyboard. You can attach actions to the computer keyboard (*keystate*) or build a GUI with sliders and buttons. The code required for each of these methods is a bit too involved for this section of the text. For now I would like to focus on internal automatic controls.

Practice

The patch below uses only two *SinOsc* and one *LFNoise0*, but can generate very rich sounds because of the way they are linked together (frequency modulation, covered later).

There are four mouse controls and the arguments for the patch are taken from the mouse position. Use the mouse position to explore sounds. Try the four corners first, then some spots in the middle.

Try changing the high (12) and low ([1, 1]) ranges for the *MouseX*, which control the rate of change, or the high (1000) or low (30) for *MouseY*, which changes the range of pitches. The only difference between the third and fourth examples is the *exprand* (exponential random values) and *rrand* (linear random values). Notice that with the *rrand* it seems most of the pitches are high, while the *exprand* seems (to our "octave" ears) to space them evenly across the spectrum. This shows that we hear an exponential increase as being "even" and linear as being biased toward high values.

```
   10.11. Practice sci-fi computer

(
{
PMOsc.ar(
   LFNoise1.kr(
      MouseX.kr([1, 1], 12),
      mul: MouseY.kr(10, 1000),
      add: 1000),
   LFNoise0.kr(
      MouseX.kr([1, 1], 12),
      mul: MouseY.kr(30, 1000),
      add: 1000),
   MouseY.kr(0.1, 5.0),
   mul: 0.3)
}.play
)

// Can you hear the difference between these two?

{Mix.ar( SinOsc.ar({rrand(100.0, 64000.0)}.dup(50)))*0.05}.play
```

```
{Mix.ar( SinOsc.ar({exprand(100.0, 64000.0)}.dup(50)))*0.05}.play

// This does the same thing, but I've added a few lines
// that print out the frequencies in octave bands. Try
// changing the rrand to exprand

(
{
o = 100.0;
"Octave Band ".post; [o, o*2].postln;
Mix.ar(
    SinOsc.ar(
        // replace rrand with exprand
        {rrand(o, 6400)}.dup(50).sort.round(0.01).do(
            {arg i;
                if(i < (o*2),
                    {i.post; " ".post},
                    {o = o*2; "\n\n".post; "Octave band ".post;
                        [o, o*2].postln; i.post; " ".post;}
                    )
            })
    )
)*0.1}.play
)

(
// exponential random
{Mix.fill(12, // number of oscillators
    {arg i;
    Pan2.ar(SinOsc.ar(SinOsc.ar(
            freq: MouseX.kr(rrand(0.1, 5.0), rrand(3.0, 20.0)), // speed of vibrato
            mul: MouseY.kr(10, 50), // width of vibrato
            add: exprand(200, 5000)), // freq of exponential random oscillators
        mul: max(0, LFNoise0.kr(MouseX.kr(rrand(1, 6), rrand(6, 1))))), 1.0.rand2)
})*0.03
}.play
)


(
// linear random waves
{Mix.fill(12, // number of oscillators
    {arg i;
    Pan2.ar(SinOsc.ar(SinOsc.ar(
            freq: MouseX.kr(rrand(0.1, 5.0), rrand(3.0, 20.0)), // speed of vibrato
            mul: MouseY.kr(10, 50), // width of vibrato
            add: rrand(200, 5000)), // freq of linear random oscillators
        mul: max(0, LFNoise0.kr(MouseX.kr(rrand(1, 6), rrand(6, 1))))), 1.0.rand2)
})*0.03
}.play
)
```

10. Exercises

10.1. In the following patches, insert a MouseX.kr and MouseY.kr in place of any two static numbers. Swap these objects: SinOsc, LFTri, LFPulse, LFSaw, LFNoise0, LFNoise1. When your experiments result in something interesting, copy and paste the patch to a separate file and hand it in. You may also want to record sections for later use.

```
(
{
    SinOsc.ar(
        freq: SinOsc.ar(freq: 512, mul: 638,
            add: LFNoise0.kr(freq: [13, 12], mul: 500, add: 600
            )), mul: 0.6)}.play
)

(
{
var out, delay;
out = SinOsc.ar(freq:
        abs(LFNoise0.kr(freq: 0.5, mul: 600,
                add: SinOsc.kr(freq: 1.5, mul: 50, add: 500
                ))),
        mul: 0.1);
delay = CombN.ar(out, 3.0, [1.35, 0.7], 6);
Pan2.ar(out, 0) + delay
}.play
)

(
{
CombN.ar(
    SinOsc.ar(
        freq: LFNoise1.ar(freq: 4, mul: 24,
            add: LFSaw.ar(freq: [8,7.23], mul: 3, add: 80)
        ).midicps, mul: 0.04), 0.2, 0.2, 4)}.play
)
```

11 - Variables, Comments, Offset and Scale using Mul and Add

Variables and Comments

It is often useful in code to define and use your own terms and functions. Variables are used for this purpose. Remember that variable names can be anything but must begin with a lower case letter (no numbers) and they must be contiguous. It is tempting to use short cryptic names such as *tri, beg, pfun, freq*, or even single letters such as a, b, c. More descriptive names take more time but you will be glad you used them in the long run; *firstPitch*, *legalDurations*, *maximumAttack*, etc.

Variables are declared (identified to the program) with the syntax *var* followed by a list of variables separated by commas and terminated by a semicolon. Variables are assigned values (the value is stored in the memory location associated with the variable) using the "=" sign followed by the value you want to store and a semicolon. The variable (or more accurately, the value contained in the variable) can then be used throughout the program.

The scope of a variable is the function in which it is declared. A variable cannot be used outside that function. You can declare global variables using the tilde (~). These do not require the *var* syntax.

The second example below also introduces expressions. An expression is a line of code punctuated with a semicolon. It delineates the order of execution; basically "do this;" "then this;". The order of execution then is; the variables are declared, then three variables are assigned values, then the SinOsc is put together using those variables and played. Variables can be given a value at declaration, illustrated by the next patch.

```
   11.1.  Variable declaration, assignment, and comments

//First patch

{SinOsc.ar(LFNoise0.ar(10, mul: 400, add: 800), 0, 0.3)}.play

//First patch with variables

(
{

var freqRate, freqRange, lowValue;
freqRate = 10; //rate at which new values are chosen
freqRange = 1200; //the range of frequencies chosen
lowValue = 60; //the lowest frequency

SinOsc.ar(
   LFNoise0.ar(freqRate, freqRange/2, freqRange + lowValue),
   0, 0.3)
}.play
)
```

```
//More concise

(
{

var freqRate = 10, freqRange = 1200, lowValue = 60;

SinOsc.ar(
   LFNoise0.ar(freqRate, freqRange/2, freqRange + lowValue),
   0, 0.3)
}.play
)
```

One reason for using variables is clarification. Earlier this example required explanation for arguments in *LFNoise0*. But variables are self-documenting. By using descriptive names the programmer can show in the code not only where values are used but also what they mean.

Variables can also be used for consistency. One value, say a base frequency may be used hundreds of times in a program or patch. Using a variable ensures that when a change to that value is made, it affects all pertinent parts of the patch.

Another important use for variables is to link arguments. For example, you might want to reduce the volume of a bell sound when higher frequencies are played and increase the volume with lower frequencies. You might also want the cutoff for a filter to change in proportion to the fundamental frequency. You might want the decay rate of an envelope to decrease with higher frequencies (which is what happens on many instruments). Without a common variable you would have to change them all by hand. For example, here are two possible sets of parameters for one sound.

100 //low frequency
0.7 // amp
600 //filter cutoff
2.0 //decay rate

1200 //higher frequency
0.3 //lower amplitude
7200 //higher cutoff
0.1 //decay rate

With a single variable, they can be linked using expressions. When frequency changes, the other variables change accordingly.

var freq;

freq = 100 //frequency may change
amp = freq/1200 //amp changes as freq changes
*cut = freq*600 //cutoff changes as freq changes*
rate = 100/freq //decay changes as freq changes

Here is an example showing two parameters that are linked using a variable. The second SinOsc is controlling the amplitude of the first. The idea of the patch is that the frequency of the flashing is related to the frequency of the pitch we hear; higher pitches will have a faster flashing. But the first three do not use a variable, so they have to be explicitly indicated. In the fourth version, you only change one value and the variable links that change to both parameters. And finally, a group of these mixed down.

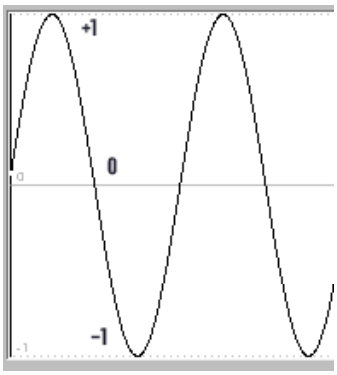
```
   11.2.  Variable declaration, assignment, and comments

{SinOsc.ar(400, mul: max(0, SinOsc.ar(4)))}.play

{SinOsc.ar(100, mul: max(0, SinOsc.ar(1)))}.play

{SinOsc.ar(1000, mul: max(0, SinOsc.ar(10)))}.play

(
   {
      var frequency = 1000; // change this single value
      SinOsc.ar(frequency, mul: max(0, SinOsc.ar(frequency/100)))
   }.play
)

(
   {
   Mix.ar(
      {
         var frequency;
         frequency = exprand(50, 1000);
         Pan2.ar(
            SinOsc.ar(frequency, mul: max(0, SinOsc.ar(frequency/100))),
            1.0.rand2
         )
      }.dup(5)
   )*0.5
   }.play
)
```

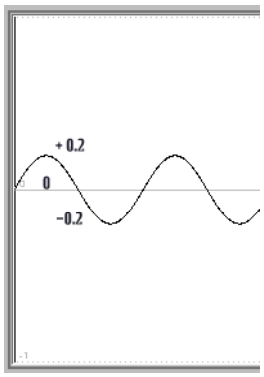
Offset and Scale using Mul and Add

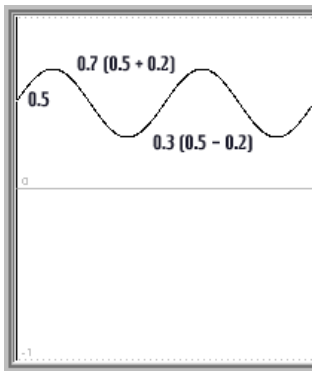
Most Ugens generate values either between 1 and -1 or 0 and 1. Earlier I used the mul argument to change the amplitude of a wave from +/-1 to +/- 0.5, +/- 0.3, and so on. The result was a change in amplitude or size of the wave. The *mul* argument multiplies or scales the wave. A larger scale means a louder sound, smaller scale quieter.

The *add* argument offsets the output of a Ugen. It changes the center from 0 to whatever the *add* value is. A *mul* of 0.2 would scale the range so that it no longer produced values from +/–1 but rather to values ranging +/–0.2, with a center value of 0. What would be the final result if that scaled output were then offset by 0.5? The middle value would become 0.5 (0 +

0.5), the bottom would be 0.3 (-0.2 + 0.5) and the top would be 0.7 (0.5 + 0.2). Here are three graphs illustrating how the *add* and *mul* arguments change a *SinOsc* wave.



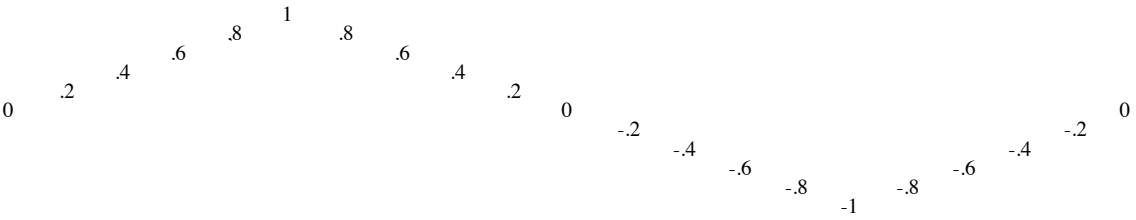




a) default *SinOsc* b) *SinOsc* scaled to 0.2 c) scaled to 0.2 offset to 0.5

Why would you want to offset a wave? Earlier we learned that values above 1.0 are inappropriate for amplitude. But you may have noticed the *LFNoise* has a *mul* value of 400. It normally generates values between –1 and 1. What happens to those values if they are multiplied, or scaled by 400? They change to +/-400. How could values that large be used in a patch? Maybe frequency?

Remember from our mul example earlier that the SinOsc just spits out a series of numbers. Those numbers can be modified with mathematical expressions. Here is an array of numbers roughly describing a triangle wave (with a low sampling rate):



Or

[0, 0.2, 0.4, 0.6, 0.8, 1, 0.8, 0.6, 0.4, 0.2, 0, -0.2, -0.4, -0.6, -0.8, -1, -0.8, -0.6, -0.4, -0.2, 0]

Notice the values go from 0, to 1, to 0, to -1, then 0. This is similar to the series of numbers a *LFTri* generates. If those numbers were all multiplied by 500 the result would be: [0, 20, 40, 60, 80, 100, 80, 60, 40, 0, -20, -40, -60, -80, -100, -80, -60, -40, -20, 0]. As a matter of fact, we can let SC do the math for us:

```
    11.3.  Offset and scale

// scaling an array
[0, 0.2, 0.4, 0.6, 0.8, 1, 0.8, 0.6, 0.4, 0.2, 0,
    -0.2, -0.4, -0.6, -0.8, -1, -0.8, -0.6, -0.4, -0.2, 0] * 100
```

```
// offsetting and scaling an array

[0, 0.2, 0.4, 0.6, 0.8, 1, 0.8, 0.6, 0.4, 0.2, 0,
    -0.2, -0.4, -0.6, -0.8, -1, -0.8, -0.6, -0.4, -0.2, 0] * 100 + 600
```

It has the same shape and proportions, but larger values due to the higher scale. If all of those values were offset by adding 600 the result would be: [600, 620, 640, 660, 680, 700, 680, 660, 640, 600, 580, 560, 540, 520, 500, 520, 540, 560, 580, 600]

Why offset and scale? This final array of numbers can't be used as an oscillator, because our speakers only accept values between -1 and 1. But they would be correct for the frequency input for another Ugen, since they have all been scaled and offset to values that are appropriate for pitch. That is one example of how *mul* and *add* are used.

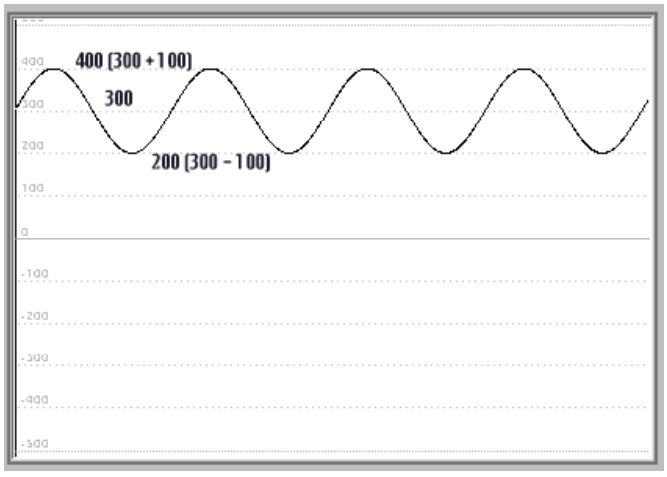
Scale and offset can be confusing because the resulting range of the Ugen output is different from the actual offset and scale. You use an offset of 0.5 and a scale of 0.2, but the range of values ends up being 0.3 to 0.7.

The line of code below shows an offset of 300 and scale of 100. But the *range* is 200 to 400. In this example a *SinOsc* is being used as the *freq* argument for another *SinOsc*. To understand how it is being used, first calculate what values are being returned by this inner *SinOsc* with a *mul* of 100 and an *add* of 300. Rather than the default of 1 to -1, it is scaled to 100 and -100. The offset makes the 100 400, and the -100 200. So the *range* is 200 to 400. With a *freq* of 4 this Ugen will move between 200 and 400 four times per second. How will that affect the *freq* of the outer *SinOsc*?

The term for this effect is voltage control, which is discussed in more depth later.

```
   11.4.  Offset and scale with mul and add

{SinOsc.ar(SinOsc.ar(4, 100, 300), 0, 0.3)}.play
```

Here is a one second graph of the *SinOsc.ar(4, 100, 300)*.



offset (*add*): 300, scale (*mul*): 100, actual result: 200 to 400

Offset and scale can be even more confusing because some ugens, such as an *Env* and *EnvGen*, or *LFPulse*, have a default range of 0 to 1 not +/–1. Even more confusing, Max/MSP has a few objects whose default values are 0.5 and -0.5 (the reasoning of which escapes me). So the affects of a scale and offset are different (though more straightforward). An *EnvGen* that is scaled by 200 and offset by 100 will have final values of 100 to 300. If this is confusing read this section a few more times. You can't just glance over it and move on. The concepts of scale and offset are used over and over, not only in SC, but other synthesis programs.

Our students have found it useful to think of scaling in these terms:

For Ugens with a default range of +1 to -1:
–The *add* is the center value of a range.
–The *mul* is the amount of deviation above and below the center.
–The lowest value is the *add* minus the *mul*.
–The highest value is the *add* plus the *mul*.
–The range is the *mul* times 2.

For Ugens with a default range of 0 to 1:
-The *add* is the low end of the range
-The *mul* is the amount of deviation above the low.
-The *mul* + *add* is the high end of the range
-The *mul* is the range

There is a message that can do the calculation for you. It is called *range* and converts one range of values to another. The arguments are the low and high range. In most cases it's just about as easy to do the calculation in your head.

```
   11.5.  Map a range
{
SinOsc.ar(
       SinOsc.kr(4).range(300, 500)
   )
}.play(s)
```

For an example of *mul* and *add* I repeat the first patch, but this time broken down so you can see the components. The first line plots the shape of an *LFNoise0* wave. We see that it is random step noise. Using the mouse, click on the plot window at several of the steps to confirm that they are all values between 1 and -1. The second one plays the patch. It should just sound like low frequency noise. The next line plots the same Ugen after it has been offset by 1000 and scaled by 400. It looks the same as the first plot, but this time the values are a much wider range. Confirm by clicking in the plot window on these steps that they are values between 600 (1000 – 400) and 1400 (1000 + 400). That range is not useful for amplitude, since our speaker can only move +/-1, but a range of 600 to 1400 could be used for frequency. The last version of the patch places the *LFNoise0* in the frequency position of the *SinOsc*. As it feeds values from 600 to 1400 to the *SinOsc*, that Ugen in turn uses those

values for frequency. How fast does the *LFNoise0* send a new value to the *SinOsc*? The *LFNoise0* has a frequency of 10 in the left channel and 15 in the right, so those are the rates that each channel moves to a new value, and new pitch.

11.6. First patch showing mul and add

```
{LFNoise0.ar(2000)}.plot

{LFNoise0.ar(2000)}.play

{LFNoise0.ar(2000, mul: 400, add: 1000)}.plot

{SinOsc.ar(LFNoise0.ar([10, 15], mul: 400, add: 800), 0, 0.3)}.play;
```

Practice

This practical example illustrates how variables are used, and how a ugen can be offset and scaled to values appropriate for a particular control. This patch builds a single sound with 20 sine waves. Without the variable *fundamental* you would have to enter the frequency of each wave. But this single variable allows you to generate a series of multiples based on one frequency. Try changing it to values between 10 and 300. Also try changing the number of partials. This patch also uses an argument *i*, which is similar to a variable. It counts each harmonic as it is created. To link the speed of each flashing harmonic with its number, try replacing the first argument of the LFNoise1 (6) to *(i + 1)*. The results will be high harmonics that have a fast flashing, and low harmonics that have slow flashing rates. Reverse this with *(partials –i)*, for fast lows and slow highs. This is taken an example, written by James, that was included with an early version of SC. It is one of many patches that caught my attention when I first toyed with SC.

The second example also illustrates how a single variable can be used to link two parameters together; frequency and decay. Each sine wave has an envelope (covered next chapter) with a decay time. A random frequency is chosen, and the decay for that frequency is calculated as a function of the frequency, such that high pitches have a short decay and low pitches have a longer decay.

11.7. Harmonic swimming from the examples folder, variable decay bells

```
// Remove the commented sections for variations.

(
// harmonic swimming
play({
    var fundamental, partials, out, offset;
    fundamental = 100;      // fundamental frequency
    partials = 20;       // number of partials per channel
    out = 0.0;       // start of oscil daisy chain
    offset = SinOsc.kr(1/60, 0.5pi, mul: 0.03, add: 0.01);
    partials.do({ arg i; var p;
```

```
        p = fundamental * (i+1);      // freq of partial
        // p = exprand(20, 3000); // variation
        out = FSinOsc.ar(
               p,
               0,
               max(0,           // clip negative amplitudes to zero
                  LFNoise1.kr(
                     6 + [4.0.rand2, 4.0.rand2],      // amplitude rate
                     0.02,               // amplitude scale
                     offset              // amplitude offset
                  )
               ),
               out
        )
    });
    out
// * EnvGen.kr(Env.perc(0, 4), Impulse.kr(1/4)) // variation
})
)


// Decay rates linked to frequency using a variable. (Low freq; long decay. High
freq; short decay.)

(
{
Mix.fill(15,
{
var freq;
freq = exprand(100, 3000);
    Pan2.ar(
        SinOsc.ar(
            freq * LFNoise1.kr(1/6, 0.4, 1),
            mul: EnvGen.kr(
                Env.perc(0, (freq**(-0.7))*100), Dust.kr(1/5))
        ), LFNoise1.kr(1/8)
    )
})*0.3
}.play
)
```

11. Exercises

11.1. Rewrite this patch replacing all numbers with variables.

```
{SinOsc.ar(
    freq: SinOsc.ar(512, mul: 673, add: LFNoise0.kr(7.5, 768, 600)),
    mul: 0.6
)}.play
```

11.2. Rewrite the patch above with variables so that the scale of the SinOsc is twice the frequency, and the offset of LFNoise0 is 200 greater than the scale.

11.3. How would you offset and scale an LFNoise0 so that it returned values between the range of C2 and C6 (C4 is 60)?

11.4. An envelope is generating values between 400 and 1000. What is the offset and scale?

11.5. As in previous chapters, make modifications to the practice examples and save any interesting variations in your collection. You can also record examples for later use in a composition.

12 - Voltage Control, LFO, Envelopes, Triggers, Gates, Reciprocals

You've now probably done enough editing to see the value of the shortcuts I mentioned earlier. Here they are again, as a reminder:

```
Com-,                       Go to line number
Com-/                       Make selected lines a comment
Opt-Com-/                   Remove comment marks on selected lines
Shift-Com-B                 Balance enclosures
Com-]                       Shift code right
Com-[                       Shift code left
Com-.                       Stop all playback
Shift-Com-/ (Com-?)         Open help file for selected item
Com-'                       Syntax colorize
Double click enclosure      Balance enclosures
Com-\                       Bring post window to front
Shift-Com-K                 Clear post window
Shift-Com-\                 Bring all windows to front
```

Early synthesizer modules had names such as VCO, VCA, and VCF. They are acronyms that stand for voltage controlled oscillator to control pitch, amplifier to control amplitude, and filter to control upper harmonic structure. The terms are a bit archaic, but the concepts are at the core of all synthesis systems and SC patches specifically. Control simply means changing a parameter. When you turn the volume of your stereo up you are manually controlling the amplitude parameter. When you press your finger down on the string of a guitar you manually control the length of the string and therefore the frequency. The vision of electronic music pioneers was the precision, accuracy, speed, and flexibility one could achieve using electricity to control these parameters of sound. They realized that the limitations inherent in humans and many instruments (how high we can sing, how fast we can play, how loud a piano can play) could be surpassed with electrically generated and controlled sound. They were right, and today there are virtually no limits to music generated on machines. In fact, they can easily exceed our ability to perceive.

Voltage control is where one ugen is used to control a parameter or argument of another. We saw it in the first patch. The *LFNoise0* generated values at a given rate and those values were used as the frequency for the *SinOsc*. When we used a *MouseX* to control frequency it was manual control. Using an *LFNoise0* is voltage or mechanical control. Nearly every object in SC can be "voltage controlled." Some modules allow you to control just one aspect, but most components allow control over all three; frequency, filter, or amp, so they act as a combination VCO, VCA, and VCF. The following examples will focus on pitch, or VCO.

We've already seen examples of VCO, VCA, and VCFs in previous patches, but we didn't call them that. Now that we know what they are, let's look at three examples. From the previous chapter we saw how an *LFNoise0*, if scaled correctly, would produce numbers in a range that could be used for pitch. Similarly I can scale and offset it to produce numbers useful for amplitude or timbre. The first patch below shows our original patch with the *LFNoise0* controlling frequency. Next, the same *LFNoise0* is scaled by 0.5 and offset by 0.5,

resulting in values between 0 and 1. This ugen is then placed not in the slot for frequency, but for *mul*, which in this case controls amplitude. Finally, the same *LFNoise0* is placed in the correct position for *knumharmonics* in a *Blip*. This argument effects the number of harmonics, and therefore the timbre. The effective range of harmonics is 0 to 20, so *LFNoise0* is offset and scaled to generate values between 2 and 18.

```
12.1. VCO, VCA, VCF
```

```
{Blip.ar(LFNoise0.kr(10, 500, 700), 7, 0.9)}.play

{Blip.ar(300, 7, LFNoise0.kr(10, 0.5, 0.5))}.play

{Blip.ar(300, LFNoise0.kr(10, 8, 10), 0.9)}.play
```

One of the easiest controls to conceptualize is a *Line.kr*. The arguments are starting point, ending point, and duration. The example below shows a sine oscillator ugen with static values, then the same patch with the *Line.kr* ugens in place of the static *freq* and *mul* arguments. Note that the start and end values are appropriate for each control: 0 to 1 for amplitude and 200 to 2000 for frequency. The last example shortens the *Line* so that it sounds like an envelope, covered below.

```
12.2. Line.kr
```

```
{SinOsc.ar(freq: 400, mul: 0.7)}.play

{SinOsc.ar(freq: Line.kr(200, 2000, 10), mul: Line.kr(0.9, 0.2, 10))}.play

{SinOsc.ar(freq: Line.kr(800, 400, 0.5), mul: Line.kr(0.9, 0, 0.5))}.play
```

Vibrato

One simple application of voltage control is vibrato. Vibrato is a slight oscillation (convenient, since we have oscillators) in either the amplitude (in the case of voice) or pitch (in the case of stringed instruments). String musicians roll the finger forward and backward during vibrato. The actual pitch then moves between, say 435 and 445, essentially moving in and out of tune about 5 times per second. In some previous *SinOsc* patches the frequency argument was a static value (such as 440). For a vibrato we need to replace the static value with some function or ugen that will change over time smoothly between 435 and 445. The shape of a vibrato is really like a sine wave: it moves back and forth between two values at a periodic and predictable rate. Could we use a slow moving *SinOsc*, scaled and offset as the input or *freq* argument for another *SinOsc*? Draw a sine wave with 0 in the center, 1 at the top and –1 at the bottom. Change the 0 to 440, the 1 to 445 and the –1 to 435. Can you calculate what the scale and offset need to be to achieve those values?

The answer is scale of 5 and offset of 440. Remember that with +/-1 style ugens the offset is the center value and the scale is the deviation. So 440 is the center value, 5 is the deviation.

There is one other value we need to consider, the speed of the vibrato. What determines the speed of the vibrato? The frequency of the control source *SinOsc*. How often should it move between 435 and 445? About 5 times per second should make a natural vibrato. So the frequency of the vibrato *SinOsc* should be 5 Hz. The vibrato section of the patch then would look like this. I'll use keyword arguments for clarity:

```
   12.3.  SinOsc as vibrato

SinOsc.ar(freq: 5, mul: 5, add: 440)
```

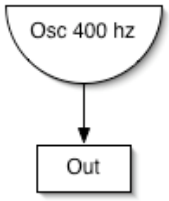
In the patch below the *SinOsc* with a frequency of 5 is stored in the variable "vibrato." If we were to listen to just the vibrato part of the patch we wouldn't hear anything because 5 is not a frequency what we hear as pitch, and 450 is a value too high for the hardware to convert to sound. But when it is placed in the frequency argument for the *SinOsc* then that outer Ugen actually produces audio we can hear.

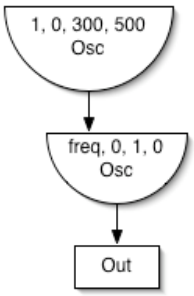
```
   12.4.  Vibrato

(
{
   var vibrato;
   vibrato = SinOsc.ar(freq: 5, mul: 5, add: 440);
   SinOsc.ar(vibrato, mul: 0.5)
}.play
)
```

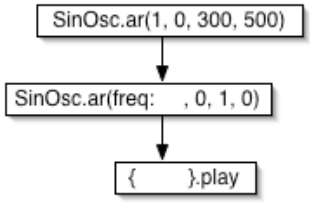
Experiment with each of the controls in the vibrato *SinOsc* above to see how they affect the sound. Which value affects the speed of the vibrato? Which value affects the depth? This is a fairly believable vibrato, if a bit crude, manufactured, or flat; like a lot of early electronic music. Real musicians don't dive right into a vibrato; they usually start with a pure tone then gradually increase it. Which values would you change, and what values would you use to make the vibrato more natural?

Block Diagrams

Block diagrams are used to illustrate the routing and connections of components in a patch. There are several styles currently in use, but all use boxes with inputs and outputs, then lines or arrows showing how they are connected together. They were used with early electronic patches to help visualize the connections of electronic components. I always feel like I should cover them, but every time I start writing a section on block diagrams I wonder if there's any point, because code based synthesis as easy to read as a diagram. Take for example a simple sine oscillator. It might be diagramed like the first example below. If we expanded that to include all the arguments for a *SinOsc* and added a control oscillator, it would be diagramed as in the second example. It may be more visual, but if you think of the enclosures of the coded version as boxes (as in the third example), it’s not much different.







{SinOsc.ar}.play {SinOsc.ar(SinOsc.ar(1, 0, 300, 500), 0, 1, 0)}.play

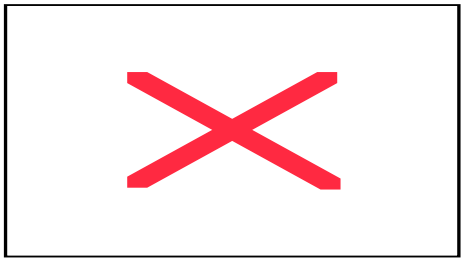
But for the visually oriented, I'll add some diagrams in this chapter.

Theremin

The Theremin was one of the first electronic instruments. The sounds it created were so "alien" that it was often used in sci-fi sound tracks (and, notably, the Beach Boys hit "Good Vibrations"). I include it here as an application of vibrato, because the two characteristics we most identify with this instrument are the glissando between pitches (a continuum rather than discrete pitches) and a solid wide vibrato. The vibrato was not the result of a voltage control (as in the these examples), but generated by the performers hand. Perhaps they felt the need for some added dimension to compensate for the pure, plastic tone of a sine wave.

Below is a virtual Theremin with controls connected to mouse position: up and down for pitch, left and right for amplitude. The vibrato is activated using a voltage control oscillator, or VCO. You can practically see the spaceship landing.

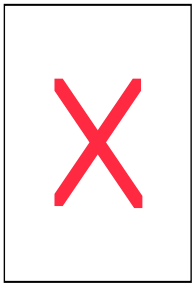
```
    12.5.  Theremin

(
{
var vibrato;
vibrato = SinOsc.kr(6, mul: 0.02, add: 1);
    SinOsc.ar(
        freq: MouseY.kr(3200, 200, lag: 0.5, warp: 1) *
        vibrato, //Vibrato
        mul: abs(MouseX.kr(0.02, 1)) //Amplitude
    )
}.play
)
```



The *mul* argument of the control *SinOsc* will change the depth of the vibrato. If it is set to 0 there is no vibrato, if it is set to 5 it is at a good depth. But both values are static. They don't change with time as natural vibratos do. In order for the vibrato depth to change over time we need to replace the static 5 with another time variant control.

There are countless Ugens that will work. For now we can use another *Line.kr*, (see also *XLine.kr*). Here is where it would fit into the patch (this time with variables):

```
    12.6.  Better vibrato

(
//Vibrato
{
var depthChange, vibrato;
depthChange = Line.kr(0, 5, 3);
vibrato = SinOsc.ar(freq: 5,  mul: depthChange, add: 440);
SinOsc.ar(
    vibrato,
    mul: 0.5)
}.play
)
```



Would it be more natural to control the speed (frequency) of the vibrato over time rather than the depth? What would you change to get such an effect? Could you control both? Could you also control the overall pitch (i.e. the add value), moving between, say, 300 to 900 using either a line or another *SinOsc*?

For those of use who learned on vintage patch synthesizers, this is the moment where we feel a bit giddy at the unlimited number of combinations available in SC. Even on current synthesizers you rarely have more than two or three *Line* style ugens to create an effect. SC, while limited to processor capacity, is not limited in terms of ugens. I could replace every static value in this patch with *Line* ugens (I see 5). Then I could replace values within those ugens with additional *Lines*. I could have literally thousands of *Line* ugens controlling some aspect of the patch. Don't believe me? Here it is:

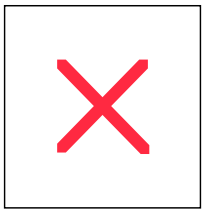
The vibrato *SinOsc* and the *Line.kr* are both "voltage" controls in the classic sense. That is they are controlling a parameter (frequency) of another ugen. We tried to recreate a natural sounding vibrato. A violin will rarely exceed a speed of 10 times per second or a depth beyond 40. But with synthesis, it is easy, even inviting, to go beyond those limits. Try increasing the *mul* and the *freq* arguments to values beyond what you would expect from a natural vibrato: 33, 100, 1200, etc., for *freq*, and the same for *mul*. Try changing the *add* to other pitches. Try replacing the *add* with another control; a *SinOsc* or a *Line*. Any static value can be replaced with a control. Try using controls other than *Line*. (Calculate the offset and scale for each carefully so as not to create negative values.) The sounds generated by these excessive values are unique to synthesis and the electronic age. I'm sure you may be accustomed to them, because they are so pervasive (a testament to the vision of early crackpot composers), but imagine hearing them for the first time.

Of the two *SinOsc* objects one is generating frequencies that we hear. The other is generating frequencies below audio range (e.g. 10 Hz, 5 Hz). We don't actually hear that sound, but we do hear the affect it has on the audio range oscillator. Many synthesizers use the term LFO or Low Frequency Control to describe such an effect. Nearly any oscillator in SC can be used at LFO rates. Here are several examples using different wave shapes. Experiment with the frequency, scale, and offset for each LFO. Make sure the offset (*add*) is greater than the scale

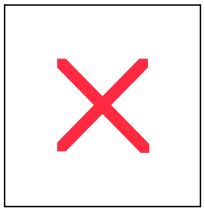
(*mul*). If the scale is greater you'll get negative values. Don't exceed 20 Hz for the frequency of these controls (lest you stumble onto something interesting). We'll cover that later.

12.7. Other LFO controls

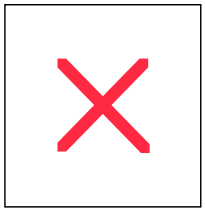
```
(
{ //SinOsc (sci-fi)
var lfo;
lfo = SinOsc.ar(freq: 10,  mul: 100, add: 400);
SinOsc.ar(lfo, mul: 0.5)
}.play
)
```



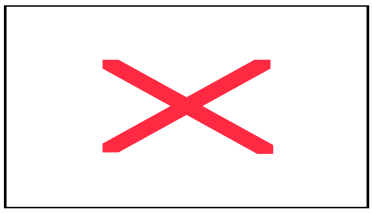
```
(
{ //Pulse (phone ring)
var lfo;
lfo = LFPulse.ar(freq: 15, mul: 200, add: 1000);
SinOsc.ar(lfo, mul: 0.5)
}.play
)
```



```
(
{ //Saw
var lfo;
lfo = LFSaw.ar(freq: 2, mul: -100, add: 600);
SinOsc.ar(lfo, mul: 0.5)
}.play
)
```



```
(
{ //Noise (computer)
var lfo;
lfo = LFNoise0.ar(freq: [28, 27], mul: 1000, add: 2000);
SinOsc.ar(lfo, mul: 0.5)
}.play
)
```



```
(
{ //Noise (manic birds)
var lfo;
lfo = LFNoise1.ar(freq: [28, 27], mul: 400, add: 2000);
SinOsc.ar(lfo, mul: 0.5)
}.play
)
```

Envelopes

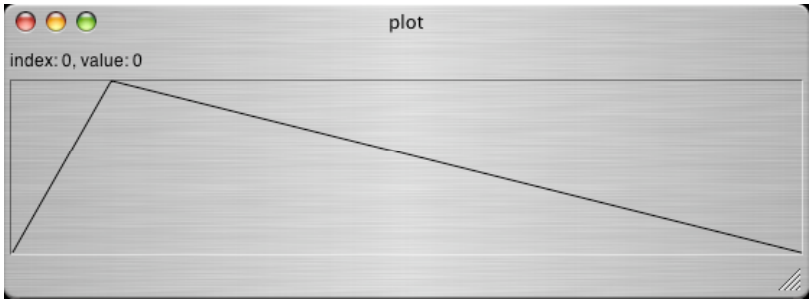
The *Line* ugen in the previous patch is a type of envelope. Envelopes[25] describe a single event (as opposed to periodically repeating events) that changes over time. Typically they are used to control amplitude (a VCA), but they can be applied to any aspect of sound such as frequency, or as in the last patch, the depth of a vibrato.

An envelope in SC has an additional function: when it has ended it can signal the server to stop playing that ugen and free up the CPU. Without this feature the events in a composition would add up and eventually max out the CPU.

We all know the amplitude of a musical event decreases over time (this is called decay), but there is also an attack, which is the beginning of a sound. A marimba has a sharp attack. An organ has a softer or long attack. A violin may have a short or a very long gradual attack depending on how the bow is used. A piano has a sharp attack, but not as sharp as a wood block. The difference may be only a few milliseconds, but it is easy to distinguish between very small variations in attack time. All presets on all synthesizers use envelopes to describe the change of volume over time.

There are two types of envelopes. Fixed duration and sustain envelopes. The sections of a fixed duration envelope will always be the same duration. Fixed envelopes usually have an attack and a release. The duration of a sustain envelope will change depending on the length of a gate, e.g. how long a key is held down. Percussion instruments such as cymbals, marimba, chimes, etc., have fixed duration envelopes. Once they are set in motion you can't control the duration of the decay. Organs, voice, violins, brass, etc., have sustaining envelopes, where each note continues to sound until the key is released or the musician stops blowing or bowing. At that point the pitch dies away.

The most common terms used to describe envelopes are attack, decay, sustain, and release or ADSR. Simpler envelopes may include only AR, or attack and release. Below is a graph showing an envelope with an attack and decay.



Attack Release

[25] The name comes from early composition techniques when sound samples were stored on sections of magnetic tape and kept in envelopes. When you wanted a sound you would select an envelope.

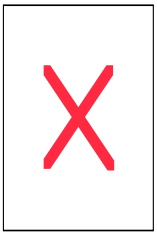
SC uses *EnvGen* and *Env* to create envelopes. *Env* describes the shape of the envelope and *EnvGen* generates that shape when supplied a trigger or gate.

Triggers, Gates, messages, ar (audio rate) and kr (control rate)

When you press the key of a consumer synthesizer the first piece of information that is sent to the processor is a trigger to begin that event. Below are examples of two triggers. The first is a periodic trigger, the second is a random trigger. The first two examples do not use the trigger to activate an envelope. Rather they simply play the triggers, which will sound like pops (not really intended for audio output). They both take one argument. For *Impulse* it is the number of pulses per second. For *Dust*, it is the average density of pulses.

Previously we saw that the *mul* argument in a *SinOsc* corresponds with volume. An *EnvGen* and *Env* can be used in conjunction with a trigger to supply events with an envelope for the *mul* or amplitude of the *SinOsc*, as in the third and fourth examples below.

```
   12.8.  Trigger and envelope

{Impulse.ar(4, mul: 0.5)}.play

{Dust.ar(5)}.play

(
{
   SinOsc.ar(
      440,
      mul: EnvGen.kr(Env.perc(0.001, 1.0), Impulse.kr(2))
   )
}.play
)

(
{
   SinOsc.ar(
      440,
      mul: EnvGen.kr(Env.perc(0.001, 0.3), Dust.kr(2))
   )
}.play
)
```

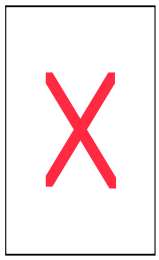


Try changing the arguments for *Impulse* and *Dust*. The arguments for *Env.perc* are attack and decay. Try changing both. Notice that the decay of a current event is cut off if the next trigger begins before it can finish its decay.

SinOsc uses the *ar* message while the ugens *Dust* and *Impulse* use *kr*. The *ar* stands for audio rate while *kr* is control rate. The *SinOsc* is generating actual audio so it needs to have a sample rate of 44.1k. But controls that are not heard as audio, but rather shape or trigger other ugens, do not need such high resolution and are more efficient if generated at a lower rate. A *SinOsc.kr* will generate a sine wave at 1/64th the rate of a *SinOsc.ar*.

Any positive value will be interpreted by *EnvGen* as a trigger, as illustrated by the patch below. A *SinOsc* moves repeatedly from positive to negative values. When the *SinOsc* moves from negative to positive the *EnvGen* is triggered. In the second example a *MouseX* is used to map the X axis of the screen to values between –0.1 and 0.1 (the center of the screen is 0). When the mouse crosses the middle of the screen a trigger is sent to *EnvGen*. The next example shows the same implementation using *MouseButton*. When the mouse is clicked, a trigger is sent. The last shows how you can do this with any keyboard key *KeyState*. This links the control to a map of the keyboard. The numbers that correspond with a key are a bit confusing, and not worth explaining here. Just use the list below the patch for key state numbers.

12.9. Trigger with MouseX



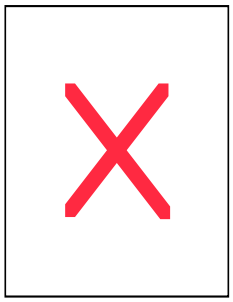
```
(
{
    SinOsc.ar(
        440,
        mul: EnvGen.kr(Env.perc(0.001, 0.3), SinOsc.ar(2))
    )
}.play
)

(
{
    SinOsc.ar(
        440,
        mul: EnvGen.kr(Env.perc(0.001, 0.3), MouseX.kr(-0.1, 0.1))
    )
}.play
)

(
{
    SinOsc.ar(
        440,
        mul: EnvGen.kr(Env.perc(0.001, 2), MouseButton.kr(0, 1))
    )
}.play
)

(
{
    SinOsc.ar( // type "j" to activate this envelope
        440,
        mul: EnvGen.kr(Env.perc(0.001, 2), KeyState.kr(38, 0, 1))
    )
}.play
)

// Here is a list of key state numbers:

1-18;  2-19;  3-20;  4-21;  5-23;  6-22;  7-26;  8-28;  0-29;  --27;  =-24
```

```
q-12;  w-13;  e-14;  r-15;  t-17;  y-16;  u-32;  i-34;  o-31;  p-35
a-0;   s-1;   d-2;   f-3;   g-5;   h-4;   j-38;  k-40;  l-37;  ;-41;  '-39
z-6;   x-7;   c-8;   v-9;   b-11;  n-45;  m-46;  ,- 43; .-47;  /-44;
```

The ugen *TRand* responds to a trigger, returning a random value within a given range. It can be used as a control source for the frequency argument of a SinOsc.

The example below uses a variable to link several arguments together. The *TRand* is generating a new pitch at every *trigger*. The envelope is also being generated at every trigger, so they correspond. Also, the decay rate (second argument of *Env.perc*) is proportional to the trigger speed, in this case, 1/2. This is done so that each triggered event will decay before the next is started. If the decay is longer than the duration of the trigger, then the events will overlap. This is what happens in real life, but I've made this adjustment to illustrate an application of variables.

Try changing the trigger speed and the attack rate. Notice that the decay is longer when the trigger is slower. How would you link the duration of the attack to the trigger speed? Can you expand it to two channels such that the left channel has lower notes and a faster rate, the right higher notes and a slower rate?



```
    12.10. Envelope with trigger

(
{
var triggerSpeed, trigger;

triggerSpeed = 8;
trigger = Impulse.kr(triggerSpeed);

SinOsc.ar(
    TRand.kr(100, 2000, trigger),
    mul: EnvGen.kr(
        Env.perc(0.001, 1/triggerSpeed),
        trigger
    )
)
}.play
)
```

Duration and Frequency [New]

An event or series of events can be expressed either as duration (how many seconds each event lasts) or as frequency (the number of events in a given period, e.g. a second). In the patch above the trigger speed is a frequency, that is to say it is expressed as a number of events per second. The decay argument in the *Env.perc* however is a duration, that is how long each event lasts. But in this patch the two are connected; the duration must last as long as each trigger. And each new trigger must occur at the end of each duration. This is common

in patches, so it is important to understand how they relate, how to express frequencies that are slower than one time per second or durations that are shorter then a second, and how to convert one to the other. First we have to understand how to express either as a ratio.

In the case of frequency the number of events is the numerator and the time (usually in seconds) is the denominator. For example, 40 Hz is 40 events in 1 second, or 40/1. When a frequency is lower than one time per second you can express it as a decimal, such as 0.1. But you can also continue to use the numerator and denominator method, increasing the denominator (time) rather than decrease the number of events. For example, 2 Hz is 2 events in 1 second, or 2/1. 1 Hz is 1 event in 1 second or 1/1, and one event in 5 seconds is 1/5, therefore 0.2 Hz (0.2 times in 1 second).

For duration, time is the numerator and the number of events is the denominator. An event that lasts 5 seconds can be expressed as 5 seconds for 1 event, or 5/1. When duration is lower than a second you can do the same: 5/1 is 5 seconds for one event, 1/1 is 1 second for one event, 1/2 is one second for two events, each event lasting ½ second (0.5).

Duration and frequency then are reciprocals. To convert between the two simply swap numerator and denominator. Take for example the frequency of 40 Hz. What is the duration of each wave? There are 40 waves in 1 second, or 40/1. The duration of each is the reciprocal; 1/40 (1 second for 40 events). The duration then of each individual event is 1/40th of a second.

To convert a duration to a frequency do the same. What, for example, is the frequency of an event that has a 5 second duration? It takes 5 seconds for 1 event, so that is 1 event in 5 seconds, or a frequency of 1/5 (0.2).

Since SC understands both decimal and fractional notation my students have found it easier to always use fractional notation and let SC do the math. So rather than set the frequency of an LFO to 0.1, just enter 1/10 (one event in ten seconds). 3/5 is 3 events in 5 seconds, 35/41 is 35 events in 41 seconds. If the denominator is 1 (15/1 or fifteen events in one second), you just drop the 1 and use 15.

Synchronized LFO Control

Expressing frequencies and durations as fractions facilitates synchronization of LFO frequencies; something that is very difficult on vintage synths. Imagine, for example, six pitches of a chord, each with a trigger, attack, decay, and duration. If the first has a trigger rate of 2/1 (2 in one second), the next has 3/1, then 4/1, 5/1, etc., then they will all be synched up every second. Likewise, triggers of 3/15, 4/15, 7/15, 11/15, etc., will all be in synch every 15 seconds.

The same is true for frequency. If 10 *SinOsc* ugens have related frequencies, expressed as ratios, they will be in synch periodically at the interval expressed by the denominator. That

sentence is a mouthful, so here is an example. Five *SinOsc's* with frequencies of 4/10, 3/10, 6/10, 7/10, will fall back into synch every ten seconds[26].

12.11. Synchronized LFOs and Triggers[27]

```
(
{
SinOsc.ar(SinOsc.ar(4/10, mul: 100, add: 1000), mul: 0.1) +
SinOsc.ar(SinOsc.ar(2/10, mul: 100, add: 1000), mul: 0.1) +
SinOsc.ar(SinOsc.ar(5/10, mul: 100, add: 1000), mul: 0.1)
}.play
)


(
{
var scale = 300, offset = 500;
SinOsc.ar(SinOsc.ar(4/3, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(7/3, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(2/3, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(8/3, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(6/3, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(5/3, mul: scale, add: offset), mul: 0.1)
}.play
)


(
{
var scale = 600, offset = 1000, synch = 10;
SinOsc.ar(SinOsc.ar(4/synch, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(7/synch, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(2/synch, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(8/synch, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(6/synch, mul: scale, add: offset), mul: 0.1) +
SinOsc.ar(SinOsc.ar(5/synch, mul: scale, add: offset), mul: 0.1)
}.play
)

( // synchronized triggers
{
var synch = 5;
SinOsc.ar(100, mul: EnvGen.kr(Env.perc(0, 1), Impulse.kr(3/synch))) +
SinOsc.ar(300, mul: EnvGen.kr(Env.perc(0, 1), Impulse.kr(7/synch))) +
SinOsc.ar(500, mul: EnvGen.kr(Env.perc(0, 1), Impulse.kr(5/synch))) +
SinOsc.ar(700, mul: EnvGen.kr(Env.perc(0, 1), Impulse.kr(2/synch))) +
```

---

[26] They may appear to synch up every 5 seconds. This is because some of the sine waves will be entering their downward phase. But they are truly in synch every 10 seconds.

[27] These are examples of additive synthesis, discussed later.

```
SinOsc.ar(900, mul: EnvGen.kr(Env.perc(0, 1), Impulse.kr(9/synch))) +
SinOsc.ar(1100, mul: EnvGen.kr(Env.perc(0, 1), Impulse.kr(6/synch))) +
SinOsc.ar(1300, mul: EnvGen.kr(Env.perc(0, 1), Impulse.kr(1/synch))) * 0.1
}.play
)
```

Frequency and Duration Linked

To link duration (such as attack and decay) with frequency in a patch, assign variables to each at the beginning of the program.

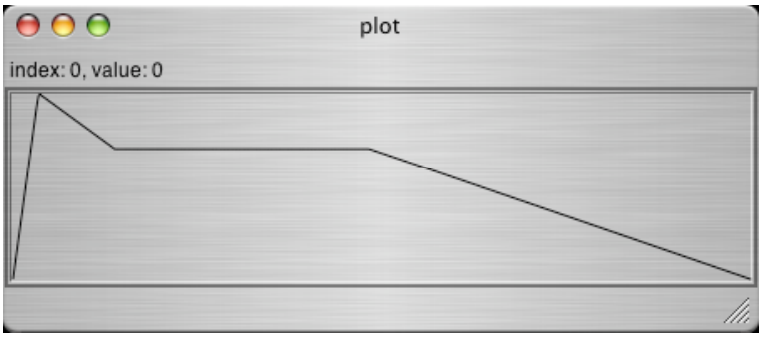
```
   12.12. Duration, attack, decay

var dur, att, dec, trigFreq;

dur = 10; // ten seconds long
att = dur*0.1;
dec = dur*0.9;
trigFreq = 1/dur;

or

freq = 10; // ten in one second
att = 1/freq*0.1;
dec = 1/freq*0.9;
duration = 1/feq;
```

Gates

A gate is a trigger and duration. (A trigger is a gate with 0 duration.) Triggers are used for fixed length envelopes. A gate can be used for both fixed and sustaining envelopes, but it is necessary for sustained envelopes. The duration of the gate defines sustain time.

An organ is a good example of a gated envelope. The "gate" is defined by pressing, holding, and releasing the keys. When you press the key it takes a little time for the pipe to speak. It may decay slightly while still holding the key, but eventually settles at a sustain level. It will remain at that level as long as you hold down the key. When you release the key it may take some time for the sound to die away.

Env.adsr is a gated envelope. It remains "on" for the duration of the gate. The arguments are attack, decay, sustain and release. The attack is the length of the initial rise to 1.0. It begins at the same time as the gate. The decay is the length of time it takes to fall to the sustain phase. It is in the middle of the gate. Note that sustain is not the length of sustain, but the level at which the sustain portion remains until the gate is closed. The release is the time it takes to move from the sustain level back to 0 once the gate is closed. Below is a graph of an adsr.



MouseX can be used as a trigger[28], as seen above, but it can also be a gate. As long as the *MouseX* value is in the positive range the gate will remain "open." While the gate is open the Env.adsr will attack, then decay to and remain at the sustain value for as long as the gate is open. When the *MouseX* falls below 0 the gate closes and the *Env.adsr* releases.

Try changing the values for *Env.adsr*.

The second example places the *EnvGen* and *Env* in *SinOsc* as the *freq* argument (with appropriate changes in scale and offset). Since it is easier to distinguish changes in pitch you can hear the shape of the envelope better.

```
   12.13. Envelope using a gate

(
{
   SinOsc.ar(440,
      mul: EnvGen.kr(
         //Envelope is attack, decay, sustain level, release
         Env.adsr(0.001, 0.3, 0.2, 0.1),
         MouseX.kr(-0.1, 0.1) //gate
      )
   )
}.play
)

(
{
   SinOsc.ar(
      400 + EnvGen.kr(
         Env.adsr(0.3, 0.4, 0.2, 1),
         MouseX.kr(-0.1, 0.1),
         1000
      ),
      mul: 0.5
   )
```

[28] You might be thinking we should use a MIDI keyboard to supply the gate, but I think it is valuable while learning the basics to work outside of that context. See the section on MIDIIn below for examples that use an external keyboard for gate and note values.

```
}.play
)
```

Finally, *LFNoise0*[29] is a random source that wanders between –1 and 1. It can be used to supply a random gate (random triggers and random sustain times). When it is positive the gate is open, when it falls below 0 the gate is closed.

```
   12.14. Envelope with LFNoise as gate

(
{
   SinOsc.ar(
      LFNoise0.kr(13, 1000, 1200),
      mul: EnvGen.kr(
         //Envelope is attack, decay, sustain level, release
         Env.adsr(0.001, 0.3, 0.2, 0.1),
         LFNoise0.kr(3) //gate
      )
   )
}.play
)
```

With even the most sophisticated synthesizers you rarely have envelopes with more than 3 or 4 break points (where the envelope changes direction). Real instruments are capable of much more complex variations in amplitude. SC allows for this level of complexity with the "new" message. The first two arguments are arrays. The first array contains levels for each break point, the second array contains durations between each break point. Note that there must be one more value in the levels array than there is in the times array. This patch uses a complex envelope to control frequency. The EnvGen creates values between 0 and 1, so it is scaled and offset using the * and + at the end. The graph below shows the shape of the envelope.

```
   12.15. Complex envelope

(
{
SinOsc.ar(
   EnvGen.kr(Env.new(
      [ 0, 0.5, 0, 0.3, 0.3, 0, 0.1, 0, 0.2, 0.3, 0.15, 0.5, 0.25 ],
      [1, 0.5, 0.5, 2, 0.7, 1, 0.3, 0.6, 0.5, 0.8, 0, 0.4])
   ) * 1000 + 200
)
}.play
)

// This will plot the array
```

[29] LFNoise1 can also be used. See the help file for an explanation of the differences.

```
[ 0, 0.5, 0, 0.3, 0.3, 0, 0.1, 0, 0.2, 0.3, 0.15, 0.5, 0.25 ].plot
```

Finally, below are examples of bells where the pitch or frequency for each event is chosen from a range. With many instruments the sustain of an event is short in higher frequencies and longer for lower notes. This patch demonstrates how to link the duration of each event with the frequency using a variable. There are a few ugens you haven't seen before. Just read the help files.

The Experimental Process

Now you have the essential tools for experimentation and discovery. Synthesis using SC is understanding the parameters (arguments) of a ugen in a patch, and experimenting with different kinds of controls. It is often reassuring to me that all synthesis can be broken down to controls for the three elements of sound: frequency, amplitude, timbre.

The last few chapters presented concepts in the order of my experimental process: I typically start with a very simple patch, maybe just one oscillator. Then I put together two or three linked controls. I replace and existing static value with a *MouseX* to try a range of values. Then I think about what other control sources I could use to automatically move through those values. I do the calculation for the offset and scale, then place it in the patch. My final step is to thicken the patch with duplication, stereo expansion, random values, and mixing. Feel free to do the same with the patches in this tutorial and the help and example files.

Practice, Bells

```
   12.16. Bells

(
//frequency linked to envelope length
  //high notes short, low long
{
var frequency;
Mix.ar(
   {
      frequency = rrand(100, 5000);
      Pan2.ar(
         SinOsc.ar(
            frequency,
            mul: EnvGen.kr(
               Env.perc(0.001, 500/frequency),
```

```
                Dust.kr(0.05),
                0.2
            )
        ),
        rrand(-1.0, 1.0)
    )
  }.dup(100)
)
}.play
)


(
//frequency linked to decay length
  //basically the same code but more compact
  //low notes short, high long
{var frequency;
Mix.ar({
       frequency = rrand(100, 3000);
       Pan2.ar(SinOsc.ar(frequency,
          mul: EnvGen.kr(Env.perc(0.001, frequency/1000),
             Dust.kr(0.05), 0.2)), rrand(-1.0, 1.0)) }.dup(100))}.play
)

(//high notes short, low long
{var frequency;
Mix.ar({
       frequency = rrand(100, 3000);
       Pan2.ar(SinOsc.ar(frequency,
          mul: EnvGen.kr(Env.perc(200/frequency, 0.0001),
             Dust.kr(0.05), 0.2)), rrand(-1.0, 1.0)) }.dup(100))}.play
)

(//low notes short, high long
{var frequency;
Mix.ar({
       frequency = rrand(100, 1000);
       Pan2.ar(SinOsc.ar(frequency,
          mul: EnvGen.kr(Env.perc(frequency/500, 0001),
             Dust.kr(0.05), 0.05)), rrand(-1.0, 1.0)) }.dup(100))}.play
)
```

12. Exercises

12.1. In this patch, what is the speed of the vibrato?
{SinOsc.ar(400 + SinOsc.ar(7, mul: 5))}.play

12.2. Rewrite the patch above with a MouseX and MouseY to control the depth and rate of vibrato.

12.3. Rewrite the patch above using an LFPulse with a slower rate and greater depth (maybe 100) rather than a SinOsc to control frequency deviation.

12.4. Rewrite the Theremin patch so that vibrato increases with amplitude.

12.5. Write a patch using a Pulse and control the width with a Line moving from 0.1 to 0.9 in 20 seconds.

12.6. Write a patch using a SinOsc and EnvGen. Use the envelope generator to control frequency of the SinOsc and trigger it with Dust.

12.7. Start with this patch: {SinOsc.ar(200)}.play. Replace the 200 with another SinOsc offset and scaled so that it controls pitch, moving between 400 and 800 3 times per second. Then insert another SinOsc to control the freq of that sine so that it moves between 10 and 20 once every 4 seconds. Then another to control the frequency of that sine so that it moves between once every 4 seconds (a frequency of 1/4) to 1 every 30 seconds (a frequency of 1/30), once every minute (a frequency of 1/60). Can you hear the affect of each control?

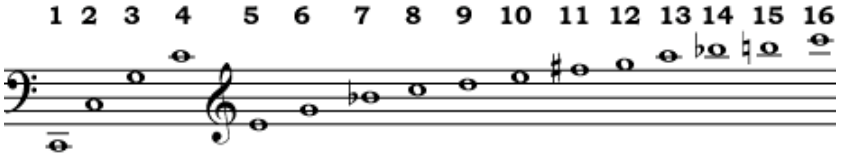
12.8. What is the duration of a single wave with a frequency of 500 Hz?

12.9. What is the frequency, in Hz (times per second), of two and a half minutes?

13 - Just and Equal Tempered Intervals, Multi-channel Expansion, Global Variables

As we discovered in a previous chapter, pure musical intervals are ratios. An octave is 2:1. That is to say an octave above A 440 is double, or 880. The octave above that is 1760. Interval inversions are reciprocals: an octave up is 2:1, so an octave down is 1:2. The intervals we use in western music come from the harmonic series, which are multiples of the fundamental. If the fundamental pitch (the pitch you hear) is 200 then the harmonic partials are 200, 400, 600, 800, 1000, and so on. The musical intervals they produce above the fundamental (the fundamental is the unison, or the pitch itself) are an octave, fifth, octave, third, fifth, seventh, octave, second, as illustrated in the chart below.

Harmonic series



C2 is the bottom pitch on the chart. The first fifth we encounter is G3, the third harmonic. In order to calculate a G2 (the fifth just above C2) you would multiply that frequency by 3 (G3) then divide by 2 to move down an octave to G2, or 3/2. Let's say C2 is 60 Hz (to make the math easy), C3 would be 120, G3 180, G2 would be one half that: 60 * 3 / 2 = 90.

The fifth harmonic is the first major third in the series and is actually two octaves and a third (E4). To calculate a third above C2 (E2), first multiply by 5 to get E4, then divide by 2 for each octave down, a total of 4 for E2: 60 * 5 / 4 = 75.

There are a couple tricks to memorizing interval ratios. An octave is 2:1, fifth is 3:2, fourth is 4:3, third is 5:4, minor third is 6:5, second is 9:8. Notice that all the numerators are one greater than the denominators. So if you can remember the sequence octave, fifth, fourth, third, minor third, and second, you can usually reconstruct the ratios (remember 9:8 for a second).

Also, you can refer to the harmonic series above to locate intervals we use in western tuning along with their corresponding ratios. The interval between harmonic 1 and 2 is an octave (2:1), between 2 and 3 is a fifth (3:2), between 3 and 4 a fourth (4:3), 4 and 5 a major third (5:4), 5 and 6 a minor third (6:5)… but then the system breaks down. What is the interval between 6 and 7? 7 and 8? Why don't we use them? Should we use them? Between 8 and 9 is the interval we use for a second (9:8). The last two illustrate the half-step we use between 15 and 16 (16:15). Note also that the interval between harmonics 5 and 8 is a minor sixth (8:5). A major sixth (5:3) can be found between the 6^{th} and 10^{th} harmonics, a major seventh (15:8) between 8 and 15, and a minor seventh (9:5) between 9 and 5.

The patch below allows you to test different intervals. The r = 2/1 represents the ratio of the second frequency. If f = 400 and r = 2:1 ("two to one"), then f*r is 800, or 400 multiplied by 2 and divided by 1. Even though the division by 1 has no effect I express it this way so that you can try different ratios to see if they do indeed represent common intervals. Change the r value to 3/2 (a fifth), 4/3 (a fourth), 5/4, etc. Try 64/45 for a tritone.

```
   13.1.  Intervals

(
{
   f = 400; //fundamental
   r = 2/1; //ratio for second note in interval
   FSinOsc.ar([f, f*r], mul: 0.6)
}.scope(2)
)
```

The first argument for *FSinOsc* (fast sine oscillator) is not a single value but an array with variables. Remember that an array is a comma-separated list of values within brackets. When an array is used anywhere as an *argument* in a message SC expands the entire patch into two identical patches for each channel. It uses the first value of the array for the left channel and the second for the right. This wonderful feature is called multi-channel expansion. An array with more than two values will expand to more channels; as many as your hardware will handle. SC matches all the arrayed arguments, duplicating where necessary, to create parallel patches.

```
   13.2.  Multi-channel expansion

//This code

(
{SinOsc.ar(
   LFNoise0.ar([10, 12, 14, 6], 400, [800, 1000]),
   mul: [0.3, 0.5])}.scope(4)
)

//Becomes this in each channel:

(
//channel 1
{SinOsc.ar(
   LFNoise0.ar(10, 400, 800),
   mul: 0.3)}.scope(1)
)

(
//channel 2
{SinOsc.ar(
   LFNoise0.ar(12, 400, 1000),
   mul: 0.5)}.scope(1)
)
```

```
(
//channel 3
{SinOsc.ar(
   LFNoise0.ar(14, 400, 800),
   mul: 0.3)}.scope(1)
)

(
//channel 4
{SinOsc.ar(
   LFNoise0.ar(6, 400, 1000),
   mul: 0.5)}.scope(1)
)
```

Look at the interval patch once more. Because both frequencies are periodic, and are a mathematical ratio, the destructive and constructive points become an aggregate, and we "hear" the resulting pattern in the peaks and valleys. Here is the same example with a three channel arrayed expansion, one with the first frequency, one with the second, then the third with both added together. If you are playing back on two channels you will only hear 1 and 2. The third[30] is just for analysis (though it would sound the same). Change the *r = 2/1* to other ratios such as 3/2, 4/3, 5/4, 6/5, 9/8, etc.

```
   13.3.  Intervals

(
{
   f = 400;  r = 2/1;
   a = FSinOsc.ar(f, 0.3);
   b = FSinOsc.ar(f*r, 0.3);
[a, b, a+b]*0.3
}.scope(3, zoom: 4)
)
```

In the past few patches you may have noticed I was able to use variables without declaring them. The letters a-z are global variables, declared when SC launches, and are useful for quick experiments. Fair warning; you will regret using them instead of more meaningful names as your patch becomes more complex.

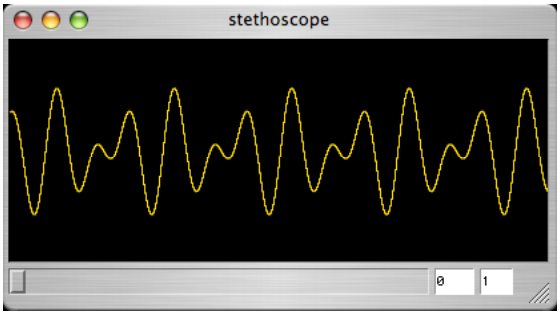
Also notice that the last line of code in the function is simply the array of variables. This last line is called the “return.” A function “returns” the results of the last line of code regardless of what happened in the body of the function, as illustrated below.

```
   13.4.  Function return: last line
```

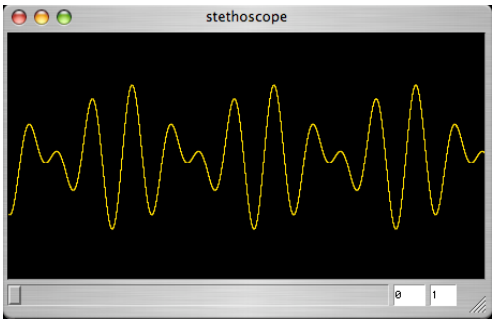
[30] I'm using bus 3 for convenience in scope. It is connected to the audio input on most machines. This may cause interference with the wave. If so, set your audio input to line in and make sure nothing is connected.

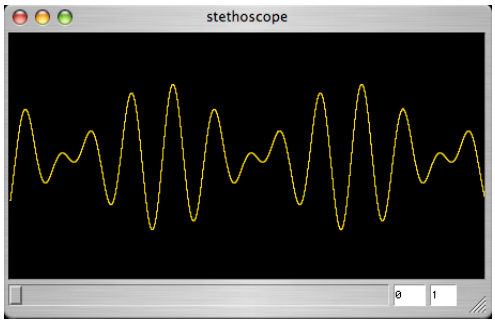
```
(
{ // This code will only play "c"
    a = FSinOsc.ar(200);
    b = FSinOsc.ar(1000);
    c = Pulse.ar(45);
    d = FSinOsc.ar(400);
c
}.scope(1)
)

(
    1 + 25;
    1000 * 13 / 4;
    1 + 1;
    7; // This last line is the only one that prints. The others are meaningless.
)
```

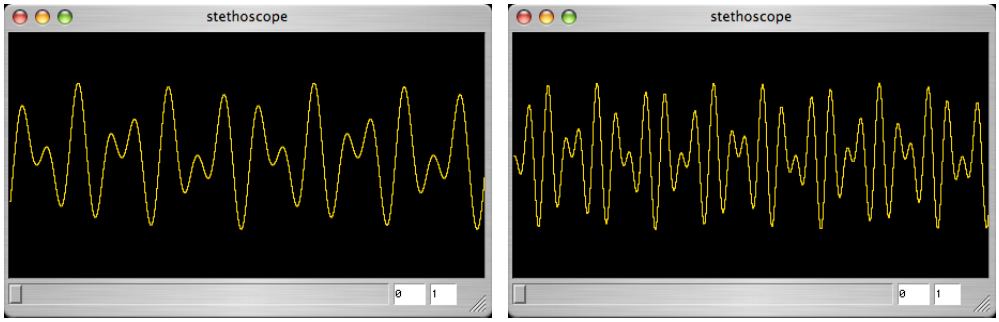
Here are some screen shots that plot the wave forms of various intervals. The first is a fifth over one tenth of a second.



There are lots of peaks and valleys in this example. The highest peaks and lowest valleys are where there is the greatest amount of constructive interference. The waves are adding together to push or pull farther. We hear the entire pattern as a fifth. Following are a fourth (4/3), third (5/4), minor sixth (8/5), and finally a tritone (64/45).







With lower ratios there is more constructive interference, so fewer peaks and valleys. With high ratios there is more deconstructive interference and more peaks and valleys. Even with complex peaks and valleys we still hear the pattern. Dissonance and consonance are directly related and can be defined as the number of peaks in a pattern for a given amount of time, or the amount of time between the peaks that define the pattern. More peaks and valleys; higher dissonance.

In each of the graphs above our eyes (brains) recognize the visual pattern, and likewise our ears pick up on the aural pattern that results from the constructive and destructive interference.

The next example illustrates this concept in real time. It allows you to move two pitches from low frequency to audio range. This way you can first hear the pattern as a set of clicks or pulses in low frequency. With a pulse or saw you hear a single click each period of the wave when the speaker pops back to its original position. (We hear the separate pulses as rhythmic patterns of 3/2, 4/3 etc.) Then as you move the mouse to the right, bringing the frequencies into audio range, you hear the resulting audio interval. First try running the line below to hear the difference between LFO and audio rates. Move the mouse all the way to the left for low frequency, to the right for audio frequency.

```
    13.5.  Audio frequencies

{Saw.ar(MouseX.kr(1, 1200, 1), 0.5)}.scope(1)
```

The patch below uses two frequencies defining an interval. I've set the ratio to 3:2 using *ratioNum* and *ratioDenum*. Change these to 2:1, 4:3, 5:4, etc. Try non-super particular ratios such as 8:3, 7:2. Remember that with higher numbers there will be fewer coincident constructive interference peaks and we will hear that as a more dissonant interval.

Listen to the pattern and look at the periods in the scope. Next move the mouse slowly to the right until the frequencies become pitches. Move all the way to the right to confirm that it is the interval you were trying for.

```
    13.6.  Ratios from LF to audio rate

(
```

```
{
var freq, ratioNum, ratioDenum; //declare two variables
ratioNum = 3; //assign numerator
ratioDenum = 2; //assign denominator
freq = MouseX.kr(1,440, warp: 1); //freq is mouse control
LFSaw.ar(
   [freq, freq*(ratioNum/ratioDenum)], 0,
   0.3)
}.scope(2)
)
```

Just vs. Equal Intervals

We now have a slight dilemma (or more choices, depending on your outlook). Do we use pure, just intervals? Or do we use equal temperament? In a previous chapter we used *midicps* to convert MIDI numbers to frequencies. But MIDI values are equal tempered. The ratios we've been experimenting with result in a Just tuning. Which do we use? Do we have to choose one or the other? Are they really that different?

Yes, they are really that different. Each half step in an equal tempered scale is 100 cents. MIDI numbers use one integer for each half step, so we can use a precision of two decimal points to describe the cents. The quarter tone between C4 and C#4, for example, is 60.5 (60 + 50 cents). The lines of code below show how to calculate a major scale using pure ratios, then convert those frequencies to MIDI values (which show the cents), which can then be compared to equal tempered values.

```
   13.7.  Equal Tempered compared to Pure Ratios

(261.6 * [1, 9/8, 5/4, 4/3, 3/2, 5/3, 16/15, 2/1]).round(0.01)

[261.6, 294.3, 327, 348.8, 392.4, 436, 279.04, 523.2].cpsmidi.round(0.01)
```

The results are 60, 62.04, 63.86, 64.98, 67.02, 68.84, 61.12, 72 for just, or pure ratios, compared to 60, 62, 64, 65, 67, 69, 70, and 72. The major third is 14 cents flat, and the major sixth is 16 cents flat.

Let's see how they sound using the low frequency to audio rate experiment. There are two very important differences in the example below. First, the *MouseX* no longer has an exponential warp. This is because MIDI numbers are linear. The frequencies are automatically converted to exponential values in the *midicps* message. The next is that the interval (expressed in half steps) is *added* to the value in the left speaker, not multiplied as it was with the ratio.

```
   13.8.  Ratios from LF to audio rate
```

```
(
{
var midiNum, interval;
interval = 7;
midiNum = MouseX.kr(0.1, 84);
LFSaw.ar(
    [midiNum.midicps, (midiNum + interval).midicps], 0,
    0.3)
}.scope(2)
)
```

When working with other equal tempered instruments (guitar, piano, MIDI keyboards), you should use MIDI values. When working with fretless strings or vocals, why not use just intonation? (Why has no one ever presented this choice to you before? Because before computers we haven't had instruments precise enough, or flexible enough, to allow the choice.) Or why not quarter tones, or mean tone, or Chinese Lu scale by Huai Nan Zi, Han era (P. Amiot 1780, Kurt Reinhard)?

If you tune to successive just intervals, you will encounter pitch drift. If you are adventurous, you should see this as a challenge, and delve into the world of free just intonation, in which the drift is accepted as correct.

For the examples below I jump farther ahead in terms of code. You will just have to suppress your need to understand each detail. This patch plays two simple tones, one in the left, the other in the right. The left channel is operating with free-just tuning where each interval is calculated using Pythagorean ratios without regard to any adjustments. The right channel is using equal tempered intervals. The Pythagorean comma will affect the left channel, and the frequencies will drift (from the equal counterpart in the right channel). They should be playing the "same" pitch, and if they were both set to the same tuning, they would. When they drift apart, this is not because they are making different choices, but because of the tuning, and because the Pythagorean tuning drifts. I've triple checked my calculations, and I'm sure you'll correct me if I'm wrong, but yes, they go out of tune that quickly, and that much.

The first example chooses random intervals, the second is just a scale, which illustrates a little better the pitch shift. When we hear the two together we can tell they are out of tune. But can you hear the drift if you listen to just the left channel?

Practice, Free-Just, and Equal-Tempered Tuning

```
   13.9.  Tuning

( //double click to select the entire example
SynthDef("PureTone",
{arg justFreq = 440, equalFreq = 440;
Out.ar(0, SinOsc.ar([justFreq, equalFreq], mul: 0.4)
       *EnvGen.kr(Env.perc(0, 1), doneAction:2));
}).load(s);
Task({
```

```
var jfreq = 440, efreq = 69, next = 6, equalInt, justInt;
equalInt = [-10, -8, -7, -5, -3, -1,
   0, 2, 4, 5, 7, 9, 11];
justInt = [9/16, 5/8, 2/3, 3/4, 5/6, 15/16, 1/1,
   9/8, 5/4, 4/3, 3/2, 5/3, 15/8];
   {
   [equalInt[next], justInt.at(next).round(0.01)].post;
   Synth("PureTone", [\justFreq, jfreq.round(0.01),
      \equalFreq, efreq.midicps.round(0.01)].postln);
   next = 13.rand;
   jfreq = jfreq*justInt.at(next);
   efreq = efreq + equalInt.at(next);
   if(jfreq < 100, {jfreq = jfreq*2; efreq = efreq + 12});
   if(jfreq > 1000, {jfreq = jfreq/2; efreq = efreq - 12});
   [0.125, 0.125, 0.124, 0.25, 0.5, 1].choose.wait
   }.loop;
}).play(SystemClock);
)

// Same example with just a scale.
( //double click to select the entire example
SynthDef("PureTone",
{arg justFreq = 440, equalFreq = 440;
Out.ar(0, SinOsc.ar([justFreq, equalFreq], mul: 0.4)
      *EnvGen.kr(Env.perc(0, 1), doneAction:2));
}).load(s);
Task({
var jfreq = 440, efreq = 69, next = 0, equalInt, justInt;
equalInt = [-12, 2, 2, 1, 2, 2, 3];
justInt = [1/2, 9/8, 9/8, 16/15, 9/8, 9/8, 6/5];
   {
   [equalInt.wrapAt(next), justInt.wrapAt(next).round(0.01)].post;
   Synth("PureTone", [\justFreq, jfreq.round(0.01),
      \equalFreq, efreq.midicps.round(0.01)].postln);
   next = next + 1;
   jfreq = jfreq*justInt.wrapAt(next);
   efreq = efreq + equalInt.wrapAt(next); 0.25.wait
   }.loop;
}).play(SystemClock);
)

// Free just intonation only
( //double click to select the entire example
SynthDef("PureTone",
{arg justFreq = 440, equalFreq = 440;
Out.ar(0, SinOsc.ar([justFreq, equalFreq], mul: 0.4)
      *EnvGen.kr(Env.perc(0, 1), doneAction:2));
}).load(s);
Task({
var jfreq = 440, efreq = 69, next = 0, equalInt, justInt;
equalInt = [-12, 2, 2, 1, 2, 2, 3];
justInt = [1/2, 9/8, 9/8, 16/15, 9/8, 9/8, 6/5];
   {
```

```
    [equalInt.wrapAt(next), justInt.wrapAt(next).round(0.01)].post;
    Synth("PureTone", [\justFreq, jfreq.round(0.01),
        \equalFreq, jfreq.round(0.01)].postln);
    next = next + 1;
    jfreq = jfreq*justInt.wrapAt(next);
    efreq = efreq + equalInt.wrapAt(next); 0.25.wait
    }.loop;
}).play(SystemClock);
)
```

13. Exercises

13.1. What is the 6th harmonic of a pitch at 200 Hz?

13.2. What is the phase of the sine wave in channel 4?
SinOsc.ar([10, 15, 20, 25], [0, 1.25], 500, 600)

13.3. What value is returned from this function?
{a = 10; b = 15; c = 25; a = c + b; b = b + a; b}

13.4. Arrange these frequencies from most consonant to most dissonant.
180:160, 450:320, 600:400, 1200:600

13.5. In the first practice example replace the code SinOsc.ar([justFreq, equalFreq]) with SinOsc.ar([justFreq, *justFreq*]), so that both channels play the same pitch, and there is no "correct" reference. Do you notice any drift? Play it for three or four trained (but not perfect pitch abled) musicians. Ask if they notice anything unusual. Is free just intonation a viable tuning system?

14 - Additive Synthesis, Random Numbers, CPU usage

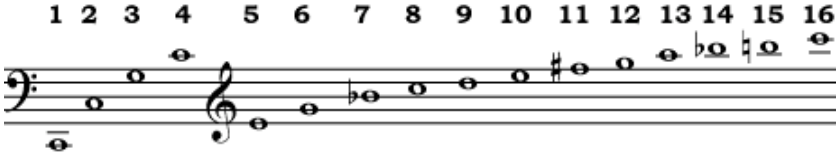
Why, after nearly 50 years of electronic music and instrument design, can we still tell if an instrument is synthesized? What most synthetic models lack are complexity and chaos. Even sampled instruments do not sound real because an exact duplication of a single note from any instrument is not a true representation of that instrument, rather a single instance of many possibilities. A single high resolution digital photo of a leaf duplicated hundreds of times would not look like a tree. All the leaves on a tree are different, and they change in real time. The same is true for a piano. Each key has its own harmonic fingerprint. Even the same key when struck successively will produce different partials depending on the position of the vibrating string from the previous note when it is struck a second time. Real instruments are complex and chaotic. Each note is unique. Though the general upper harmonic content remains constant, the small details change. SC is the first real-time synthesis package I've encountered that is capable of, even invites and encourages this level of complexity. We will start the theory of synthesis with the most intricate and computation intensive method (and most rewarding in terms of results); additive synthesis.

Harmonic Series and Wave Shape

In an earlier chapter we learned that the harmonic series is responsible for the character of a sound. The presence and strength of each harmonic defines this character. But how does a single string or vibrating body generate a series of related frequencies?

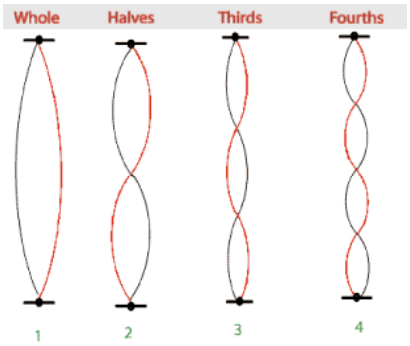
Density and tension being equal, short strings vibrate faster than long. A string that would normally vibrate at 100 Hz divided in half (e.g. by pressing your finger on the finger board at the half-way point) will vibrate twice as fast, or at 200 Hz. If you shorten it to one third its original length, by pressing your finger higher, it will vibrate three times as fast or 300 Hz. Shortening it to one fourth generates a frequency four times faster; 400 Hz. Dividing a string in such a way will produce harmonics. Harmonics are the foundation for scale degrees of western music. The illustration below shows the pitches that correspond with each upper harmonic. The lowest C is about 65 Hz, the next is 130, then 195, 260 (middle C), 325, 390, 455, 520, 585, 650, 715, 780, 845, 910, 975, 1040.

14.1. String Vibration and Upper Harmonics

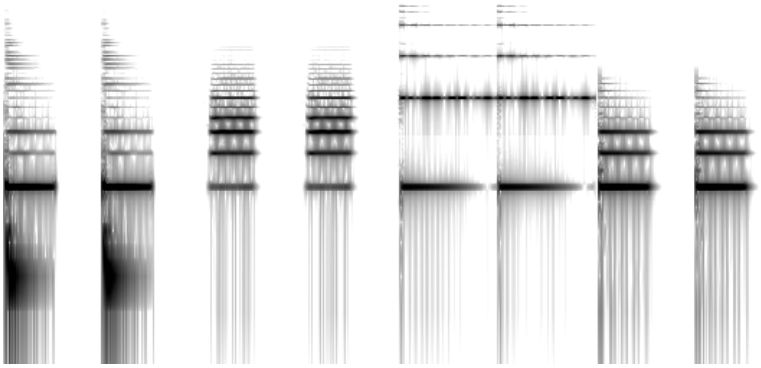


Most strings (and musical instrument bodies) vibrate at all those frequencies simultaneously. While a string is vibrating at its full length it is also vibrating in halves (see the illustration below–note that all of those motions happen at once). That half vibrating motion is twice as fast. It is also vibrating in thirds, fourths, fifths, and so on. So a single string vibrating at 65 Hz is also producing the frequencies 130, 195, etc.

14.2. Vibrating Strings

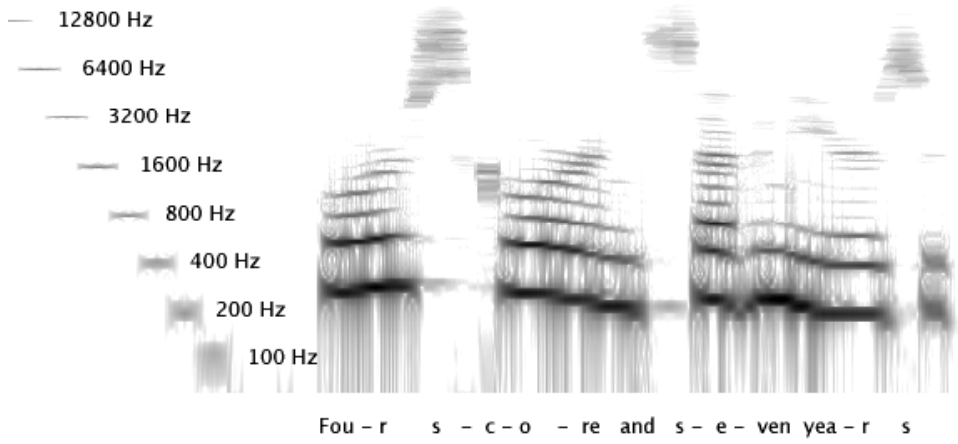


Timbre is not only the presence, but also the amplitude of these harmonics. Each of the harmonics is a sine wave. As a general rule the higher harmonics have a lower amplitude, but the actual amplitude of each harmonic will differ with each timbre. The presence and amplitude of upper harmonics are the tonal fingerprint of any sound. A violin may have a stronger third harmonic, weaker fourth, no fifth, weak sixth, while a trombone will have a weak third, strong fourth, and so on. Below are sonograms of a guitar, saxophone, bell, and electric piano playing two notes each. (They are synthesized, but clearly recognizable.) Note the different collections of upper harmonics.



Below is the graph we examined earlier of a voice saying “four score and seven years.” The ribs are harmonics. Notice the difference between the “e” in seven and the two “r” sounds in score and years. In the “r” of year the first three harmonics are strong, then there is a span of about four missing harmonics, then about four harmonics of relative strength. Also notice that each harmonic band is independent, and evolving with each word.

14.3. Spectral Analysis of “Four Score”

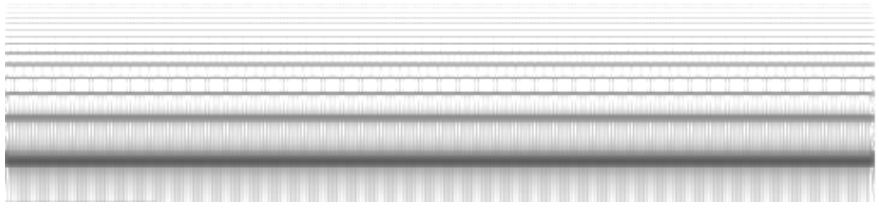


It is also important to understand that timbre (hence the shape of the wave and presence of upper harmonics) changes in real time. Most musicians strive for, but never achieve perfect tone quality. Yet it is this very failure that makes the sound natural. Synthesized sounds have perfectly homogonous tones and we don't like that[31]. For example, a sawtooth wave is actually a series of sine waves tuned to the harmonic series with relatively lower amplitudes. The same is true of other complex waves; square, triangle, even noise. All can be constructed using individual sine waves. That is the theory of additive synthesis.

Additive Synthesis

Most synthesizers offer a variety of wave forms for building sounds but if the shape of the wave is predefined you have no control (outside of filtering) over the individual harmonics. But they are also incapable of additive synthesis, since they have only four or five sine oscillators and maybe two or three envelope generators. But additive synthesis allows for independent control over each wave's parameters such as amplitude or envelope, and SC is a virtual warehouse of synthesis VCOs. With only a few lines of code you can create hundreds of partials with individual envelopes. But let's start with a dozen.

Below is a sonogram of a sawtooth wave at 200 Hz. Not only does it have all harmonics, but they are very even and consistent. Each of these upper harmonics can be thought of as sine waves with frequencies in multiples to the lowest, or fundamental.



[31] Then what, if we don't like perfect sounds, and they are impossible, is the goal of a musician working to improve tone quality?

To reconstruct this sawtooth we need individual sine waves that are harmonic, or multiples of the fundamental. Here is a crude version built on a fundamental of 400. Adding together 12 sine waves would normally distort, so they are all scaled down by 0.1.

```
   14.4.  Adding sines together

(
{
(
   SinOsc.ar(400) + SinOsc.ar(800) + SinOsc.ar(1200) +
   SinOsc.ar(1600) + SinOsc.ar(2000) + SinOsc.ar(2400) +
   SinOsc.ar(2800) + SinOsc.ar(3200) + SinOsc.ar(3600) +
   SinOsc.ar(4000) + SinOsc.ar(4400) + SinOsc.ar(4800)
)*0.1
}.scope
)

// Harmonics adjusted

(
{
(
   SinOsc.ar(400, mul: 1) + SinOsc.ar(800, mul: 1/2) +
   SinOsc.ar(1200, mul: 1/3) + SinOsc.ar(1600, mul: 1/4) +
   SinOsc.ar(2000, mul: 1/5) + SinOsc.ar(2400, mul: 1/6) +
   SinOsc.ar(2800, mul: 1/7) + SinOsc.ar(3200, mul: 1/8) +
   SinOsc.ar(3600, mul: 1/9) + SinOsc.ar(4000, mul: 1/10) +
   SinOsc.ar(4400, mul: 1/11) + SinOsc.ar(4800, mul: 1/12)
)*0.1
}.scope
)
```

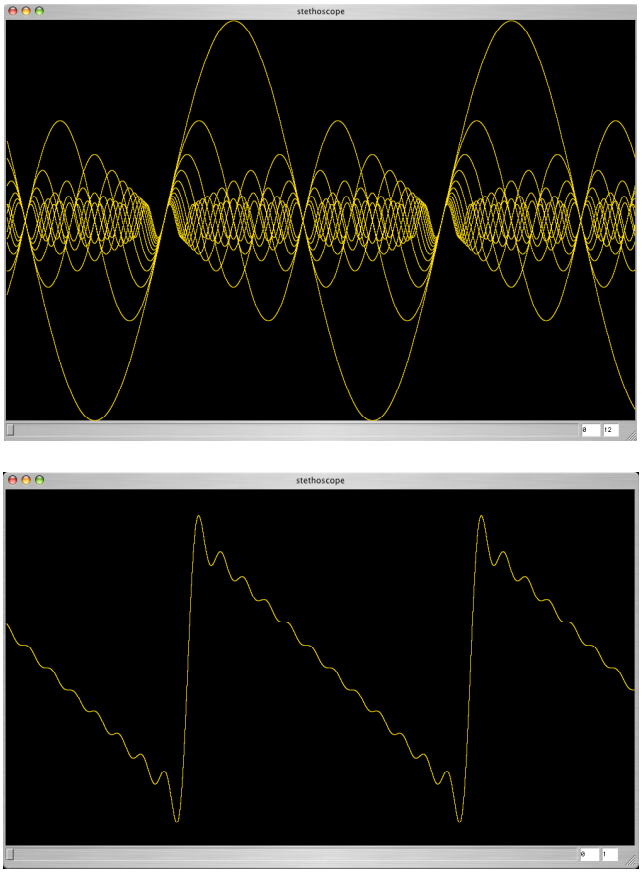
This second version makes one more adjustment. The amplitude of each sine should decrease in proportion to the partial number. The second partial should be ½ the volume of the first, the third is 1/3rd, the fourth 1/4th, etc.

This next example uses a variable to calculate the upper harmonics and it uses an array to spread the sine waves across different output busses. You probably don't have 12 outputs, but that's ok, it is intended to be seen and not heard. Before running it, turn your speakers down and also open the sound system preferences and choose line in for an input (and don't connect anything). This will insure no input signal will interfere with the waves. Also remember you can resize the scope window to get a better look at each wave.

```
   14.5.  Additive synthesis with a variable

(
{
f = 100;
[
   SinOsc.ar(f*1, mul: 1), SinOsc.ar(f*2, mul: 1/2),
   SinOsc.ar(f*3, mul: 1/3), SinOsc.ar(f*4, mul: 1/4),
```

```
    SinOsc.ar(f*5, mul: 1/5), SinOsc.ar(f*6, mul: 1/6),
    SinOsc.ar(f*7, mul: 1/7), SinOsc.ar(f*8, mul: 1/8),
    SinOsc.ar(f*9, mul: 1/9), SinOsc.ar(f*10, mul: 1/10),
    SinOsc.ar(f*11, mul: 1/11), SinOsc.ar(f*12, mul: 1/12)
]
}.scope(12)
)
```

After resizing, change the style to "overlay" by pressing "s." This will overlay all the sine waves so you can see them on one scope. It should look something like this:



This patch illustrates how harmonic waves interact. About a quarter of the way across this graph you can see how all the sines are in synch and are pushing toward their peak. This is what creates the sharp edge of a saw tooth wave. As the higher harmonics move downward they begin to cancel out the energy of the lower waves. The aggregate of this process is a gradually descending ramp of energy as more harmonics move into the negative phase of their cycle, until the end of the pattern where you see they are all more or less in the negative part of the graph. This is the bottom of the saw wave. The second graph is the same patch mixed down. I've tried to line them up so that the cycles match.

Here is another way to think of it: We know that because all the upper partials are multiples of the fundamental, there will be some point where they are all at 0. (100 Hz, 200 Hz, 300

Hz, and so on, will all be at 0 every second.) There will also be some point where they will all be in synch at their peaks; 1, likewise -1. This is a mathematical fact, since they are multiples. The point where they are all at 1 is the peak of the saw wave. Where they are all 0 is the middle, and all at -1 is the valley of the sawtooth. We can roughly illustrate this idea using square waves.

Changing the harmonics will change the character of the sound. Changing the amplitudes of any single harmonic will subtly change the character. Try replacing any *mul* argument (a higher one) with a *MouseX.kr*, isolating the amplitude of that single harmonic. Notice the change. Below is the same patch, but each amplitude is being gradually controlled with an *LFNoise1*. Notice that the fundamental pitch stays the same, while the tone quality changes. Our brains add together all the sine waves and we hear it as a single pitch. Also note that this is not filtering (covered below), but additive.

```
   14.6.  Additive saw with modulation

(
{
var speed = 14;
f = 300;
t = Impulse.kr(1/3);
Mix.ar([
   SinOsc.ar(f*1, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/1),
   SinOsc.ar(f*2, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/2),
   SinOsc.ar(f*3, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/3),
   SinOsc.ar(f*4, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/4),
   SinOsc.ar(f*5, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/5),
   SinOsc.ar(f*6, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/6),
   SinOsc.ar(f*7, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/7),
   SinOsc.ar(f*8, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/8),
   SinOsc.ar(f*9, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/9),
   SinOsc.ar(f*10, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/10),
   SinOsc.ar(f*11, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/11),
   SinOsc.ar(f*12, mul: LFNoise1.kr(rrand(speed, speed*2), 0.5, 0.5)/12)
])*0.5
}.scope(1)
)
```

Try changing the value for *speed*, which controls the frequency of the *LFNoise*. Then try replacing all the *LFNoise0* ugens with an *LFNoise1* (use find and replace). Finally, replace *f = 100* with *f = someOtherControl* such as a sine wave, another *LFNoise* or a mouse.

Next we are going to add an envelope. We could replace the last *0.5 with a single envelope to control the amplitude of all the oscillators at once. And that's how most synthesizers with limited modules do it. But since we are working with code we can assign an envelope to each oscillator, making each harmonic independent, resulting in a more natural sound.

These examples are unnecessarily redundant (a sort of shotgun approach) to illustrate additive methods clearly. Even so, with these few dozen lines we have already surpassed the

capacity of most commercial synthesizers. Granted, this exercise is a bit tedious, perhaps giving you a glimpse of the pioneering efforts of early electronic composers working on patch style synthesizers.

```
   14.7.  Additive saw with independent envelopes

(
{
f = 100;
t = Impulse.kr(1/3);
Mix.ar([
   SinOsc.ar(f*1, mul: EnvGen.kr(Env.perc(0, 1.4), t)/1),
   SinOsc.ar(f*2, mul: EnvGen.kr(Env.perc(0, 1.1), t)/2),
   SinOsc.ar(f*3, mul: EnvGen.kr(Env.perc(0, 2), t)/3),
   SinOsc.ar(f*4, mul: EnvGen.kr(Env.perc(0, 1), t)/4),
   SinOsc.ar(f*5, mul: EnvGen.kr(Env.perc(0, 1.8), t)/5),
   SinOsc.ar(f*6, mul: EnvGen.kr(Env.perc(0, 2.9), t)/6),
   SinOsc.ar(f*7, mul: EnvGen.kr(Env.perc(0, 4), t)/7),
   SinOsc.ar(f*8, mul: EnvGen.kr(Env.perc(0, 0.3), t)/8),
   SinOsc.ar(f*9, mul: EnvGen.kr(Env.perc(0, 1), t)/9),
   SinOsc.ar(f*10, mul: EnvGen.kr(Env.perc(0, 3.6), t)/10),
   SinOsc.ar(f*11, mul: EnvGen.kr(Env.perc(0, 2.3), t)/11),
   SinOsc.ar(f*12, mul: EnvGen.kr(Env.perc(0, 1.1), t)/12)
])*0.5
}.scope(1)
)
```

Shortcuts

There is always an easier way to do things in SC. Any limitation usually lies in the person writing the code. For example, we can use multi-channel expansion, covered in the previous chapter, to expand a single line of code into an array of items. In the patch above the array (everything between the two brackets) is filled with oscillators that are written out fully. The *Mix* combines them all into one channel. Remember that if any argument is an array, that ugen is expanded into an array of ugens, each ugen inheriting the argument corresponding to it's position in the array.

```
   14.8.  additive synthesis with array expansion

Mix.ar([SinOsc.ar(100), SinOsc.ar(200), SinOsc.ar(300)])

// Could be written as

Mix.ar(SinOsc.ar([100, 200, 300]))
```

In the example below, the first *midicps* returns a single value. The second returns an array of values. The next line shows a shortcut for writing an array that contains a series of numbers.

The last shows the patch above written using this technique. To the compiler, they are the same. The advantage is economy in typing. It is also more concise and easier to read.

```
   14.9.  additive synthesis with array expansion

midicps(60);

midicps([60, 62, 64, 65, 57, 69, 71]);

(1..12) // shortcut for [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

midicps((1..24)*60)

(1, 3..13) // means [1, 3, 5, 7, 9, 11, 13]

midicps((60, 63..72))

// additive synthesis with shortcuts

(
{
f = 100;
t = Impulse.kr(1/3);
Mix.ar(
   SinOsc.ar(
      f*(1..12),
      mul: EnvGen.kr(
         Env.perc(0, 1),
         t,
         levelScale: 1/(1..12),
         timeScale: [1.4, 1.1, 2, 1, 1.8, 2.9, 4, 0.3, 1, 3.6, 2.3, 1.1]
      )
   )
)*0.5
}.scope(1)
)
```

The *timescale* needs to be written out because it is not a logical series that can be expressed with a shortcut. But if is the values don't need to be specific, as long as it's between 1 and 3, I could use *rrand(1.0, 3.0).dup(12)* to generate the array.

One situation where you might need to enter each item of an array is with a collection of values that have no simple geometric pattern, such as a Cmaj 9 #11 chord (C, E, G, B, D, F#), or the whole-step, half-step relationships in a diatonic minor scale (C, D, Eb, F, G, Ab, Bb, C). Both of these are shown below as arrays of midi values.

```
   14.10. additive synthesis with array expansion

(
{
t = Impulse.kr(1/3);
Mix.ar(
```

```
    SinOsc.ar(
        [60, 64, 67, 71, 74, 78].midicps,
        mul: EnvGen.kr(
            Env.perc(0, 1),
            t,
            levelScale: 1/(1..6),
            timeScale: rrand(1.0, 3.0).dup
        )
    )
)*[0.3, 0.3]
}.scope(1)
)

(
{
Mix.ar(
    Pan2.ar(
        SinOsc.ar(
            [60, 62, 63, 65, 67, 68, 71, 72].midicps,
            mul: LFNoise1.kr(rrand(0.1, 0.5).dup(8), 0.5, 0.5)
        ),
        1.0.rand2.dup(8)
    )
)*0.2
}.scope(1)
)
```

Filling an array

Another version of the patch below uses *Array.fill* to do all the typing, so to speak, for us. *Array.fill* generates an array based on a function. Each item in the array is a result of the function. The first example simply fills the array with random numbers. The second fills an array with sine waves. The first argument for *Array.fill* is the number of items in the array and the second is the function that generates each value. Think of the function as a small machine that is given a task; perhaps to spit out numbers based on your instructions.

```
    14.11. Array.fill

Array.fill(16, {100.rand})

Array.fill(16, {SinOsc.ar(100.rand)})
```

When you give someone a task, such as cutting up apples, you often want to supply different instructions for subsequent repetitions. For example, the task might be to cut up ten apples. But you may want each one cut up differently; leave the first one whole, cut the second one in half, the third in threes, the fourth in fourths, and so on. You can see the pattern. Rather than tell the person each time the number of cuts to make for each apple, you could just include in the instructions to count the number of repetitions and cut each apple into that many pieces. In order to do this, the system needs to know which iteration it is on. It needs a counter. We could build our own using a variable, as shown in the example below.

14.12. Array.fill with counter

```
( // fills an array with the number counter, then increases the count
var counter = 0;
Array.fill(16, {counter = counter +1; counter})
)

( // fills an array with the counter * 100
var counter = 0;
Array.fill(16, {counter = counter +1; counter*100})
)
```

This is such a common procedure that the *Array* object comes with a built in counter to keep track of the number of repetitions. That number is passed to the function by way of an argument. An argument acts just like a variable; you can name it anything you want, and can even change its value, but there is usually little reason to do so. The difference is that arguments are usually supplied from some outside process and are passed to the function. If there is more than one argument then they are identified by position (like arguments in the argument list we've seen so far). You see this new syntax within the function below: *arg myNameForCount*. Run each of the lines below and check the results in SCs data display window. The last two examples show an alternate syntax: *|myNameForCount|*. The second example generates an array of arrays.

14.13. Array.fill with arg

```
//fill an array with the number of this iteration
Array.fill(16, {arg myNameForCount; myNameForCount})

//fill an array with random numbers
Array.fill(16, {arg myNameForCount; [myNameForCount, rrand(20, 100)]})

//fill an array with multiples of myNameForCount
Array.fill(16, { |myNameForCount| myNameForCount*3})

//fill an array with SinOsc objects, each with a freq of the counter*100
Array.fill(16, { |myNameForCount| SinOsc.ar(myNameForCount*100)})
```

To build a harmonic series you want to multiply some fundamental by 1, then 2, 3, 4, etc. The counter can be used to calculate the *freq* of each *SinOsc* and the *levelScale* of each *EnvGen*. The counter begins at 0, not 1, and I don't want to start with freq*0, so I use the variable partial to increase it by 1. The code below generates 16 upper partials, each proportionally softer, with a random decay between 0.1 and 2.1. The *fund*partial* ensures they are all harmonic. The *1/partial* is the *levelScale* for the envelope. The first will be 1/1, the next 1/2, 1/3, 1/4, etc. *Array.fill* returns an array of *SinOsc* ugens and *Mix.ar* mixes them all down to one channel.

14.14. Additive saw wave, separate decays

```
(
{
var gate, fund;
gate = Impulse.kr(1/3);
fund = MouseX.kr(50, 1000);
Mix.ar(
    Array.fill(16,
    {arg counter;
    var partial;
    partial = counter + 1;
    SinOsc.ar(fund*partial) *
    EnvGen.kr(Env.adsr(0, 0, 1.0, TRand.kr(0.2, 2.0, gate)),
        gate, 1/partial)
    })
    )*0.2 //overall volume
}.scope(1)
)

// Same patch, using a few shortcuts.

(
{
var gate, fund;
gate = Impulse.kr(1);
fund = MouseX.kr(50, 1000);
Mix.ar(
    SinOsc.ar(fund*(1..16).postln) *
    EnvGen.kr(Env.perc(0, TRand.kr(0.2, 2.0, gate)),
        gate, 1/(1..16))
    )*0.2 //overall volume
}.scope(1)
)
```

I began this section praising SC's capacity for complexity and richness. The patch above isn't that astonishing, but you should recognize the more natural sound in the different decay rates for each partial. Compare it with this patch, which uses only one envelope for all partials.

```
    14.15. Additive saw wave, same decays

(
{
var gate, fund, env;
gate = MouseButton.kr(0, 1, 0);
fund = MouseX.kr(50, 1000);
env = Env.adsr(0, 0, 1.0, 2.0);
Mix.ar(
    Array.fill(16,
    {arg counter;
    var partial;
    partial = counter + 1;
    SinOsc.ar(fund*partial) *
    EnvGen.kr(env, gate, 1/partial)
```

```
    })
    )*0.2 //overall volume
}.scope(1)
)
```

When working with code based synthesis (as opposed to graphics) it is easy to duplicate a single simple idea to achieve complex sounds quickly. This is especially evident with additive synthesis and array expansion. Take for example this simple patch, where the amplitude of an oscillator is being controlled by another sine wave.

```
    14.16. Single sine with control

{SinOsc.ar(400, mul: SinOsc.ar(1/3, mul: 0.5, add: 0.5))}.play
```

We can use this single model to build a much more complex sound[32] by adding together sine waves as harmonics. *Mix.fill* combines the *fill* logic used with *Array.fill* and the *Mix* used in a previous patch. It generates an array of ugens and mixes them down. It also has a counter argument that can be used to calculate upper harmonics. This patch uses an *FSinOsc*, which is more efficient.

```
    14.17. Gaggle of sines

(
{
    var harmonics = 16, fund = 50;
    Mix.fill(harmonics,
           { arg count;
              Pan2.ar(
                 FSinOsc.ar(
                    fund * (count + 1), // calculates each harmonic
                    mul: FSinOsc.kr(rrand(1/3, 1/6), mul: 0.5, add: 0.5 )),
                 1.0.rand2)
           }
    ) / (2*harmonics)
}.play;
)
```

See the practice section below for variations on this patch.

Inharmonic spectra

Each of the sine waves in the examples above is a multiple of the fundamental (f*1, f*2, f*3, f*4, etc.). This is a harmonic spectrum. Most pitched instruments have harmonic spectra. Non-pitched instruments such as gongs, bells, and cymbals tend toward inharmonic spectra,

[32] Based on an example from the Tour of Ugens in the Help folder.

or a set of frequencies that are not multiples of the fundamental. To generate an inharmonic spectrum you need to enter unrelated values for each sine wave, for example, 135, 173, 239, 267, 306, 355, 473, 512, 572, and 626. How did I arrive at these unrelated frequencies? I used a shuffled deck of cards, dealing three at a time, ignoring royalty.

In the examples above the amplitude of each upper harmonic was calculated in relation to it's position; higher harmonics were softer. But in enharmonic spectra the amplitudes don't have a pattern. For example, 0.25, 0.11, 0.12, 0.04, 0.1, 0.15, 0.05, 0.01, 0.03, 0.02, and 0.12. I used a similar "random" method to generate this series.

Additive synthesis can be used to generate pure wave forms, such as saw tooth, square, or triangle, but these are available on commercial synthesizers. It is true, you have more control over the upper partials, but at least these other wave forms do exist. But wave forms with enharmonic spectra do not. Commercial synthesizers go straight from perfect wave forms to perfect noise. Nothing in between. This is one area where SC is unique.

Here is a patch that adds together these frequencies and amplitudes from my random process above. There is no envelope, so you don't get the sense of a bell, but it has a very metallic resonance.

```
   14.18. Inharmonic spectrum

{Mix.ar(
   SinOsc.ar(
      [72, 135, 173, 239, 267, 306, 355, 473, 512, 572, 626],
      0, //phase
      [0.25, 0.11, 0.12, 0.04, 0.1, 0.15, 0.05, 0.01, 0.03, 0.02, 0.12]
   ))}.scope(1)
```

Random Numbers, Perception

We call the sets of frequencies above "random" because there is no apparent pattern. Randomness then is a matter of context and perception. We decide if a number is random by how it corresponds with the numbers that came before or after. A single number with no context cannot really be random.

As an illustration: Can, you, predict, the, next, number, in, this, series?

226, 966, 7733428, 843, 6398, 686237, 46, 8447, _____

Seem random? It is not. There is a very clear system. (Hint: there are no 1s or 0s). When you get the trick you can predict the next number and maybe the next three or four. When you see the pattern the random quality evaporates, even though the series or process hasn't changed. Then what has changed? Your perception. A good lesson to apply in other areas of music appreciation (and life).

When I shuffle a deck of cards they are "randomized" so that I can't predict what the sequence of cards would be. But if I began with a fresh deck of cards, to which I know the

order, and I used a precise formula when shuffling (e.g. split in half and alternate exactly one from each pile in turn), or kept close track of the order in which the cards fall during shuffling, I would then be able to predict the order. If that were the case, the deck would not be randomized. The only difference between a real shuffle and a carefully orchestrated one is my grasp of the process. So what random really means is mixed up to the point where a particular human can't predict the outcome.

During a recent lecture on enharmonic spectra I showed a class a set of random values chosen by the computer. One student said he didn't understand how that could happen. Other class members didn't understand why he didn't understand; the computer picks a random number, that's all. But his observation was astute; though computers do seem sometimes to act randomly, they never are. Every action, every process, every bit of code, every 0 and 1 can be accounted for. It is not possible for a computer to act in a random fashion. But sometimes their complexity may result in behavior that seems random to us.

To generate random sequences composers use a similar method to my shuffled deck of cards, but applied to a very large collection of numbers. These numbers are mixed up using a mathematical formula (see Moore page 409) in the form of an algorithm inside the computer. Since the process was described to the computer, and the computer executes the method precisely, and since computers, while bereft of intuition, are very good at keeping close track of numbers, it "knows" the sequence, because it is a formula. However, the resulting series is so complex that we humans don't recognize the pattern. This series of numbers is called a random number generator, and is executed and stored when the computer or program first runs. Since the same formula is used each time, the series of numbers is the same, and should even be the same on different computers[33]. So to the computer it is not random at all.

How did we do this before computers? We used a book called "A Million Random Digits" which contained, drum roll, a million random digits. No kidding. Look it up on Amazon and read the reviews.

When you run a program that uses some "random" series of events (such as a dice or card game), the computer chooses those values from this sequence of numbers.

The next problem is that it always starts at the beginning, so you always get the same values at first, like using the same deck of cards from the top of the deck without cutting or reshuffling, or reading "A Million Random Digits" at the same page every day. The order will seem randomized the first few times through the series. After two or three repetitions we will remember the pattern and it won't be random anymore. Part of random, to us, is different numbers each time.

So how do you get it to do different numbers? You cut the deck. To cut the deck of a random number generator you start at a different point in the series. This is called a random seed. You give the computer a number as a seed which is used to count into the random number sequence. The computer then starts using numbers from that point.

---

[33] We have confirmed this in our multi-user labs.

There is one more problem. If you know the seed, then the resulting sequence of numbers is still not really random to you. You need to choose a random seed. I mean, you need to choose a random seed randomly. I mean, that random seed has to be random (perplexing?).

The solution to this problem is to use the internal clock of a CPU, which is just a series of numbers rapidly flying by. If each number on the clock were matched to a number in the random sequence, then in a sense you are zipping through the random sequence (like riffling the shuffled deck). Grabbing a number from the clock and using it as a seed is like sticking your finger in the riffling deck: you get a random position at that moment. It seems a little convoluted, but that's how it was (and is) done.

SC does a random seed automatically behind the scenes. Each time you run a line like *10.rand* SC first reads the number on its internal clock to use as a seed. It then moves into the random number generator sequence and starts its series of choices from that point. It is random to us because we can't predict what number the clock gave or the order of the numbers at that point.

That's how you get pseudo-random choices. But there are also situations where you want to repeat a set of random choices. In this case you would use the same seed over and over to reproduce the same series of events. (For example, many computer card games allow you to replay the previous hand. This is how it's done.) A given random seed is useful when you are debugging a crash and want to reproduce the same error every time. Also you might find a particular variation of random events that you like and will want to reproduce it exactly.

First I'll demonstrate some random choices and then some random choices using a seed. There are a number of random functions in SC. The message *rand* sent to any number will return a random choice between 0 and that number. *55.rand* will return an integer (e.g. 1, 2, 3) between 0 and 55 (not including 55). *55.0.rand* will return a floating point number (e.g. 3.3452, 1.2354) between 0.0 and 55.0. Try both lines several times each to see how random numbers are chosen.

```
    14.19. rand

10.rand;

10.0.rand;
```

Running the lines several times over and over is a bit cumbersome, so below is a method of testing random choices using the dup message, which fills an array with duplicate returns of the function.

```
    14.20. Test a random array

{100.rand}.dup(20)
```

Here is a typical beginner error. Try running the example below and see how different the results are.

14.21. Error from not using a function

```
(100.rand).dup(20)
```

This code picks a random number, but it uses that same number each time. The first example uses a random number choice enclosed in a function. A function means "run this line of code" while *100.rand* on its own means pick a random number. *{100.rand}* means pick a random number each time, *100.rand* means pick a random number and use it each time.

Below is the same example using a seed on the client side (the language and programming side). Run the first line several times to see how it fills an array with random values. Then run the second line several times. Since you are seeding the random generator each time you will get the same numbers in the array. Try changing the seed to something other than 5. You'll get a new series, but the same series each time. (Not very "random" after four runs.)

14.22. Client random seed

```
{100.rand}.dup(20);

thisThread.randSeed = 5; {100.rand}.dup(20);
```

This does not, however, affect the server, which plays the sounds once you design a patch. To seed a random process on the server use *RandSeed*. This ugen can be triggered, so you can reset the seed over and over. The first argument is the trigger, the second the seed. Run the first and second examples several times. Let the third one run. It will reset every 5 seconds.

14.23. Server random seed

```
// different every time
{SinOsc.ar(LFNoise0.kr(7, 12, 72).midicps, mul: 0.5)}.play

// same every time
(
{
RandSeed.kr(1, 1956);
SinOsc.ar(LFNoise0.kr(7, 12, 72).midicps, mul: 0.5)
}.play
)

// resets every 5 seconds
(
{
RandSeed.kr(Impulse.kr(1/5), 1956);
SinOsc.ar(LFNoise0.kr(7, 12, 72).midicps, mul: 0.5)
}.play
)
```

Finally, what if you want the seed to be chosen in the regular way (using the clock), but you want to know what it is so you can reproduce that series later for debugging or because you liked that version? Here's how.

```
   14.24. Post clock seed

thisThread.randSeed = Date.seed.postln; {100.rand}.dup(20);

(
{ // one random seed chosen and used repeatedly
RandSeed.kr(Impulse.kr(1/5), Date.seed.postln);
SinOsc.ar(LFNoise0.kr(7, 12, 72).midicps, mul: 0.5)
}.play
)
```

The seed is taken from the clock and posted to the data window. If you want to reproduce that version, replace *Date.seed.postln* with the number posted to the window.

In conclusion, one could argue that a generative composition is not really random, since it can be duplicated with the same seed. It seems random because we haven't heard that series of numbers before. But each seed (billons of them), represents a specific repeatable version. So instead of a random process, it can be thought of as a billion possible variations, each identified by that seed. The code you write then is the DNA for billions of variations, of which one is chosen at each performance, and can be identified and repeated with the seed.

Bells

Upper harmonic structure is a "timbre" fingerprint. It helps us distinguish what the sound is (violin, flute, voice), and even subtle variations between instances of the same sound (e.g. your mom with a cold). Even though a collection of frequencies are inharmonic, a "random" collection, we still can easily distinguish it from any other "random" set of frequencies as illustrated with the bell patches below.

The code below generates a set of bell like sounds, each with it's own pseudo-random spectrum. Try reading the code from the center out. The innermost *SinOsc* has a random frequency between 50 and 4000. It is multiplied by an envelope with a decay between 0.2 and 3.0 and a volume between 0 and 1. This code is inside a function with the dup message with 12 as an argument. The dup message creates an array of *SinOsc* ugens with differing frequencies and envelopes. Since the trigger is created outside the function, all the Ugens share this single trigger. The attacks will be the same and it will appear as a single sound. Those 12 sine waves, or inharmonic frequencies, are mixed down using *Mix.ar* and panned with *Pan2*. Try stopping playback and running the first example again. Repeat this four or five times. Can you tell that some frequencies have longer or shorter decay rates?

With each execution the computer chooses a different set of random numbers from the random number generator to create each bell. Could you enter a line of code that would use a random seed and produce the same bell collection twice in a row?

Finally, run the example four or five times without stopping the previous bell, so that 5 or 6 bells are running at once. Notice that even though each single bell is actually a collection of supposedly independent and randomly chosen sine waves our brains meld them into a single recognizable sound. Also, even though each collection is random, we distinguish it as a separate entity, and we can keep track and recognize a group of them.

I can't help raising this question: When is the "random" collection no longer random? When you hear it the second time?

```
   14.25. random frequencies (Pan2, Mix, EnvGen, Env, fill)

(  // Let it run for a while, the strikes are random
{
var trigger, partials = 12;
trigger = Dust.kr(3/7);
Pan2.ar(
   Mix.ar(
      {
      SinOsc.ar(exprand(50.0, 4000)) *
      EnvGen.kr(
         Env.perc(0, rrand(0.2, 3.0)),
         trigger,
         1.0.rand
         )
      }.dup(partials)
   )/partials,
   1.0.rand2
)
}.play
)
```

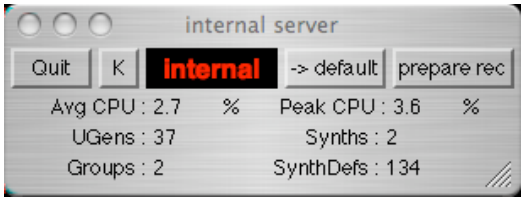
And finally, to illustrate how a harmonic spectrum can become inharmonic, and the resulting transformation from pitch to non-pitch, the patch below adds one single ugen to the previous patch; a mouse control that detunes the upper harmonics as you move the mouse from right to left. The amount of detune is randomly chosen from a range between -1.4 and 1.4, to insure an inharmonic spectrum.

```
   14.26. Harmonic to inharmonic spectra

(
// Let it run for a while, the strikes are random
// Or change Dust.kr to Impulse.kr
{
var trigger, partials = 12, fund;
trigger = Dust.kr(3/7);
fund = exprand(50, 700);
Pan2.ar(
   Mix.fill(partials,
      {arg count;
      SinOsc.ar(count + 1 * fund *
         MouseX.kr(1.0, 1 + 0.4.rand2)) * // detune
```

```
        EnvGen.kr(
            Env.perc(0, rrand(0.2, 3.0)),
            trigger,
            1.0.rand
            )
        }
    )/partials,
    1.0.rand2
)
}.play
)
```

CPU Usage

I lied. SC is not an unlimited supply of synthesis components. We are limited by processing power. Additive synthesis is costly, and the cpu information in the server window (shown below) is now important. It shows average and peak cpu and number of Ugens. Run the patches above several times (without stopping the previous sound) or increase the number of partials to see what happens when you max out the cpu load.



On my laptop I can get it up to 3000 ugens before it slows down. That's a far cry from the two dozen modules on vintage gear.

Go back to the single bell patch and increase the number of partials. Watch the CPU increase, but also notice how the sound changes as you add partials. What is the result of 200+ inharmonic partials?

As the number of random partials increase the sound becomes more and more unfocused until it approaches noise. That is covered in the next chapter.

Just a footnote about the last practice examples below, diverging, converging, and decaying gongs: I like these examples because they sound so cool, and they are so different from anything in real life, but also because they were artificially conceived. During a class on additive synthesis we brainstormed on ways to use the power of additive techniques and hit on the theory of these sounds before realizing them.

Practice: flashing sines, gaggle of sines, diverging, converging, decaying gongs

14.27. flashing[34] (MouseButton, Mix, Array.fill, Pan2, EnvGen, Env LFNoise1)

```
(
{
var trigger, fund;
trigger = Dust.kr(3/7);
fund = rrand(100, 400);
Mix.ar(
    Array.fill(16,
    {arg counter;
    var partial;
    partial = counter + 1;
    Pan2.ar(
        SinOsc.ar(fund*partial) *
        EnvGen.kr(Env.adsr(0, 0, 1.0, 5.0),
            trigger, 1/partial
        ) * max(0, LFNoise1.kr(rrand(5.0, 12.0))), 1.0.rand2)
    })
    )*0.5 //overall volume
}.play
)

//Several of the above mixed down

(
{
var trigger, fund, flashInst;
flashInst = Array.fill(5,
{
    trigger = Dust.kr(3/7);
    fund = rrand(100, 400);
    Pan2.ar(
        Mix.ar(
            Array.fill(16,
            {arg counter;
            var partial;
            partial = counter + 1;
                SinOsc.ar(fund*partial) *
                EnvGen.kr(Env.adsr(0, 0, 1.0, 5.0),
                    trigger, 1/partial
                ) * max(0, LFNoise1.kr(rrand(5.0, 12.0)))
            })
            )*0.2,
     1.0.rand2)
});
Mix.ar(flashInst)*0.6
}.play
)
```

[34] Based on McCartney's patch in the examples folder.

```
// Gaggle of sines varations

(
{
    var harmonics = 16, fund = 50, speeds;
    speeds = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]/5;
    Mix.fill(harmonics,
           { arg count;
              Pan2.ar(
                 FSinOsc.ar(
                    fund * (count + 1),
                    mul: max(0, FSinOsc.kr(speeds.wrapAt(count)))),
                 1.0.rand2)
           }
    ) / (2*harmonics)
}.play;
)

(
{
    var harmonics = 16, fund, speeds;
    speeds = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]/20;
    fund = (MouseX.kr(0, 36).round(7) + 24).midicps;
    Mix.fill(harmonics,
           { arg count;
              Pan2.ar(
                 FSinOsc.ar(
                    fund * (count + 1),
                    mul: max(0, FSinOsc.kr(speeds.choose))),
                 1.0.rand2)
           }
    ) / (2*harmonics)
}.play;
)

// Use mouse to change fundamental

(
{
    var harmonics = 16, fund;
    fund = (MouseX.kr(0, 36).round(7) + 24).midicps;
    Mix.fill(harmonics,
           { arg count;
              Pan2.ar(
                 FSinOsc.ar(
                    fund * (count + 1),
                    mul: max(0, FSinOsc.kr(rrand(1, 1/3), mul: 20).softclip)),
                 1.0.rand2)
           }
    ) / (2*harmonics)
}.play;
)
```

```
(
{
   var harmonics = 16;
   Mix.fill(harmonics,
          { arg count;
             Pan2.ar(
                FSinOsc.ar(
                   exprand(100, 2000),
                   mul: max(0, FSinOsc.kr(rrand(1/3, 1/6))*rrand(0.1, 0.9))),
                1.0.rand2)
          }
   ) / (2*harmonics)
}.play;
)



// Dissipating and converging gongs illustrates how a patch can be built
// from duplicating one idea; classic additive synthesis. It also shows
// how additive synthesis can be used to control each harmonic.
// Listen in stereo to hear the harmonics diverge.

(

{
var dur = 6, base, aenv, fenv, out, trig;
base = Rand(40, 100);
trig = SinOsc.ar(1/10);
out = Mix.fill(15,{
   var thisDur;
   thisDur = dur * rrand(0.5, 1.0);
   aenv = EnvGen.kr(Env.perc(0, thisDur), trig);
   fenv = EnvGen.kr(Env.new([0, 0, 1, 0], [0.25*thisDur, 0.75*thisDur, 0]), trig);
   Pan2.ar(SinOsc.ar( Rand(base, base * 12) *
      LFNoise1.kr(10, mul: 0.02 * fenv, add: 1), // freq
      mul: aenv // amp
   ), ([1, -1].choose) * fenv)
}) * 0.05;
out
}.play(s);


{
var dur = 6, base, aenv, fenv, out, trig, detune;
base = Rand(40, 60);
detune = 0.1; // increase this number to detune the second bell
trig = SinOsc.ar(1/10, pi);
out = Mix.fill(15,
{ arg count;
   var thisDur;
   thisDur = dur * rrand(0.5, 1.0);
   aenv = EnvGen.kr(Env.perc(0, thisDur), trig);
   fenv = EnvGen.kr(Env.new([1, 1, 0, 1], [0.05*thisDur, 0.95*thisDur, 0]), trig);
```

```
    Pan2.ar(SinOsc.ar( base*(count+1+ detune.rand) *
        LFNoise1.kr(10, mul: 0.02 * fenv, add: 1), // freq
        mul: aenv // amp
    ), ([1, -1].choose) * fenv)
}) * 0.05;
out
}.play(s);
)

// Decaying bell

(
{
var aenv, fenv, out, trig, dur, base;
dur = rrand(1.0, 6.0);
base = exprand(100, 1000);
trig = Impulse.kr(1/6);
out = Mix.ar(
    Array.fill(15,{
        arg count;
        var thisDur;
        thisDur = dur * rrand(0.5, 1.0);
        aenv = EnvGen.kr(
            Env.new([0, 1, 0.4, 1, 0], [0, 0.5, 0.5, 0]), trig,
            timeScale: thisDur);
        fenv = EnvGen.kr(
            Env.new([0, 0, 0.5, 0.5, 0], [0.25, 0.5, 0.25, 0]),
                trig, timeScale: thisDur);
        Pan2.ar(SinOsc.ar( Rand(base, base * 12) *
            LFNoise1.kr(10, mul: 0.1 * fenv, add: 1), // freq
            mul: aenv // amp
        ), ([1, -1].choose) * fenv)
    })
) * EnvGen.kr(Env.linen(0, dur, 0), Impulse.kr(6), timeScale: dur,
        levelScale: 0.05, doneAction: 2);
out*0.3;
}.play;
)
```

14. Exercises

14.1. Using any of the additive patches above as a model, create a harmonic collection of only odd harmonics, with decreasing amplitudes for each partial. What is the resulting wave shape?

14.2. Begin with the "additive saw with modulation" patch above. Replace all the LFNoise1 ugens with any periodic wave (e.g. SinOsc), offset and scaled to control amplitude, with synchronized frequencies (3/10, 5/10, 7/10).

14.3. Modify the "additive saw with independent envelopes" patch so that all the decays are 0, but the attacks are different (e.g. between 0.2 and 2.0 seconds). Use a random function if you'd like.

14.4. Use Array.fill to construct an array with 20 random values chosen from successive ranges of 100, such that the first value is between 0 and 100, the next 100 and 200, the next 200 and 300, etc. The result should be something like [67, 182, 267, 344, 463, 511, etc.].

14.5. In the patch below, replace 400 and 1/3 with arrays of frequencies, and 0.3 with an array of pan positions. You can either write them out, or use {rrand(?, ?)}.dup(?).

```
(
{Mix.ar(Pan2.ar(
        SinOsc.ar(400, // freq
                mul: EnvGen.kr(Env.perc(0, 2),
                        Dust.kr(1/3) // trigger density
                )*0.1),
        0.3 // pan position)
)}.play)
```

14.6. Devise a method of ordering numbers and/or letters that would appear random to someone else, but is not random to you.

15 - Noise, Subtractive Synthesis, Debugging, Modifying the Source

Additive synthesis, from the previous chapter, builds complex sounds by adding together individual sine waves. In subtractive synthesis you begin with a sound or wave that has a rich spectrum and modify it with a filter. The results are more natural because natural sounds come about in a similar way. The tone of a plucked string or bell, for example, begins with a source of random excitation. The body of the instrument, which has natural resonant frequencies, filters and shapes the character of the sound. Speech works through filtering. The vocal chords alone represent an unintelligible but rich fundamental sound source. The tongue, sinus cavities and shape of the mouth determine the rich consonants and vowels required for speech.

All synthesizers, including SC, come equipped with a collection of spectrum rich sources. If richness is defined by the number of sine waves that are represented in the sound, then arguably the richest is pure noise. In the previous chapter you may have noticed that the greater the number of enharmonic spectral elements there were, the more the sound approached noise. Noise is often defined as all possible frequencies having equal representation. We have also defined it as a wave with no pattern. I like to think of it as any series of numbers with a pattern I don't recognize.

Noise

"White" noise is said to have equal power across the frequency spectrum. "Pink" noise is exponentially biased toward lower frequencies; equal energy, or number of sine waves, per octave band. Remember that musical octaves are exponential. The frequencies 100 and 200 are an octave apart, a difference of 100. But an octave above 2000 is 4000, a difference of 2000. White noise represents frequencies evenly, so if there were theoretically 100 frequencies between 100 and 200, then there would be 2000 between 2000 and 4000. But that's not the way we hear music. We perceive the distance between 100 and 200 to be the same as 2000 to 4000. Pink noise adjusts the distribution of frequencies to match the way we hear. So if there were 100 frequencies between 100 and 200 (an octave band), then there would only be 100 between 2000 and 4000 (an octave band). Pink noise sounds more natural and full. White noise sounds brighter and thinner.

We can create noise through additive synthesis. Pink noise would require an exponential random process with more low frequencies. The lines of code below illustrate a linear random set and an exponential set. Notice there are more low numbers represented with the exponential set. Next, these two functions are used to mix down 1000 randomly chosen sine waves to generate white and pink noise. (Notice the amount of cpu, and compare it with the single *PinkNoise* object by itself.)

```
   15.1.  noise from scratch (rrand, exprand, Mix, fill, SinOsc)

{rrand(1, 1000).round(1)}.dup(100).sort

{exprand(1, 1000).round(1)}.dup(100).sort
```

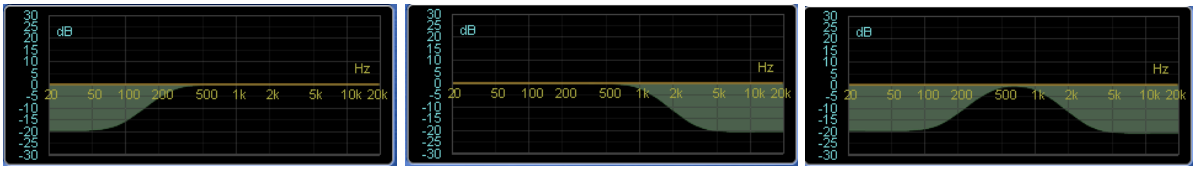
```
{Mix.fill(1000, {SinOsc.ar(rrand(1.0, 20000))})*0.01}.play

{Mix.fill(1000, {SinOsc.ar(exprand(1.0, 20000))})*0.01}.play

{PinkNoise.ar}.play // compare cpu
```

There are 15 or so different types of noise available in SC. They can be used for audio signal (sounds you hear) and control (patterns that affect other sine waves, that you hear). The common denominator is that they appear to be random, that is, we don't recognize any pattern in the wave or series of numbers. Listen to each of these examples of noise. Notice the differences in how they look on the scope. Turn down your monitors or headsets. They can be loud.

```
   15.2.  Types of noise

{WhiteNoise.ar}.scope(1)

{PinkNoise.ar}.scope(1)

{BrownNoise.ar}.scope(1)

{GrayNoise.ar}.scope(1)

{Dust.ar(12)}.scope(1)
```

Subtractive Synthesis

Subtractive synthesis uses filters to remove or shape the frequencies of a rich wave. Three common filters are low pass, high pass, and band pass. The name describes what they do. A low pass filter allows low frequencies to pass, filtering or removing the highs. A high pass filter allows high frequencies to pass, filtering the lows. A band pass filters lows and highs, allowing a band to pass. See examples of each bellow. The first is a high pass; the brown area represents frequencies being filtered out. Everything above, "higher" than the cutoff value is allowed to pass. Likewise, the second example shows the lower frequencies unaffected while the brown area, highs, are filtered.



These names have always been confusing to me because they indicate the frequencies that are *not* affected. We *use* them to modify the opposite end of the spectrum. So when I use a high pass filter, I think of it as a *bass* filter; increasing or decreasing low frequencies. The third example, a band pass filter, illustrates this confusion. It shows low and highs are cut, leaving a band unaffected. But if you increase the entire spectrum, raising the lows and highs to 0, and the band above 0 db, you are essentially boosting the middle band of frequencies

that are presumably, from the name, being passed; band *pass*. So here are images illustrating how high, low, and band pass filters are often used, to increase the band that is not being "passed" despite their name.

The objects in SC that represent these devices are *RLPF*, *RHPF*, and *BPF*. The arguments for each are input to be filtered, frequency cutoff, and resonance or the reciprocal of q (bandwidth/cutoff frequency). The input is the signal being filtered. The cutoff frequency is the point where frequencies begin to be filtered. Resonance (rq) effects how narrow the focus of the filter is. The practical result of a very narrow resonance (e.g. 0.05) is a pitch at the filter cutoff frequency. Here are examples of a band pass filter with broad and narrow rq values.

In the examples below the X axis controls the frequency cutoff, the Y axis controls the q. The top of the screen is a narrow q (0.01) and the bottom is wide (2.0).

15.3. Filtered noise

```
(
{
var signal, filter, cutoff, resonance;

signal = PinkNoise.ar(mul: 0.7);
cutoff = MouseX.kr(40, 10000, 1);
resonance = MouseY.kr(0.01, 2.0);

RHPF.ar(signal, cutoff, resonance)}.scope(1)
)

(
{
var signal, filter, cutoff, resonance;

signal = PinkNoise.ar(mul: 0.7);
cutoff = MouseX.kr(40, 10000, 1);
resonance = MouseY.kr(0.01, 2.0);

RLPF.ar(signal, cutoff, resonance)}.scope(1)
)
```

```
(
{
var signal, filter, cutoff, resonance;

signal = PinkNoise.ar(mul: 0.7);
cutoff = MouseX.kr(40, 10000, 1);
resonance = MouseY.kr(0.01, 2.0);

BPF.ar(signal, cutoff, resonance)}.scope(1)
)
```

Here are similar examples using a saw wave (which is a rich sound by virtue of the upper harmonic content). When a low number is used for resonance on noise you hear a single continuous frequency that corresponds with the filter cutoff. With a saw wave each of the upper harmonics will resonate as the cutoff frequency passes near and over that harmonic. This familiar filter sweep was pretty cool back in the 70s.

The volume is quite low when the resonance is narrow. Maybe you could add a variable that adjusts the volume as the resonance decreases?

```
   15.4.  Filtered saw

(
{
var signal, filter, cutoff, resonance;

signal = Saw.ar([50, 75], mul: 0.7);
cutoff = MouseX.kr(40, 10000, 1);
resonance = MouseY.kr(0.01, 2.0);

RLPF.ar(signal, cutoff, resonance)}.scope(2)
)

(
{
var signal, filter, cutoff, resonance;

signal = Saw.ar([50, 75], mul: 0.7);
cutoff = MouseX.kr(40, 10000, 1);
resonance = MouseY.kr(0.01, 2.0);

BPF.ar(signal, cutoff, resonance)}.scope(2)
)

{RLPF.ar(Saw.ar([100,250],0.1), XLine.kr(8000,400,5), 0.05) }.play;
```

Voltage Controlled Filter

On classic synthesizers these modules were called VCFs, or voltage controlled filters. The parameters of a filter can be controlled in the same way we did frequency and amplitude in previous examples.

With a narrow rq a pitch emerges at the filter cutoff, so the same types of controls that worked in previous patches for frequency will work for a filter cutoff, with similar results.

```
   15.5.  Filter cuttoff as pitch

// Frequency control (first patch)

{SinOsc.ar(LFNoise0.kr([12, 12], 500, 500), mul: 0.5)}.play

// Same control source applied to filter cutoff with narrow rq

{RLPF.ar(PinkNoise.ar(0.3), LFNoise0.kr([12, 12], 500, 500), 0.02)}.play

// Wider rq does not result in a focused pitch

{RLPF.ar(PinkNoise.ar(0.9), LFNoise0.kr([12, 12], 500, 500), 1)}.play
```

These filters are single band, and the results are rather simple, synthetic, vintage. Real sounds have many resonant frequencies. In order to achieve that level of richness we need a filter that will allow many resonant nodes.

Component Modeling and Klank

Component modeling shifts the focus of synthetic components from the sound you want to create to the body of an instrument you want to imitate. That is, rather than shape a saw wave, shape the body of a guitar, then excite it with a rich source.

Klank is a component modeling filter that imitates the resonant nodes of a body or space. It allows you to specify a set of frequencies to be filtered (resonated), matching amplitudes, and decay rates.

The first argument for *Klank* is the frequency specifications. It is an array of arrays (two dimensional array). That is each item in the array is also an array (like a box filled with boxes). The outer array begins with a "`" (back tick? accent grave?) to show that I don't want multi-channel expansion. (If this character were left off, SC would expand the array into different channels. Go ahead and try it.) The first inner array is the set of frequencies that resonate, the second is the amplitude of each frequency (default is 1), and the last is the decay rates of each frequency.

With steady state sounds such as noise, the decay has little affect, so it's left off in the first examples. Since we are adding 10 resonant frequencies the noise source has to be very quiet: 0.01 to 0.1. The second example fills the arrays using *Array.series*. Look at the help file for

an explanation. The third example loads an array with random frequencies. Run it several times to see how it is different every time.

In these examples the sound is constantly evolving because the excitation source (input) is noise, or a random set of numbers flying by at 44k a second (the random number series mentioned above). I like to think of these examples as bowing a resonant body. The bow is the random number series, the Klank is the resonant body, which shapes the sound. The physical body or space described by *Klank* extracts patterns from the noise, but the noise gives variety to the body or space.

```
   15.6.  Resonant array

(
{
Klank.ar(
    `[[100, 200, 300, 400, 500, 600, 700, 800, 900, 1000], //freq array
    [0.05, 0.2, 0.04, 0.06, 0.11, 0.01, 0.15, 0.03, 0.15, 0.2]],
    PinkNoise.ar(MouseX.kr(0.01, 0.1)))
}.scope(1)
)

(
{
Klank.ar(
    `[Array.series(10, 50, 50),
      Array.series(10, 1.0, -0.1)],
    ClipNoise.ar(0.01)
)
}.scope(1)
)

(
{
Klank.ar(
    `[{exprand(60, 10000)}.dup(15)],
    PinkNoise.ar(0.005)
)
}.scope(1);
)
```

Chimes

In the additive synthesis chapter we built the characteristic ring of a bell by adding together enharmonic (random) frequencies. In this patch we approach a bell from the opposite direction; removing or filtering all but a set of random frequencies. The results are nearly identical. So what is the difference? When would you use additive and when subtractive? With additive, you have control over each sine wave. This can result in complex unearthly sounds, but at the price of CPU. Subtractive synthesis may sound more natural because you can't change each frequency. It is also a closer approximation to sounds in the wild. The

biggest reason is efficiency. Additive chimes may use up to 400 ugens and 10% CPU, but the *Klank* chime below uses only 20 ugens and 1% CPU.

The clapper striking the side of a bell is a burst of noise. (Try holding a bell so it doesn't resonate and strike it with the clapper. You just hear a short tick.) The physical construction of the bell or chime resonates certain frequencies. Those frequencies are sustained and all others die away quickly. The remaining frequencies are what make up the character of the bell or chime. So for this patch we begin with an extremely short burst of noise.

```
   15.7.  chime burst (Env, perc, PinkNoise, EnvGen, Spawn, scope)

(
{
var burstEnv, att = 0, burstLength = 0.0001, signal; //Variables
burstEnv = Env.perc(0, burstLength); //envelope times
signal = PinkNoise.ar(EnvGen.kr(burstEnv, gate: Impulse.kr(1))); //Noise burst
signal;
}.play
)
```

Try different noise sources such as *WhiteNoise*, *BrownNoise*, *ClipNoise*. Also try longer attacks and decay rates.

For a resonator we use *Klank.ar*, generating the arrays with the dup message. The input argument for *Klank* is the noise burst from above. Since in these examples we use a short burst as an exciter, rather than the steady state noise, decay is now in play, and the decay argument is used.

```
   15.8.  chimes (normalizeSum, round, Klank, EnvGen, MouseY)

(
{
var chime, freqSpecs, burst, totalHarm = 10;
var burstEnv, att = 0, burstLength = 0.0001;

freqSpecs = `[
   {rrand(100, 1200)}.dup(totalHarm), //freq array
   {rrand(0.3, 1.0)}.dup(totalHarm).normalizeSum.round(0.01), //amp array
   {rrand(2.0, 4.0)}.dup(totalHarm)]; //decay rate array

burstEnv = Env.perc(0, burstLength); //envelope times
burst = PinkNoise.ar(EnvGen.kr(burstEnv, gate: Impulse.kr(1))); //Noise burst

Klank.ar(freqSpecs, burst)*MouseX.kr(0.1, 0.8)
}.scope(1)
)
```

The number of harmonics, amplitudes, and decay rates are set with *totalHarm*. Each array in the *freqSpecs* array is filled automatically with values generated by the function and the *dup*

message. The variable *burst* is inserted as the second argument to *Klank.ar*, along with the *freqSpecs* array, and that result is played.

Of course, you can also "tune" the bell by using a harmonic series.

```
   15.9.  Tuned chime (or pluck?)

(
{
var chime, freqSpecs, burst, totalHarm = 10;
var burstEnv, att = 0, burstLength = 0.0001;

freqSpecs = `[
   {rrand(1, 30)*100}.dup(totalHarm),
   {rrand(0.1, 0.9)}.dup(totalHarm).normalizeSum,
   {rrand(1.0, 3.0)}.dup(totalHarm)];
burstEnv = Env.perc(0, burstLength);
burst = PinkNoise.ar(EnvGen.kr(burstEnv, gate: Impulse.kr(1)));

Klank.ar(freqSpecs, burst)*MouseX.kr(0.1, 0.8)
}.scope(1)
)
```

To fill the arrays I just pick a number between 1 and 30 and multiply that by a fundamental. There might be some duplicated harmonics. This sounds a little more like a plucked instrument, which is covered in the next chapter. If the harmonics are detuned slightly, it will sound a little more like a tuned bell. How would you detune the harmonics by + or – 10%?

You may have noticed there are several places where you can adjust the amplitude of the signal in a patch. The same is true with most electro-acoustic music studios. In this patch you can adjust the volume in the *EnvGen*, or at the *PinkNoise*, at the *Klank* or at the final *EnvGen*.

Try adjusting the total number of harmonics and *burstLength*. Try adding a variable for attack. A softer attack may represent a softer strike. Change the range of frequencies in the *freq* array. Change the range of choices for decay rates. Finally, try replacing any of the *freqSpecs* functions with your own array of frequencies. (You just have to make sure you enter the same number of frequencies as stipulated with *totalHarm*, since those arrays will be of that size.) Try odd harmonics, or varying degrees of detuned harmonics.

Debugging, commenting out, balancing enclosures, postln, postcln, postf, catArgs

How can you be sure the frequency collection is what you wanted (random, odd, even, detuned, etc.)? Likewise, how do you go about isolating problems in a patch that isn't doing what you want?

The first thing I do when something seems out of whack is isolate and evaluate small pieces of code. If you are careful you can select and run a single line or even a section of a line. The example below shows an entire line of code with smaller sections offset in bold. These

smaller sections can be selected and evaluated individually. Just select the bolded areas and press enter.

```
15.10. running a selection of a line

Array.fill(5, {exprand(10, 100)}).round(1) * ([1, 2, 5, 7].choose)
Array.fill(5, {exprand(10, 100)}).round(1) * ([1, 2, 5, 7].choose)
Array.fill(5, {exprand(10, 100)}).round(1) * ([1, 2, 5, 7].choose)
Array.fill(5, {exprand(10, 100)}).round(1) * ([1, 2, 5, 7].choose)
```

You have to be careful about commas and variables. For example, if you select and evaluate this section: "{rrand(100, 1200)}.dup(totalHarm)," you will get an error because of the comma and missing variable declaration for totalHarm. But if you replace the totalHarm with 10 and select everything but the comma, it can be evaluated separately and SC will print the results in the post window.

You can also isolate problems by replacing suspect parts of the code with safer values, and by "commenting out" a section or line of code[35]. Again, you have to be careful about semicolons and commas. Comments are identified by "//" to the end of the line or "/*" to begin the comment and "*/" to end the comment. If you suspect a problem with the freqSpecs arrays you could insert a single "safe" value in place of the random array: *{0.2 /*rrand(0.3, 1.0)*/}.dup(totalHarm)*. This will fill the entire array with 0.2 rather than the rrand(0.3, 1.0).

Select Syntax Colorize under the Format menu to colorize your code based on syntax. You can use this to see if your comments are correct. Comments should be red. Remember the program will ignore all commented items. In the example below I have the original line, then the same line with the round, multiply, and choose functions removed. Select and run both, then colorize.

```
15.11. running a selection of a line

Array.fill(5, {exprand(10, 100)}).round(1) * ([1, 2, 5, 7].choose)
Array.fill(5, {exprand(10, 100)}) /* .round(1) * ([1, 2, 5, 7].choose) */
```

I sometimes also use a sort of A/B switch that takes advantage of the order in which code is run: top to bottom. The "safe" section is something that I know works, which I place below the lines I suspect are causing problems. Since it is below, it replaces any values above, the "suspect" values. Turn the "A/B" switch on and off by commenting out the bottom line. Run the first example, then uncomment the "this I know works" and run it again. The sentence “this is experimental” is replaced.

```
15.12. commenting out

(
```

[35] See "comment", "uncomment", and "insert /* */" menu items under the Format menu.

```
a = "this is experimental";
// a = "this I know works";
a.postln
)

( // an example in real code
{
var chime, freqSpecs, burst, totalHarm = 10;
var burstEnv, att = 0, burstLength = 0.0001;

freqSpecs = `[ //"experimental" collection
   {rrand(100, 1200)}.dup(totalHarm), //freq array
   {rrand(0.3, 1.0)}.dup(totalHarm).normalizeSum.round(0.01), //amp array
   {rrand(2.0, 4.0)}.dup(totalHarm)]; //decay rate array

//freqSpecs = `[ //"safe" collection
// [100, 200] //freq array
// [0.5, 0.5] //amp array
// [0.1, 0.1]]; //decay rate array
```

When the second *freqSpecs* is commented it is not included in the patch. If you remove the comments then the safe section will replace the experimental section. (You don't have to comment out the experimental section, it is just reassigned.) If the patch then runs correctly, there is something wrong with the experimental part.

I also use this when I'm not getting any sound from several pages of complex code. To check to see if it's a system problem, or something in my code, I place a simple *SinOsc.ar* as the last line of the function and that line will be played while everything above it is ignored.

Another common error is mismatched enclosures. Com-shift-b "balances" enclosures by highlighting everything between the nearest outer enclosure. When it cannot find a match it gives an error flash. To locate the problem place the cursor inside an obviously correct enclosure (in the example above, perhaps the rrand(2.0, 4.0) and repeatedly press com-shift-b. It will balance as far as it can giving a system error when it can't.

The last debugging strategy uses post message to print the results of an operation to the post window. This is similar to selecting and evaluating a section. The difference is you can evaluate several places at once. There are intentional errors in the patch below. When you run it, it will explode into several screen's worth of error data. But at the top the first lines should reveal the items attached to the postln. Can you spot them by examining the posted data? (Hint: nil means no data has been assigned.)

```
   15.13. debugging using postln

(
{
var chime, freqSpecs, burst, totalHarm = 10;
var burstEnv, att = 0, burstLength;

freqSpecs = `[
```

```
    rrand(100, 1200).dup(totalHarm).postln,
    {rrand(0.3, 1.0)}.dup(totalHarm).normalizeSum.round(0.1).postln,
    {rrand(2.0, 4.0)}.dup(totalHarm).round(0.1).postln];

burstEnv = Env.perc(0, burstLength.postln); //envelope times
burst = PinkNoise.ar(EnvGen.kr(burstEnv, gate: Impulse.kr(1)));

Klank.ar(freqSpecs.postln, burst)*MouseX.kr(0.1, 0.8)
}.scope(1)
)
```

The postln message can be nested within a message chain, as shown in this example. You can remove them using find and replace.

```
    15.14. debugging using postln in message chains

(
[[60, 62, 68], [65, 67], [61, 63, 55, 23]]
.choose.postln
.midicps.postln
.round(0.1).postln
.reciprocal.postln
.round(0.0001).postln
)
```

The *post* message prints to the post window, *postln* prints the object then begins a new line, *postcln* precedes that line with a comment ("//"). This is useful if you are developing code in the post window. Your new post won't affect the code on subsequent trials.

If you use *post* repeatedly the information is merged into one blob. The obvious solution is to use *postln*, but that eats up the post window quickly. It is also difficult to examine related data if everything is on a separate line. Here are some methods for formatting posted information. The first is to collect everything in an array. The second is to use *catArgs*, which concatenates a string with a list of arguments, and finally *postf*, which has as its first argument a string containing text interspersed with the "%" character. Following are additional arguments that will be placed in the string where the "%" characters were (in order). Combined with the "\t" combination (for tab), your lines can be very readable. Programmers who have worked in C should recognize these conventions.

```
    15.15. Formatting posted information
(
// unintelligible
var pitch, duration, amplitude, voice;
20.do({
pitch = [60, 62, 64, 65, 67, 69, 71].choose.post;
duration = 4.0.rand.round(0.1).post;
amplitude = 1.0.rand.round(0.1).post;
voice = 10.rand.post;
})
)
```

```
(
// too spread out
var pitch, duration, amplitude, voice;
20.do({
pitch = [60, 62, 64, 65, 67, 69, 71].choose.postln;
duration = 4.0.rand.round(0.1).postln;
amplitude = 1.0.rand.round(0.1).postln;
voice = 10.rand.postln;
})
)


(
// cumbersome
var pitch, duration, amplitude, voice;
20.do({
pitch = [60, 62, 64, 65, 67, 69, 71].choose.post;
" ".post;
duration = 4.0.rand.round(0.1).post;
" ".post;
amplitude = 1.0.rand.round(0.1).post;
" ".post;
voice = 10.rand.postln;
})
)


(
// adequate
var pitch, duration, amplitude, voice;
"pitch, duration, amplitude, voice".postln;
20.do({
pitch = [60, 62, 64, 65, 67, 69, 71].choose;
duration = 4.0.rand.round(0.1);
amplitude = 1.0.rand.round(0.1);
voice = 10.rand;
[pitch, duration, amplitude, voice].postln;
})
)

(
// better?
var pitch, duration, amplitude, voice;
"pitch, duration, amplitude, voice".postln;
20.do({
pitch = [60, 62, 64, 65, 67, 69, 71].choose;
duration = 4.0.rand.round(0.1);
amplitude = 1.0.rand.round(0.1);
voice = 10.rand;
"P, D, A, V ".catArgs(pitch, duration, amplitude, voice).postln
})
)
```

```
(
// useful
var pitch, duration, amplitude, voice;
20.do({
pitch = [60, 62, 64, 65, 67, 69, 71].choose;
duration = 4.0.rand.round(0.1);
amplitude = 1.0.rand.round(0.1);
voice = 10.rand;
"pch %\tdur %\tamp %\tvce %\n".postf(pitch, duration, amplitude, voice)
})
)

(
// useful
var pitch, duration, amplitude, voice;
"pch\t\tdur\t\tamp\t\tvce".postln;
20.do({
pitch = [60, 62, 64, 65, 67, 69, 71].choose;
duration = 4.0.rand.round(0.1);
amplitude = 1.0.rand.round(0.1);
voice = 10.rand;
"%\t\t%\t\t%\t\t%\n".postf(pitch, duration, amplitude, voice)
})
)
```

Modifying the source code

Now we are getting into dangerous territory. Proceed with caution. The last trick I use requires you to modify the source code. The problem with the last two techniques is that when examining the data in the post window you're not sure which value is which. You could place strategic string posts, such as *"choice".postln*, *"array".postln*. But that wouldn't help with the message chain example. What you need is a post message that can take an argument, such as *postn(id)*, where id is an identifier. So that *45.postn("freq")* would result in *freq: 45*. No such message? Write your own.

Before we do the examples let me give a small word of warning. I find that I modify SC less and less, for two reasons. The first is that I work in labs with lots of machines. If I modify the source on my laptop then I have to modify the source on all the machines I will ever work on, otherwise, the code won't run. Similarly, if I ever update SC then I have to update those files. I've found, over time, that the modifications are not worth the work of keeping up with newer versions and lab machines. But that shouldn't stop you.

Open the String.sc and Object.sc files. In the String.sc file find the "post" sections of code and add a third line shown below:

```
postcln { "// ".post; this.postln; }
postc { "// ".post; this.post; }
postn {arg note; note.post; ": ".post; this.postln; }
```

Next, add this line to the Object.sc file:

```
postc { this.asString.postc }
postcln { this.asString.postcln; }
postn {arg note; this.asString.postn(note); }
```

Quit and relaunch SC, or choose Compile Library from the Lang menu. Now you can add a note to your posted items. You can also say you are part of the development team.

```
   15.16. postn

(
[[60, 62, 68], [65, 67], [61, 63, 55, 23]]
.choose.postn("choice")
.midicps.postn("midi conversion")
.round(0.1).postn("rounded value")
.reciprocal.postn("recip")
.round(0.0001).postn("rounded recip")
)
```

Practice, Chimes and Cavern

The patch below uses steady state noise and noise bursts as inputs for *Klank*. The chimes and cavern are mixed by simply adding them together. This patch uses lots of ugens and may max out your cpu. If so, reduce the number of harmonics, instruments, etc.

```
   15.17. Subtracitive Synthesis (Klank, Decay, Dust, PinkNoise, RLPF, LFSaw)

(
{
var totalInst, totalPartials, baseFreq, ampControl, chimes, cavern;
totalInst = 5; //Total number of chimes
totalPartials = 12; //Number of partials in each chime
baseFreq = rrand(200, 1000); //Base frequency for chimes

chimes =
   Mix.ar(
       {
       Pan2.ar(
           Klank.ar(`[
               {baseFreq*rrand(1.0, 12.0)}.dup(totalPartials),
               Array.rand(totalPartials, 0.3, 0.9),
               Array.rand(totalPartials, 0.5, 6.0)],
               Decay.ar(
                   Dust.ar(0.2, 0.02), //Times per second, amp
                   0.001, //decay rate
                   PinkNoise.ar //Noise
```

```
            )), 1.0.rand2) //Pan position
        }.dup(totalInst)
    );

cavern =
    Mix.ar(
        {
        var base;
        base = exprand(50, 500);
            Klank.ar(
                `[ //frequency, amplitudes, and decays
                   {rrand(1, 24) * base *
                       rrand(1.0, 1.1)}.dup(totalPartials),
                   Array.rand(10, 1.0, 5.0).normalizeSum
                ],
                GrayNoise.ar( [rrand(0.03, 0.1), rrand(0.03, 0.1)])
            )*max(0, LFNoise1.kr(3/rrand(5, 20), mul: 0.005))
        }.dup(5));
cavern + chimes
}.play
)

[New]

// Cavern variation
(
{
var totalPartials = 12;

    Mix.ar(
        {
        var base;
        base = exprand(50, 1000);
        Pan2.ar(
            Klank.ar(
                `[ //frequency, amplitudes, and decays
                   {rrand(1, 24) * base *
                       rrand(1.0, 1.1)}.dup(totalPartials),
                   Array.rand(10, 1.0, 5.0).normalizeSum
                ],
                GrayNoise.ar( rrand(0.03, 0.1))
            )*max(0, LFNoise1.kr(6, mul: 0.005)),
            LFNoise0.kr(1))
        }.dup(5));

}.play
)

// Rotating wheels

{
var totalPartials = 4;
```

```
    Mix.ar(
        {
        var base;
        base = exprand(50, 10000);
        Pan2.ar(
            Klank.ar(
                `[ //frequency, amplitudes, and decays
                   {rrand(1, 24) * base *
                       rrand(1.0, 1.1)}.dup(totalPartials),
                   Array.rand(10, 1.0, 5.0).normalizeSum
                ],
                GrayNoise.ar( rrand(0.03, 0.1))
            )*max(0, SinOsc.kr(6/rrand(1, 10), mul: 0.005)),
            LFNoise1.kr(1))
        }.dup(8));

}.play


// This one floats in and out, so if there is no
// sound let it run a while.
// Execute it four or five times to get a bunch
// of them going.

{
var totalPartials = 3;

    Mix.ar(
        {
        var base;
        base = exprand(50, 100);
        Pan2.ar(
            Klank.ar(
                `[ //frequency, amplitudes, and decays
                   {rrand(1, 24) * base *
                       rrand(1.0, 1.1)}.dup(totalPartials),
                   Array.rand(10, 1.0, 5.0).normalizeSum
                ],
                GrayNoise.ar( rrand(0.03, 0.1))
            )*max(0, SinOsc.kr(10/rrand(1, 5), mul: 0.005)),
            LFNoise1.kr(1))
        }.dup(8)) * max(0, LFNoise1.kr(1/10));

}.play
```

15. Exercises

15.1. Write a patch using an RLPF with a 50 Hz Saw as its signal source. Insert an LFNoise0 with frequency of 12 to control the frequency cutoff of the filter with a range of 200, 2000.

15.2. Write a patch similar to "tuned chimes" using Klank with PinkNoise as its input source. Fill the frequency array with harmonic partials but randomly detuned by + or – 2%. In other words, rather than 100, 200, 300, etc., 105, 197, 311, 401, etc. (Multiply each one by rrand(0.98, 1.02)).

15.3. Without deleting any existing code, rewrite the patch below with "safe" values to replace the existing trigger with a 1 and the fund with 100.

```
({var trigger, fund; trigger = Dust.kr(3/7); fund = rrand(100, 400);
Mix.ar(
    Array.fill(16,
    {arg counter; var partial;  partial = counter + 1;
    Pan2.ar(
            SinOsc.ar((fund*partial), mul: 0.7) *
            EnvGen.kr(Env.adsr(0, 0, 1.0, 5.0), trigger, 1/partial
            ) * max(0, LFNoise1.kr(rrand(5.0, 12.0))),
            1.0.rand2)
    }))*0.5;
}.play)
```

15.4. In the patch above monitor the frequency arguments only of all the SinOscs, LFNoise1s, and the pan positions by only adding postlns.

15.5. In the patch above comment out the EnvGen with all its arguments, but not the max(). Be careful not to get two multiplication signs in a row (* *), which means square. Check using syntax colorize under the format menu.

15.6. Without deleting, replacing any existing code, or commenting out, replace the entire Mix portion with a "safe" SinOsc.ar(400).

16 - FM/AM Synthesis, Phase Modulation, Sequencer, Sample and Hold

AM and FM synthesis or "Ring" Modulation

In a previous chapter I created a vibrato with voltage control; using one oscillator in low frequency to control another in audio range. At that time I warned you not to exceed 20 Hz for the control frequency. Hopefully you didn't take my suggestion and experimented with higher frequencies. Of course you can have control rates above 20 Hz, but something unexpected, almost magical occurs: new frequencies appear above and below the one being controlled. The sounds you get with a control source above LFO range are similar to tuning a radio. That's because you are entering the realm of frequency and amplitude modulation; the same technology used by am and fm bands on your radio to transmit signals through the air. (It also sounds surprisingly similar to bird song. Some birds can produce two frequencies that interact in the same way.)

```
16.1.  From LFO to FM

{SinOsc.ar(SinOsc.ar(MouseX.kr(1, 500), mul: 300, add: 800))}.play
```

What distinguishes am and fm from LFO are upper and lower sidebands. They are additional frequencies that appear as a product of the two modulated frequencies.

In amplitude modulation there are two sidebands; the sum and difference of the carrier frequency (the audio frequency that is being modulated) and the modulator frequency (the frequency that is controlling the audio frequency). A carrier frequency of 500 and a modulating frequency of 112 could result in two sidebands: 612 and 388. If there are overtones in one of the waves (e.g. a saw wave being controlled by a sine wave), then there will be sidebands for each overtone.

```
16.2.  AM Synthesis (SinOsc, scope, mul, Saw)

{SinOsc.ar(500, mul: SinOsc.ar(50, mul: 0.5))}.scope(1)

{Saw.ar(500, mul: SinOsc.ar(50,  mul: 0.5))}.scope(1)
```

In the example above the outer SinOsc is the carrier and the inner SinOsc is the modulator. The sidebands should be 550 and 450. Change the argument 50 to other values to see how it changes the sound. Replace the 50 with a *MouseX.kr(1, 500)*. As you move the mouse from left slowly to the right you should hear two sidebands (frequencies); one ascending and the other descending.

In frequency modulation a similar effect takes place. But with FM many more sidebands can be generated. The exact number depends on the modulation index. The modulation index is how far the modulating frequency deviates from the carrier (the amplitude of the modulating wave). In SC it is a little more difficult to recognize the components of FM synthesis because

both the carrier and modulator frequencies can appear as arguments in a single SinOsc. For clarity I've used this form: *400 + SinOsc(124, mul: 100)*. The 400 is the carrier frequency, 124 is the modulator frequency and 100 is the index. (A higher index results in more sidebands.)

```
   16.3.  FM Synthesis

{SinOsc.ar(400 + SinOsc.ar(124, mul: 100), mul: 0.5)}.scope(1)
```

Phase Modulation

With the *PMOsc* object it is easier to see carrier, modulator, and modulation index, since they are the first three arguments. Phase modulation controls the phase of the carrier rather than the frequency. The differences are academic, and the effect is nearly the same as FM. Try replacing each of the arguments below (400 is the carrier frequency, 124 is the modulator frequency, 1 is the index in radians) with a *MouseX* and *MouseY* to better understand how they correlate with the sidebands.

```
   16.4.  PM Synthesis

{PMOsc.ar(400, 124, 1, mul: 0.5)}.scope(1)
```

The ratio of the carrier and control frequency determine the sidebands and hence the quality or timbre, the index controls the number of sidebands, i.e. the brightness of the wave. Different frequencies of the carrier (i.e. changing the carrier while the modulator frequency and index remain the same) will sound like one instrument with a changing pitch. Different frequencies of the modulator only will sound like different instruments playing the same pitch. Different indexes will sound like one instrument being filtered. Of course, you can change all three. In short: carrier = pitch, modulator = timbre, index = filter.

```
   16.5.  Controls for carrier, modulator, and index

(
{PMOsc.ar(LFNoise0.kr(5, 300, 700),// carrier
   134, 4, mul: 0.4)
   }.scope(1)
)

(
{PMOsc.ar(700,
       LFNoise0.kr(5, 500, 700),//modulator
       12, mul: 0.4
   )}.scope(1)
)

(
{PMOsc.ar(700, 567,
       LFNoise0.kr(5, 6, 12),  //index
```

```
      mul: 0.4
   )}.scope(1)
)

(
// All three. This is the type of sound that
// got me hooked on synthesis in the first place.

{PMOsc.ar(LFNoise0.kr([9, 9], 300, 700),
      LFNoise0.kr([9, 9], 500, 700),
      LFNoise0.kr([9, 9], 6, 12),
      mul: 0.5
   )}.scope(1)
)
```

The sidebands are the sum and difference of the carrier and modulator wave. Given a carrier of 400 Hz and a modulator of 50 Hz, the sidebands would be 450, 500, 550, 600, 650, etc., depending on the modulation index, and 350, 300, 250, 200, etc., depending on the modulation index. If the modulation index is high enough to generate negative numbers on the lower spectrum, they wrap around. A quick SC formula for calculating 30 sidebands above and below a given frequency would be *abs((-30..30)*m+c)* where m is the modulator and c is the carrier frequency. The example above would be *abs((-30..30)*50+400)*.

While it's important to understand how sidebands are produced and what those sidebands are in relation to carrier, modulator, and index, I have to say I've never had to actually do those calculations in real life. And by real life I mean my experiments and compositions. What I do remember are these three things, as they relate to how the sound changes: carrier = fundamental pitch, modulator = character of the wave, index = number of partials or filter.

A similar result could be achieved through additive synthesis; 20 sine waves each tuned to the frequencies listed above. A major advantage to FM is efficiency. It uses only two sine waves to generate literally hundreds of sidebands, which are equivalent to sine waves. Compare the two patches below, one using phase modulation, the other additive synthesis. Note the number of ugens and CPU load for each.

```
   16.6.  Efficiency of PM

{PMOsc.ar(1000, 1367, 12, mul: EnvGen.kr(Env.perc(0, 0.5), Impulse.kr(1)))}.play

(
{
Mix.ar(
   SinOsc.ar(abs((-20..20)*1367 + 1000),
      mul: 0.1*EnvGen.kr(Env.perc(0, 0.5), Impulse.kr(1)))
)}.play
)
```

The sidebands are heard as overtones of a single pitch. This raises the issue of harmonic and inharmonic spectra from an earlier chapter. If the sidebands are related by a low ratio, then

they will sound more harmonic. For this reason it is useful to calculate the modulating frequency as a ratio of the carrier frequency. In the patch below, change the ratio to whole numbers such as 1, 2, 3, 4, then ratios such as 3/2, 5/4, 8/7, then try numbers such as 2.317, 1.576, etc. A near round value, such as 2.01, will detune a harmonic spectrum slightly. Since the modulating frequency ratio stays the same we hear a single timbre playing different pitches. The modulation index is still controlled by the *MouseY*. Changing the index with *MouseY* changes the brightness but not the timbre because it adds and removes sidebands. The other advantage of a carrier that is a ratio of the modulator is that if the carrier frequency changes, so will the modulator, and the collection of sidebands will remain the same in relation to the carrier. The practical result is that we here the same timbre, even though the pitch changes. Notice I am using MIDI numbers for frequency.

```
   16.7.  Carrier and modulator ratio

(
{var freq, ratio;
freq = LFNoise0.kr(4, 20, 60).round(1).midicps;
ratio = 2/1;
PMOsc.ar(
   freq, //carrier
   freq*ratio, //modulator
   MouseY.kr(0.1, 10), //index
   Mul: [0.4, 0.4]
)}.play
)
```

The next example adds an envelope to control the amplitude of the *PMOsc*, but it also controls the modulation index so that the number of sidebands decay as the sound decays. The *TRand* is generating values between 36 and 86, appropriate for MIDI. Since the arguments are integers and not floats, it will return integers, which in terms of MIDI constrains the pitches to the equal tempered scale. (I'm not avoiding notes between the keys, just not what I wanted here. Go ahead and change it.). Again, try unusual values for the ratio to change the character of the instrument.

```
   16.8.  Envelope applied to amplitude and modulation index

(
{var freq, ratio, env, rate = 5, trig;
trig = Impulse.kr(5);
freq = TRand.kr([36, 60], [72, 86], trig).midicps;
ratio = 2;
env = EnvGen.kr(Env.perc(0, 1/rate), gate: trig);
PMOsc.ar(
       freq,
       freq*ratio,
       3 + env*4,
       mul: env
)}.play
)
```

FM synthesis was developed by John Chowning during the 60s and used extensively in the 80s, specifically in the Yamaha DX7 series. Its success comes from its efficiency; using only two waves to generate theoretically unlimited sidebands of harmonic or inharmonic spectra.

Sequencer

In each chapter I have introduced a few common control sources. We used a *SinOsc* as an LFO to control the frequency of another *SinOsc*. We used *LFNoise*, envelopes, *Line*, and other wave shapes as controls. There are two more classic controls I would like to illustrate. The first is a sequencer. The sequencer moves through a set of values at a given rate or trigger. This is done using a combination of *Select* and *Stepper*.

The sequence is defined as an array. The sequencer can be used to control any aspect of sound. In this example it is used to control frequency using MIDI pitch numbers. You can use any set of frequencies or midi numbers (i.e. other tunings), or even random frequencies. You can also fill the array automatically using *Array.fill, rand, series*, etc., or modify the array in a number of ways (illustrated below).

```
   16.9.  Sequencer (array, midicps, SinOsc, Sequencer, Impulse, kr)

(
var pitchArray; //Declare a variable to hold the array
   //load the array with midi pitches
pitchArray = [60, 62, 64, 65, 67, 69, 71, 72].midicps;
{
   SinOsc.ar(
      Select.kr(
         Stepper.kr(Impulse.kr(8), max: pitchArray.size-1),
         pitchArray),
      mul: 0.5)
}.play
)

(
var pitchArray; //Declare a variable to hold the array
   //load the array with midi pitches
pitchArray = Array.rand(24, 100, 2000);
{
   SinOsc.ar(
      Select.kr(
         Stepper.kr(Impulse.kr(8), max: pitchArray.size-1),
         pitchArray),
      mul: 0.5)
}.play
)
```

You can modify the array using *reverse*, *scramble*, or *fill*. Consider the lines below. First an array is filled with random numbers between 60 and 84. *Array.rand* takes three arguments; first the number of items in the array, then the low and high ends of the random range. Next

we scramble the array and post it. Then reverse it and post it. Last we add 12 to the entire array and post it.

```
   16.10. scramble, reverse (Array.rand, postln, scramble, reverse)

(
var pitchArray;
pitchArray = Array.rand(10, 60, 84);
pitchArray.postln.scramble.postln.reverse.postln;
(pitchArray + 12).postln
)
```

Here are examples of each process in an actual patch. Try changing the pitch array to a list of other scale degrees.

```
   16.11. sequencer variations

(
var pitchArray;
pitchArray = [60, 62, 64, 65, 67, 69, 71, 72];
pitchArray = [ //pick one of the following
   (pitchArray + rrand(1, 12)).midicps, //transpose
   pitchArray.reverse.midicps, //reverse
   pitchArray.scramble.midicps, //scramble
   Array.rand(12, 36, 72).midicps, //random midi
   Array.rand(12, 100, 1200) //random frequency
   ].choose;
{
   SinOsc.ar(
      Select.kr(
         Stepper.kr(Impulse.kr(7), max: pitchArray.size-1),
         pitchArray),
      mul: 0.5)
}.play
)

// [New] a more interesting example using PM

(
{
var freq, freqArray, ratioArray, indexArray,
   env, rate = 5, trig;
trig = Impulse.kr(rate);
freqArray = [48, 50, 52, 53, 55, 57, 59, 60,
          62, 64, 65, 67, 69, 71, 72].scramble.midicps;
ratioArray = {rrand(1.0, 3.0)}.dup(20);
indexArray = {rrand(1.0, 4.0)}.dup(20);
env = EnvGen.kr(Env.perc(0, 1/rate), gate: trig);
freq = Select.kr(Stepper.kr(trig, max: freqArray.size-1), freqArray);
PMOsc.ar(
      freq,
      freq*Select.kr(Stepper.kr(trig, max: ratioArray.size-1), ratioArray),
```

```
        Select.kr(Stepper.kr(trig, max: indexArray.size-1), indexArray)
         + env*4,
        mul: env
)}.play
)
```

Stepper has an argument to control how it advances through the array. A value of 1 will move forward one index at a time. If that value is negative, it will move backward one step at a time. If it is 3, it will move three steps.

Sample and Hold

Sample and hold is another classic synthesis control source available on the old modular synthesizers. It didn't survive the transition from modular to integrated units. But I think the effect is worth preserving. The SC equivalent to a sample and hold is *Latch.kr*.

A sample and hold uses a periodic wave as an input. The wave is sampled at regular intervals. The resulting values are used as a control. It "samples" the wave and "holds" that value until a new value is sampled. It can be thought of as an analog to digital converter with a low frequency sampling rate (though technically the wave being sampled is also digital). The effect is similar to a strobe light or motion picture film taking snapshots of a smooth process. That smooth process is then quantized into discrete steps.

How is this useful in synthesis? Why would you want to freeze-frame a wave form? Because if you sample a periodic wave with a periodic rate the results are constantly evolving patterns. The wave being sampled can generate aliases, similar to the aliasing discussed in chapter 6. Take that same example we used of watching the second hand of a clock, opening your eyes at a given "sample" rate. If you open your eyes on time per second, you get these numbers: [1, 2, 3, 4, 5, 6, 7, …]. Not very interesting. What if we "sampled" once every 5 seconds. [5, 10, 15, 20, …] Better, but still pretty predictable. But when the sample rate is near the rate of the clock revolutions, interesting patterns result. Take, for example, a sample rate of 46. What numbers would result? SC can tell us: Array.fill(20, {arg i; i*46}).mod(60); = [0, 46, 32, 18, 4, 50, 36, 22, 8, 54, 40, 26, 12, 58, 44, 30, 16, 2, 48, 34]. Are they random? No, because the sample rate and series of numbers are both periodic, so there is, by definition, a pattern. Its just a more interesting pattern than a sequencer or random walk.

The idea is that even though the sample rate is too low to accurately represent the true shape of the wave, patterns will still emerge because the wave is periodic, and the sample rate is periodic. Now imagine a saw wave that moves from 0 to 1 once every second, returning to 0 at each second. If the sample rate for that wave were 10 times per second the actual values returned would be 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.0, 0.1, 0.2 etc. The shape of the saw is evident with a sample rate of 10 times per second. But what if the sample rate were 0.9 times per second (9 samples in 10 seconds)? That is to say the amount of time between each sample, or opening your eyes to see the second hand, is 0.9 seconds. The first value would be 0, then 0.9, but the next, taken 0.9 seconds later, would be 0.8, then 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0, then 0.9 again, and so on. They look like multiples of 9, but since the saw

wave resets to 0 each second the values never go above 1 (like a modulo). So even though the original wave was an ascending ramp, these samples describe a descending ramp over the space of 10 seconds[36]. If the sample rate were 1.65, then the actual values returned would be 0, 0.65, 0.3, 0.95, 0.6, 0.25, 0.9, 0.55, 0.2, 0.85, 0.5, 0.15, 0.8, 0.45, 0.1, 0.75, 0.4, 0.05, 0.7, etc. (i.e. multiples of 1.65 but never above 1)[37]. This series isn't even close to the original saw, but there is a clear pattern, actually, three interleaved patterns. These array examples show some of the patterns that emerge from two periodic sources. The 100 is, in essence, the periodic wave, the 95, 124, 165 are the sampling rates. (Don't think about this example too long. It's kind of confusing. Just look at the resulting patterns.)

```
   16.12. Latch patterns

Array.fill(100, {arg i; i*95}).mod(100);

Array.fill(100, {arg i; i*124}).mod(100);

Array.fill(100, {arg i; i*165}).mod(100);
```

I like a sample and hold because it strikes a nice balance between variation and repetition. The sequences that emerge are more varied than a single sequence, but more cohesive than an *LFNoise* control.

Try the example below and listen for the shape of the Saw wave in the frequencies. In the paragraph above I changed the sample rate to produce a different pattern, but in this patch I change the frequency of the wave being sampled. (Shortening the wave rather than lengthening the sample time.) The effect on the patterns is the same, but this way the rate remains the same. I use a *MouseX.kr* to control the frequency of the sampled *LFSaw*. It begins at 0.1, or once every ten seconds and then moves to 20. When the mouse is at the left of the screen you should hear the shape of the wave in the resulting frequency control. As you move to the right, you are essentially shortening the sampled wave in relation to the sample rate, and the smooth ramp will begin to disappear, but there will always be a pattern. The second example replaces the mouse control with a *Line.kr*, which illustrates nicely the gradual transition from one pattern to the next.

```
   16.13. Latch

(
{
SinOsc.ar(
```

---

[36] If this sounds familiar, it is because the same process causes aliasing; ghost frequencies which occur when a signal is digitized at a sampling rate below the frequencies of the source. In this case the resulting "patterns" are heard as frequencies.

[37] The code to reproduce this series is *(0, 1.65..20).mod(1).round(0.01)*

```
    Latch.kr(
        LFSaw.kr(MouseX.kr(0.1, 20), 0, 500, 600),
        Impulse.kr(10)),
    mul: 0.3 //Volume of Blip
)
}.scope(1)
)

(
{
SinOsc.ar(
    Latch.kr(
        LFSaw.kr(Line.kr(0.1, 20, 60), 0, 500, 600),
        Impulse.kr(10)),
    mul: 0.3 //Volume of Blip
)
}.scope(1)
)
```

As the relationship between the frequency of the wave and the rate of sample changes you hear different patterns, some obvious, others surprising. But any two values will result in some kind of pattern since both the trigger and the wave are periodic.

The *Latch* can be used to control amplitude, frequency of modulator or carrier, or index. The third example scales and offsets the control to work with all three. The right channel is delayed by two seconds for greater variety.

```
    16.14. Latch

(
{ // controlling modulator and carrier freq
f = Latch.kr(
        LFSaw.kr(MouseX.kr(1.1, 30), 0, 1000, 1100),
        Impulse.kr(10));
PMOsc.ar([f, f*3/2], f*2, 12, mul: 0.3)
}.scope(2)
)

(
{  // controlling index
i = Latch.kr(
        LFSaw.kr(MouseX.kr(1.1, 30), 0, 5, 5),
        Impulse.kr(10));
PMOsc.ar([300, 350], 356, i, mul: 0.3)
}.scope(2)
)

(
{  // controlling ratio
r = Latch.kr(
        LFSaw.kr(MouseX.kr(1.1, 30), 0, 2.0, 3.0),
        Impulse.kr(10));
```

```
PMOsc.ar(300, [300*r, 400*r], 12, mul: 0.3)
}.scope(2)
)

(
{  // and of course, scaled to control all three
c = Latch.kr(
       LFSaw.kr(MouseX.kr(1.1, 30), 0, 0.5, 0.5),
       Impulse.kr(10));
f = c*1000+200;
o = PMOsc.ar(f, f*(c*3+4), c*5+6, mul: 0.3);
[o, CombL.ar(o, 2, 2)]
}.scope(2)
)
```

In a previous chapter on intervals we saw that it is the ratio of the two frequencies that determines the relative complexity or simplicity of the destructive and constructive interference patterns, and therefore the level of consonance or dissonance. The same is true for a S&H. The ratio between the frequency of the sampled wave and the sample rate will determine the patterns. If we linked the two values the same pattern will persist when the rate is changed. In the patch below try replacing the *MouseX* with explicit ratios. I find values around 1.6 to 1.75 are pleasing (the golden mean, in case you're one of *those* guys, is 1.61803399).

The second example adds one more control so the ratio of the samples will gradually move through interesting patterns.

16.15. Latch sample and speed ratio (Blip, Latch, LFSaw, Impulse, mul)

```
(
{
var rate, speed, ratio;
rate = MouseX.kr(1, 24);
ratio = 1.61803399;
c = Latch.kr(
       LFSaw.kr(rate*ratio, 0, 0.5, 0.5),
       Impulse.kr(rate));
f = c*1000+200;
o = PMOsc.ar(f, f*(c*3+4), c*5+6, mul: 0.3);
[o, CombL.ar(o, 2, 2)]
}.scope(1)
)

(
{
var rate, speed, ratio;
rate = MouseX.kr(1, 24);
ratio = SinOsc.ar(2/7, mul: 0.2, add: 1.75);
c = Latch.kr(
       LFSaw.kr(rate*ratio, 0, 0.5, 0.5),
       Impulse.kr(rate));
```

```
f = c*1000+200;
o = PMOsc.ar(f, f*(c*3+4), c*5+6, mul: 0.3);
[o, CombL.ar(o, 2, 2)]
}.scope(2)
)
```

Another method for generating interest is to sample a more complex waveform. Even though it is complex, if it is periodic, patterns in the wave will be evident in the control values. Take the following wave as an example. Three waves are mixed together to create a more complex, but still periodic, wave. The first example below shows four waves at 100, 200, 300, and 550 Hz, mixed to one wave. The second part places the mixed wave into a Latch.

```
   16.16. Complex Wave as Sample Source (Mix, SinOsc, Blip, Latch, Mix, Impulse)

{Mix.ar(SinOsc.ar([100, 200, 300, 550], mul: 0.1))}.scope(1)

(
//Used in a sample and hold
{
f = Latch.kr(
       Mix.ar(SinOsc.ar([100, 200, 300, 550], mul: 100, add: 110)),
       Impulse.kr(7));
e = EnvGen.kr(Env.perc(0, 0.2), Impulse.kr(7));
PMOsc.ar(f, f*1.25, e*5, mul: e*0.3)
}.play
)
```

Practice S&H FM

In case you haven't guessed I really like the results of an S&H. We should use it more. I can't think of any other method of organization that generates such evolving patterns. In the example below try an attack of 1/rate and a decay of 0. Try the alternate *latchrates*, which move either gradually or stepwise through random values between 1.53 and 1.63. The patch is expanded to stereo with the *[freq, freq * 1.5]*. This creates a fifth in the right channel. Try other intervals such as 1.2, 1.7, 2, 1.25, etc. Uncomment the alternative latchrates, frequency, and ratios, or add your own.

The key is to find that level of complexity that keeps your interest; that sweet spot where you can still follow the emerging patterns, yet have enough variety to keep your attention for long periods of time. Isn't that really what composition is all about; a blend of predictability and surprise?

The second example does not use an S&H or sequencer (maybe you could add one?), but illustrates PM. It is a variation on one of James McCartney's original SC2 examples. James Harkins then ported it to SC3 and I've made further modifications. It generates PM events with complex ratios, therefore inharmonic spectra (like a crotale) with a collection of envelopes for amplitude and modulation index.

The code that actually generates crotale events is a routine that uses random functions, but remember it gets values from a random number generator, as described earlier. Remember that you can reset the seed of the random number series, reproducing precisely the sequence of values. The third task does this, and the occasional repetition gives a sense of cognizant improvisatory gestures. It sends "seeds" periodically, either chosen from the clock (70 percent of the time), or repeating the previous seed (30%). You can think of this task as a conductor, instructing the performer to either move on to something new (from the collection of billions of possible gestures) or repeating something you already did.

Try both examples at once. In which case you might want to adjust the overall volume of the first one.

```
   16.17. Practice, Sample and Hold, FM

(
{var freq, latchrate, index, ratio, env, rate;
rate = 9;
latchrate = rate*1.61803399;
// latchrate = rate*LFNoise1.kr(1/7, mul: 0.03, add: 1.6);
// latchrate = rate*LFNoise0.kr(1/3, mul: 0.03, add: 1.6);

index = Latch.kr(
   LFSaw.kr(latchrate, mul: 4, add: 8),
   Impulse.kr(rate)
   );
freq = Latch.kr(
   LFSaw.kr(latchrate, mul: 36, add: 60),
   Impulse.kr(rate)
   ).round(1).midicps;
// freq = 200; //uncomment this line to hear just the index

ratio = 2.01;
// ratio = LFNoise1.kr(1, mul: 2.3, add: 3.0);
// ratio = LFNoise0.kr(1, mul: 2.3, add: 3.0);
// ratio = LFNoise1.kr(1/5, mul: 2.0, add: 5.0);

env = EnvGen.kr(Env.perc(0, 2/rate), gate: Impulse.kr(rate));

PMOsc.ar([freq, freq * 1.5],
       [freq*ratio, freq*1.5*ratio],
       index,
       mul: env*0.5
)}.play
)

// Variation. I love this example.

(
{var freq, latchrate, index, ratio, env, rate;
rate = 9;
latchrate = rate*LFNoise0.kr(1/10, mul: 0.03, add: 1.6);
index = Latch.kr(
```

```
    LFSaw.kr(latchrate, mul: 5, add: 6),
    Impulse.kr(rate)
    );
freq = Latch.kr(
    LFSaw.kr(latchrate,
        mul: max(0, LFNoise1.kr(1/5, 24, 10)),
        add: LFNoise0.kr(1/7, 12, 60)),
    Impulse.kr(rate)
    ).round(1).midicps;
ratio = LFNoise1.kr(1/10, mul: 2.0, add: 5.0);

env = EnvGen.kr(Env.perc(0, LFNoise0.kr(rate, 1, 1.5)/rate), Impulse.kr(rate),
LFNoise1.kr([5, 5], 2, 1).max(0).min(0.8));
PMOsc.ar(
    [freq, freq * 1.5],
    freq*ratio,
    index,
    mul: env //overall volume
)}.play
)


// Crotale
// Run this code first to define the instrument,
// then execute the "task" below as many times
// as you want.

(
SynthDef("crotale", {
arg param = #[500, 3, 2, 0, 6, 5, 0, 0.9];
var factor, env, out, freq, index, dur;
var bus, ratioa, ratiob, attack, decay, panCont;
freq = param.at(0); index = param.at(1); dur = param.at(2);
bus = param.at(3); ratioa = param.at(4); ratiob = param.at(5);
attack = param.at(6); decay = param.at(7);

env = Env.perc(attack, decay);
factor = gcd(ratioa, ratiob);
ratioa = div(ratioa, factor);
ratiob = div(ratiob, factor);

panCont = (EnvGen.kr(env, timeScale: dur*1.1,
    levelBias: -1, levelScale: 2))
           * (IRand(0, 1) * 2 - 1);  // 0*2-1 = -1, 1*2-1 = 1

out = PMOsc.ar(
    ratioa*freq, //or try ratioa*freqCont,
    ratiob*freq, //or try ratioa*freqCont,
    pmindex: EnvGen.kr(env, timeScale: dur,
        levelBias: 1, levelScale: index),
    mul: EnvGen.kr(env, timeScale: dur, levelScale: 0.3));

out = Pan2.ar(out, panCont);
```

```
out = out * EnvGen.kr(env, timeScale: 1.3*dur,
   levelScale: Rand(0.1, 0.5), doneAction:2);
Out.ar(0, out); //Out.ar(bus, out);

}).play;
)

// Define the instrument by running the code
// above, then run this task to actually play
// the instrument.

(
r = Task({
   var freq, indexDepth, indexRange, synthIndex, dur, repeat;
   var next, count, countDown, offset, ratioa, ratiob, envs, env;
   var range = 60, outBus = 0;
   count = 0; countDown = 0; offset = 0;
   envs = [[0, 0.9], [0.01, 0.9], [0.1, 0.8], [0.8, 0.01]];
   repeat = Array.fill(10,
      {[rrand(range, range+24).round(1).midicps, 3,
         2.1 - exprand(0.1, 2.0), 0, 1, 1, 0, 0.9]});
   next = Array.fill(10, {[3, 0.75, 0.5, 0.25, 0.125].choose});
   freq = rrand(range, range*2); // these two are just starting points
   indexDepth = 1;

   inf.do({
      if(countDown <= 0,
         {
         env = envs.choose;
         next.put(count%10, [3, 0.5, 0.25, 0.125, 0.125].choose);
         repeat.put(count%10, [
            rrand(range, range + 24).round(1).midicps,
            rrand(0.1, 12.0),
            2.1 - exprand(0.1, 2.0), outBus, rrand(1, 12),
            rrand(1, 12), env.at(0), env.at(1)]);
         });

         Synth("crotale").setn(\param, repeat.wrapAt(count));
         next.wrapAt(count).wait;
         if((count > 10).and(countDown <= 0),
            {offset = countDown = [0, 3.rand, 6.rand].choose;
            count = count - offset});
   count = count + 1;
   countDown = countDown - 1;
   });
}).play(SystemClock);
)
```

16. Exercises

16.1. What are the AM sidebands of a carrier at 440 Hz and a modulator of 130?

16.2. What are the PM sidebands of a carrier at 400 Hz and a modulator at 50?

16.3. Create a PM patch where the modulator frequency is controlled by an LFNoise1 and the modulation index is controlled by an LFSaw ugen.

16.4. Create a PM patch where the carrier frequency, modulator frequency, and modulation index are controlled by separate sequencers (with arrays of different lengths) but all using the same trigger.

16.5. Create a PM patch where the carrier frequency is controlled by a TRand, the modulator is controlled by an S&H, and the index a sequencer, all using the same trigger.

17 - Karplus/Strong, Synthdef, Server commands

Like the chime and *Klank*, the Karplus/Strong "pluck" is a physical model that begins with noise. It models a plucked string, which when set in motion by the chaotic pluck of a finger or pick, reproduces that event as it swings from one side to the other.

In an earlier chapter we created a periodic wave from noise by copying a millisecond sample and pasting into a new file over and over. Karplus-Strong achieves the same effect using a delay ugen. A burst of noise is released into an echo chamber and the repeating and decaying echoes comprise the periodic repetition of the burst, which if short enough in duration is then perceived as a pitch.

Karplus-Strong Pluck Instrument

We begin with a very short burst of noise as in the chime.

```
   17.1.  noise burst

(
{
   var burstEnv, att = 0, dec = 0.001; //Variable declarations
   burstEnv = EnvGen.kr(Env.perc(att, dec), gate: Impulse.kr(1)); //envelope
   PinkNoise.ar(burstEnv); //Noise, amp controlled by burstEnv
}.play
)
```

Delays

The next step is to send the burst of noise through an echo chamber; *CombL*, which has these arguments *in, maxdelaytime, delayTime, decayTime, mul, add*. The input is going to be the burst of noise we just created. The *delaytime* and *maxdelaytime* are the same for this example. They represent the amount of time, in seconds, the signal is delayed (the echo). The *decaytime* is how long it takes for the echo to die away. Try changing the *delayTime* and the *decayTime*. The gate frequency is set to the reciprocal of the duration *delayDecay* so a new impulse triggers another burst of noise when the previous one has died away.

```
   17.2.  Noise burst with delay

(
{
   var burstEnv, att = 0, dec = 0.001;
   var out, delayTime = 0.5, delayDecay = 10;
   burstEnv = EnvGen.kr(Env.perc(att, dec), gate: Impulse.kr(1/delayDecay));
   out = PinkNoise.ar(burstEnv);
   out = out + CombL.ar(
      out,
      delayTime,
      delayTime,
```

```
      delayDecay); //Echo chamber
   out
}.play //End Ugen function
)
```

Why use noise as an excitation source? Notice that each pluck sound is a little different. This is because each new burst of noise is slightly different than the previous. (Not the echoes, but the initial burst.) The result is subtle, but natural variations to the sound. In the example below I add a *RandSeed* with the same trigger as the noise burst. When uncommented, this seed will reset the number generator and you will get the same burst of noise each time. Notice how these plucks all sound alike.

Change the delay time to 0.1 (ten times per second, or 10 Hz), 0.01 (100 times per second, or 100 Hz), then 0.001 (1000 Hz), etc. The delay time is the reciprocal to the pitch we hear (1/100th of a second, 100 Hz). What delay time would we enter for A 440?

Notice that we have a similar problem as discussed in previous chapters regarding duration and frequency. Since we are creating a pitched event, we think in terms of frequency. But the delay argument in *CombL* is a duration. To achieve a certain pitch, say, 200 Hz, we have to enter the duration of each cycle for that pitch: 1/200th of a second.

SC can do the calculation with the *reciprocal* message. The line *440.reciprocal* will return the duration of each period of a wave with a frequency of 440. And *3.5.reciprocal* will return the frequency of an event that is 3.5 seconds long. If we also use *midicps* to convert midi numbers to frequency we can raise the language level one more step toward our thinking.

```
   17.3.  midi to cps to delay time

// This will return the duration of each cycle of a wave
// that is the frequency of midi value 69, or A 440

69.midicps.reciprocal;

440.reciprocal; // same thing

// Insert this section into the pluck instrument.

(
{
   var burstEnv, att = 0, dec = 0.001;
   var burst, delayTime, delayDecay = 0.5;
   var midiPitch = 69; // A 440
   delayTime = midiPitch.midicps.reciprocal;
//  RandSeed.kr(Impulse.kr(1/delayDecay), 111);
   burstEnv = EnvGen.kr(Env.perc(att, dec), gate: Impulse.kr(1/delayDecay));
   burst = PinkNoise.ar(burstEnv);
   CombL.ar(burst, delayTime, delayTime,
      delayDecay, add: burst);
}.play
)
```

Insert postln messages to check values.

We don't have to use a *PinkNoise* burst as a noise source. Try other types of noise and other types of waves. Try mixing down a complex collection of sine oscillators also.

Delays for complexity

Earlier we saw how quickly a simple patch can be fleshed out using multi-channel expansion or *Mix* in conjunction with a function and the *dup* message. In this case they are duplicated at the same point in time. Delays have a similar effect. They duplicate a patch, but sequentially. Notice the subtle difference between these two examples. The first creates a single process, and duplicates it in time. The second creates five independent processes.

```
   17.4.  Delay to add complexity

(
{
t = Impulse.kr(5);
o = SinOsc.ar(TRand.kr(2000, 4000, t), mul: EnvGen.kr(Env.perc(0.001, 0.1),
t))*0.1;
Mix.ar(Pan2.ar(
   CombL.ar(o, 2.0,
      Array.fill(5, {rrand(0.2, 1.9)})
   ),
   Array.fill(5, {1.0.rand2})
));
}.play
)

// Compare with

(
{
t = Impulse.kr(Array.fill(5, {rrand(4.0, 7.0)}));

Mix.ar(Pan2.ar(
SinOsc.ar(TRand.kr(2000, 4000, t), mul: EnvGen.kr(Env.perc(0.001, 0.1), t))*0.1,
   Array.fill(5, {1.0.rand2})
));
}.play
)
```

Synth definitions

In the post window you have probably noticed something like the following.

Synth("-429504755" : 1008)

Synth("-1768726205" : 1009)

Synth("-2052671376" : 1010)

Synth("-713843897" : 1011)

Synth("2119841241" : 1012)

The numbers in quotes are synth definition names followed by colon, then a node number. These are created automatically each time an example is run. So far we've been using a backward compatibility (for SC2 users) short cut, allowing SC to create a new arbitrary synth definition each time a section of code is run. It creates a new synth even if you run the same example a second time, which is inefficient. It would be better to replay a definition that already exists. Try these lines of code, replacing the number with what you see in your post window. (You can use copy and paste, or you can just remove the ":" and node number and run that statement in the post window.)

17.5. playing a synthDef

```
//This first
{SinOsc.ar(rrand(700, 1400), mul: 0.1)}.play

//You will see something like this in the post window
//
//Synth("1967540257" : 1012)

//Then this, replacing 1967540257 with the number in your post window
Synth("1967540257");
```

We have also been stopping each example using command period, which stops all processes. If the Synth is assigned to a variable you can stop it explicitly with the free message. Run these two lines separately; again replace the number with any of the numbers you see in the post window.

17.6. stopping a synthDef

```
a = Synth("1967540257");

a.free;
```

We can bypass finding the synth name by assigning the original patch to a variable. We can also run several examples and stop them separately or one at a time.

17.7. playing a synthDef

```
//Run these one at a time
a = {SinOsc.ar(rrand(700, 1400), mul: 0.1)}.play

b = {SinOsc.ar(rrand(700, 1400), mul: 0.1)}.play
```

```
c = {SinOsc.ar(rrand(700, 1400), mul: 0.1)}.play

d = {SinOsc.ar(rrand(700, 1400), mul: 0.1)}.play

//Stop several

a.free; b.free;

//or one at a time

c.free;

d.free
```

The *SynthDef* object allows you to give the patch a name and add arguments. An argument is similar to a variable and is used to pass values to the function. You can set arguments when a synth is first created or after it is running. The first example below is just the patch in the style we have been using so far. The next couches it in a synth definition with a name and arguments.

17.8. SynthDef

```
(//Original patch
{
var rate = 12, att = 0, decay = 5.0, offset = 400;
var env, out, pan;
pan = LFNoise1.kr(1/3);
env = EnvGen.kr(Env.perc(att, decay));
out = Pan2.ar(
          Blip.ar(LFNoise0.ar(rate, min(100, offset), offset),
             (env)*12 + 1, 0.3),
          pan)*env;
out
}.play
)

//SynthDef (naming it) and arguments

(
SynthDef("SH",
{
arg rate = 12, att = 0, decay = 5.0, offset = 400;
var env, out, pan;
pan = LFNoise1.kr(1/3);
env = EnvGen.kr(Env.perc(att, decay), doneAction: 2);
out = Pan2.ar(
          Blip.ar(LFNoise0.ar(rate, min(100, offset), offset),
             (env)*12 + 1, 0.3),
          pan)*env;
Out.ar(0, out)
}).play
```

```
)
```

Watch the post window when you run this example. You will see that the long number is replaced by the name "SH."

With *SynthDef* a few things that were previously done automatically need to be explicit. The first is assigning an output bus using *Out.ar*. (More on busses later.) The next is stopping the patch when it is done playing. We use the envelope to generate the event, but you might have noticed that the CPU and Ugens remained the same. That is because even though the envelope terminated the sound we hear, the patch is still running. There are a number of possible results when the envelope terminates. The default is to do nothing. Changing it to *doneAction: 2* signals the server to deallocate and free up the processing power used for that ugen when it is done. (To see its importance try commenting out this line of code when running the examples below. Watch the peak CPU in the synth server add up and eventually explode.)

Now that the code is loaded into the server we can run several copies of the instrument with control values (for the arguments) using *set*. The arguments are placed in an array with the syntax symbol[38], value: *[\rate, 10, \offset, 200]*. Run these lines singly but one right after the other. (SC makes this easy because when you run one line the cursor moves to the next line. So you can just keep pressing enter.)

```
   17.9.  Multiple nodes of SH

//Three separate nodes of "SH" with different arugments.
//Run these three lines in sequence then stop.
a = Synth("SH", [\rate, 10, \offset, 200]);
b = Synth("SH", [\offset, 400, \att, 3.0, \decay, 0]);
c = Synth("SH", [\rate, 30, \offset, 2000]);

//Let them die out or run these lines to stop them.
a.free;
b.free;
c.free;

//Changing a parameter of an existing node. Run these lines in sequence.
a = Synth("SH", [\rate, 23, \offset, 30, \decay, 20]);
a.set(\offset, 1000)
a.set(\offset, 300)
a.set(\offset, 800)

a.free;

//Two nodes with arguments
a = Synth("SH", [\rate, 7, \offset, 200, \decay, 20]);
b = Synth("SH", [\rate, 23, \offset, 1200, \decay, 20]);
```

[38] A symbol is a word preceded by a backslash: \rate.

```
a.set(\offset, 40)
b.set(\offset, 1000)
a.set(\offset, 800)
b.set(\offset, 600)
a.set(\offset, 1200)
b.set(\offset, 50)

a.free; b.free
```

You can also set arguments using the argument number followed by value, or string, value, as shown below.

17.10. Syntax for passing arguments

```
//Same thing
a = Synth("SH", [\rate, 10, \offset, 200]);
a = Synth("SH", [0, 10, 3, 200]);
a = Synth("SH", ["rate", 10, "offset", 200]);
```

Finally you can indicate a lag time between control changes for a smooth transition.

17.11. Transition time between control changes

```
//SynthDef, arguments, transition

(
SynthDef("SH",
{
arg rate = 12, att = 0, decay = 5.0, offset = 400;
var env, out, pan;
pan = LFNoise1.kr(1/3);
env = EnvGen.kr(Env.perc(att, decay), doneAction: 2);
out = Pan2.ar(
          Blip.ar(LFNoise0.ar(rate, min(100, offset), offset),
             (env)*12 + 1, 0.3),
          pan)*env;
Out.ar(0, out)
},
[0.5, 0.1, 0, 4] //transition for each argument above
).play
)

a = Synth("SH", [\rate, 6, \decay, 20, \offset, 200]);
a.set(\rate, 18);
a.set(\offset, 1000);
```

We could have changed the parameters in the original patch and run it again. But that would be less flexible. Using a *SynthDef* with arguments now allows us to create copies of instruments with different control values automatically.

Now that we have a synth definition loaded into the server we can send a series of "play" commands that build on a compositional process. To repeat these commands we will use a loop.

The loop message repeats a function. Task allows you to pause the loop. (Don't try a loop without the pause, or at least save your work before you do.) With each loop a new *SH* is created with random values as arguments, or controls.

```
   17.12. Multiple nodes of SH

(
r = Task({
   {
      Synth("SH", [
      \rate, exprand(3.0, 22.0),
      \decay, rrand(0.5, 15.0),
      \att, [0, rrand(0, 3.0)].choose,
      \offset, rrand(100, 2000)]);
      rrand(1.0, 5.0).wait; //wait time between repetitions
   }.loop; //repeat this function
}).play
)

r.stop;
```

Try changing any of the values in these examples. Also try inserting postln messages to monitor values (e.g. *rrand(1.0, 5.0).wait.postln*).

Finally, once you have honed the instrument down to where it is worth saving, you can write a copy to disk (synthdefs folder of the SC folder). The next time you launch SC you can just run the synth by name. (It is important to send it to the correct server, whichever is running. If you click on the default button of the running server it will go to that server. Or you can use the target argument to send it to one or the other. I've included the boot expression below to illustrate the second method.)

```
   17.13. Multiple nodes of SH

(//Save to file and load in server "s"
SynthDef("SH",
{
arg rate = 12, att = 0, decay = 5.0, offset = 400;
var env, out, pan;
pan = LFNoise1.kr(1/3);
env = EnvGen.kr(Env.perc(att, decay), doneAction: 2);
out = Pan2.ar(
          Blip.ar(LFNoise0.ar(rate, min(100, offset), offset),
             (env)*12 + 1, 0.3),
          pan)*env;
Out.ar(0, out)
}).load(s)
```

```
)

//Now quit SC, look in the synthdefs folder for "SH.scsyndef"

//Launch SC and run these lines

s = Server.internal; s.boot;

a = Synth("SH", [\rate, 10, \offset, 200], target: s);
```

After you've compiled a nice library of instruments like the *KSpluck* you will probably forget what control busses, output busses, and arguments they had. You can look up all that information using these lines.

```
   17.14. SynthDef Browser

(
// a synthdef browswer
SynthDescLib.global.read;
SynthDescLib.global.browse;
)
```

Below is the KSpluck in a *SynthDef* and then a repeating routine.

```
   17.15. KSpluck SynthDef (EnvGen, Env, perc, PinkNoise, CombL, choose)

(
//First load the synth and save to disk
SynthDef("KSpluck",
   {
      arg midiPitch = 69, delayDecay = 1.0;
      var burstEnv, att = 0, dec = 0.001;
      var signalOut, delayTime;
      delayTime = [midiPitch, midiPitch + 12].midicps.reciprocal;
      burstEnv = EnvGen.kr(Env.perc(att, dec));
      signalOut = PinkNoise.ar(burstEnv);
      signalOut = CombL.ar(signalOut, delayTime, delayTime,
         delayDecay, add: signalOut);
      DetectSilence.ar(signalOut, doneAction:2);
      Out.ar(0, signalOut)
   }
).play;
)

(
//Then run this playback task
r = Task({
   {Synth("KSpluck",
      [
      \midiPitch, rrand(30, 90), //Choose a pitch
      \delayDecay, rrand(0.1, 1.0) //Choose duration
```

```
        ]);
        //Choose a wait time before next event
        [0.125, 0.125, 0.25].choose.wait;
    }.loop;
}).play(SystemClock)
)

//Stop it

r.stop;
```

The right channel is doubled at the octave (+ 12 in MIDI). Try adding an argument allowing you to double at other intervals (fifth, fourth, third, etc.) Change the choices of wait time. Why in this patch did I put two values of 0.125 and one of 0.25? Add postln messages to check variables. Add a statement that links MIDI pitch with decay time such that high notes have short decays and low notes long decays.

Listen carefully to each attack and the character of each pitch. It is clearly the same instrument, yet each note has a slightly different timbre. The character changes because each time the function "pluckInst" is run a new burst of noise is used, which has a different wave shape. Traditional presets on synthesizers lack this complexity, which is inherent in natural instruments.

Practice: Karplus-Strong Patch

In the practice K-S patch the argument *midiPitch* is set from an array of "legal" pitch choices. It is then added to an array of octave choices. The variable *art* (articulation) has replaced *delayDecay* because it is used to shorten or lengthen each attack. Notice that the burst envelope supplies a stereo (array) signal to out. Even though you hear one pitch, the left and right channel should be slightly different. Also the *delayTime* is set using a stereo array; the right an octave higher (MIDI numbers). The entire patch is passed through an *RLPF* with *LFNoise1* control for the filter sweep. The wait array uses a quick method for biasing choices: load the dice. Since there are 5 instances of 0.125 and one of 1, then 0.125 will be chosen about 86% of the time.

Try changing the midiPitch array to various scales: whole tone, diatonic, chromatic, octatonic, quarter tone, etc. Try adding stereo arrays to other aspects of the patch, e.g. *LFNoise* rate or filter cutoff.

```
    17.16. Practice: K S pluck (EnvGen, PinkNoise, LFNoise1, Out, DetectSilence)

//Load this definition
(
SynthDef.new("KSpluck3",
    { //Beginning of Ugen function
    arg midiPitch, art;
    var burstEnv, att = 0, dec = 0.01, legalPitches; //Variable declarations
    var out, delayTime;
    delayTime = [midiPitch, midiPitch + 12].midicps.reciprocal;
```

```
    burstEnv = EnvGen.kr(Env.perc(att, dec));
    out = PinkNoise.ar([burstEnv, burstEnv]); //Noise burst
    out = CombL.ar(out, delayTime, delayTime,
        art, add: out); //Echo chamber
    out = RLPF.ar(out, LFNoise1.kr(2, 2000, 2100), 0.1); //Filter
    DetectSilence.ar(out, doneAction:2);
    Out.ar(0, out*0.8)
    }
).play;
)


//Then run this routine

(
r = Task({
    {Synth("KSpluck3",
    [
    \midiPitch, [0, 2, 4, 6, 8, 10].choose + [24, 36, 48, 60].choose,
    \art, [0.125, 0.25, 0.5, 1.0, 2.0].choose
    ]);
    //Choose a wait time before next event
    [0.125, 0.125, 0.125, 0.125, 0.125, 1].choose.wait;
    }.loop;
}).play(SystemClock)
)
```

17. Exercise

17.1. In a KS patch, what delay time would you use to produce these frequencies: 660, 271, 1000, 30 Hz?

17.2. Rewrite three patches from previous chapters as synth definitions written to the hard disk with arguments replacing some key variables. Then write several Synth() lines to launch each instrument with different arguments. Don't forget the Out.ar(0, inst) at the end. For example:
{SinOsc.ar(400)}.play
Would be:
SynthDef("MySine", {arg freq = 200; Out.ar(0, SinOsc.ar(freq))}).play
Then:
Synth("MySine", [\freq, 600])

17.3. Modify the KS Pluck patch (the Task portion) so that it chooses different pitches articulations, and wait times. Try different types of scales.

17.4. Create a Task that generates multiple events of one of your other synth definitions. For example:
Task({{Synth("Sine", [\freq, 1000.rrand]); 3.wait}.loop;}).play(SystemClock)

18 - Busses and Nodes and Groups (oh my!); Linking Things Together

Disclaimer

This is a very difficult chapter. Turn off the TV.

Busses and nodes are new in SC3, and there is a lot to cover. I may betray my tendency to know just enough of the programming side to do what I want compositionally. My goal is to get you familiar enough with busses, nodes, and groups to do some easy examples and also allow you to follow the examples and help files.

Just a reminder before we dive into this discussion; the code below shows a useful tool for remembering the structure of synth definitions. It can also be used to review or learn bus assignments.

```
   18.1.  Browsing Synth Definitions

(
// a synthdef browswer
SynthDescLib.global.read;
SynthDescLib.global.browse;
)
```

Synth definitions

Creating synth definitions is like building a little synthesizer; virtual metal boxes that make sound. Like real hardware devices the wiring, once sent to the server, can't be changed. Look at the first example from the book, duplicated below. It will always have an *LFNoise* as the frequency control and the arguments will always be 10, 15, 400, and 800. It's as if we had built the synthesizer in a closed box with no controls or inputs. We can redo the code, but then it is a different synth.

```
   18.2.  First Patch (play, SinOsc, LFNoise0, .ar)

{SinOsc.ar(LFNoise0.ar([10, 15], 400, 800), 0, 0.3)}.play
```

We gained more flexibility by replacing the static values with arguments, and using *set* to change the values as shown below. Now our virtual box at least has knobs for changing parameters.

```
   18.3.  First SynthDef

//SynthDef (naming it) and arguments

(
SynthDef("RandSine",
```

```
{
arg rate = 9, scale = 300, offset = 600, pan = 0, out;
out = Pan2.ar(SinOsc.ar(LFNoise0.ar(rate, scale, offset), mul: 0.3), pan);
DetectSilence.ar(out, doneAction:2);
Out.ar(0, out)
}).load(s)
)

// execute these separately
a = Synth("RandSine", [\pan, -1, \rate, 8]);
b = Synth("RandSine", [\pan, 1, \rate, 13]);
b.set(\offset, 2000);
a.set(\rate, 20);
b.set(\rate, 6);
a.set(\scale, 550);
```

But the internal wiring, the diagram or flow chart of the *RandSine* will remain the same. It will always have an *LFNoise0* as a control.

To change the control source to something else, an *LFNoise1*, a sequencer, or *S&H*, we would have to rewrite the patch with those controls; build a new box, so to speak. This was one of the limitations with SC2. A modular approach would be more efficient, that is build the *LFNoise0*, *LFNoise1*, and *S&H* controls as separate patches from the *SinOsc*, then patch whichever one we want to use to the *SinOsc*. SC3's architecture was designed with this in mind. It allows you to run several patches at a time, even the same patch several times, and it allows you to connect those patches together using a bus.

In all patches there are outs and ins. The *LFNoise0* has an output and the *SinOsc* is using that output in one of its inputs. I've used the terms send, control, source, or modulator for an output, and receive, filter, carrier, for an input. But no matter how you use it, there is an output from one unit and an input for another. So far we have connected them by nesting them inside one another. Now we will learn how to use busses to connect them together.

Busses are common in mixers and are used to route several signals to or from a common destination. They don't usually change the signal in any way, they just route it. They are like city busses[39] or shuttles. A worker boards a bus, two or three other workers join along the way and ride to "work" or their destination. They can all get off at one place, or several places. There are 16 or so busses that take people to different factories. We can do the same with signals. The output of one ugen would be the people, and the destination might be a reverb unit.

I've had students ask, after routing the signal to a bus, where it has gone. (That is, what fx unit or headset send have I routed it to?) The answer is nowhere. They don't really go

[39] I only made this connection a few years ago after reading a Mackie manual. Duh. Though it might be more apt to think of them as shuttles in a town with just one street where all the businesses are and everyone lives.

anywhere specific. They are just patch points or cables, and like city shuttles it is important to understand that you define what they are; who gets on and where they go.

Audio and Control Busses

There are two kinds of busses in SC: audio and control. In an earlier example the scope window showed 12 sine waves. These were playing on 12 busses. We could hear the top two because they were routed to the computer's output. The others were routed to audio buses 4 through 12, which were probably not connected to an output that you could hear.

You inter-connect audio busses using *Out.ar* and *In.ar*. The control busses use *Out.kr* and *In.kr*. There are 128 audio busses and 4096 control busses. If you're used to classic studios imagine a snake (or patchbay) with 128 patch points numbered 0 through 127. One end of the snake is labeled "In.ar" and the other "Out.ar," and another completely separate snake with 4096 patch points numbered 0 through 4095. What are they connected to? Nothing yet, with this exception: On most systems the audio busses 0 and 1 are routed to the left and right channel output of your hardware (computer sound out). Confirm this with *{Out.ar(0, SinOsc.ar)}.play* and *{Out.ar(1, SinOsc.ar)}.play*. Similarly, 2 and 3[40] by default are connected to the hardware input. Run *{In.ar([2, 3])}.play* to confirm this. (Use headsets to avoid feedback.) If you tried *{In.ar(0)}.play* you would be listening to your speaker. Likewise *{Out.ar(2)}.play* would send signal to your internal mic. When you do *{anything}.play* it is automatically routed to outs 0 and 1. So *{In.ar([2, 3]}.play* is the same as *{Out.ar([0, 1], In.ar([2, 3])}.play*.

The control busses have no default connections.

To illustrate control and audio busses, run these lines of code. Remember that an array will be expanded to that many channels, as is evident in the scope. We only hear the first two channels (unless you have a multi-channel interface). We don't hear any of the control busses. Note that a control scope is blue, audio scopes are yellow.

```
   18.4.  Audio and Control Busses

(
{
   [
      SinOsc.ar,
      PinkNoise.ar,
      LFNoise1.ar,
      LFNoise0.ar,
      LFTri.ar,
      WhiteNoise.ar
   ]*0.4
}.scope
```

[40] Is it just me, or is it really annoying that buss numbers begin with 0 so that left is even and right odd. That's so wrong.

```
)

// Control busses

(
{
   [
      SinOsc.kr(100),
      Dust.kr(50),
      Impulse.kr(78),
      LFNoise0.kr(100),
      LFNoise1.kr(100),
      WhiteNoise.kr
   ]*0.4
}.scope(zoom: 10)
)
```

Below is the first patch broken up into two components. The *SinOsc* and the *LFNoise0*. The *Out* object indicates which bus the signal is routed to and the first argument is the bus number. You can enter a single number for a mono signal, or an array for a stereo signal. But even if you don't enter an array the correct number of busses is automatically allocated. If the patch is a stereo signal (e.g. using multi-channel expansion) and the bus number in *Out* is 0, then it will actually use 0 and 1. If the signal has four channels and the bus out is 4 it will use 4, 5, 6, and 7. You are responsible for making sure there are no conflicts.

The *LFNoise0* module is sent to control bus 9. (Remember there are control busses and audio busses. The *.kr* message will use control busses.) I've written the patch with the *mul* and *add* outside the LFNoise so that it displays on the scope.

When I isolate the LFNoise in the third example, you hear it in both channels, but it is just a series of pops. When the *SinOsc* is moved to bus 5, you won't hear it, but will see it in the scope. Neither will you hear the last example because control bus 9 isn't connected to an audio out. Why busses 5 and 9? To illustrate that the busses are arbitrary. You can use whatever bus you want, as long as you connect the destination to the same bus number.

18.5. Assigning busses

```
//Entire patch
{SinOsc.ar(LFNoise0.kr([10, 15]) * 400 + 800, 0, 0.3)}.scope

//Just the SinOsc
{SinOsc.ar(800, 0, 0.3)}.scope

//Just the LFNoise0
{LFNoise0.ar([10, 15])}.scope

//Just Sine patched to audio out 0
{Out.ar(0, SinOsc.ar(800, 0, 0.3))}.scope;

//Just Sine patched to audio out 5
{Out.ar(5, SinOsc.ar(800, 0, 0.3))}.scope(16);
```

```
//Just the LFNoise def
{Out.kr(9, LFNoise0.kr([10, 15], 1, 0))}.scope(16, zoom: 10)
```

Now two synths are running. One is producing audio on 5 the other is producing a control rate signal that is riding around on control bus 9 waiting for us to tell it where to get off. It needs to get off at the *SinOsc* factory. To get it there we use *In.kr* (not *In.ar*, which would connect it to the audio an bus). In the example below the first argument for *In.kr* is the bus number (9) and the second is the number of channels (2). Why 9? Because that's where the *LFNoise* is connected. The bus, so to speak, that it boarded. Why 2 channels? Because the *LFNoise0* is a stereo signal (frequency argument array), so it is using 9 and 10. *In.kr* reads from control bus 9 and 10, which is where *LFNoise0* is connected and uses that as a control. *SinOsc* then plays to audio busses 0 and 1, because it is now a stereo signal too.

If you haven't already done so, press command-period to stop all processes.

I reverse the order in this example so you can start the *LFNoise0* first. Watch the server window to see that the cpu increases and a Synth is created with 5 or so Ugens even though we don't hear it. Then start the *SinOsc* to hear the two patched together. Then reverse the order so you can hear the Sine without the *LFNoise0* patched in, then start the *LFNoise0*.

18.6. Patching synths together with a bus

```
{Out.kr(20, LFNoise0.kr([8, 11], 500, 1000))}.scope

{Out.ar(0, SinOsc.ar(In.kr(20, 2), 0, 0.3))}.scope
```

The next step is to use an argument for bus numbers so we can change it on the fly. Why change the bus number while the synth is running? To connect it to a different control, which I've added below, assigned to different busses. You could assign the busses on the controls using an argument too. It's just not necessary for this example. You can define all the synths at once inside parentheses. Once they are running note the server window displays *Synths: 4* and quite a few Ugens. We only hear one of them: the sines without controls. The other two are on control busses we don't hear. Run each *a.set* to assign a new bus number for the *PatchableSine* input for *In*, thereby connecting whichever synth control is connected to that bus and changing the control for frequency.

18.7. Patching synths together with a bus, dynamic control sources

```
(
//Start all the synths
SynthDef("LFN0Control",
   {Out.kr(20, LFNoise0.kr([8, 11], 500, 1000))}).play(s);


SynthDef("SineControl",
   {Out.kr(22, SinOsc.kr([3, 3.124], mul: 500, add: 1000))}).play(s);
```

```
SynthDef("MouseControl",
   {Out.kr(24, MouseX.kr([100, 200], 1000))}).play(s);


a = SynthDef("PatchableSine", {arg busInNum = 0;
   Out.ar(0, SinOsc.ar(In.kr(busInNum, 2), 0, 0.3))}).play(s);
)

a.set(\busInNum, 20); //set to LFNoise0
a.set(\busInNum, 22); //set to SineControl
a.set(\busInNum, 24); //set to MouseControl
```

Why are the control busses numbered 20, 22, and 24? Try using 20, 21, and 22 to see what happens.

It is less efficient to have all three controls running at once, so this will also work[41].

```
   18.8.  Patching synths together with a bus, dynamic control sources

a = Synth("PatchableSine", [\busInNum, 20]); b = Synth("LFN0Control");

b.free; b = Synth("SineControl"); a.set(\busInNum, 22);
b.free; b = Synth("MouseControl"); a.set(\busInNum, 24);
```

It is possible to patch several controls to a single bus, or control several synths with a single bus. In this example I use *Out.kr(0)* to illustrate that it is separate from *Out.ar(0)*. Notice that both controls are connected to bus 0, so they both control the receiver. In the second example both receivers are connected to some arbitrary bus, so they are both controlled by that synth. (It may sound like one synth so listen to the left headset then the right.)

```
   18.9.  Several controls over a single bus

(
SynthDef("SendControl1", {
   Out.kr(0, SinOsc.ar(0.3, mul: 1000, add: 1100))}).send(s);
SynthDef("SendControl2", {Out.kr(0, LFNoise0.ar(12, 200, 500))}).send(s);
SynthDef("Receive", {Out.ar(0, SinOsc.ar(In.kr(0)))}).send(s);
)

Synth("Receive");
Synth("SendControl1");
Synth("SendControl2");

//Or
```

[41] See also *ReplaceOut*.

```
Synth("Receive");
Synth("SendControl2");
Synth("SendControl1");

//Or

(
SynthDef("SendControl", {Out.kr(1275, LFNoise0.ar(12, 200, 500))}).send(s);
SynthDef("Receive1", {
   Out.ar(0, RLPF.ar(PinkNoise.ar, In.kr(1275), 0.05))}).send(s);
SynthDef("Receive2", {Out.ar(1, SinOsc.ar(In.kr(1275)))}).send(s);
)

Synth("Receive1");
Synth("Receive2");
Synth("SendControl");

//Stop all using command-. Then try executing them in reverse order

// Open the browser and examine the bus assignments.

(
// a synthdef browswer
SynthDescLib.global.read;
SynthDescLib.global.browse;
)
```

Nodes

It works in either order because it is a control bus. When working with audio buses order does matter. When you create a synth SC stores it on a node or memory location. Each new synth creation results in a new node (you see them in the post window when you start a synth). Nodes are linked together. The computer knows the order and can add a new node either at the beginning of the list (head) or at the end (tail) or anywhere else, for that matter. Be aware that node number and node order are not related. The number is just an address, and doesn't affect or refer to the ordering of the nodes. In the examples we've done so far this connection process has remained behind the scenes. But now that we are patching synths together we have to take control. We are not talking about in what order the synths are sent to the server, but in what order they are created and start generating sound on the server.

Create the two synth definitions below then run the following examples. Notice that if you run the *Saw* first, then the filter, you get no sound.

```
   18.10. node order, head, tail

(
//The definitions can be any order
SynthDef("Saw", { arg filterBus = 16;
   Out.ar(filterBus, LFSaw.ar([60, 90]))
```

```
}).send(s);

SynthDef("Filter", {arg filterBus = 16;
   Out.ar(0, RLPF.ar(In.ar(filterBus, 2), LFNoise0.kr(12, 300, 350), 0.2))
}).send(s);
)

//won't work
Synth("Saw"); //source
Synth("Filter"); //filter

//works
Synth("Filter"); //filter
Synth("Saw"); //source

//or
Synth("Filter"); Synth("Saw");
```

In this example we are linking an audio source to a filter. I use audio busses for both signals (*Out.ar*) because they both need to be audio rate. The source is sent out on 16. I can't use 0, 1, 2, or 3 because they are connected to my hardware out and in. Why not 5? Because I may eventually use this on a machine that can handle 8 outs and 8 ins (0 through 15), so I move to 16 (though I could use 60 for that matter). See also dynamic bus allocation below.

The only difference between the two examples above is execution order. In the first example *Saw* (the source) is started first, then *Filter* (the filter). The second creates *Filter*, then *Saw*.

When you create a new synth it is given a node id and placed at the head of the list. Any synths that are already in the list are moved down. So if you execute the *Saw* first it is given a node and placed at the head. Then when you execute *Filter* it is given a node number, placed at the head bumping *Saw* to the tail. But signal flows from head to tail, so the *Saw* has to be at the head and *Filter* at the tail.

I think of the head as the top of a flow chart because we used to diagram patches with signal flowing down. Unfortunately this is confusing because it's opposite of the execution order, as shown below.

```
   18.11. Execution order, node order

// This execution order

Synth("Receive");
Synth("SendControl1");
Synth("SendControl2");

// Results in these nodes printed to the post window

Synth("Receive" : 1000)
Synth("SendControl1" : 1001)
Synth("SendControl2" : 1002)
```

But their order, head to tail, top to bottom is

Group(0)
    Group(1)
        Synth 1002 //head
        Synth 1001
        Synth 1000 //tail

So the execution order is the reverse of the node list, top to bottom (head to tail).

Here is what I try to remember: Order of execution is inputs first, outputs second; receive first, send second; destination first, departure second. By that I mean execute the synth that is going to receive signal from another at its input bus. Execute the synth that is sending a signal to be used second. (How about this: you have to build the factory before putting workers on the bus. If you put them on a bus before their destination exists you can't tell them where to get off[42]. Absurd method for remembering, I'll admit, but no more so than all cows eat grass before dinner.)

If you want to think of them as head or tail: inputs at the tail (bottom), outputs at the head (top), because signal flows from top to bottom.

Execution order: Ins then outs. Node order: Ins at the tail, outs at the head.

When using *Synth* the node is placed at the head of the group by default. But you can also explicitly assign a node to the tail or head. With the example below the *Saw* is placed at the head (which would happen anyway). The second expression places *Filter* at the tail instead of the head which would move *Saw* to the tail. Using *tail* and *head* mean order of execution doesn't matter.

When using the message *head* or *tail* the first argument is the group, the head or tail of which the synth is placed. We haven't created any groups yet, so it is placed on the server itself, which is a default first node with id 0 and has a default group of 1.

```
   18.12. node order, head, tail

Synth.head(s, "Saw");
Synth.tail(s, "Filter");

//Same

Synth.tail(s, "Filter");
Synth.head(s, "Saw");
```

[42] They don't have cell phones.

I can see your brains are full. Walk down to the corner for a bagel. I'll wait.

[I'll watch a Twilight Zone episode while you're gone; the one where Will from Lost In Space sends people to the cornfield.]

Ok, more confusion.

Dynamic bus allocation

In the previous example I used bus 16 in order to leave room for an 8 channel interface, 8 in, 8 out. What if you're not sure how many outs and ins you already have? Naturally, there is an automated method for managing busses. *Bus.control* and *Bus.audio* can be used to return the next available control or audio bus index. *Bus.index* returns that index. The first example below results in this line in the post window: *Bus(internal, audio, 4, 1)*, meaning a single audio bus on the internal server at index 4 was returned. Why 4? because 0 through 3 are in use by the computer's out and in, so 4 is the next available bus. The *free* messages frees that bus to be allocated again. Next I reallocate a two channel bus to c, then a two channel bus to b. They should be 4, 5, and 6, 7, but your mileage may differ; they will actually be whatever is free on your system. I then send a *Saw* and *SinOsc* to each of these allocated busses, using scope[43] with 12 channels so you can see them all. Notice that both are stereo busses, but the *Saw* is a mono patch. Even so, the second c bus is kept available.

Be sure to free each bus after each example, just so my explanations jive.

```
   18.13. Bus allocation and reallocation

b = Bus.audio;

b.index;

b.free;

c = Bus.audio(s, 2);
b = Bus.audio(s, 2);

{Out.ar(c.index, Saw.ar)}.play;
{Out.ar(b.index, SinOsc.ar([500, 1000])).scope(8)

b.free; c.free;
```

If you bypass the *Bus* and just enter an index number it will not register with the bus allocation. This is called hardwiring. To illustrate this I'll first take over some busses with a hard wired dummy synth. Then I use *Bus* to get the next available bus, print its index, then

[43] Here are some tricks for scope: When you generate a scope, the scope window comes to the front. Use command-` to quickly cycle back to your code window. Use the up and down arrow keys to scroll through busses in the window, use tab and arrow to show more busses.

use its index to create a new synth. Notice that it is sent to the same busses as the first *SinOsc*.

18.14. Bus allocation and reallocation

```
{Out.ar(4, SinOsc.ar([100, 200, 300, 400]))}.scope(8);

b = Bus.audio(s, 2);

b.index; // should still be 4 regardless of above code

{Out.ar(b.index, Saw.ar([800, 1000]))}.scope(8); // adds to sines

b.free;
```

Follow along and watch the post window while executing this set of instructions for bus allocation.

18.15. Bus allocation

```
a = Bus.audio(s, 2) // Get the next available 2 channels
b = Bus.audio(s, 1) // Get next 1 channel
c = Bus.audio(s, 2) // Get two more
c.index // print c
a.index // print a
a.free // free a
b.free // free b
d = Bus.audio(s, 1) // a and b are now free, so these
e = Bus.audio(s, 2) // should take over those indexes
a = Bus.audio(s, 2) // reallocate a and b, will probably
b = Bus.audio(s, 1) // 9 through 11
[a, b, c, d, e].postln; // print all of them
s.scope(14); // start a scope

// Now we start some synths. I'm going to mix them all
// down to bus 0, 1, so we have to start it first. Remember
// In first, Out second, receive first, send second.

{Out.ar(0, Mix.ar(In.ar(2, 12))*0.1)}.play
{Out.ar(a.index, SinOsc.ar)}.play
{Out.ar(b.index, SinOsc.ar(1000))}.play
{Out.ar(c.index, Saw.ar([400, 800]))}.play
{Out.ar(d.index, Pulse.ar(200))}.play
{Out.ar(e.index, [Saw.ar(500), FSinOsc.ar(900)])}.play
// You can get a bus without assigning it to a variable,
// you just won't be able to free it later.
{Out.ar(Bus.audio.index, Saw.ar(2000))}.play
// You can write two signals to a single bus
{Out.ar(a.index, Saw.ar(2000))}.play


[a, b, c, d, e].do({arg each; each.free}) // free all of them
```

I have to admit I don't have a lot of experience with busses. My gut feeling is that dynamic allocation could be either indispensable or just another unnecessary level of complexity. You decide.

Using busses for efficiency

Busses can make your patches more efficient. You can use a single control for several synths, or send several synths to a single global fx. If an fx is built into a synth then duplicating the synth also duplicates the fx. Consider this patch with a trigger and a reverb. Run it and check the cpu usage and number of Ugens in the server window. It's not too bad when running one copy of the synth, but try six of them using *Synth("inefficient")* and watch the cpu and Ugens increase with each one. (Note that since the trigger is random you might have to let it run a while before you hear anything.)

And here is another trick for executing the same line of code several times: type *Synth("inefficient")* at the bottom of a new window but *don't* type a return. Now when you press enter the cursor won't go to the next line, but will stay on the *Synth* line. Each time you press enter a new synth will be created.

```
   18.16. inefficient patch

(
SynthDef("inefficient",
{
var out, delay;
out =
    SinOsc.ar(LFNoise0.kr(15, 400, 800), mul: 0.2)
    *
    EnvGen.kr(
        Env.perc(0, 1),
        gate: Dust.kr(1)
    );

delay = CombC.ar(out, 0.5, [0.35, 0.5]);
out = Pan2.ar(out, Rand(-1.0, 1.0));
Out.ar(0, (out + delay))
}).play;
)

// Type this into a new window with no return and keep pressing enter
Synth("inefficient")
```

If you run six versions you get six reverbs (*CombC*) and six triggers (Dust.kr). On my machine that totals 96 Ugens and 16% of cpu.[44] If we break up the patch so that the trigger, source (*SinOsc*), and reverb (*CombC*) are all separate units then we can get virtually the same effect using one trigger and one reverb.

```
   18.17. more efficient modular approach using busses

(
//define all the synths in one pass
SynthDef("singleTrigger", {
   Out.kr(
      //output busses are 1560 through 1565
      LFNoise0.kr(5, mul: 4.0, add: 1563).div(1),
      Dust.kr(6)
   )
}).send(s);

SynthDef("source",
{ arg trigIn, rate;
var out, delay;
out =
   SinOsc.ar(LFNoise0.kr(rate, 400, 800), mul: 0.1)
   *
   EnvGen.kr(
      Env.perc(0, 1),
      gate: In.kr(trigIn)
   );

out = Pan2.ar(out, Rand(-1.0, 1.0));
Out.ar(16, out)
}).send(s);

SynthDef("singleReverb",
{
var signal;
signal = In.ar(16, 2);
   Out.ar(0, (signal + CombC.ar(signal, 0.5, [0.35, 0.5])))
}).send(s);

)

// start the trigger
Synth("singleTrigger", [\rate, 1/4])

// start the reverb
Synth("singleReverb")

// start the sources "watching" trigger busses 4-9
// start 4 and 9 first to make sure they're working (with
```

[44] And yes, I'm including cpu as an illustration but also so we can laugh about it in two years.

```
// a slow and fast rate so you can keep track)

Synth("source", [\trigIn, 1560, \rate, 4])
Synth("source", [\trigIn, 1565, \rate, 25])
Synth("source", [\trigIn, 1561, \rate, 10])
Synth("source", [\trigIn, 1562, \rate, 8])
Synth("source", [\trigIn, 1563, \rate, 17])
Synth("source", [\trigIn, 1564, \rate, 7])
```

In this patch I first define a trigger that is playing on six busses: 1560 through 1565 (with a little overlap). Why 1560s? because they can be anything. The *LFNoise* generates values between 1559 and 1566, converted to integers with *div(1)* and is used to "spray" *Out* bus numbers. (I'm not sure if this is clever or just a hack. It works.) Imagine busses 1560 through 1566 with leds lighting up one at a time as the *LFNoise0* erratically jumps around between them. They will act as triggers. When *LFNoise0* "hits" one of the busses it sends a trigger to that bus.

The *source* synth gets its trigger from a bus, set by the argument *trigIn*. When we launch each copy of the source we assign it to "watch" one of the busses. When the *LFNoise0* trigger connects with, say, bus 1563, then the source that is listening to that bus will fire.

My machine shows 79 Ugens and 6%; half the cpu of the less efficient version.

When do you break a patch up into components? When would you not? In this case the reverb is exactly the same. If you wanted each of the delay times to be different, i.e. if it is critical to the composition that they be different (the practice example below might fall into this category), then they should be part of the patch, so that each synth has a unique reverb. But in this case, I believe that the *CombC* in all of the inefficient duplications are precisely the same, with the same delay times, so it is worth separating that component and using just one.

Groups

Hopefully by now you have tens or possibly hundreds of nodes running in a patch. It would be difficult to manage the nodes and controls of each one individually. This is where groups come in, because controls can be sent to an entire group. They also have an order in the linked list and all the nodes in the group inherit the order of the group. All your sources could be in a group that is "ahead" of a group containing filters. You wouldn't have to worry about the position of each individual source and each filter. All of the filters in the group will be after all sources if the filter group is after the source group.

Group Manipulation

A group could contain ten or so duplications of one synth that you would want to act in unison. A message controlling amplitude, frequency, attack, decay, etc. sent to the group will change all of its children (the individual synth defs) at once.

```
(
//Create a synth
SynthDef("ping",
{arg fund = 100, harm = 1, rate = 0.2, amp = 0.1;
a = Pan2.ar(SinOsc.ar(fund*harm, mul: amp) *
EnvGen.kr(Env.perc(0, 0.2), gate: Dust.kr(rate)), Rand(-1.0, 1.0));
Out.ar(0, a)
}).load(s);
)

// Define a group using a global variable (~)
~synthGroup = Group.head(s);

// Run this 8 or so times, adding a new ping to the group
Synth("ping", [\fund, rrand(100, 1000), \rate, 1], ~synthGroup);
Synth("ping", [\fund, rrand(100, 1000), \rate, 1], ~synthGroup);
Synth("ping", [\fund, rrand(100, 1000), \rate, 1], ~synthGroup);
Synth("ping", [\fund, rrand(100, 1000), \rate, 1], ~synthGroup);
Synth("ping", [\fund, rrand(100, 1000), \rate, 1], ~synthGroup);
Synth("ping", [\fund, rrand(100, 1000), \rate, 1], ~synthGroup);
//etc.

// Change all the rates of the group
~synthGroup.set(\rate, 3/5);
~synthGroup.set(\rate, 8);

// Change the amplitude
~synthGroup.set(\amp, 0.2);
~synthGroup.set(\amp, 0.01);

//Command-period stops the synths and the group, so use this so that the
//group still exists.
~synthGroup.freeAll;
```

All items in the group that have a *rate* or *amp* argument will recognize the change. If we had added a synth that did not have those arguments, the command would be ignored.

I use global variables (identified with a tilde) for the synth groups.

The argument for *Group.head* is the target. Using *.head* means that this group is at the head of its parent; the server. It could be added to another existing group, but this is the only one we have, so it is added to the running server (which is the default group). We add *pings* to the group using *Synth*. We've used *Synth* to create new nodes before. They were added to the default group without us knowing it. Now we are explicitly adding it to a group.

The last command, *freeAll*, is used rather than command-. because we want the group to remain active even though we stop all the nodes.

The next example creates nodes using *Array.fill*, adding each to the group. This allows us to use the arguments *i* to manipulate each harmonic as the synth is created. I could have used *12.do*, but filling an array with the synths allows access to each one in the array. If I want to change just one I can then use *~all.at(n).set(args)*.

I'm assuming the group is still active. If it isn't, run that line of code again before doing this example.

```
   18.19. Automated node creation

~all = Array.fill(12,
   {arg i; Synth("ping", [\harm, i+1, \amp, (1/(i+1))*0.4],~synthGroup)});

~synthGroup.set(\rate, 0.8);
~synthGroup.set(\rate, 5);

Array.fill(12, {arg i; i/2+1})

// Change the amp of one node
~all.at(6).set(\amp, 1);
~all.at(6).set(\amp, 0.1);

// Change all the harmonics using a formula. I checked the formulas
// using this Array.fill(12, {arg i; i/2+1})

~all.do({arg node, count; node.set(\harm, count/2+1)}); //1, 1.5, 2, etc.
~all.do({arg node, count; node.set(\harm, count*2+1)}); //1, 3, 5, 7, etc.
~all.do({arg node, count; node.set(\harm, count*1.25+1)});
~all.do({arg node, count; node.set(\harm, count*1.138+1)});

// Change the fundamental
~synthGroup.set(\fund, 150);
~synthGroup.set(\fund, 250);
~synthGroup.set(\fund, 130);

// Stop the nodes but not the group
~synthGroup.freeAll;

// Create a task that adds new synths
r = Task({{Synth("ping",
   [\fund, rrand(100, 2000), \rate, 2], ~synthGroup); 1.wait}.loop}).play

// Slow down the attacks when it gets to be too many
~synthGroup.set(\rate, 0.2);

// Turn them all down. Note that new ones still have the old volume
~synthGroup.set(\amp, 0.01);

// Stop everything but the task
~synthGroup.free;

// Stop the task
r.stop;
```

Now we will add another group to the tail of the default group (the server) for all fxs. Note that if I wanted to route the fx in series, that is source -> fx1 -> fx2, -> fx3, then I would need to have them and/or their groups in the correct order. But here they are in parallel: the source is routed to both echoes and both echoes are mixed to bus 0, 1.

I've changed the output bus for the ping to 16 in order to route it to the echoes. I send all the definitions at once, create a source group for the pings and an fx group for the echoes. The order of creation doesn't matter because their nodes are determined by the group; *synthGroup* is at the head where it should be and *fxGroup* is at the tail. I can start and stop them in any order.

There are actually going to be three "echoes" in the fx group. Two are echoes using *Comb*, but in neither of those do I mix in the dry signal. So echo1 and echo2 are just the wet signal, no source. I use a *dry* synth that just reroutes the source to bus 0 and 1 so that I can control it separately, adding or removing it from the final mix. This could also have been done using *Out.ar([0, 1, 16, 17], a)*, routing the source to 16 and 17 for fx, 0 and 1 for the dry signal.

```
   18.20. Source Group, Fx Group

(
SynthDef("ping",
{arg fund = 400, harm = 1, rate = 0.2, amp = 0.1;
a = Pan2.ar(SinOsc.ar(fund*harm, mul: amp) *
EnvGen.kr(Env.perc(0, 0.2), gate: Dust.kr(rate)), Rand(-1.0, 1.0));
Out.ar(16, a)
}).load(s);

SynthDef("dry",
{var signal;
signal = In.ar(16, 2);
   Out.ar(0, signal);
}).load(s);

SynthDef("echo1",
{
var signal, echo;
signal = In.ar(16, 2);
echo = CombC.ar(signal, 0.5, [0.35, 0.5]);
   Out.ar(0, echo);
}).load(s);


SynthDef("echo2",
{
var signal, echo;
signal = In.ar(16, 2);
echo = Mix.arFill(3, { CombL.ar(signal, 1.0, LFNoise1.kr(Rand(0.1, 0.3), 0.4, 0.5),
15) });
   Out.ar(0, echo*0.2)
```

```
}).load(s);
)


~synthGroup = Group.head(s);
~fxGroup = Group.tail(s);

// 12.do will not allow me to access each one, but it doesn't matter
(
12.do({arg i;
    Synth("ping", [\harm, i+1, \amp, (1/(i+1))*0.4],
    ~synthGroup)});
)

// "ping" is playing on bus 16, so we don't hear it

// Start the echo1 (wet), echo2 (still wet), then dry
a = Synth("echo1", target: ~fxGroup);
b = Synth("echo2", target: ~fxGroup);
c = Synth("dry", target: ~fxGroup);

b.free; // remove each in a different order
a.free;
c.free;

// The original ping is still running, so stop it.
~synthGroup.freeAll;

// This also works
a = Synth("echo1", target: ~fxGroup);
b = Synth("echo2", target: ~fxGroup);
12.do({arg i; Synth("ping", [\harm, i+1, \amp, (1/(i+1))*0.4],~synthGroup)});
c = Synth("dry", target: ~fxGroup);

~synthGroup.freeAll; // Stop the source, but the echoes are still running

// Start the source again
12.do({arg i; Synth("ping", [\harm, i+1, \amp, (1/(i+1))*0.4],~synthGroup)});

~synthGroup.set(\rate, 0.8);
~synthGroup.set(\rate, 5);

~synthGroup.free;
~fxGroup.free;
```

Practice: Bells and Echoes

```
    18.21. Bells and echoes

(
SynthDef("bells",
{arg freq = 100;
```

```
var out, delay;
out = SinOsc.ar(freq, mul: 0.1)
*
EnvGen.kr(Env.perc(0, 0.01), gate: Dust.kr(1/7));

out = Pan2.ar(Klank.ar(`[Array.fill(10, {Rand(100, 5000)}),
   Array.fill(10, {Rand(0.01, 0.1)}),
   Array.fill(10, {Rand(1.0, 6.0)})], out), Rand(-1.0, 1.0));

Out.ar(0, out*0.4); //send dry signal to main out
Out.ar(16, out*1.0); //and send louder dry signal to fx bus

}).load(s);

SynthDef("delay1", // first echo
{var dry, delay;
dry = In.ar(16, 2);
delay = AllpassN.ar(dry, 2.5,
   [LFNoise1.kr(2, 1.5, 1.6), LFNoise1.kr(2, 1.5, 1.6)],
   3, mul: 0.8);
Out.ar(0, delay);
}).load(s);

SynthDef("delay2", // second echo
{var delay, dry;
dry = In.ar(16, 2);
delay = CombC.ar(dry, 0.5, [Rand(0.2, 0.5), Rand(0.2, 0.5)], 3);
Out.ar(0, delay);
}).load(s);

SynthDef("delay3", // third echo
{
var signal, delay;
signal = In.ar(16, 2);
delay = Mix.arFill(3, { CombL.ar(signal, 1.0, LFNoise1.kr(Rand([0.1, 0.1], 0.3),
0.4, 0.5), 15) });
   Out.ar(0, delay*0.2)
}).load(s);
)

//define groups
~fxGroup = Group.tail;
~bellGroup = Group.head;

// start one of the echoes and 4 bells
f = Synth("delay3", target: ~fxGroup);
4.do({Synth("bells", [\freq, rrand(30, 1000)], target: ~bellGroup)})

// stop existing echo and change to another
f.free; f = Synth("delay1", target: ~fxGroup);
f.free; f = Synth("delay2", target: ~fxGroup);
f.free; f = Synth("delay3", target: ~fxGroup);
Synth("delay1", target: ~fxGroup); // add delay1 without removing delay3
```

18. Exercises

18.1. Modify this patch so that the LFSaw is routed to a control bus and returned to the SinOsc using In.kr.
{Out.ar(0, SinOsc.ar(LFSaw.kr(12, mul: 300, add: 600)))}.play

18.2. Create an additional control (perhaps, SinOsc) and route it to another control bus. Add an argument for the input control bus on original SinOsc. Change between the two controls using Synth or set.

18.3. Create a delay with outputs routed to bus 0 and 1. For the delay input use an In.ar with an argument for bus number. Create another stereo signal and assign it to 4/5. Set your sound input to either mic or line and connect a signal to the line. Use set to switch the In bus from 2 (your mic or line) to 4 (the patch you wrote).

18.4. Assuming the first four bus indexes are being used by the computer's out and in hardware, and you run these lines:
a = Bus.ar(s, 2); b = Bus.kr(s, 2); c = Bus.ar(s, 1); c.free; d = Out.ar(Bus.ar(s), SinOsc.ar([300, 600]));
-Which bus or bus pair has a SinOsc at 600 Hz?
-What variable is assigned to audio bus 6?
-What variable is assigned to control bus 3?
-What variable is assigned to audio bus 4?

18.5. Assuming busses 0 and 1 are connected to your audio hardware, in which of these examples will we hear the SinOsc?
({Out.ar(0, In.ar(5))}.play; {Out.ar(5, SinOsc.ar(500))}.play)
({Out.ar(5, SinOsc.ar(500))}.play; {Out.ar(0, In.ar(5))}.play)
({Out.ar(0, In.ar(5))}.play; {Out.kr(5, SinOsc.ar(500))}.play)
({Out.ar(5, In.ar(0))}.play; {Out.ar(0, SinOsc.ar(500))}.play)
({Out.ar(0, SinOsc.ar(500))}.play; {Out.ar(5, In.ar(0))}.play)

Section III: Computer Assisted Composition

19 - Operators, Precedence, Arguments, Expressions, and User Defined Functions

The previous section dealt with synthesis. From here on we will use SC to construct systems for organizing events, using both the synth definitions from previous sections, or outboard MIDI instruments.

Operators, Precedence

The examples below combine numbers with operators +, /, -, and *. Evaluate each line separately.

```
   19.1.  Operators (+, /, -, *)

1 + 4

5/4

8*9-5

9-5*8

9-(5*8)
```

The last three expressions look similar, but have different results. The third is 8*9, then -5 = 67, the fourth is 9-5 then * 8 = 32. The difference is precedence. Precedence is the order in which each operator and value is realized in the expression. Precedence in SC is quite simple (and it is different from the rules you learned in school); enclosures first, then left to right. Since the 8*9 comes first in the third expression it is calculated before the -5. In the line below it 9-5 is calculated first and that result is multiplied by 8. The last line demonstrates how parentheses force precedence. First 5*8, because it is inside parentheses then that result is subtracted from 9.

Can you predict the result of each line before you evaluate the code?

```
   19.2.  More operators

1 + 2 / 4 * 6

2 / 4 + 2 * 6

(2 * 6) - 5

2 * (6 - 5)
```

Here are some other binary operators. Run each line and see what SC returns: > greater than, < less than, == equals, % modulo.

```
   19.3.  Binary operators (>, <, ==, %)

10 > 5

5 < 1

12 == (6*2)

106%30
```

The > (greater than) and < (less than) symbols return: *true* and *false*. SC understands that the number 10 is greater than 5 (therefore *true*) and 5 is not less than 1 (*false*). We will use this logic in later chapters.

Modulo (%) is a very useful operator that returns the remainder of the first number after dividing by the second. For example, 43%10 will reduce 43 by increments of 10 until it is less than 10, returning what is left. 12946%10 is 6. A common musical application is a modulo 12, which reduces numbers to below 12, or an octave.

Can you predict the results of these expressions?

```
   19.4.  Predict

(8+27)%6

((22 + 61) * 10 )%5
```

All of these examples use integers. Integers are whole numbers. (If you don't remember math, that's numbers used for counting items, without the decimal point: 1, 2, 3, 4, etc.) Numbers that use a decimal are called floating-point values. In SC you express integers by just writing the number (7, 142, 3452). Floating point values must have the decimal with numbers on both sides, even for values below 1: 5.142, 1.23, 456.928, 0.0001, 0.5 (not .0001 or .5)

Messages, Arguments, Receivers

You should be comfortable with messages, arguments, and receivers. Remember that numbers can be objects. The message usually has a meaningful name such as *sum* or *abs*, and is followed by parentheses that enclose arguments separated by commas. Following are typical messages used in computer music. In previous chapters messages have used the syntax *Object.message*. This is called receiver notation. The object is the receiver. An equivalent syntax is shown in the examples below, where I use functional notation: *message(argument)*. The object is placed as the first argument in the argument list.

5.rrand(10) can be expressed rrand(5, 10). Likewise, min(10, 100) can be expressed 10.min(100).

19.5. Music related messages

```
cos(34)   //returns cosine

abs(-12)  //returns absolute value

sqrt(3)  //square root

midicps(56)  //given a midi number, this returns
//the cycles per second in an equal tempered scale

cpsmidi(345)  //given cps, returns midi

midiratio(7)  //given a midi interval, returns ratio

ratiomidi(1.25)  //given a ratio, returns midi number

rand(30)  //returns a random value between 0 and 29

rand2(20)  //returns a random value between -30 and 30

rrand(20, 100) //returns a random value between 20 and 100

// Here are examples in receiver notation.

30.cos //same as cos(30)

0.7.coin //same as coin(0.7)

20.rand //same as rand(20)

7.midiratio

// Binary functions have two arguments.

min(6, 5) //returns the minimum of two values

max(10, 100) //returns maximum

round(23.162, 0.1) //rounds first argument to second argument

// Arguments can be expressions

min(5*6, 35)

max(34 - 10, 4) //returns the maximum of two values
```

Practice, Music Calculator

SC is useful even when you're not producing music[45]. I often launch it just as a music calculator; to calculate the frequency of an equal tempered Ab, how many cents an equal tempered fifth is from a just fifth, or the interval, in cents, of two frequencies. Here are some examples.

```
   19.6.  Music calculator

// Major scale frequencies
([0, 2, 4, 5, 7, 9, 11, 12] + 60).midicps.round(0.01)

// Major scale interval ratios
[0, 2, 4, 5, 7, 9, 11, 12].midiratio.round(0.001)

// Phrygian scale frequencies
([0, 1, 3, 5, 7, 8, 10, 12] + 60).midicps.round(0.01)

// Phrygian scale interval ratios
[0, 1, 3, 5, 7, 8, 10, 12].midiratio.round(0.001)

// Equal and Just Mixolydian scale compared
[0, 2, 3, 4, 5, 7, 9, 10, 12].midiratio.round(0.001)

[1/1, 9/8, 6/5, 5/4, 4/3, 3/2, 8/5, 7/4, 2/1].round(0.001)

// Just ratios (mixolydian) in equal tempered cents
// (and therefor their deviation from equal temperament)
[1/1, 9/8, 6/5, 5/4, 4/3, 3/2, 8/5, 7/4, 2/1].ratiomidi.round(0.01)

// Retrograde of a 12-tone set
[0, 11, 10, 1, 9, 8, 2, 3, 7, 4, 6, 5].reverse

// Inversion of a 12-tone set
12 - [0, 11, 10, 1, 9, 8, 2, 3, 7, 4, 6, 5]

// And of course, retrograde inversion (see where I'm heading?)
(12 - [0, 11, 10, 1, 9, 8, 2, 3, 7, 4, 6, 5]).reverse

// Random transpositions of a 12-tone set
([0, 11, 10, 1, 9, 8, 2, 3, 7, 4, 6, 5] + 12.rand)%12

// Random permutation of a 12-tone set (out of 479,001,600)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].permute(479001600.rand) + 60
```

[45] Or should I say sound? These examples fall under the category of imaginary music.

Functions, Arguments, Scope

A function is a series of expressions enclosed in two braces. The entire function is usually (but not always) assigned to a variable. The lines of code are executed in order and the results of the last line is returned. When the function is called or run anywhere in the program it is as if all the lines were inserted in place of the function. A function is evaluated by using the *value* message.

A simple function with no variables or arguments, with its call:

```
   19.7.  Function

(
var myFunc;

myFunc = {100 * 20};

myFunc.value.postln;
)
```

The first line declares the variable name that will be used for the function. The second line is the function assignment ("make myfunc equal to the line {100 * 20}, or store the expression 100 * 20 in myFunc). Every place you put myFunc in your code the value 2000 will be used.

Arguments are like variables but they can be passed to the function when it is called. Declare arguments right after the opening brace. Next you declare variables if you want to use them.

Here is a function with arguments.

```
   19.8.  Function with arguments

(
var func;
func = { arg a, b;
   var r;
   r = (b * 20)%a;
   r.postln;
   };
func.value(15, 5);
)
```

It is often useful to pass an array containing a collection of items to the function, for example, when the number of elements you want to use in the function is subject to change. Consider the first example below, which sums three numbers. What if you wanted to sum three values one time, then ten the next, then two or six? Using arguments you would have to change the number of arguments each time, or write a new function. The solution is to pass an array as an argument. To do this you must first declare the array argument by preceding it with three dots, as seen in the last examples.

19.9. Function with array arguments

```
(
var func;
func = { arg a = 1, b = 2, c = 4;
    [a, b, c].sum;
    };
func.value(15, 5, 100);
)


(
var func;
func = { arg ... a;
    a.postln;
    a.sum.postln;
    [a.sum, 1.0.rand.sum].sum
    };
func.value(15, 5, 100);
func.value(15, 5, 100, 3, 78, 18, 367);
func.value(1, 2);
)

// Combine both syntaxes

(
var func;
func = { arg a = 0, b = 0 ... c;
    [a, b, c].postln;
    c.sum.postln;
    [c.sum, 3.0.rand.postln].sum.postln;
    (a/b*c).postln;
    };
func.value(15, 5, 100, 45);
func.value(15, 5, 100, 3, 99, 754, 78, 18, 367);
func.value(1, 2, 3, 4, 5);
)
```

Scope describes the effective range of a variable or argument. A variable can only be used inside the function where it is declared. Functions inside that function can make use of the outer variable. That is to say a function can use a variable that was declared in a function where it is enclosed. A global variable, declared with a tilde, will work everywhere.

19.10. Function with arguments and variables

```
var func, outside = 60;
~myGlobal = 22;
func = { arg first = 5, second = 9;
    var inside = 10;
    inside = (first * 11)%second;
    [first, second, inside, outside, ~myGlobal].postln; // all of these work
```

```
    (outside/inside).postln; //works
    };
//inside.postln; // uncomment this, it will not work
func.value(15, 6); // arguments passed to the function.
```

Can you predict the values of the three *myFunc* calls before running this code? The first has no arguments passed to the function and will use the defaults (10 and 2). The next two use defaults.

```
    19.11. Function calls

(//line 1
var myFunc;
myFunc = { arg a = 10, b = 2;
    b = (b * 100)%a;
    b.postln;
    };
myFunc.value; //line 7
myFunc.value(15); //line 8
myFunc.value(11, 30); //line 9
)
```

You can also use keywords with your own functions.

```
    19.12. Keywords

(
var myFunc;
myFunc = { arg firstValue = 10, secondValue = 2;
    firstValue = (firstValue * 100)%secondValue;
    firstValue.postln;
    };
myFunc.value;
myFunc.value(firstValue: 15);
myFunc.value(firstValue: 30, secondValue: 11);
myFunc.value(secondValue: 30, firstValue: 11);
myFunc.value(secondValue: 23);
)
```

In the previous examples the last line prints the final result. But in most functions there is a value returned to the place where the function is called. The last line of the function is returned. Here is an example that makes a little more musical sense.

```
    19.13. Return

(
var octaveAndScale;
octaveAndScale = { arg oct = 4, scale = 0;
    var scales, choice;
```

```
    oct = (oct + 1)*12; //translate "4" (as in C4) to MIDI octave (60)
    scales = [
        [0, 2, 4, 5, 7, 9, 11], //major
        [0, 2, 3, 5, 6, 8, 9, 11], //octatonic
        [0, 2, 4, 6, 8, 10] //whole tone
    ];
    scale = scales.at(scale); //more on the "at" message below
    choice = scale.choose; //choose a pitch
    choice = choice + oct; //add the octave
    choice //return the final result
    };

octaveAndScale.value; //choose from major scale, C4 octave
octaveAndScale.value(3); //choose from C3 octave, major scale
octaveAndScale.value(7, 2); //choose from C7 octave, whole tone scale
octaveAndScale.value(scale: 1); //choose from C4 octave, octatonic scale
)
```

When should you use a function? We have used them in just about every example so far. Messages such as *max*, *choose*, *midicps*, are functions (in receiver notation) that were put together by the authors of SC. When do you write your own functions? Convenience or clarity, for example when you use a section of code you developed, and you use it over and over. Rather than repeat the code each place it would be clearer and more efficient to write a function and use that single function each time you need a matrix. The other situation is when there is no existing message or function that does exactly what you need, so you have to tailor your own[46]. There is *rand*, *rand2*, *rrand*, *bilinrand*, *exprand*, etc., but maybe you need a random number generator that always returns pitches from a major scale in the bass clef. In that case, you could tailor your own function to your needs.

Practice, just flashing

In this patch the function returns the next frequency to be used in a flashing instrument. Each new pitch has to be calculated based on the previous pitch, because it uses pure ratios, as we did in a previous chapter. I've added a short printout that shows what the actual pitch is, and what the nearest equal tempered equivalent would be[47].

It begins with a patch from a previous chapter placed in a *SynthDef* with added arguments; *fundamental*, *decay*, and *filter*. This instrument is played successively using a task and loop. Before entering the loop I define the *freqFunc*, which picks a new pitch based on the previous pitch and a new pure interval ratio. This is called free just intonation (very difficult to do on natural instruments, very easy for computers). I add a *wrap* to keep the frequency within a certain range. The *nextEvent* not only controls when the next event will be, but that

[46] Check carefully, there probably is one that does what you want.

[47] If rounded to the nearest MIDI value. The actual difference will probably be greater if it strays more than a half step. I couldn't figure out how to do that (without an if statement). You do it and send me the code.

value is passed to the *Flash* instrument as a decay, ensuring one sound will decay shortly after the next sounds. At this writing I'm not convinced the filter works the way it should.

```
   19.14. Function practice, free, just tempered flashing

(
//run this first
SynthDef("Flash",
{
arg fund = 400, decay = 4, filter = 1;
var out, harm;

out = Mix.ar(
   Array.fill(7,
   {
      arg counter;
      var partial;
      partial = counter + 1;
      SinOsc.ar(fund*partial) *
         EnvGen.kr(Env.linen(0, 0, decay + 2),
         levelScale: 1/(partial*filter)
         ) * max(0, LFNoise1.kr(rrand(5.0, 12.0)))
   })
   )*0.3; //overall volume
out = Pan2.ar(out, Rand(-1.0, 1.0));
DetectSilence.ar(out, doneAction:2);
Out.ar(0, out)
}
).play(s);
)

(
//then this
r = Task({
var freqFunc, pitch = 440, nextEvent;

freqFunc = {arg previousPitch;
   var nextPitch, nextInterval;
   nextInterval = [3/2, 2/3, 4/3, 3/4, 5/4, 4/5, 6/5, 5/6].choose;
   nextPitch = (previousPitch*nextInterval).wrap(100, 1000);
   nextPitch.round(0.01).post; " != ".post;
   nextPitch.cpsmidi.round(1).midicps.round(0.01).postln;
   nextPitch
};

{
   nextEvent = [0.5, 0.25, 5, 4, 1].choose;
   pitch = freqFunc.value(pitch);
   Synth("Flash",
      [\fund, pitch, \decay, nextEvent, \filter, rrand(1.0, 4.0)]);
   //Choose a wait time before next event
   nextEvent.wait;
}.loop;
```

```
}).play
)
```

Practice: Example Functions

Here are some other functions you can try. They should all be inserted into the patch above (replacing the existing var list and freqFunc).

```
   19.15. Pitch functions

var freqFunc, pitches, pitch = 440, count, midiNote, nextEvent;
pitches = [60, 61, 62, 63, 64]; //declare an array of pitches
freqFunc = {
   midiNote = pitches.choose; //pick a pitch from the array
   midiNote.midicps; // return the cps for that pitch
};

var freqFunc, pitches, pitch = 440, count, midiNote, nextEvent;
pitches = [60, 62, 64, 67, 69, 72, 74, 76]; //declare an array of pitches
count = 0; //initialize count
freqFunc = {
   midiNote = pitches.wrapAt(count); // wrapped index of count
   if(count%30 == 29, //every ninth time
8     {pitches = pitches.scramble} //reset "pitches" to a scrambled
                  //verion of itself
   );
   count = count + 1; //increment count
   midiNote.midicps; //return cps
};


// My favorite:
var freqFunc, pitches, pitch = 440, count, midiNote, nextEvent;
pitches = [60, 62, 64, 67, 69, 72, 74, 76].scramble;
freqFunc = {
   midiNote = pitches.wrapAt(count); // wrap index of count
   if(count%10 == 9, //every tenth time
      {pitches.put(5.rand, (rrand(60, 76)))}//put a new pitch between
                     //65 and 75 into the array pitches
                     //at a random index
   );
   count = count + 1; //increment count
   midiNote.midicps; //return cps
};
```

19. Exercises

19.1. Write a function with two arguments (including default values); low and high midi numbers. The function chooses a MIDI number within that range and returns the frequency of the number chosen.

19.2. Write a function with one argument; root. The function picks between minor, major, or augmented chords and returns that chord built on the supplied root. Call the function using keywords.

20 - Iteration Using *do*, MIDIOut

Functions and messages use arguments. Sometimes one of the arguments is another function. The function can be assigned to a variable, then the variable used in the argument list, or it can be nested inside the argument list. Here are both examples.

20.1. function passed as argument

```
// function passed as variable

var myFunc;

myFunc = {
   (10*22).rand
};

max(45, myFunc.value);

// function nested

max(45, {(10*22.rand)})
```

The *do* function or message is used to repeat a process a certain number of times. The first argument is a list of items, typically an array, to be "done." The second argument is a function, which is repeated for each item in the list. The items in the list need to be passed to the function, so they can be used inside the function. This is done with an argument. You can name the argument anything you want. But it has to be the first argument.

20.2. do example

```
do(["this", "is", "a", "list", "of", "strings"], {arg eachItem; eachItem.postln;})

// or

do([46, 8, 109, 45.8, 78, 100], {arg whatever; whatever.postln;})
```

If the first argument is a number, then it represents the number of times the do will repeat. So *do(5, {etc})* will "do" 0, 1, 2, 3, and 4.

20.3. do example

```
do(5, {arg theNumber; theNumber.postln;})
```

Be sure the second argument is a function by enclosing it in braces. Try removing the braces in the example above. Notice the difference.

To me it makes more sense using receiver notation where the first argument, the 5, is the object or receiver and do is the message. The two grammars are equivalent.

20.4. do in receiver

```
do(5, {"boing".postln})
//same result
5.do({"boing".postln;})
```

In an earlier chapter we found it useful to keep track of each repetition of a process (when generating overtones). The *do* function has a method for doing this. The items being done are passed to the function, and the do also counts each iteration and passes this number as the second argument to the function. You can also name it anything you want, but the two have to be in the correct order. This next concept is a bit confusing because in the case of the *10.do* the first and second argument are the same; it "does" the numbers 0 through 9 in turn while the counter is moving from 0 through 9 as it counts the iterations.

20.5. do(10) with arguments

```
do(10, {arg eachItem, counter; eachItem.postln; counter.postln})
```

It is clearer when using an array as the first argument, or object being done. Remember that the position of the argument, not the name, determine which is which. Notice the second example the names seem incorrect, but the items being iterated over is the first argument, and the count is the second. With numbers (*5.do*) the difference is academic. But with an array the results can be very different, as shown in the third example. Note also the last item in this array is also an array; a nested array.

You can use the term *inf* for an infinite *do*. Save everything before you try it!

20.6. array.do with arguments

```
[10, "hi", 12.56, [10, 6]].do({arg eachItem, counter; [counter, eachItem].postln})

[10, "hi", 12.56, [10, 6]].do({arg count, list; [count, list].postln}) //wrong

[10, 576, 829, 777].do({arg count, items; (items*1000).postln});

[10, 576, 829, 777].do({arg items, count; (items*1000).postln});

inf.do({arg i; i.postln}) //this will, of course, crash SC
```

MIDIOut

Though the distinction is blurring, I still divide computer music into two broad categories; synthesis, that is instrument or sound design, and the second category is event/structure design. Event/structure design deals with pitch organization, duration, next event scheduling, instrument choice, and so on. If you are more interested in that type of composition then there really isn't much reason to design your own instruments, you can use just about anything[48]. This is where MIDI comes in handy.

MIDI is complicated by local variations in setup. You may have four external keyboards connected through a midi interface while I'm using SimpleSynth[49] on my laptop. Because of this I'll have to leave the details up to each individual, but the example below shows how I get MIDI playback.

To start and stop notes we will use *noteOn* and *noteOff,* each with arguments for MIDI channel, pitch, and velocity. The *noteOff* has a velocity argument, but note-off commands are really just a note-on with a velocity of 0, so I think it is redundant. You could just as well do the note on and the note off with a note on and velocity of 0: *m.noteOn(1, 60, 100); m.noteOn(1, 60, 0)*. The rest of the examples in this text that use MIDI will assume you have initialized MIDI, assigned an out, and that it is called "m."

The second example shows a simple alternative instrument.

```
   20.7.  MIDI out

(
MIDIClient.init;
m = MIDIOut(0, MIDIClient.destinations.at(0).uid);
)

m.noteOn(1, 60, 100); //channel, MIDI note, velocity (max 127)

m.noteOff(1, 60); //channel, MIDI note, velocity (max 127)

// Same thing:

m.noteOn(1, 60, 100); //channel, MIDI note, velocity (max 127)

m.noteOn(1, 60, 0); //channel, MIDI note, velocity (max 127)

// Or if you don't have MIDI

(
```

[48] Many of Bach's works have no instrumentation, and even those that do are transcribed for other instruments. In these works his focus was also on the events, not instrumentation.

[49] pete.yandell.com

```
SynthDef("SimpleTone",
   { //Beginning of Ugen function
   arg midiPitch = 60, dur = 0.125, amp = 0.9;
   var out;
   out = SinOsc.ar(midiPitch.midicps, mul: amp);
   out = out*EnvGen.kr(Env.perc(0, dur), doneAction:2);
   Out.ar(0, out)
   }
).play(s);
)

//Then in the examples replace this

m.noteOn(arguments)

//with

Synth("SimpleTone", arguments)
```

The next example is conceptual or imaginary[50] music. In the first few years of my exploration of computer assisted composition I began to think about writing every possible melody. I was sure this was within the scope of a computer program. After realizing all the possible variables (rhythms, lengths, variations, register, articulation), I decided to try a limited study of pitches only; every possible 12-tone row. That experiment was one of the many times I locked up the mainframe. Then I wondered if it were really necessary to actually *play* every row, or would writing it count as composition? What constitutes creation? Would a print out do? How about a soft copy? (Which I think is actually what crashed the machine back then.) Why not the idea of writing every melody? Couldn't the code itself, with the potential of every 12-tone row be a *representation* of every possible 12-tone melody? Conceptual music; that's where I left it.

And here is the latest incarnation, which actually plays every row, given enough time. It doesn't include inversions, retrograde, or inverted-retrograde because theoretically those will emerge as originals. It begins somewhere in the middle, but then wraps around, so it will eventually play every variation.

The *total.do* will repeat for every possible variation. It differs from a loop in that I can pass the argument *count* as a counter which is used to advance to each new permutation. The *permute* message takes a single argument, the number of the permutation. It returns an array with all the elements reordered without repetition. Try the short example first as an illustration.

I calculate 12 factorial as 479,001,600; the total possible variations. Once the permutation is chosen a second *do* steps through that array playing each note. The *r.start* and *r.stop* start and

---

[50] See Tom Johnson's wonderful *Imaginary Music*, unfortunately out of print. Thank heaven for interlibrary loan.

stop the *Task*. The last *127.do* is an "all notes off" message to stop any hanging pitches if you execute the *r.stop* in the middle of an event. The *thisThread.clock.sched* is used to turn off the MIDI pitch. It schedules the off event for *next*art* where *art* is the articulation. So if articulation is set to 0.9, it is rather legato. If it is set to 0.1 it will be staccato.

Practice, do, MIDIOut, Every 12-Tone Row

```
   20.8.  Every row

// Permute

25.do({arg count;
   postf("Permutation %: %\n", count, [1, 2, 3, 4].permute(count));})

//Every row

(
//run this first
var original, total, begin, next, art;
original = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11];
total = 479001600;
begin = total.rand;
next = 0.125;
art = 0.9;
("Total playback time = " ++ (total*next/3600).asString ++ " hours.").postln;
r = Task({
   total.do({arg count;
      var thisVar;
      thisVar = original.permute(count+begin);
      thisVar.postln;
      (thisVar + 60).do({arg note;
         m.noteOn(1, note, 100);
         thisThread.clock.sched(next*art, {m.noteOff(1, note, 100); nil});
         (next).wait
      });
   })
})
)

//then these
r.start;
r.stop; 127.do({arg i; m.noteOff(1, i, 0)})
```

This is conceptually correct because it steps through every possible permutation. It is, however, a bit pedantic since the variations from one permutation to the other are slight.

The following chooses a row at random. Not strictly 12-tone since it is possible for two pitches to be repeated. Theoretically this would take longer to present every possible permutation in a row, since it could repeat permutations.

```
   20.9.  Every random row
```

```
(
var original, total, begin, next, art;
original = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11];
total = 479001600;
next = 0.125;
art = 0.9;
("Total playback time = " ++ (total*next/3600).asString ++ " hours.").postln;
r = Task({
   total.do({
      var thisVar;
      thisVar = original.permute(total.rand);
      thisVar.postln;
      (thisVar + 60).do({arg note;
         m.noteOn(1, note, 100);
         thisThread.clock.sched(next*art, {m.noteOff(1, note, 100); nil});
         (next).wait
      });
   })
})
)

r.start;
r.stop; 127.do({arg i; m.noteOff(1, i, 0)})
```

20. Exercises

20.1. Write an example using two *do* functions, nested so that it prints a multiplication table for values 1 through 5 (1*1 = 1; 1*2 = 2; . . . 5*4 = 20; 5*5 = 25).

20.2. Write another nested *do* that will print each of these arrays on a separate line with colons between each number and dashes between each line: [[1, 4, 6, 9], [100, 345, 980, 722], [1.5, 1.67, 4.56, 4.87]]. It should look like this:
1 : 4 : 6 : 9 :
---
100 : 345 : etc.

21 - Control Using *if*, *do* continued, Arrays, MIDIIn, Computer Assisted Analysis

Control message "if"

Artificial intelligence and computer-assisted composition begin with logic controls. That is, telling the machine what to do in certain circumstances. *If* you are hungry *do* open the fridge. *If* there is no bread then *do* go to the store. *If* you have enough money *do* buy 4 loaves. *Do* come home and open the fridge again. *If* there is jam and jelly *do* choose between the two and make a sandwich. *Don't* use marinated tofu. There are several methods of iteration and control such as *while*, *for*, and *forBy*, but *do* and *if* are the most common.

The *if* message or function takes three arguments: an expression to be evaluated, a *true* function, and a *false* function. It evaluates the expression to be true or false (as in the previous examples where 10 < 20 returned *true* or *false*) and returns the results of the first function if true, the second if false.

if(expression, {true function}, {false function})

The true or false often results form a comparison of two values separated by the operators covered earlier such as "<" for less than, ">" for greater than, "==" for equals, etc. (Note the difference between two equals signs and one. "=" means store this number in the variable, "**==**" means "is it equal to?"). Run both examples of code below. The first one evaluates a statement which returns *true* (because 1 does indeed equal 1) so it runs the first function. The second is false, so the *false* function is run.

```
   21.1.  if examples

if(1 == 1, {"true statement";},{"false statement";})

if(1 == 4, {"true statement";},{"false statement";})

// Commented:

if(
   1 == 1,   //expression to be evaluated; "1 is equal to 1" true or false?
   {"true statement";},       //if the statement is true run this code
   {"false statement";}       //if it is false run this code
)
```

Here are other Boolean operators
< less than
> greater than
<= less than or equal to
>= greater than or equal to

!= not equal to[51]
== equal to

The message *or* combines two statements, returning true if either are correct; or(a > 20, b > 100). The message *and* combines two statements, returning true only if both are correct; and(a > 20, b > 100). The word *true* is true and *false* is false.

```
   21.2.  if examples

if((1 == 1).and(5 < 7), {"both are true"},{"maybe only one is true";})

if((1 == 20).and(5 < 7), {"both are true";},{"one or both are false";})

if((1 == 20).and(24 < 7), {"both are true";},{"one or both are false";})

if((1 == 4).or(true), {"true is always true";},{"1 does not equal 4";})

if(false.or(true), {"true is always true";},{"true wins with or";})

if(false.and(true), {"true is always true";},{"but false wins with and";})

if(or(10 > 0, 10 < 0), {34},{78})

if((1 == 1).and((10 > 0).or((5 < 0).or(100 < 200))), {78},{88})
```

These isolated numerical examples seem moot without a context (i.e. why would I ever use the expression 10 > 0? and why would you just post "true statement" or "false statement"). The *if* function is usually used in combination with some iterative process such as *do*. Here is a real musical example. The code below begins at MIDI 60 (C4). It then picks a new MIDI interval, adds it to *m* and returns that new value. Watch the results.

```
   21.3.  do 50 MIDI intervals

(
m = 60;
50.do(
   {
   m = m + [6, 7, 4, 2, 11, 8, -2, -6, -1, -3].choose;
   m.postln;
   }
)
)
```

I've biased the choices so that there are more intervals up than down. So eventually the MIDI values exceed a reasonable range for most instruments. Even if I carefully balanced the

[51] In computer parlance the exclamation point, or "bang" means "not." The inside joke is that it lends a more accurate representation of advertising, such as "The must see movie of the year! (not!)"

choices[52] it is conceivable that a positive value is chosen 20 times in a row. The *if* statement below checks the value during each iteration and reduces it by two octaves if it exceeds 84 and increases it by two octaves if below 36.

```
   21.4.  do 50 MIDI intervals

(
m = 60;
50.do(
   {
   var next;
   next = [6, 17, 14, 2, 11, 8, -12, -16, -1, -3].choose;
   "next interval is : ".post; next.postln;
   m = m + next;
   "before being fixed: ".post; m.post;
   if(m > 72, {m = m - 24});
   if(m < 48, {m = m + 24});
   " after being fixed: ".post; m.postln;
   }
)
)
```

When writing a little piece of code like this it is worth poking around in SC to see if a function already exists that will do what you want. There is *wrap*, which wraps a value around, but in this example we are buffering, not wrapping. So in this case we really do need the extra lines of code.

Below shows a *do* iteration over an array of pitch classes with an *if* test to look for C, D, or E. The computer doesn't understand these as actual pitches (see the discussion on strings below), but just text. Even so it does know how to compare to see if they are equal.

```
   21.5.  pitch class do

(
["C", "C#", "D", "Eb", "E", "F", "F#", "G", "Ab", "A", "Bb", "B"].do(
   {arg item, count;
      if((item == "C").or(item == "E").or(item == "G"), //Boolean test
         {item.post; " is part of a C chord.".postln;}, //True function
         {item.post; " is not part of a C chord".postln;} //False function
         )
      }
   )
)
```

[52] There are several better schemes for managing interval choices. For example, you could always choose positive values to add, reduce that result to a single octave using modulo, so that all values are between 0 to 12, then choose the octave separately.

You might say we have taught the computer a C chord. This is where AI begins.

A series of *if* statements can be used to define a region on the screen. When the mouse enters the region with an x greater than 0.3, but less than 0.5, and a y that is greater than 0.3, but less than 0.7 (all conditions are "true") it generates a positive value, or a trigger. In this example the * is equivalent to *and*. There are no true or false functions, just the values 1 (on) and 0 (off).

```
   21.6.  Mouse Area Trigger

(
{
var aenv, fenv, mgate, mx, my;
mx = MouseX.kr(0, 1);
my = MouseY.kr(0, 1);
mgate = if((mx>0.3) * (mx<0.5) * (my>0.3) * (my<0.7), 1, 0);
aenv = EnvGen.kr(Env.asr(0.1, 0.5, 2), mgate);
fenv = EnvGen.kr(Env.asr(1, 1, 2), mgate, 1000, 100);
RLPF.ar(Saw.ar(100)*aenv, fenv, 0.1)
}.play
)
```

One nice trick with *if* is to control when to *postln* rather than *post,* which does not print a new line:

```
   21.7.  new line

(
100.do(
   {
      arg count;
      100.rand.post;
      if(count%10 == 9, //Every 9th time
         {" new line: ".postln;}, //print a carriage return
         {" : ".post;} //just " * " without a return
         );
   }
)
)
```

while

The *while* function repeats the second argument (a function) until the first argument (an evaluation function) returns a false. This is a dangerous tool for beginners. If you write a condition that has no chance of ever returning false, SC will run forever. (This is often the cause of many applications locking up.) The first example below shows a common error. This line will repeat until the second function stores the value 10 in the variable a. But *10.rand* picks numbers between 0 and 9, so it will never pick 10, and will run until you force it to quit. It is therefore a good idea to put an additional condition that will stop it after a

specified number of repetitions. But if you're like me, I forget to include the c = c + 1 and it crashes anyway.

The last line shows a typical use for *while*. This line should tell you how long it took to pick the number 13 out of 100.

```
   21.8.  while

a = 0; while({a != 10}, {a = 10.rand}) // save before doing this

// Safer?

c = 0; a = 0;
while({and(a != 10, c < 100)}, {a = 10.rand; c = c + 1;})


(
a = 0; c = 0;
while({(a != 13).and(c < 10000)}, {a = 100.rand; c = c + 1;});
c
)
```

for, forBy

The *for* allows you to specify the start and end points of the iteration, and *forBy* includes an argument for step value. In a previous example we built a patch with odd harmonics. The counter was used to calculate each harmonic, but since it began at 0 we had to add 1 (*counter +1*). We also had to include a bit of math for odd harmonics only. But with *forBy* the calculation can be integrated.

```
   21.9.  for, forBy

34.for(84, {|i| i.midicps.postln});

34.forBy(84, 5, {|i| [i, i.midicps].postln});

1.forBy(24, 2, {|i| [i, i*500].postln}); // odd harmonics

(  // additive square wave
{
o = 0;
1.forBy(24, 2, {|i| o = o + SinOsc.ar(i*500, mul: 1/i)});
o*0.4
}.scope
)
```

MIDIIn

MIDIIn reads input from a MIDI source. There are a number of ways this can be used. The first is to trigger an instrument you have designed previously, as in this example:

```
   21.10. MIDI input to trigger SynthDef

(
// First define this instrument
SynthDef("simpleInst",
{
arg midi = 60, vel = 60, inst;
inst = {
   SinOsc.ar(
      freq: (midi.midicps)*(LFNoise0.kr(12) + 2).round(1),
      mul: vel/256)
   }.dup(5) *
   EnvGen.kr(Env.perc(0, 3), doneAction: 2);
Out.ar(0, inst)
}
).play
)

MIDIIn.connect;

(
// Then link it to MIDI input

MIDIIn.noteOn = {
arg src, chan, midiNum, midiVelo;
Synth("simpleInst", [\midi, midiNum, \vel, midiVelo]);
};

)
```

To control any other instrument that has already been defined using SynthDef, just replace the "simpleInst" with the instrument name and be sure to match the arguments with those appropriate for that instrument.

Real-Time Interpolation

The next use of MIDI input is to reflect the input back to your MIDI equipment (or different equipment), after being modified in some way. At first blush this may seem like a pretty good practical joke, but it can also be used for interpolation. Interpolation capitalizes on the success of existing compositional components such as structure or intuition of an improviser. It uses this proven component by mapping it to some or all musical elements of new material. For example, a pianist may show facility with diatonic scales common to jazz during improvisation, but loses that clarity if asked to incorporate unusual scales or designs, 12-tone systems, synthetic scales, or complex canonic techniques. By reading the input from his real

time improvisation using materials he is familiar with, you can map that fluid intuition to the more difficult experiments.

When interpolating data you often have to apply a logical filter using an *if* statement, allowing some material to pass unchanged while modifying others. While this is usually a practical consideration, the following examples apply the filter in a novel way. The first is a simple delay, but it is only applied to velocities above 60, using an *if* statement as a filter. Velocities below 60 pass through unchanged.

The second takes MIDI input and reflects it to the MIDI device inverted on the axis of C4. However, the *if* statement searches for white keys only (*[0, 2, 4, 5, 7, 9, 11]%12*) and only applies the inversion to those MIDI values. Black keys are not inverted. So the blues scale C4, D4, Eb4, E4, F4, F#4, G4, A4, Bb4, C5 would invert to C4, Bb3, *Eb4*, Ab3, G3, *F#4*, F3, Eb3, *Bb4*, C3. Needless to say this could really throw off a performer, so to be effective they should be isolated with headsets monitoring the *correct* midi information, to encourage intuitive improvisation with accurate feedback, while the audience hears only the interpolated version. Try this with multiple layers, classical works, improvisation, unusual scales, tunings, modified rhythms, etc. The possibilities are endless.

21.11. MIDI input interpolation with *if()* filter

```
// Simple delay for velocities above 60
(
MIDIIn.noteOn = {arg src, chan, num, vel;
var delay = 0.5;
thisThread.clock.sched(delay, {m.noteOn(1, num, vel)});
};
MIDIIn.noteOff = {arg src, chan, num, vel;
var delay = 0.5;
thisThread.clock.sched(delay, {m.noteOff(1, num, vel)});
};
)


// Interpolation: inversion of white keys only

(
var white;
white = [0, 2, 4, 5, 7, 9, 11];

MIDIIn.noteOn = {arg src, chan, num, vel;

if(white.includes(num%12),
   {m.noteOn(1, (60 - num) + 60, vel);},
   {m.noteOn(1, num, vel)});
};

MIDIIn.noteOff = {arg src, chan, num, vel;

if(white.includes(num%12),
   {m.noteOff(1, (60 - num) + 60, vel);},
```

```
    {m.noteOff(1, num, vel)});
};
)
```

Analysis

The last use of MIDI input is to perform analysis. The example below reads MIDI values, calculates the interval from the previous value, then stores that value in an array, thus keeping track of how often a composer or improviser uses each interval. There are two versions. The first counts an interval as motion to any pitch, whether up or down, as the same interval since they are inversions; C4 up to G4 (a fifth) is counted as the same as C4 down to G3 (a fourth down). The second version, commented out, counts intervals up and down as the same; C4 up to G4 (a fifth up) is the same as C4 down to F3 (a fifth down).

```
    21.12. MIDI input interpolation with if() filter

(
// Run this to start reading input

var previousPitch = 62, thisInterval;
~intervals = Array.fill(12, {0});

MIDIIn.noteOn = {arg src, chan, num;
// Counts inversions as the same
thisInterval = (12 - (previousPitch - num))%12;
// Counts intervals up or down as the same
// thisInterval = abs(previousPitch - num);

// uncomment to monitor values
// [previousPitch, num, thisInterval].postln;

~intervals.put(thisInterval, ~intervals.at(thisInterval) + 1);
previousPitch = num;

// uncomment to watch in real time
// ~intervals.postln;

};
)

// When done, run this line

~intervals;
```

Practice

[Insert example]

```
    21.13. Example
```

Example

21. Exercises

21.1. True or false?
if(true.and(false.or(true.and(false))).or(false.or(false)), {true}, {false})

21.2. Write a *do* function that returns a frequency between 200 and 800. At each iteration multiply the previous frequency by one of these intervals: [3/2, 2/3, 4/3, 3/4]. If the pitch is too high or too low, reduce it or increase it by octaves.

21.3. Write a *100.do* that picks random numbers. If the number is odd, print it, if even, print an error. (Look in the *SimpleNumber* help file for odd or even.)

21.4. In the patch below, add 10 more SinOsc ugens with other frequencies, perhaps a diatonic scale. Add *if* statements delineating 12 vertical regions of the screen (columns) using *MouseX*, and triggers for envelopes in the ugens, such that motion across the computer screen will play a virtual keyboard.

```
(
{
var  mx, mgate1, mgate2, mgate3, mgate4;
mx = MouseX.kr(0, 1);
mgate1 = if((mx>0) * (mx<0.5), 1, 0);
mgate2 = if((mx>0.5) * (mx<1.0), 1, 0);
Mix.ar(
SinOsc.ar([400, 600],
   mul: EnvGen.kr(Env.perc(0, 1), [mgate1, mgate2]))
)*0.3}.play;
)
```

21.5. Using existing SynthDefs, modify the MIDI example above so that it executes those instruments, passing MIDI number, noteOn and noteOff, and velocity.

21.6. Modify the interpolation example with new rules for which notes are filtered and how they are modified.

22 - Collections, Arrays, Index Referencing, Array Messages

A collection or array is a group of items. Arrays are enclosed in brackets and each item is separated by a comma. Here is an array of integers.

[1, 4, 6, 23, 45]

You can have arrays of strings. A "string" is a group of characters that the computer sees as a single object.

["One", "Two", "Three", "Four"]

Or you can have a mixture. Note the "34" (in quotes) is not understood by SC as the integer 34, but a string consisting of a character 3 and a 4. But 1, 56, and 3 are integers.

[1, "one", "34", 56, 3]

You can also perform math on entire arrays. That is to say, the array understands math messages.

```
   22.1.  array math

(
a = [1, 2, 3, 4]; //declare an array
b = (a + 12)*10; //add 12 to every item in the array, then multiply them
          //all by 10 and store the resulting array in b
b.postln;
)
```

Can you predict the outcome of each of these examples?

```
   22.2.  array.do and math

(
a = [60, 45, 68, 33, 90, 25, 10];
5.do(
   {
      a = a + 3;
      a.postln;
   }
)
)

   22.3.  array + each item

(
a = [60, 45, 68, 33, 90, 25, 10];
5.do(
   {arg item;
```

```
        a = a + item;
        a.postln;
    }
)
)
```

This is a little harder; predict the outcome of the example below before running it. I'm using two arrays. The first is stored in the variable *a* and used inside the *do* function. The second is the object being used by the *do* function. So the *item* argument will be 2 on the first iteration, 14 on the second, 19, and so on.

```
    22.4.  two arrays

(
a = [60, 45, 68, 33, 90, 25, 10];
b = [2, 14, 19, 42, 3, 6, 31, 9];
b.do(
    {arg item;
        item.post; " plus ".post; a.post; " = ".post;
        a = a + item;
        a.postln;
    }
)
)
```

It is also possible to test an array for the presence of a value. The message we need to use is *includes*. This message answers true if the array contains an item or object. It takes one argument; the object you are looking for. So that *[1, 2, 3, 4].includes(3)* will return a *true* and *[1, 2, 3, 4].includes(10)* will return a *false*. These true or false returns can be used in an *if* function (as we saw in the previous chapter).

```
    22.5.  testing an array

(
a = [60, 45, 68, 33, 90, 25, 10];
b = [25, 14, 19, 42, 33, 6, 31, 9];

100.do(
    {arg item;
        if(a.includes(item), {item.post; " is in a ".postln});
        if(b.includes(item), {item.post; " is in b ".postln});
    }
)
)
```

Arrays can be used to store collections of pitches, frequencies, durations, articulations, or instrument choices. They can be used in musical styles that make use of scales, cells, collections, series, or rows.

To retrieve a single value within an array, use the *at* message. The argument for *at* is the index number. Remember that computers begin counting with 0. So the array [1, 2, 3, 4] has four items with index numbers 0, 1, 2, and 3. Given the array [9, 12, "this", 7], index 1 is 12. At index 2 is "this." If the index is too large for the array you will get a *nil*. This is called a wild pointer. But we often have counters that exceed the size of an array, and in this case we use *wrapAt* to return the value at the index modulo the size of the array.

```
   22.6.  referencing an item in an array

[12, 4, 871, 9, 23].at(3) //index 3 is "9"

[12, 4, 871, 9, 23].at(124) //wild pointer, will return nil

[12, 4, 871, 9, 23].wrapAt(124) //will wrap around and return 23
```

Array messages

Here is a collection of array messages.

```
   22.7.  arrays messages

a = [1, 2, 3, 4]; //assigns the array to the variable "a"

a.post; //prints the array

a + 5; //adds five to each item in the array

a*3; //multiplies it, etc.

a.do({arg item; function}) //iterates over each item passing each item
                           //to the function

a.at(index) //refers to item at index number

// Here are some new ones. Run each of them to see what they do:

[1, 2, 3, 4].reverse.postln; //reverses the array

[1, 2, 3, 4].rand.postln;

[1, 2, 3, 4].scramble.postln; //scrambles the array

[1, 2, 3, 4].size.postln;// returns the size (number of items)

Array.fill(size, function); //fills an array with "size"
//number of arguments using function

a = Array.fill(5, {10.rand}); a.postln;

a = Array.rand(12, 0, 12)
```

```
[1, 2, 3, 4].add(34).postln; //adds an item to the array

//Note about add. You have to pass it to another array variable to
//make sure the item is added. So the code would have to be:

a = [1, 2, 3];
b = a.add(10);
a = b;

[1, 2, 3, 4].choose; //chooses one of the values

[1, 2, 3, 4].put(2, 34).postln; //puts second argument at
//index of first argument

[1, 2, 3, 4].wrapAt(index) //returns item at index with a wrap

//example:

30.do({arg item; [1, 2, 3, 4].wrapAt(item).postln});
```

One practical use for an array reference is to define available choices. Previously we have used *[array].choose* which limited the choices to items in the array. The following does the same thing using the message *rand*. The first returns a series of numbers, 0 through 10. But if the *rand* is placed in the *at* then its values become the index of the array and the value at that index number is returned.

As I mentioned earlier the index should not be larger than the array size. We fixed one example using *wrapAt*. Another method is to include the message *size*, which returns the size of the array, and can be used with *rand*. The number returned by *size* will be one larger than the actual index numbers. The array [3, 5, 7, 1, 8, 12] has a size of 6, but the index values are 0 through 5. That's ok because *rand* chooses a number from 0 to, but not including, its receiver.

```
   22.8.  array of legal pitches

20.do({12.rand.postln;}) // random numbers 0 through 11

// random numbers chosen, but array reference returns only pitches
// from a major scale

20.do({[0, 2, 4, 5, 7, 9, 11].at(6.rand).postln})

// Be sure the rand range is not too large

20.do({[0, 2, 4, 5, 7, 9, 11].at(12.rand).postln})

// To protect against this, use array.size or wrapAt

a = [0, 2, 4, 5, 7, 9, 11];
20.do({a.at((a.size).rand).postln})
```

There are shorthand symbols for *at*, *wrapAt*, and a fold and clip index (which do not have message equivalents). For these use the "@" and "|" symbols. The /*i*/ is shorthand for *arg i;*.

```
   22.9.  Array index shorthand

[45, 37, 99, 367, 9] @ 3 // "at" index 3

[45, 37, 99, 367, 9] @@ 25 // "wrapAt" index 25

[45, 37, 99, 367, 9] @|@ 25 // fold at index 25

[45, 37, 99, 367, 9] |@| 25 // clip at index 25

30.do({[0, 2, 4, 5, 7, 9, 11] @ (12.rand).postln})

30.do({ |i| ([0, 2, 4, 5, 7, 9, 11] @@ i).postln})

30.do({ |i| ([0, 2, 4, 5, 7, 9, 11] @|@ i).postln})

30.do({ |i| ([0, 2, 4, 5, 7, 9, 11] |@| i).postln})
```

Practice, Bach Mutation

The *at* message also allows you to reference array items at varying degrees: every other one, every third one, etc., as illustrated in the minimalist interpolation of Bach's invention.

```
   22.10. array of legal pitches

// (engage midi)
(
var pitch;

r = Task({
// Try other versions
pitch = [12, 0, 2, 4, 2, 0, 7, 4, 12, 7, 4, 7, 5, 4, 5, 7, 0, 4, 7, 11] + 60;
inf.do({arg h, i;
   pitch.size.do({arg j;
   var n;
      //every one, then every other one, then every third, etc.
      n = pitch.wrapAt(j*(i+1));
      if((j%20 == 19), {n.postln}, {n.post; " ".post});
      m.noteOn(1, n, 100);
      thisThread.clock.sched(0.1, {m.noteOff(1, n, 100); nil});
      0.1.wait;
   });
});
})
```

```
)

r.start;
r.stop; 127.do({arg i; m.noteOff(1, i)});
```

22. Exercises

22.1. Write a function that returns 50 random midi values, 10 at a time from these scales; 10 from a whole tone scale, then 10 from minor, chromatic, and pentatonic.

22.2. Write a function that scrambles 12 MIDI pitches 0 through 11 then prints out that array transposed to random values between 0 and 11, but kept within the range 0 to 11 (if transposition is 10 then an original pitch 8 would become 6). Print the transposition interval and the transposed array.

23 - Strings, String Collections

A character string is stored as an array. The last item in the array is a 0. The 0, or termination, indicates the end of the string. The string can be any series of character combinations (words or numbers) identified by quotes, such as "my 4 strings." Internally SC stores the ascii integers for each character in an array (see the chart in a later chapter). The first position is the 'm' (176), the second 'y' (121), third ' ' (space or 32), fourth is the *character*, not the number, '4' (111). Since it is an array, it can be included in a *do* function.

```
   23.1.  String as array


"CharacterArray".at(0) // change the index to numbers between 0 and 13

"CharacterArray".at(0).ascii

"This is a string that is actually an array".do(
   {arg each, count;
      [count, each.ascii, each].postln;
   })
```

An array of strings is an array of arrays or multi-dimensional array. The array *["one", "two" "three"]* holds three arrays. The first array contains the characters 'o', 'n', 'e', and 0. The second is 't', 'w', 'o', and 0.

Arrays understand math. Strings, as arrays understand math, but not the way you would expect. If you wanted to transpose a set of pitches you might try adding the string or array of string to a number, (e.g. "C#" + 5), as in the example below:

```
   23.2.  "C" + 5?

("C#" + 5)

(
a = ["C#", "D", "Eb", "F", "G"];
a = a + 5;
a.postln;
)
```

This example runs, but the results are not what we wanted. What if we want the program to **generate** numbers, but **print** strings? This can be done using an array of strings as a reference.

Here is an array of strings[53] and a line of code posting one of the strings followed by a random choice, and a *do* with random choices:

```
   23.3.  pitch array index

(
a = ["C", "D", "E", "F", "G"];
a.at(3).postln; //post item at index position 3 in the array a
)

(
a = ["C", "D", "E", "F", "G"];
a.at(5.rand).postln;
)

(
a = ["C", "D", "E", "F", "G", "A", "B"]; //pitch class array
"count\trandom\tpitch at index:".postln; //header
10.do( //do 10 items
   {arg item, count; //use arguments item and count
      var pick;
      pick = a.size.rand;
      count.post; "\t\t".post; //print the number of this iteration
      pick.post; "\t\t".post; //print the number I picked
      a.at(pick).postln; //print the item at that array position
   })
)
```

You can save on *postln* messages by using the concatenate message ++. This is used to put two strings together. So "this " ++ "string" will be "this string." Another method for combining data is to print an entire array containing a group of items, e.g. *[count, b, a.at(b)].postln;*. Finally, you can use *postf*, which formats a post line by inserting a list of arguments into a string, replacing "%" characters. Try the example several times to confirm it is indeed choosing different values. The second example is a more concise version.

```
   23.4.  concatenated string

(
a = ["C", "D", "E", "F", "G", "A", "B"];
10.do(
   {arg item, count; var b;
      b = a.size.rand;
      ("Item " ++ count ++ " : " ++ b ++ " = " ++ a.at(b)).postln;
      // or
      // postf("Item % : % = %\n", count, b, a.at(b))
   })
)
```

[53] Even though there is only one character it is still called a string; the character, then the terminating 0.

```
// More concise

do(10, { ["C", "D", "E", "F", "G", "A", "B"].wrapAt(100.rand).postln;})
```

Now I can combine the string array with the 12-tone melodies.

```
   23.5.  Every 12-tone row with pitch class strings

(
//Initiate MIDI, run this first
var original, total, begin, next, art, pcstrings, count;
original = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11];
pcstrings = ["C  ", "C# ", "D  ", "Eb ",
   "E  ", "F  ", "F# ", "G  ", "Ab ", "A  ", "Bb ", "B  "];
total = 479001600;
count = 0;
begin = total.rand;
next = 0.125;
art = 0.6;
("Total playback time = " ++ (total*next/3600).asString ++ " hours.").postln;
r = Task({
   inf.do({
      var thisVar;
      thisVar = original.permute(begin + count);
      thisVar.do({arg e; pcstrings.at(e).post});
      "".postln;
      (thisVar + 60).do({arg note;
         m.noteOn(1, note, 100);
         thisThread.clock.sched(next*art, {m.noteOff(1, note, 100); nil});
         next.wait
      });
      count = count + 1;
   })
})
)

//then these
r.start;
r.stop; 127.do({arg i; m.noteOff(1, i, 0)})
```

A Moment of Perspective.

For me, the goal of computer assisted composition is for the machine to do all the boring detailed parts so I can get on with the creative fun part. In the example above the result is just printed pitches, not music. Even so, we could stop here and it would be useful in composition exercises. This is close to how it was done forty years ago. Composers began with what you can do now; get the computer to at least do the numbers. But it took four pages of code or hundreds of punch cards. With SC it takes one line. Here is such an example (the first value

is duration in eighths, the second is time until next in eighths, the next is midi pitch, the next is volume):

```
   23.6.  Illiac suite?

60.do({[8.rand, 8.rand, (rrand(36, 72)), 10.rand].postln;})
```

That's quite an advance over Hiller's program in the 50s. But I'm still stuck transcribing the stuff into manuscript and getting someone to play it. The advantage of current technology is immediate playback. Even if my final product is intended for real musicians, I can use the computer for trials. I can try ideas at a rate of about four a minute in contrast to a few months (when using live musicians).

Ten years ago we could get the cpu to generate actual sounds, but it took overnight. In 2000 that amount of time was cut down to 5 minutes, but that still took pages of code and two separate programs (one to crunch the numbers, another to generate the sounds). Today it's in real time, and it takes about ten lines of code.

Practice, Random Study

Here is an example. A complete composition based on a few simple ideas: three instruments, a set number of events, random choices for pitch, duration, next event[54], and amplitude. Set your MIDI playback so that channels 1, 2, and 3 have different instruments.

I should add that this experiment is not very interesting as is. (Random walks never are.) But our goal is fast experimental turnaround. So listen to this example for a while then tweak the random choices and listen again. Change the range for durations, next events, pitch choices, etc. Instead of a random range (e.g. *duration = rrand(0.5, 2.0)*) try a random index to an array of choices (e.g. *duration = durationArray.at(10.rand)*). Try different scales in a pitch array. Try frequencies rather than MIDI values. Rather than an infinite *do* set up a series of defined *do* loops for formal design, changing the parameters for each one (20 long durations then 5 short durations, high notes then low, etc.).

```
   23.7.  (Biased) random study

(

a = Task({
   inf.do({arg i;
   var note, dur, next, amp, inst;
   note = rrand(24, 84);
   dur = rrand(0.1, 0.5);
   amp = rrand(30, 127);
   next = rrand(0.1, 0.5);
```

[54] Duration and next event are different. The current event could have a duration of 0.5 while the next event is in 3 seconds. The current event could be 3 seconds and the next event 0, or now.

```
    m.noteOn(1, note, amp);
    thisThread.clock.sched(dur, {m.noteOff(1, note); nil});
    next.wait
    })
});

b = Task({
    inf.do({arg i;
    var note, dur, next, amp, inst;
    note = rrand(24, 84);
    dur = rrand(0.1, 0.5);
    amp = rrand(30, 127);
    next = rrand(0.1, 0.5);
    m.noteOn(2, note, amp);
    thisThread.clock.sched(dur, {m.noteOff(2, note); nil});
    next.wait
    })
});

c = Task({
    inf.do({arg i;
    var note, dur, next, amp, inst;
    note = rrand(24, 84);
    dur = rrand(0.1, 0.5);
    amp = rrand(30, 127);
    next = rrand(0.1, 0.5);
    m.noteOn(3, note, amp);
    thisThread.clock.sched(dur, {m.noteOff(3, note); nil});
    next.wait
    })
});
)

a.start;
b.start;
c.start;
a.stop; 127.do({arg i; m.noteOff(1, i)})
b.stop; 127.do({arg i; m.noteOff(2, i)})
c.stop; 127.do({arg i; m.noteOff(3, i)})
```

23. Exercises

23.1. Using this sentence set up an iteration that chooses one of the characters converts it to an ASCII number, reduces that to below an octave (12) and prints that value.

23.2. With the random study above, add lines of code that print out all the information you would need to transcribe the choices made. They include instrument, pitch, amplitude, start time, and duration. You will have to add a "time" variable that sums up each "next" value as it is chosen.

23.3. Use an array of pitch class strings, and an array of numbers 0 through 11 to generate a random 12-tone row and print a pitch class matrix showing all transpositions of original, retrograde, inversion, and retrograde-inversion. Here's a hint: begin with numbers only (from 0 to 11), it's easier to see the transposition and inversions. Try inverting just one row. Then do just the transpositions, then the inversions, then scramble the original row. Add the pitch class string last. User interface always comes last because it's the hardest. It is deceptively simple; about six lines of code. The printout should look like something like this.

```
C  B  Bb F  Eb F# E  A  D  C# Ab G
C# C  B  F# E  G  F  Bb Eb D  A  Ab
D  C# C  G  F  Ab F# B  E  Eb Bb A
G  F# F  C  Bb C# B  E  A  Ab Eb D
A  Ab G  D  C  Eb C# F# B  Bb F  E
F# F  E  B  A  C  Bb Eb Ab G  D  C#
Ab G  F# C# B  D  C  F  Bb A  E  Eb
Eb D  C# Ab F# A  G  C  F  E  B  Bb
Bb A  Ab Eb C# E  D  G  C  B  F# F
B  Bb A  E  D  F  Eb Ab C# C  G  F#
E  Eb D  A  G  Bb Ab C# F# F  C  B
F  E  Eb Bb Ab B  A  D  G  F# C# C
```

24 - More Random Numbers

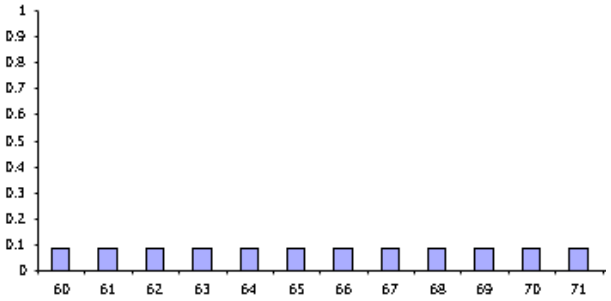
In this section we will talk about different types of random processes. Not many people write purely random music (though it seems to be). It's not very interesting anyway. There is usually some type of filtering, skew, or bias. Even what might seem to be a very broad choice, all audible frequencies (30 Hz to 20000 Hz), has a bias toward those frequencies. The With the "random" music example above we have biased the choices to a range of MIDI values (rather than continuous frequencies), a range of possible durations, and so on. Even the works of Cage, often thought of as random, are shaped by bias. 4'33" has a strong bias: no notes from the piano.

What is random? What is the same? Imagine a scale going from exactly the same to completely random. What would be at both ends? In terms of digital audio, where a sound is represented by a 16 bit number, completely random would mean an equal chance of any number in the 16 bit range occurring for each new sample generated. Such a system results in noise. Exactly the same I guess would be all 0s.

I like to think in degrees of randomness. Webern *sounds* more random than Mozart. What I mean is that the patterns in Webern are more difficult to follow. What we look for in a musical experience is a satisfying balance between similarity and variety. That changes with each person, mood, or context (e.g. concert hall vs. movie theatre where we tend to be more tolerant). There are two ways to approach this balance. One is to add levels of complexity (as in additive synthesis), the other is to filter complex processes (as in subtractive synthesis). In this chapter we will filter.

Biased Random Choices

A biased random choice limits or filters a broad possible set of outcomes. We have seen it several times, as in a random range: *rrand(60, 72)*. Another method is to weight the possibilities toward one value, or a set of values. In these systems the weight of each possibility is expressed as a value between 0 and 1, where all possibilities add up to 1. In the example above there is an equal probability that any value between 60 and 71 is chosen (these are integers), so each individual value has a 1/12, or 0.08333 chance of being chosen. The probability distribution is described as a graph, shown below.



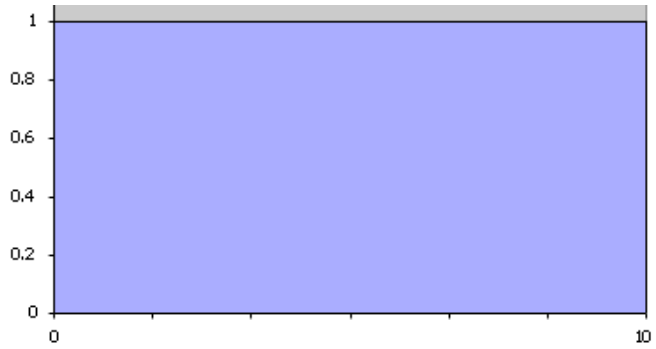
Likewise, in *[60, 65, 69, 54].choose* each value has a 0.25 chance of being chosen, since there are four values (0.25*4=1.0).

One method for skewing the choices toward a single value is to load the dice by repeating choices:

```
   24.1.  loaded dice

[60, 60, 65, 69, 54].choose
```

This code still divides the random choice between all elements in the array (each has a 0.2 or 1/5th chance of being chosen), but since 60 is entered twice there is a 0.4 chance of 60 being chosen.

In the case of *10.0.rand*, the choices are not discrete integers, but rather a range of possible floating point values. The chart showing these probabilities would have a straight line across the top, indicating that any value between the low number (0.0) and high number (10.0) has an equal chance of being chosen.



Biasing the outcome of this model requires a bit of math.

At the end of The Price is Right there is a good example of a random bias. There are three people spinning a wheel to get the highest number. If you spin a low number you try again. The results are biased toward higher numbers because the highest number wins.

Apply this process to a pair of dice: roll both dice, but use the higher of the two numbers. The possible combinations[55] are.

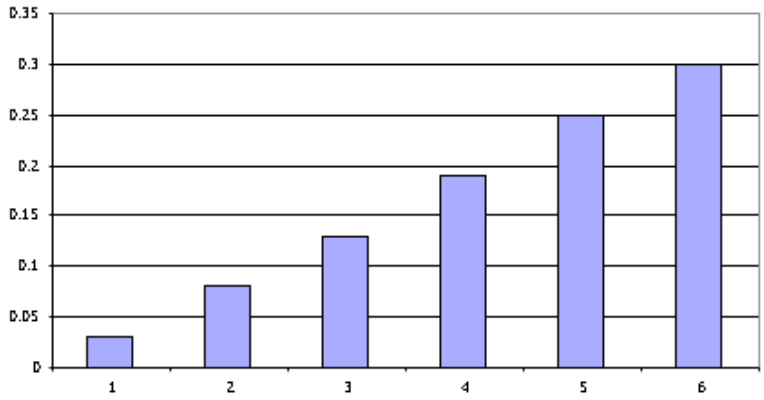
[1, 1] [1, 2] [1, 3] [1, 4] [1, 5] [1, 6]
[2, 1] [2, 2] [2, 3] [2, 4] [2, 5] [2, 6]
[3, 1] [3, 2] [3, 3] [3, 4] [3, 5] [3, 6]
[4, 1] [4, 2] [4, 3] [4, 4] [4, 5] [4, 6]
[5, 1] [5, 2] [5, 3] [5, 4] [5, 5] [5, 6]
[6, 1] [6, 2] [6, 3] [6, 4] [6, 5] [6, 6]

[55] SC can be used for lots of quick experiments. I generated this example using this line of code: 6.do({arg i; 6.do({arg j; [i + 1, j + 1].post}); "".postln;})

Any pair that contains a 6 will return a 6 (right column and bottom row). But only one pair will return a 1: [1, 1] (upper left corner). The alternate shading below shows each group returning 1s, then 2s, 3s, etc.

	1:	2:	3:	4:	5:	6:
1:	[1, 1]	[1, 2]	[1, 3]	[1, 4]	[1, 5]	[1, 6]
2:	[2, 1]	[2, 2]	[2, 3]	[2, 4]	[2, 5]	[2, 6]
3:	[3, 1]	[3, 2]	[3, 3]	[3, 4]	[3, 5]	[3, 6]
4:	[4, 1]	[4, 2]	[4, 3]	[4, 4]	[4, 5]	[4, 6]
5:	[5, 1]	[5, 2]	[5, 3]	[5, 4]	[5, 5]	[5, 6]
6:	[6, 1]	[6, 2]	[6, 3]	[6, 4]	[6, 5]	[6, 6]

There are 11 combinations that result in 6 (11 out of 36 is 0.3), 9 combinations that result in 5 (0.25), 7 of 4 (0.19), 5 of 3 (0.13), 3 of 2 (0.08), and only 1 of 1 (0.03). Here is the probability distribution chart these values.

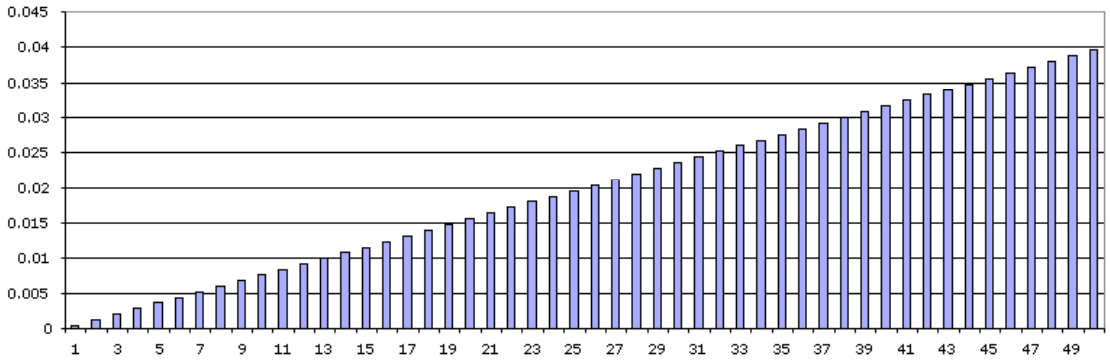


Using 100 sided dice would give even more numbers, but the trend would be the same. 100 and any other number would result in 100. But only 1 and 1 would return 1. The formula for calculating the total for each number is n*2-1. The probability for each value is (n*2-1)/total. The total is the highest value.

24.2. high bias calculation

```
n = 100; n.do({arg i; i = i + 1; ((i*2-1)/(n.squared)).postln})
```

Here is a graph for 50 choices using a "higher of the two" bias. The more choices the smoother the probability graph.



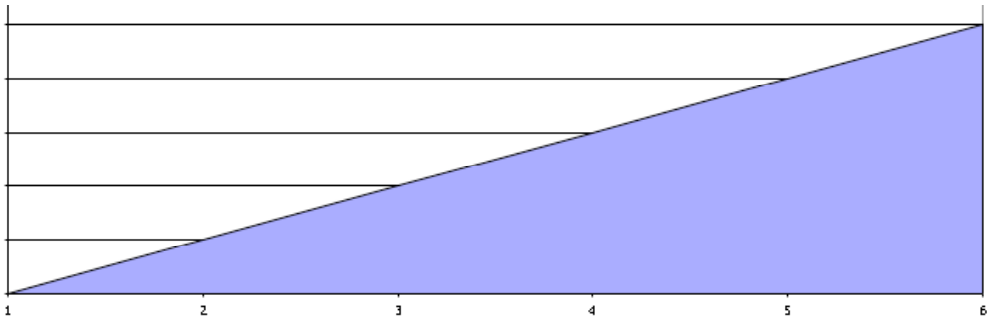
Floating point random choices yield many more possibilities (though still finite). The trend of the graph is the same.

In SC use the min or max function.

```
24.3.  bias float
```

```
max(6.0.rand.round(0.01), 6.0.rand.round(0.01));
```

Any number and 6 will result in 6, but only 0 and 0 returns 0. The graph for this function would be:



What would be the probability graph for this code?

```
24.4.  bias
```

```
min(100.0.rand, 100.0.rand);
```

A little tougher question; how would this code change the outcome (note that the second random choice is not a typo, I use 100.rand to indicate an integer choice, not a floating point):

```
24.5.  bias
```

```
max(200.0.rand, 100.rand);
```

How about this one (the possible results are 0 to 100, but ask yourself how many combinations will result in 0? how many will result in 50? how many 100?):

```
   24.6.  bias

(100.rand + 100.rand)/2
```

How would you do an inverted triangle? a choice between -10 and 0? a bias toward -10? a choice between 0 and 10 with a bias toward 7.5? How would you assign a percentage choice to each value, so that 60 has a 0.25 chance of being chosen, 61 has 0.1, 62 has 0.5, etc.?

Complicated? It used to be. SC has a good collection of biased random functions for choosing numbers. Before we look at them I want to demonstrate a system for testing. The method I typically use is to declare a counting array that can keep track of the random choices[56], then increment the position of the array as the choices are made. At the end I use the *plot* message, which plots an array on a graph. Try the first line to see how this works.

```
   24.7.  test bias

[0, 1, 0.5, 0.178, 0.9, 0].plot; // plots an array

(
a = Array.fill(100, {0}); //fill an array with 0s
1000.do( //do 1000 iterations; more choices, smoother graph
   {
      b = 100.rand; //pick a random number between 0 and 999
      a.put(b, a.at(b) + 1); //increment that position
                  //in the array
   }
);
a.plot; //plot the results.
)
```

The random choice is between 0 and 999. That number is used as an index to reference and increment that position of the array *a* using *put*. The first argument for *put* is the index of the array, the second is the value you want to store. It changes the value at index *b*, by replacing it with one greater than what it previously was (*a.at(b) +1*). Using this method you can test each of the random processes we've discussed above, or the functions listed below.

With some of the biased random choices in the examples above a float is returned. In this case, you must convert the value to an integer before using it as a reference to the array.

```
   24.8.  Test float bias
```

---

[56] The following examples use integers and not floats. For random choices the distinction is minor. The range of 0.0000001 to 1.0000000 is pretty much the same, in terms of possible choices, as 1 to 1000000. If you want to use floats just convert them to integers using *div*, such as *10.0.rand.div(1)*.

```
(
a = Array.fill(100, {0}); //fill an array with 0s
1000.do( //do 1000 iterations
   {
      b = (100.rand + 100.rand)/2.div(1);
      a.put(b, a.at(b) + 1);
   }
);
a.plot; //plot the results.
)
```

Here are the random functions and tests.

24.9. rand tests

```
(
a = Array.fill(100, {0});
1000.do({
b = 100.rand; // 0 and this, rand2 does negative values (-this to this)
   a.put(b, a.at(b) + 1);
});
a.plot;
)

(
a = Array.fill(100, {0});
1000.do({
b = 100.linrand; // linear distribution, bilinrand adds - values
   a.put(b, a.at(b) + 1);
});
a.plot;
)

(
a = Array.fill(100, {0});
1000.do({
b = rrand(30, 75); // random range
   a.put(b, a.at(b) + 1);
});
a.plot;
)

(
a = Array.fill(100, {0});
1000.do({
b = 100.exprand(1).div(1);
   a.put(b, a.at(b) + 1);
});
a.plot;
)
```

```
(
a = Array.fill(100, {0});
1000.do({
b = max(100.rand, 100.rand)
   a.put(b, a.at(b) + 1);
});
a.plot;
)

(
a = Array.fill(100, {0});
1000.do({
b = max(100.rand, 100.rand, 100.rand);
   a.put(b, a.at(b) + 1);
});
a.plot;
)

(
a = Array.fill(100, {0});
1000.do({
//this.coin returns true this percent of the time
b = if(0.2.coin, {max(20.rand, 20.rand)}, {min(80.rand, 80.rand) + 20});
   a.put(b, a.at(b) + 1);
});
a.plot;
)
```

24. Exercises

24.1. What is the most random event you can think of? What is the most predictable? Given enough information about the most random, could you predict it? Is there any chance the predictable event would not have the outcome you expected?

24.2. Draw a probability graph for this code: [1, 2, 3, 1, 3, 4, 5, 2, 3, 1].choose.

24.3. Monkeys at typewriters: Will a random process result in recognizable melodies? Devise a system that generates random 7 note melodies and tests for matches with a collection of known melodies (e.g. Mary Had a Little Lamb). How long before the system generates one of 7 or so recognizable melodies? (Use arrays.)

25 - Aesthetics of Computer Music

Why Music on Computers?

No doubt you've given this some thought, since you're this far into a computer music text. Here is why I use computers to compose: they are fast, accurate, complex, thorough, obedient, and obtuse. They will do precisely what you ask, and only what you ask without complaint or question. They will never fill in the blanks or interpolate with their own ideas.

Fast

Music software allows a fast turn around when trying the next idea. This invites experimentation. Take for example mutation (or transcription, commuting, transformation); using an existing pattern or musical work mutated into something new. A non-computer assisted example of mutation is the reworking of Bach's "Come Sweet Death." It is a simple, elegant idea. The results are stunning. The choir sings all the pitches in the original chorale with one new rule of transformation; each performer moves to the next pitch when they run out of breath. Though the composer[57] surely had an idea what the results would be he probably didn't know exactly until he talked an actual living, breathing choir into trying it. That process could take months.

Other examples include playing selected patterns of pitches (every other pitch, every third pitch), mixing styles (swap or map each interval from Bach with intervals from Webern), tunings (reduce all intervals by 1/2, just intonation, different modes), retrograde, inversion, diminution, etc. Trying these ideas with pen, paper, and performers takes time and is discouraging. Learning a computer language takes time, but the payoff is the ability to change two or three numbers in the code and hear the results of your idea immediately.

Accurate

I am fascinated with microtones and alternate tunings. This is, imho, the next big step for art music. But for hundreds of years the microtuning community has existed in all but exile. Practitioners were (are?) seen as eccentrics and outcasts living on another plane (planet?). They had to build their own instruments. Their scores gather dust, filled with excruciatingly contrived notation.

A computer has no difficulty at all distinguishing 440 and 440.001. I'm not suggesting the computer be the performer in all cases. Maybe its role is educator; training performers in the subtle differences of microtuning. It could ultimately be the notation of the future. What is notation but a convenient method of communicating the composers intention to the performer? We use paper and pen because that's the technology most performers know. Why use paper at all? Couldn't we say the 0s and 1s coming from the computer *are* the notation? If

---

[57] I can't find the composer.

we can format it in a way the performer understands then it will serve as notation. How about sound itself, the highest level language? Performers understand that format best. Previously we have had to devise complex accidentals as a code that represent a particular pitch only because we have to work with paper. Why not just give the performer the pitch? The truth is we often do; these days a score is usually accompanied by a CD as a practical aid.

Our compositions are constrained by existing technology. We use an equal tempered scale because the piano, or more precisely pianists, can't manage the number of keys and strings required for both A-flat and G-sharp. So we compose as though they were the same pitch. They are not. Shouldn't a composer make use of both A-flat and G-sharp? Arguably many do, and strings, voices, even brass can make the distinction. But it's not that way in our collective consciousness.

The same is true for rhythm. We write tempo and meter based music partly because it is satisfying but also because we have to put it on paper that we each look and realize in our own minds. There has to be a reference point that you and the performer agree on: the meter and tempo. Occasionally a composer will dare to venture into asymmetric divisions (7:8), but rarely compound asymmetry (quarter, sixteenth, sixteenth, eighth, dotted quarter of a 7:8) and rarely anything more difficult. If you do write something more complex (11/8 against 7/8), who is going to play it? You know someone? Who is going to play it correctly? Who is going to play it correctly on a new music recital in three months?

A computer has no difficulty with rhythms accurate to 0.0001 seconds. Again, I'm not suggesting the computer become the performer in all cases. Maybe it's the notation. Why not just give the performer the rhythm: when they hear the tick (or see an event scroll[58] to a stationary bar on the screen) they sing it. The computer is the notation and the conductor. Is that cheating? When monks ran out of red ink (it was expensive) they hit on the idea of using hollow notes and pretending they were red. We now use that system. Was that cheating?

Complex and Thorough: I Dig You Don't Work

Computers can manage complex ideas easily. Imagine a score with thirty-two parts, each playing a different rhythm with higher multiples of the beat: 1 to 1, 2 to 1, 3 to 1, 4 to 1, etc. (sort of like the harmonic series), all the way to 32 to 1. Performers? Not a chance. Computer? Piece of cake. As a matter of fact, as Jon Appleton once said, the limits of computer music are not in the machines, but in we humans, and our ability to perceive.

The computer will also try combinations thoroughly, some that we may dismiss because of our bias. Take out a sheet of paper and on your own write a minimalist poem using creative combinations of the words "I dig you don't work." In our classes we usually find thirty or so that make sense. But a computer can quickly show you every combination. You are then

---

[58] Spacial or time-line notation has never seemed very convincing to me. Performers guess at the timing and it becomes aleatoric. A scrolling score would match accuracy and complexity. It would communicate metered music as well as complex asymmetric or non-metric rhythms to hundredths of a second. There would be little or no learning curve; just play the pitch when it hits the cursor.

encouraged to consider each variation as valid because it is true to the design or system (every possible iteration). You might encounter unexpected combinations that make sense once you are faced with them on the page.

Below is the SC version of this exercise. This is an exhaustive iteration with no repeated words. How many make sense? Nearly all except those where you and I are consecutive. (And even they work with creative inflection, or interpreting "I" as it's homonym "eye." Having every iteration before you encourages thinking out of the box.)

```
   25.1.  I Dig You Don't Work

var text;
text = ["I", "DON'T", "DIG", "YOU", "WORK"];
121.do({arg i; i.post; text.permute(i).postln;})

// Or even better


Task({
i = 0; t = ["i ", "don't ", "dig ", "you ", "work"];
{
u = "";
t.permute(i).do({|e| u = u ++ e});
u.speak;
i = i + 1;
3.0.wait
}.loop
}).play
```

Obedient and Obtuse

Computers are obedient, obtuse servants. They make no observations or conclusions. Humans, on the other hand, can not suppress observation and conclusion. Consider this series: C C B G A B C C A G. It's almost impossible for you, a human, to look at those symbols without making meaningful assumptions or searching for a pattern. You probably assumed they were pitch classes. You might even have sang them to see if you recognized the melody. Some readers might even have said the author under their breath. But did I say they were pitches? A computer will not make any of those assumptions. If I asked you to complete the series you probably could. A computer would scratch its virtual head.

[New] This is useful in two ways. The first is proving compositional method. If you write code that produces something different from what you wanted the error is not the performer (the cpu), but your system (the code). During a composition seminar a student was explaining his methods for a work we just heard; a random walk. First he tried using a deck of cards to choose pitches. He complained that this resulted in familiar patterns such as repeated pitches, arpeggios, or even the occasional melody quotation. He said, and I quote, "it wasn't very random." I immediately corrected him. It was random. It just didn't *sound* random to him. Repeated pitches and arpeggios are a possible result of a random process. The system (a deck

of cards) did what he wanted, but did not achieve his goals. The error wasn't in the mechanics of realization, but his system, and that was evident because he used an honest method of realization. If, on the other hand, you had asked a pianist to play "random" notes, they would intuitively avoid octaves, arpeggios, and familiar tunes. Your compositional system is still flawed, but you wouldn't know it because the performer is not being honest or true to the system.

Human bias is evident in aleatoric works. How often have you performed, or seen performed, some vague instruction or graphic notation that ends up like every other aleatoric work? The performer plays what he thinks the composer wants; that avant-garde pseudo random abstract bee-blonkish whatever. My last experience (maybe this is why it was my last?) was a single page with instructions to "play" a graphic. It was produced on a draw program with a cheesy mechanical method of duplication (like a kaleidoscope, but not as cool). Something you might expect from a child. I decided to "play" the process: cheap electronic imitation of art. At the next rehearsal at my cue I turned on an annoying electronic version of Mozart from a VHS tape rewinder. The composer went ballistic. I was trying to be obedient and obtuse, but it was not what the composer wanted. Where was the missing link; me, or the instructions? (Or the lack of instructions.)

A computer will give you exactly what you ask for. If it's not what you want, then you can't blame the machine. The flaw is in your compositional methods, and understanding that can be useful in honing your skills.

When the system, if realized honestly, returns something unexpected you have two choices; accept this aberration as a new idea to consider or refine your instructions to better reflect what you want. The first time I tried to realize the "I Don't Dig You Work" exercise I made a mistake in the code and the computer printed "I", then "I, I", then "I, I, I." It was doing what I asked, but not what I expected. Should I rewrite the code or include these possibilities in my poem? Maybe annoying electronic Mozart was a good idea.

Escaping Human Bias

While between degrees I had a job writing children's songs for educational videos. Our methods were this: The producers had an idea about the music they wanted. They described it to me, then I wrote out intuitively what they described and what I imagined. These melodies were completely predictable and common; exactly what the boss wanted. This method wasn't wrong. In that job it was essential. It's just not very interesting to me now.

Intuitive composers imagine an event, describe the circumstances that will result in that event, give the description to performers, then make adjustments to the description and the performer's understanding until the event imagined is realized. I like to think of this as backward composition. You are working back to something you imagined. I also believe it is

more of a craft than an art. You are reproducing something that already exists conceptually[59]. (It's probably closer to the truth to say that composers work with a mixture of system and intuition. I delineate them here for effect.)

The systematic composer works the opposite way. Instead of imagining an event then describing the circumstances required for that event, a set of circumstances are described and the composer discovers the results. In my mind this is what constitutes forward composition. (Herbert Brün would say the difference is political.) I believe all true innovators work this way. Instead of asking "I wonder how I can get this sound" they ask "I wonder what sound would result if I did this." I'm only paraphrasing here, but I recently read an article about the composers who worked on *Forbidden Planet*. They were asked what process or plan they followed to realize their ideas. The answer was none. They just patched cords, twisted knobs and sat back marveling at what came out. "I wonder what would happen if I did this?"

How do you escape your own bias? You either use a method of iteration (like trying every interval in turn until one strikes you as useful) or you use a random system (plunking at the piano for new ideas). Both are systems. (The truth is, even intuitive composers work with systems when they get stuck for ideas.)

One way to escape your own bias is to listen to other composers. We've all seen this evolution in students. They are initially shocked, even offended by a new style. (And it doesn't have to be something radical like Crumb's *Black Angels*. On first hearing Bebop was, to many seasoned musicians, indecipherable.) But gradually they begin to like it, they study it, they study with composers who write that way, and it is infused into their own style. It becomes one of their biases.

But what happens when you have picked the brains of all of your instructors? Where do you go after you have studied all existing styles? How do you move on to ideas so new that they would surprise your instructors and you? Can you *become* the next composer that challenges and stretches your own music?

You use a system. You ask the question first.

> "My next piece, which I have not yet learned to like." –Herbert Brün
>
> "The key to real freedom and emancipation from local dependence [human bias] is through scientific method. . . . Originality is the product of knowledge, not guesswork. Scientific method in the arts provides an inconceivable number of ideas, technical ease, perfection and, ultimately, a feeling of real freedom, satisfaction and accomplishment." –Joseph Schillinger from the Scientific Basis for the Arts

---

[59] David Cope's *Virtual Music* draws into question whether this method of composition is creativity or craft, since it can be reproduced mechanically.

Integrity to the System

What is your obligation to a new idea? When the computer printed "I", "I, I", "I, I, I" was I obligated to accept that as the poem? Not really.

There is a range of attitudes one might have toward an experiment. The first is "this is what I wrote, it is true to the system, it is my next piece which I have not yet learned to like." I've come to love works by well known composers that at first hearing seemed awkward and counterintuitive. It makes me wonder if all musical taste is a matter of exposure. One could argue that our intervals and tuning system–moving from high ratios or dissonance to low ratios or consonance–is based on scientific principle, but Javanese Gamelan are tuned to irrational intervals. Their melodies are just as musical. Nurture or nature? Probably both.

The secretaries in our department once sent a visiting Russian student to my office who was excited to show someone a discovery he had made; he called it "true" music. After a few minutes of explanation I realized he had figured out (on his own, apparently) that the piano was not tuned to pure intervals and had worked out the calculations for correct just tunings. He had used a computer to calculate (on paper only) the frequencies of a work by Bach when played with these pure ratios. I reluctantly pulled down a Grove and opened it to the page describing his "true" music. (He was stunned, but at least his numbers were correct.)

But his method was to calculated each interval from the previous one. I asked him about pitch drift. He hadn't thought of that. I said "won't the pitches eventually drift from the original tonic?" He puzzled for a while and said yes. "Wouldn't you find that offensive to the ears?" More thought, then "it can't be offensive, because it is true [to the system]." I had to agree. Who am I to say what is beautiful? To him integrity to the system was more important than the actual sound. We have become acclimated to equal temperament, which is out of tune. In 100 years will we accept pitch shift in free-just intonation?

So the first level would be to say "this is the work, because it's true to they system."

The next level is to consider the system a genetic code where millions of variations co-exist. I've often been unhappy with a particular "performance" of a generative work. Other "performances" are gems. In this case the system is a seed that encapsulates hundreds, millions of flowers. You can pick one to admire, or admire them all. I tried a realization of Webern's Kinderstuck (an early work where he only uses P0) that substitutes a new version of the row at each quotation. My sense was they were all equally valid variations of the same work. Webern wrote the genetic code, I grew lots of flowers. Could that have been his intention, which was at the time out of practical reach?

When I compose using generative rather than specific methods I feel more in tune with creative forces than if I wrestle over each note.

> “Since I have always preferred making plans to executing them, I have gravitated towards situations and systems that, once set into operation, could create music with little or no intervention on my part. That is to say, I tend towards the roles

> of planner and programmer, and then become an audience to the results" -Brian Eno (cite Alpern).
>
> "When one arrives at this correct conception of art, there can be no more distinction between science and inspired creating. The farther one presses, the more everything becomes identical, and at last one has the impression of encountering no human work, but rather a work of Nature[60]." –Webern

Finally, the last level of integrity is that the results of the system are used simply as inspiration; ideas to be worked out intuitively, or discarded completely. You get to decide.

Now, on to systems.

[60] Webern loved hiking among wild flowers.

25. Exercises

25.1. Name five microtonal composers (without looking them up).

25.2. Write out and play every possible combination of the pitches B, and F resolving to C and E.

26 - Pbind, Mutation, Pfunc, Prand, Pwrand, Pseries, Pseq, Serialization

Pbind

Pbind links together different parameters of musical events. It streams those values using a pattern to the current *Environment*. The *Environment* is another behind the scenes structure we haven't worried about until now. It contains a group of global default values attached to symbols. Below is an incomplete list. You will notice that they are often the same value expressed in different ways: freq and midinote, amp and db.

\amp = 0.1, \db = -20, \degree = 0, \dur = 1, \freq = 261.62, \legato = 0.8, \midinote = 60, \note = 0, \octave = 5, \out = 0, \pan = 0, \root = 0, \scale = [0, 2, 4, 5, 7, 9, 11], \server = default, \velocity = 64, \instrument = default, \out = 0, \group = 0

You pass values to the environment with Pbind by matching a symbol (e.g. \freq) with the value (e.g. 400) or a function that returns a value (e.g. *rrand(400, 900)*). If none is supplied, then it uses defaults. The defaults are useful because they allow you to focus on one or two elements of composition. If you just want to experiment with pitch you don't have to specify amplitude, duration, instrument, etc: It will use the defaults.

Before using Pbind you have to load the synth description library. So for the examples in this chapter I will assume you have run the code below.

```
   26.1.  Read global library

SynthDescLib.global.read
```

The simplest Pbind would set just one value, in the example below, frequency. The symbol is *freq*, the value is 600.

```
   26.2.  Basic Pbind

Pbind(\freq, 600).play
```

It becomes more interesting when you use other streaming processes. The first example shows a *Pfunc* that evaluates any function, in this case random values for pitch. The second adds duration. The third passes midi pitches by way of the symbols *degree* and *octave*. Degree is a scale degree, octave is the octave, where 5 is the C4 octave.

```
   26.3.  Pbind with frequency function

Pbind(\freq, Pfunc({rrand(100, 900)})).play;

Pbind(
   \freq, Pfunc({rrand(100, 900)}),
```

```
    \dur, Pfunc({rrand(0.1, 1.5)})).play

Pbind(
    \degree, Pfunc({8.rand}),
    \oct, Pfunc({rrand(3, 7)}), //or try \octave?
    \dur, 0.2).play

Pbind(
    \scale, [0, 2, 4, 6, 8, 10],
    \degree, Pfunc({6.rand}),
    \oct, Pfunc({rrand(3, 7)}), //or try \octave
    \dur, 0.2).play

Pbind(
    \scale, [0, 2, 3, 5, 6, 8, 10, 11],
    \degree, Pfunc({8.rand}),
    \oct, Pfunc({rrand(3, 7)}), //or try \octave
    \dur, 0.2).play
```

All of the instruments you've designed up until now are available in the environment. All of the arguments that you created in conjunction with those instruments are symbols that can be matched to values. If you happened to have used arguments like midinote, then they will fit right into the existing *Environment*. If you named them something else, such as *midiPitch* and *art* for articulation, you just need to use those symbols.

```
    26.4.  Pbind with Previous Instrument Definitions

Pbind(
    \instrument, "KSpluck3",
    \midiPitch, Pfunc({rrand(34, 72)}),
    \art, Pfunc({rrand(1.0, 2.0)}),
    \dur, 0.1
).play
```

dur, legato, nextEvent

I've always made a distinction between the duration of this event and the duration until next event. The default *\dur* in the Pbind is used as both, but really means the time until the next event. The actual duration of this event is determined by *legato*, which is a percentage of *dur*. The default is 0.8. Given a duration (time until next event) of 4 the actual duration of this event would be 0.8*4 or 3.2. So legato could be used for actual duration (2.5 legato + 4 duration = 10 actual duration).

When I created the KSpluck3 the argument *art* (articulation) really meant duration, but articulation too. In this example it means actual duration while *\dur* means time until next event. Hey, they're just variables and arguments. Name them whatever you want.

You can use a Pbind to launch an fx. But since it is a single event there is really no need; just start it first (remember, In first, Out second), then run the synth that is routed to it in a Pbind.

```
   26.5.  Pbind with previous fx

Synth("delay1");

Pbind(
   \instrument, "bells",
   \freq, Pseq([100, 400, 1000, 1500, 2000])
).play;
```

Pbind is very useful in interpolation; transforming an existing style or pattern with slight adjustments. First I define a simple instrument with arguments suited to serial studies. Remember, we aren't concerned with the character of the instrument. Our focus is on pitch, duration, amplitude, etc. Remember that each time you create a new instrument you have to reread the *DescLib*.

```
   26.6.  Simple serial instrument

(
SynthDef("SimpleTone",
{arg midinote = 60, amp = 0.9, dur = 1, pan = 0, legato = 0.8;
   Out.ar(0,
      Pan2.ar(
         SinOsc.ar(midinote.midicps, mul: amp)
         *
         EnvGen.kr(Env.perc(0, dur*legato), doneAction: 2),
         pan
   )
   )
}).load(s);

SynthDescLib.global.read
)
```

Next is a model that will serve for interpolation studies in pitch. The midinote is determined by a routine; a function that remembers previous states, such as the counter. The melody is a Bach invention. If you are familiar with the original I should point out a couple of cheats: the melody is transposed to a single octave (once we get into the transformations the effect is the same) and I repeat some notes to dodge the issue if rhythmic values. Likewise, the effect is the same.

Instead of just entered the MIDI values I use the *degree* and *scale* array, which work together. *Scale* is the whole/half-step orientation of harmonic minor. *Degree* is which step of that scale to use. The advantage of these two combined is that it allows me to change the mode (e.g. from harmonic minor to Lydian) without having to reenter all the notes from scratch. The *degree* array stays the same and I can change the scale.

The degree array is adjusted by –1 only to cater to my musical sense. I would normally reference the scale array using 0, 1, 2, for scale steps 1, 2, and 3. But typing out or changing

notes in the melody is much easier if I can think in musical terms, that is, 1 is the first step of the scale, [1, 3, 5] is a tonic arpeggio, [1, 4, 6] is first inversion.

```
   26.7.  Pitch Model for Mutation

(
var degreeSeq, nextPitch, scale;

scale = [0, 2, 3, 5, 7, 8, 11];

degreeSeq = [1, 2, 3, 4, 5, 6, 7, 6, 5, 4, 3, 2,
   3, 3, 5, 5, 1, 1, 4, 4, 7, 7, 2, 2] - 1;


nextPitch = Routine({
   inf.do({arg count;
      scale.wrapAt(degreeSeq.wrapAt(count)).yield;
   })
});

Pbind(
   \instrument, "SimpleTone",
   \midinote, nextPitch + 60,
   \dur, 0.1
   ).play
)
```

Here is where you can satisfy your compositional curiosity with quick answers in terms of actual playback. What if we played it backward (*count.neg*)? How about every other pitch (*count*2*)? every third, fourth, and so on. How about an inversion (*12 – scale.wrapetc.*)? Halved intervals (*scale = [etc.]/2*), doubled intervals, different modes (*scale = [0, 2, 3, 5, 7, 8, 9]*)? Here are a few more.

```
   26.8.  Experiment

// increasing multiples
   var mul = 1;
   inf.do({arg count;
      if(count%20 == 19, {mul = mul + 1});

// occasional random values

   inf.do({arg count;
      if(count%6 == 5, {12.rand.yield},
      {(scale.wrapAt(degreeSeq.wrapAt(count*mul))).yield});

// occasionally dodge sequence

      if(count%6 == 5, {scale.choose.yield},
      {(scale.wrapAt(degreeSeq.wrapAt(count*mul))).yield});
```

```
// gradually change scale

       if(count%6 == 5, {scale.put(scale.size.rand, 12.rand)});
       (scale.wrapAt(degreeSeq.wrapAt(count))).yield;
```

Prand, Pseries, Pseq

There are a dozen or so useful patterns (look in the Streams help file). I'll demonstrate three here. *Prand* returns a random choice from an array (see also *Pwrand*, or weighted random pattern), *Pseries* returns a series of values with arguments for start, step, and length. *Pseq* seems to be used the most. It steps through an array of values. They can all be nested, as shown in the last example.

```
   26.9.  Patterns

(
f = 100;

Pbind(
   \instrument, "SimpleTone",
   \midinote, Pfunc({
      f = ([3/2, 4/3].choose) * f;
      if(f > 1000, {f = f/8}); //.fold or .wrap didn't do what I wanted
      f.cpsmidi
      }),
   \dur, 0.2
   ).play

)

(
Pbind(
   \instrument, "SimpleTone",
   \midinote, Prand([60, 62, 64, 65, 67, 69, 70], inf),
   \dur, 0.1
   ).play

)

(
// The dur array is 1s and 2s, representing eighth notes.

Pbind(
   \instrument, "SimpleTone",
   \midinote, Pseq([70, 58, 60, 62, 60, 58, 65, 62, 70,
      65, 62, 65, 63, 62, 63, 65, 58, 62, 65, 69], inf),
   \dur, Pseq([2, 1, 1, 1, 1, 2, 2, 2, 2, 2,
      2, 1, 1, 1, 1, 2, 2, 2, 2, 2] * 0.2, inf)
   ).play
)
```

And of course, you can run several Pbinds. (Bach would have wanted it that way.) Try 1 or 0.5 as the last dur for 2 and 3. Pbind also responds to *mute* and *unmute*. Listen to this example long enough to hear the phase shift:


```
(
a = Pbind(
    \instrument, "SimpleTone",
    \midinote, Pseq([70, 58, 60, 62, 60, 58, 65, 62, 70,
        65, 62, 65, 63, 62, 63, 65, 58, 62, 65, 69], inf),
    \dur, Pseq([2, 1, 1, 1, 1, 2, 2, 2, 2, 2,
        2, 1, 1, 1, 1, 2, 2, 2, 2, 2] * 0.1, inf),
    \pan, -1
    ).play;

b = Pbind(
    \instrument, "SimpleTone",
    \midinote, Pseq([70, 58, 60, 62, 60, 58, 65, 62, 70,
        65, 62, 65, 63, 62, 63, 65, 58, 62, 65, 69, 0], inf),
    \dur, Pseq([2, 1, 1, 1, 1, 2, 2, 2, 2, 2,
        2, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2] * 0.1, inf),
    \pan, 0
    ).play;

c = Pbind(
    \instrument, "SimpleTone",
    \midinote, Pseq([70, 58, 60, 62, 60, 58, 65, 62, 70,
        65, 62, 65, 63, 62, 63, 65, 58, 62, 65, 69, 0, 0], inf),
    \dur, Pseq([2, 1, 1, 1, 1, 2, 2, 2, 2, 2,
        2, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2] * 0.1, inf),
    \pan, 1
    ).play;
)

a.mute;
b.mute;
a.unmute;
c.mute;
b.unmute;
c.unmute;

// Riley?

(

p = Array.fill(20, {[0, 2, 4, 7, 9].choose + [60, 72].choose}).postln;
q = p.copyRange(0, p.size - 2).postln;
Pbind(
    \instrument, "SimpleTone",
    \midinote, Pseq([Pseq(p), Pseq(p), Pseq(p)], inf),
```

```
    \dur, 0.1,
    \pan, -1
    ).play;

Pbind(
    \instrument, "SimpleTone",
    \midinote, Pseq([Pseq(p), Pseq(p), Pseq(q)], inf),
    \dur, 0.1,
    \pan, 1
    ).play;
)

// Or gradual phase
(

p = Array.fill(20, {[0, 2, 4, 7, 9].choose + [60, 72].choose}).postln;
Pbind(
    \instrument, "SimpleTone",
    \midinote, Pseq(p, inf),
    \dur, 0.1,
    \pan, -1
    ).play;

Pbind(
    \instrument, "SimpleTone",
    \midinote, Pseq(p, inf),
    \dur, 0.101,
    \pan, 1
    ).play;

Pbind(
    \instrument, "SimpleTone",
    \midinote, Pseq(p, inf),
    \dur, 0.102,
    \pan, 0
    ).play;
)
```

Here are some nested patterns. *Pbind* is a perfect tool for serialization. Below is a 12-tone example with rhythm partially serialized (mixed with aleatory), and dynamic levels also part serialized part random.

```
    26.11. Serialism

(

a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].scramble.postln;
r = [0.1, 0.1, 1.0, 0.2, 0.3, 0.166, 0.166];
o = [48, 60, 72];

Pbind(
    \instrument, "SimpleTone",
```

```
    \midinote, Prand(
        [ //P, R, I, IR
            Pseq(a) + o.choose,
            Pseq(a.reverse) + o.choose,
            Pseq(12 - a) + o.choose,
            Pseq((12 - a).reverse) + o.choose
        ], inf),
    \dur, Pseq([Prand([0.1, 0.2, 0.5, 1.0], 7),
                Prand([Pseq(r), Pseq(r.reverse)], 1)], inf),
    \amp, Prand([
                Pseries(0.1, 0.1, 5), // cresc
                Pseries(0.9, -0.1, 6), // decresc
                Prand([0.1, 0.3, 0.5, 0.7], 5)
                ], inf)
).play;
)

(

// And of course, three at once.

// If seed is any number other than 0 that seed will be used.
// If 0, a random seed will be picked and posted. Use it to
// repeat a performance.

var seed = 0;

if(seed !=0, {thisThread.randSeed = seed},
    {thisThread.randSeed = Date.seed.postln});

a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].scramble.postln;
r = [0.1, 0.1, 1.0, 0.2, 0.3, 0.166, 0.166];
o = [48, 60, 72];

Pbind(
    \instrument, "SimpleTone",
    \pan, -1,
    \midinote, Prand(
        [ //P, R, I, IR
            Pseq(a) + o.choose,
            Pseq(a.reverse) + o.choose,
            Pseq(12 - a) + o.choose,
            Pseq((12 - a).reverse) + o.choose
        ], inf),
    \dur, Pseq([Prand([0.1, 0.2, 0.5, 1.0], 7),
                Prand([Pseq(r), Pseq(r.reverse)], 1)], inf),
    \amp, Prand([
                Pseries(0.1, 0.1, 5), // cresc
                Pseries(0.9, -0.1, 6), // decresc
                Prand([0.1, 0.3, 0.5, 0.7], 5)
                ], inf)
).play;
```

```
Pbind(
   \instrument, "SimpleTone",
   \pan, 0,
   \midinote, Prand(
      [ //P, R, I, IR
         Pseq(a) + o.choose,
         Pseq(a.reverse) + o.choose,
         Pseq(12 - a) + o.choose,
         Pseq((12 - a).reverse) + o.choose
      ], inf),
   \dur, Pseq([Prand([0.1, 0.2, 0.5, 1.0], 7),
            Prand([Pseq(r), Pseq(r.reverse)], 1)], inf),
   \amp, Prand([
            Pseries(0.1, 0.1, 5), // cresc
            Pseries(0.9, -0.1, 6), // decresc
            Prand([0.1, 0.3, 0.5, 0.7], 5)
            ], inf)
).play;

Pbind(
   \instrument, "SimpleTone",
   \pan, 1,
   \midinote, Prand(
      [ //P, R, I, IR
         Pseq(a) + o.choose,
         Pseq(a.reverse) + o.choose,
         Pseq(12 - a) + o.choose,
         Pseq((12 - a).reverse) + o.choose
      ], inf),
   \dur, Pseq([Prand([0.1, 0.2, 0.5, 1.0], 7),
            Prand([Pseq(r), Pseq(r.reverse)], 1)], inf),
   \amp, Prand([
            Pseries(0.1, 0.1, 5), // cresc
            Pseries(0.9, -0.1, 6), // decresc
            Prand([0.1, 0.3, 0.5, 0.7], 5)
            ], inf)
).play;

)
```

Ok, it's not Webern. But for 20 minutes of typing code it's pretty damn close. The point is this model has the potential of any serial works you would write out by hand. As I've said before, the code doesn't have to be the work itself (though that is my preference). It could be a testing ground. Once you've found a variation you like (change the random seed for different variations), copy it out and hand it to performers.

Serialization Without Synthesis or Server using MIDIout

Pbind communicates with the servers. I find MIDI more useful in serialization studies because those compositions usually are destined for live performers and with MIDI I have a large choice of instruments. MIDI is also more efficient, since the servers do not have to

make any calculations for instrument design and playback. You can leave the servers running, or turn them off using quit. Or if you are working with MIDI over a long space of time, just don't start them in the first place.

The next example is not total serialization in the strictest sense. Only the pitch array is being serialized according to 12-tone rule; after each quotation (12 notes, or *count%12*) a new version of the row is selected. The other serial sets are being followed for a while then scrambled, about 10% of the time (*0.1.coin*). I didn't put much thought into each series, but you get the point.

After the *r.stop* the *4.do* turns all notes off on 4 channels, which is all we are using.

Practice: Total Serialization using MIDI only

```
   26.12. Babbitt: Total Serialization (sort of)

(
MIDIClient.init;
m = MIDIOut(0, MIDIClient.destinations.at(0).uid);
)

(
var pitchSeries, octaveSeries, durationSeries, nextSeries, dynSeries, instSeries;
pitchSeries = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].scramble.postln;
octaveSeries = Array.fill(6, {[36, 48, 60, 72].choose}).postln;
durationSeries = Array.fill(23, {rrand(0.1, 3.0).round(0.1)}).postln;
nextSeries = Array.fill(30, {[0, 0, 0.1, 0.1, 0.1, 0.2, 0.2, 0.2, 0.4, 0.4,
2.0].choose}).postln;
dynSeries = Array.fill(24, {rrand(40, 120).round(0.1)}).postln;
instSeries = Array.fill(20, {4.rand}).postln;

r = Task({
   inf.do({arg count;
   var note;
   note = pitchSeries.wrapAt(count) + octaveSeries.wrapAt(count);
   if(count%12 == 0, {
      pitchSeries =  // choose a new version of the row
         [pitchSeries.reverse, // retrograde
         (12 - pitchSeries).reverse, // retrograde inversion
         12 - pitchSeries, // inversion
         pitchSeries // prime
         ].choose;
      // choose a transposition of the row
      pitchSeries = (pitchSeries + 12.rand)%12;
      pitchSeries.postln;});
   if(0.1.coin, {
      durationSeries = durationSeries.scramble.postln;
      nextSeries = nextSeries.scramble.postln;
      dynSeries = dynSeries.scramble.postln;
      instSeries = instSeries.scramble.postln;
      });
   m.noteOn(instSeries.wrapAt(count), note, dynSeries.wrapAt(count));
```

```
    thisThread.clock.sched(durationSeries.wrapAt(count),
       {m.noteOff(instSeries.wrapAt(count), note); nil});
    nextSeries.wrapAt(count).wait
    })
});

r.start;
)

r.stop; 4.do({arg j; 127.do({arg i; m.noteOff(j, i, 0)})})
```

MIDI Using Pbind

Pbind provides more functionality than *Task*. To use MIDI with a *Pbind* you first have to create an environment that can handle a MIDI instrument. This model was provided by Julian Rohrhuber, which I include without modification.

```
    26.13. Pbind and MIDI, by Julian Rohrhuber

(
var f;
f = (
    noteOn: #{ arg chan, midinote, amp;
                [chan, midinote, asInteger((amp * 255).clip(0, 255))]
                   },
    noteOff:#{ arg chan, midinote, amp;
                [ chan, midinote, asInteger((amp * 255).clip(0, 255))]
                   },
    polyTouch: #{ arg chan, midinote, polyTouch=125;
                       [ chan, midinote, polyTouch]
                   },
    control: #{ arg chan, ctlNum, control=125;
                       [chan, ctlNum, control]
                   },
    program:  #{ arg chan, progNum=1;
                       [ chan, progNum]
                   }
     touch ( chan, val )
   bend ( chan, val )
   allNotesOff ( chan )
   smpte ( frames, seconds, minutes, hours, frameRate )
   songPtr ( songPtr )
   songSelect ( song )
   midiClock (  )
   startClock (  )
   continueClock (  )
   stopClock (  )
   reset (  )
   sysex ( uid, Int8Array )

   */
```

```
 );

~midiEnv =  (
             chan: 1,
             msgFuncs: f,
             hasGate: true,
             midicmd: \noteOn,
             play: #{
                var freqs, lag, dur, sustain, strum;
                var tempo, bndl, midiout, hasHate, midicmd;

                freqs = ~freq = ~freq.value + ~detune;

                tempo = ~tempo;
                if (tempo.notNil) {
                   thisThread.clock.tempo = tempo;
                };

                if (freqs.isKindOf(Symbol).not) {
                   ~finish.value;
                   ~amp = ~amp.value;
                   strum = ~strum;
                   lag = ~lag;
                   sustain = ~sustain = ~sustain.value;
                   midiout = ~midiout;
                   hasHate = ~hasGate;
                   midicmd = ~midicmd;
                   bndl =  ~msgFuncs[midicmd].valueEnvir;

                   bndl = bndl.flop;

                   bndl.do {|msgArgs, i|
                          var latency;

                          latency = i * strum + lag;


                          midiout.performList(midicmd, msgArgs);
                          if(hasHate and: { midicmd === \noteOn }) {
                             thisThread.clock.sched(sustain) {
                                midiout.noteOff(*msgArgs)
                             };
                          };
                   };
                   }
                }

          ).putAll(
             Event.partialEvents.pitchEvent,
             Event.partialEvents.ampEvent,
             Event.partialEvents.durEvent
          )
);
```

```
// initialize midiout
(
MIDIClient.init;
m = MIDIOut(0, MIDIClient.destinations.at(0).uid);
)


// I've modified the Pbind to show more values [they aren't very good]

(
Pbind(
    \parent, ~midiEnv,
    \midiout, m,
    \chan, Pseq([0, 1, 2, 3], 60), //total number of events
    \amp, Prand([
        Pseq([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]),
        Pseq([0.8, 0.7, 0.5, 0.3, 0.1]),
        Prand([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9], 10)
        ], inf),
    \dur, Prand([0.1, 0.1, 0.1, 0.2, 0.2,1.0, 2.0], inf),
    \sustain, Pfunc({rrand(0.1, 2.0)}),
    \midinote, Prand([36, 38, 40, 42, 43, 45, 47, 49, //synthetic scale
        50, 52, 54, 56, 57, 59, 61, 63, 64, 66, 68, 70], inf)
).play;
)
```

26. Exercises

26.1. Using the pitch model for interpolation, enter your own melody and experiment with methods of interpolation.

26.2. Modify any of the serialism examples above so that they print out a list of notes, durations, instruments, etc. Run several examples using the random seed. Pick one you like and transcribe to paper the first two measures.

27 - Total Serialization Continued, Special Considerations

Absolute vs. Proportional Values, Rhythmic Inversion

[This chapter is mostly composition stuff, not much SC information.]

One of the options in serialization is to choose either absolute or proportional values. An absolute value will always be the same, for example, an absolute C4 will always result in a C4. A proportional value is calculated based on a formula and the previous value. It will use the same proportion, but return different final values. An example of a proportional value is the interval of a fifth. In this case the actual value returned may be a C4 or D-flat3, but it will always be a fifth above the previous value. Proportional values work for pitch, next event, duration, and amplitude. It is especially useful in the case of amplitude, allowing the serialization of gradual changes between volumes (i.e. crescendo and decrescendo).

Proportional events make a lot of sense in music because we perceive music as relationships: this event in relation to the previous. The logic of a melody relies on the ratio of each pitch to the previous. Rhythms are based on ratios of a beat: this beat has twice as many events than the last beat.

When choosing absolute pitches you might be able to control the probabilities of one pitch over another. For example, you could weight the choices towards C and G, which would result in a greater sense of tonic. But with proportional values you could weight choices toward consonant intervals, or more dissonant intervals, thereby controlling the level of dissonance.

The danger of exceeding your boundaries is greater when using proportional values. Since you can't specify absolute pitches within a range you are at the mercy of the system, and it will regularly go out of range. The solution is to use a buffer, or wrap around as in the examples above.

With most parameters a proportional value would be a floating point number. Proportional choices between 0.0 and 1.0 will result in a new value that is smaller than the current value. (For example given a duration of 2 seconds and a proportional value of 0.75, the final next will be 1.5.) If the proportional choice is above 1.0 then the resulting value will be greater. (Given a duration of 2 seconds as a current value and a proportional choice of 1.25, the next value will be 2.5.)

Pitch

There are two important things to consider when using a proportional pitch scheme. If you are working with frequencies you would express the intervals as fractions of the current value. For example, if you are currently playing an A 440 and you want a proportional value of an octave for the next value, then the ratio is 2:1, (value*2). Be aware that interval ratios such as 2.0 (octave), 1.5 (fifth), 1.333 (fourth), 1.25 (third), etc., are *just* intervals. We use the equal tempered scale in most modern music.

If, on the other hand, you want equal temperament, then use MIDI numbers. Each MIDI number represents a key on the piano and the tuning is determined by the synthesizer, which is usually equal. The math that you would use with a proportional system using MIDI pitches is also different. You won't multiply values, rather you will add or subtract values. Given a starting point of C4 (MIDI number 60), a fifth above C4 is G4, (MIDI number 67). To get the intervals then you add 7 for fifth, 5 for fourth, 4 for third, -4 for a third down, etc. The inversion of a set of MIDI intervals is pretty easy: midiArray.neg. This basically takes all values and inverts the positive/negative sign, such that 5 becomes -5, -10 becomes 10. Negative values will become positive; -2 becomes 2.

```
   27.1.  Proportional MIDI inversion

// If used as MIDI intervals this is unison, 4th up, M3rd up,
// 5th up, M2 down, M6th down, 2nd up, ttone down

o = [0, 5, 4, 7, -2, -9, 2, -6];

o.neg;
// Results in unison, 4th down, M3rd down, etc.
[ 0, -5, -4, -7, 2, 9, -2, 6 ]
```

Duration and next event

You would think that proportional durations would make a lot of sense since our system of notation is based on proportional values (divisions or multiples of a beat). In practice it becomes very complex very fast. A series of fractions applied proportionally to duration can be very difficult to represent in traditional notation. Here is a deceptively simple series of proportional values; 1, 1.5, 1.5, 1.5, 0.5, 0.5. Given a starting point of 1 second, or one beat at quarter note = 60 bpm: 1 * 1 = 1 (quarter note) * 1.5 = 1.5 (dotted quarter) * 1.5 = 2.25 (half tied to a sixteenth) * 1.5 = 3.375 (dotted half tied to sixteenth tied to thirty-second) * 0.5 = 1.6875 (??) * 0.5 = 0.84375 (??). How would such a passage be notated in tradition meters? Is this a failing of the computer to think in human terms, or another box we should think out of?

There is nothing inherently wrong with subdivisions of metered music. We like it because it's an easy pattern to follow. A proportional scheme could still be used for each beat: this beat has twice as many divisions as the previous. This beat has 1/2 as many events as the previous.

Another problem with a proportional duration scheme is the value 0. In SC a 0 for next event is legal, and represents a chord or a simultaneous event. But once you hit the value 0 you will never escape, because 0 * anything is 0. All of the "next" values beyond that will be 0 and therefore part of a chord; an infinite chord. The solution is to either test the value and give it a non-0 value for the next choice (but then you will never have more than two notes in a chord), or better, consider a chord a single event and make the decision about whether or not this event is a chord and how many elements there are in the chord separate from next event. For example, determine if each event is a rest, a single note, or a chord. If it is a chord,

determine how many pitches are in the chord. Make them all the current duration and then determine the next event that could also be a chord, a rest, or a single pitch.

Another possibility is to never allow 0 as a choice, but conceive simultaneous events as a function of counterpoint: devise two or three lines that will sometimes come together as a chord. The difficulty this presents (and has presented since Italian Motets) is keeping track of the intervals that result from two separate systems.

Next Event

Another method for accommodating rests and simultaneous events is to calculate duration and next event separately. In this case next event could be in one second while the current event is set to four seconds. Or next event could be 0, generating simultaneous events, while the duration of each of the simultaneous events could be different. Rests would occur when the next event is longer than the duration of the current event.

Non-Sequential Events

It is not necessary to calculate events sequentially. Too often we assume that events should be linear and successive. (For good reason; we perceive music sequentially, as a series of relationships.) Another approach might include picking a point in time where a pitch should occur rather than pick which pitch should be next. For example, if you first determine the length of a work (e.g. 4 minutes), you could place items in the stream somewhere along that continuum. Pitch 1 could be placed at 2'23", have a duration of 3 seconds, pitch 2 then might be placed at 1'13". This poses a problem in regard to coherence. As pointed out earlier we recognize style sequentially, so it would be difficult to maintain thematic consistency if events are not calculated sequentially. One solution might include a mixture of sequential and non-sequential events. Themes or motives may be fleshed out sequentially but placed within the space of time as described above. However, I think the difference in sequential and non-sequential approaches is largely academic.

Amplitude

In the case of amplitude a proportional system works well because amplitude is often a function of the previous value (in the case of cresc., decresc., etc.).

Rhythmic Inversion

One last note about serialized rhythm; inversion. Prime and a retrograde versions of a rhythmic series are obvious, but how do you invert it? Most of the solutions I've read in other texts fail to capture the sense of rhythmic inversion. It is logical that a set of durations should invert to something opposite, slow to fast, long values to short values, etc. There are two methods I've used that satisfy this requirement. Both are also problematic.

The first is inversion of density, which is a very binary view of the row. It requires that you first decide what the smallest allowable articulation is. For this example, let's use an eighth note. That means that the densest measure would be filled with all eighths, the least dense

would be a single whole note. (Or even less dense might be a whole note tied to the previous measure, such that there are no articulations in the current measure.) You might represent a measure filled with durations as 0s and 1s (very computeresque), then each possible point of articulation (each eighth note) was either 0 or 1, a 0 meaning no articulation, a 1 meaning a single articulation. Using this method, four quarter notes would be 1, 0, 1, 0, 1, 0, 1, 0. Two half notes would be 1, 0, 0, 0, 1, 0, 0, 0. The logical inversion then would be to simply swap 0s for 1s. The quarter note measure would invert to 0, 1, 0, 1, 0, 1, 0, 1, or a syncopated eighth note passage. The two half notes would invert to 0, 1, 1, 1, 0, 1, 1, 1, or an eighth rest followed by three eighths, then another eighth rest followed by three eighth notes.

The first problem with this method is that you are boxed into using only the declared legal values. The second problem is that inversions may generate an unequal number of articulations, and will therefore require different sizes of pitch, and dynamic series. (For example, suppose you were using an 8 member pitch series and a rhythmic series of two half notes, four quarters, and two whole notes. The original rhythmic series uses 8 points of articulation, but the inversion would result in 24 articulations. Where are you going to come up with the extra notes?) One solution is to make sure you always use exactly half of all the articulation points, so that the inversion would be the other half. The total number of values would always remain the same. But this seems to be a narrow constraint.

Another solution, is to represent each duration as a proportion of a given pulse. Quarter notes are 1.0, half notes are 2.0, eighth notes are 0.5, etc. The inversion to a rhythm then would be the reciprocal: 2 inverts to 1/2, 1/2 inverts to 2. This scheme maintains the number of articulations in a series and satisfies the logic of an inversion (fast to slow, slow to fast, dense to less dense; an active dense line, say, all sixteenth notes, would invert to a relaxed long line, all whole notes). The only problem with this system is that the actual amount of time that passes during an inversion may be radically different from the original row. If you are working with several voices this could throw the other elements of the row out of sync.

You could force the proportions to fit into a prescribed time frame using the normalizeSum message. Take, for example, the array [1, 1.5, 0.25, 4, 0.25]. They total 6 seconds in duration. The "reciprocal" inversion (I'm rounding to 0.01) would be [1, 0.67, 4, 0.25, 4], which totals 9.92 seconds in duration. Using normalizeSum reduces each element in the array such that they all total 1.0. The results of this action ([1, 0.67, 4, 0.25, 4].normalizeSum.round(0.01)) is [0.1, 0.07, 0.4, 0.03, 0.4]. Those values total 1.0, and we want to fit it into a duration of 6 seconds. This is done by simple multiplication: [0.1, 0.07, 0.4, 0.03, 0.4]*6 = [0.6, 0.42, 2.4, 0.18, 2.4].

The results are usually pretty complex, and again quickly stray from most composer's narrow conception of rhythmic possibilities, maybe a good thing.

```
Ex. 21.1

var rhythmArray, orLength, inversion;

rhythmArray = [1, 1.5, 2, 1.25, 0.25, 0.25, 1.5, 0.333];
orLength = rhythmArray.sum;
```

```
inversion = rhythmArray.reciprocal.normalizeSum*orLength;
inversion.postln;
rhythmArray.sum.postln;
inversion.sum.postln;
```

Terry Lee, a former student, believes my attempt at preserving the total duration is a convolution and that when you invert time you invert time and should get a different total duration. I guess if you take this view then you need to make sure you invert all voices at once. That, or not care that the outcome in other voices don't match.

27. Exercises

27.1. None at this writing.

28 - Music Driven by Extra-Musical Criteria, Data Files

Extra Musical Criteria

In this chapter we will examine the possibilities of musical choices that are linked to extra-musical criteria. It would be safe to say that nearly all music has some extra-musical influence on the composer's choices, but here we will strengthen that link. These criteria might include natural data such as electrical impulses from plants, position of stars, a mountain silhouette, or rain and snow patterns. They could include human structures such as other music, text, bus timetables; or masked input such as an audience whose motion through a performance space is being mapped to pitch, amplitude, tone, etc.

Any data stream, whether read from an existing file or from a real time input source such as a mouse or video link, can be used in influencing musical parameters. SC provides a number of outside control sources such as MouseX and Y, Wacom pad controls, MIDI input, data from a file, etc.

Why other data sources? because the patterns and structure can be transferred from the extra-musical source to the composition. A star map, for example, would provide x and y locations that would generate a fairly even distribution of events, while a mountain silhouette would provide a smooth contiguous stream of events.

In addition to the structure provided by outside criteria, there is a philosophical connection that brings additional meaning to the composition, e.g. pitch choices taken from the voice print or favorite poem of one of the performers.

History provides many examples of this type of programmatic connection, and more specifically those where text has influenced composition. Examples include not only an attempt to paint the character of a word musically, but also to generate the exact replica of the word in musical symbols, as in Bach's double fugue which begins with the four pitches Bb, A, C, B (in German musicianship: B, A, C, H). Text has been a common choice, so let's start with that.

The simplest method for linking text to music is to give each letter a MIDI value; a = 0, b = 1, etc., z = 23. This would provide a two-octave range of pitches. Letters of the alphabet are represented in computer code as ascii values. That is, the computer understands the character "a" as an integer. This is a convenient coincidence for us. They are already ascii numbers in the computer, we just need to adjust them to the correct range.

Text Conversion

The *digit* or *ascii* message can be used to convert a character into a digit. With *digit* both "a" and "A" = 10, "z" and "Z" = 35. The message *ascii* returns the actual ascii value: A = 65, a = 97, Z = 90, z = 122. Try these lines of code again to confirm the ascii correlation.

```
28.1.  ascii values
```

```
a = "Test string";
a.at(2).ascii.postln;
a.at(2).digit.postln;
a.do({arg each; each.post; " ".post; each.ascii.postln;})
```

The range of ascii values (for upper case) are also conveniently matched with midi values: 65 (midi number for F4) to 90 (midi number for F6). But the lower case values are a little out of range. To solve this you can just use upper case, or you can scale lower case, or both, using simple addition and subtraction.

The problem with this direct correlation (A = 60, Z = 90) is fairly obvious: you are stuck with the value of the letter. The letters a and e will both always be low notes. And since a and e are both very common letters a higher percentage of those pitches will result.

Mapping

A map will allow greater control over the correlation between notes and the letters. You assign values to the input regardless of the intrinsic value of each item in the stream. This way you can control the nature of the composition while retaining the link to the patterns in the stream. In text, for example, we know that vowels occur about every other letter. Sometimes two in a row, occasionally they may be separated by as many as four consonants. With this understanding we can then use a map to assign specific pitches to vowels and achieve (for example) a sense of tonality; a = C4, e = E4, i = G4, o = C5 (a = 60, e = 64, i = 67, o = 72), etc. You can also confine your map to characters in use, omitting (or including) characters such as spaces, punctuation, etc.

To create a map in SC use an *IdentityDictionary*, shown below. The variable *pitchMap* is assigned the *IdentityDictionary* array that contains pairs of elements; the original character and its associated value. The association is made with the syntax "originalValue -> associatedValue" or in this case "$b -> 4" which reads "make the character b equal to the integer 4."

```
   28.2.  pitchMap

pitchMap = IdentityDictionary[
   $H -> 6, $x -> 6, $b -> 6, $T -> 6, $W -> 6,
   $e -> 11, $o -> 11, $c -> 11, $, -> 11, $. -> 11,
   $n -> 3, $y -> 3,
   $m -> 4, $p -> 8, $l -> 9
      ];
```

The numbers associated with the characters are then used as MIDI pitches, after being raised to the appropriate octave. Alternatively, they can be used as MIDI intervals.

After using the *IdentityDictionary* for a few projects I found it cumbersome to change values. (Too much typing.) So I settled on a computationally less efficient algorithm that uses a two

dimensional array. The first element of the second dimension is the list of characters comprising a string, the second element is the associated value. They are parsed using a *do* function which looks at each of the first elements and if a match is found (using *includes*) the *mappedValue* variable is set to the second element.

```
   28.3.  mapping array

var mappedValue, intervMap;

intervMap = [
   ["ae", 2], ["io", 4], [" pst", 5], ["Hrn", -2],
   ["xmp", -1], ["lfg", -4], ["Th", -5], [".bdvu", 1]
];

intervMap.do({arg item;
   if(item.at(0).includes($o),
      {mappedValue = item.at(1)})
   });
```

Here is a patch which controls only pitched elements in an absolute relationship (that is, actual pitches rather than intervals).

```
   28.4.  Extra-Musical Criteria, Pitch Only

(
var noteFunc, blipInst, midiInst, channel = 0, port = 0, prog = 0,
   intervMap, count = 0, ifNilInt = 0, midin = 0, inputString;

//The input stream.

inputString = "Here is an example of mapping. The, them, there, these,"
   "there, then, that, should have similar musical interpretations."
   "Exact repetition; thatthatthatthatthatthat will also"
   "be similar.";

//intervMap is filled with arrays containing a collection of
//characters and a value. In the functions below the character
//strings are associated with the numbers.

intervMap = [
   ["ae", 2], ["io", 4], [" pst", 5], ["Hrn", 7],
   ["xmp", 1], ["lfg", 3], ["Th", 6], [".bdvu", 11]
];


"// [Char, Interval, ifNilInt, midi interval, octave, midi]".postln;

noteFunc = Pfunc({var parseInt, octave;

   //Each array in the intervMap is checked to see if the
   //character (inputString.wrapAt(count)) is included. If
   //it is then parseInt is set to the value at item.at(1)
```

```
    intervMap.do({arg item;
        if(item.at(0).includes(inputString.wrapAt(count)),
            {parseInt = item.at(1)})
        });

    //If parseInt is notNil, midin is set to that.
    //ifNilInt is for storing each parseInt to be used if
    //no match is found and parseInt is nil the next time around.

    if(parseInt.notNil,
        {midin = parseInt; ifNilInt = parseInt},
        {midin = ifNilInt}
    );

    octave = 60;

    "//".post; [inputString.wrapAt(count), parseInt,
        ifNilInt, midin, octave/12, midin + octave].postln;

    count = count + 1;

    midin + octave
});

Pbind(
    \midinote, noteFunc,
    \dur, 0.125,
    \amp, 0.8,
    \instrument, "SimpleTone"
).play;

)
```

I admit it's not very interesting. That is partially because pitch values alone usually do not account for a recognizable style. We are more accustomed to a certain level of dissonance, not pitch sequences alone. Dissonance is a function of interval distribution, not pitch distribution. To achieve a particular balance of interval choices the *intervMap* should contain proportional interval values rather than absolute pitches. The numbers may look pretty much the same, but in the actual parsing we will use *midin = parseInt + midin%12* (thus *parseInt* becomes an interval) rather than *midin = parseInt*. Interest can also be added by mapping pitches across several octaves, or mapping octave choices to characters.

Notice in this example the series "thatthatthatthat" results in an intervallic motive, rather than repeated pitches.

```
    28.5.  Extra-Musical Criteria, Total Control

(
var noteFunc, blipInst, midiInst, channel = 0, port = 0, prog = 0,
    intervMap, count = 0, ifNilInt = 0, midin = 0, ifNilDur = 1,
```

```
    durMap, durFunc, ifNilSus = 1, susMap, susFunc, ifNilAmp = 0.5,
    curAmp = 0.5, ampMap, ampFunc, inputString;

//The input stream.

inputString = "Here is an example of mapping. The, them, there, these,"
    "there, then, that, should have similar musical interpretations."
    "Exact repetition; thatthatthatthatthatthat will also"
    "be similar.";

//intervMap is filled with arrays containing a collection of
//characters and a value. In the functions below the character
//strings are associated with the numbers.

intervMap = [
    ["ae", 6], ["io", 9], [" pst", 1], ["Hrn", -3],
    ["xmp", -1], ["lfg", -4], ["Th", -5], [".bdvu", 1]
];

durMap = [
    ["aeiouHhrsnx", 0.125], ["mplf", 0.5], ["g.T,t", 0.25],
    ["dvc", 2], [" ", 0]
];

susMap = [
    ["aei ", 1.0], ["ouHh", 2.0], ["rsnx", 0.5], ["mplf", 2.0], ["g.T,t", 4.0],
    ["dvc", 1.0]
];

ampMap = [
    ["aeHhrsnx ", 0.8], ["ioumplfg.T,tdvc", 1.25]
];

noteFunc = Pfunc({var parseInt, octave = 48;

    //Each array in the intervMap is checked to see if the
    //character (inputString.wrapAt(count)) is included. If
    //it is then parseInt is set to the value at item.at(1)

    intervMap.do({arg item;
        if(item.at(0).includes(inputString.wrapAt(count)),
            {parseInt = item.at(1)})
        });

    //If parseInt is notNil, midin is set to that plus previous
    //midin. ifNilInt is for storing each parseInt to be used if
    //no match is found and parseInt is nil.

    if(parseInt.notNil,
        {midin = parseInt + midin%48; ifNilInt = parseInt},
        {midin = ifNilInt + midin%48}
    );
```

```
    [inputString.wrapAt(count)].post;
    ["pitch", parseInt, midin, octave/12, midin + octave].post;

    midin + octave
});

durFunc = Pfunc({var parseDur, nextDur;

    durMap.do({arg item;
        if(item.at(0).includes(inputString.wrapAt(count)),
            {parseDur = item.at(1)})
        });

    if(parseDur.notNil,
        {nextDur = parseDur; ifNilDur = parseDur},
        {nextDur = ifNilDur}
    );
    ["dur", nextDur].post;
    nextDur
});

susFunc = Pfunc({var parseSus, nextSus;

    susMap.do({arg item;
        if(item.at(0).includes(inputString.wrapAt(count)),
            {parseSus = item.at(1)})
        });

    if(parseSus.notNil,
        {nextSus = parseSus; ifNilSus = parseSus},
        {nextSus = ifNilSus}
    );
    ["sustain", nextSus.round(0.01)].post;
    nextSus
});

ampFunc = Pfunc({var parseAmp;

    ampMap.do({arg item;
        if(item.at(0).includes(inputString.wrapAt(count)),
            {parseAmp = item.at(1)})
        });

    if(parseAmp.notNil,
        {curAmp = curAmp*parseAmp; ifNilAmp = parseAmp},
        {curAmp = curAmp*ifNilAmp}
    );

    count = count + 1;
    if(0.5.coin, {curAmp = rrand(0.2, 0.9)});
    ["amp", curAmp.round(0.01)].postln;

    curAmp.wrap(0.4, 0.9)
```

```
});


Pbind(
   \midinote, noteFunc,
   \dur, durFunc,
   \legato, susFunc,
   \amp, ampFunc,
   \instrument, "SimpleTone"
).play;
)
```

Working With Files

It is not always practical to type the text or data values you want to use into the actual code file. Once you have devised an acceptable map for text you can consider the map the composition and the text a modular component that drives the music. In this case a method for reading the text as a stream of data into the running program is required. With this functionality in place the mapping composition can exist separate from the files that contain the text to be read. SC has standard file management tools that can be used for this purpose.

The type of file we are working with is text (see data types below), so we should create text files. But an MS word document, or even an rtf document contains non-text formatting information that will interfere. I tried creating "text" files using SimpleText, BBEdit, SC editor, and Word. Word and BBEdit were the only two that didn't have extra unwanted stuff.

Pathnames also require a little explanation. When SC (and most programs) opens a file it first looks for the file in the directory where SC resides. So if the file you are using is in the same folder as SC then the pathname can just be the name of the file. If it resides anywhere else a folder hierarchy must be included. The hierarchy is indicated with folder names separated by a slash. If the data file resides in a folder which is in the folder where SC resides, then the pathname can begin with that folder. If the file resides outside the SC folder, then you need to give the entire path name beginning with e.g. "Users." So a file in the same folder as SC is simply "MyFile", if it is in a subfolder it might be "/Data Files/MyFile", if in another area then perhaps "/Users/Students/Documents/Computer Music/Data Files/MyFile".

To open and read the file, first declare a variable to hold the file pointer. Then use File() to open the file and identify what mode you will be using (read, write, append, etc.). In this example we use the read mode, or "r."

Once you have opened the file you could retrieve each character one at a time using *getChar*, but I would suggest reading the entire file and storing it in an array, since in previous examples the text is stored in an array. Here is the code, assuming the text file is named "Text File." This section can be inserted in the patch above (minus the *input.postln*) in place of the input = "Here is . . . etc.

```
   28.6.  reading a file
```

```
(
var input, filePointer; //declare variables
filePointer = File("Test File", "r");
input = filePointer.readAllString;
filePointer.close;
input.postln;
)
```

The longer, but more user friendly method uses *openDialog*. I'm usually working with just one file so this is an extra step I don't need. But in many cases it takes out the pathname guesswork. It can also just be used to figure out the exact pathname of any given file. There is one glitch; either everything has to be in the *openDialog* function or you have to use a global variable and run the two sections of code separately.

```
   28.7.  reading a file

// Print any pathname for later use

File.openDialog("", { arg pathName; pathName.postln});

// Open using a dialog
(
var input, filePointer; //declare variables
File.openDialog("", {arg pathname;
    filePointer = File(pathname, "r");
    input = filePointer.readAllString;
    input.postln;
    // Everything has to be inside this function
},
{"File not found".postln});

)

// Or open file and store in global

(
var filePointer; //declare variables
File.openDialog("", {arg pathname;
    filePointer = File(pathname, "r");
    ~input = filePointer.readAllString;
},
{"File not found".postln});
)

// Then

~input.postln;

// Or include ~input in the patch.
```

28. Exercises

28.1. None at this writing.

29 - Markov Chains, Numerical Data Files

Artificial intelligence is describing human phenomena to a computer in a language it understands: numbers, probabilities, and formulae. Any time you narrow the choices a computer makes in a musical frame you are in a sense teaching it something about music, and it is making an "intelligent" or informed choice. This is true with random walks; if you narrow the choice to a MIDI pitch, for example, you have taught the patch (by way of converting MIDI values to cps) about scales, ratios, intervals, and equal tempered tuning. If we limit random choices to a C-major scale, then the cpu is "intelligent" about the whole step and half step relationships in a scale. If we biased those choices such that C was chosen more often than any other pitch, then the cpu understands a little about tonality. This bias map is usually in the form of ratios and probabilities. In a simple biased random choice there is only one level of probability; a probability for each possible choice in the scale. This is known as a Markov Process with a zeroth order probability.

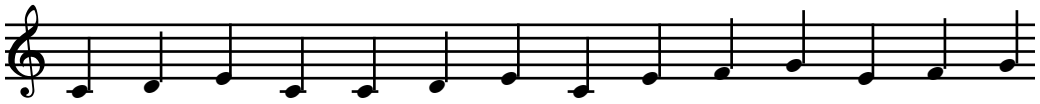
A zeroth probability system will not give us a sense of logical progression, since musical lines are fundamentally reliant on relationships between pitches, not the individual pitches themselves or general distribution of pitches in a piece. We perceive melody and musical progression in the relationship of the current pitch to the next pitch, and the last three pitches, and the pitches we heard a minute ago. In order for you to describe a melody, you have to describe the connections between pitches, i.e. intervals.

To get a connection between pitches we need to use a higher order Markov chain; first or second at least. The technique is described in *Computer Music* by Charles Dodge (page 283) and Moore's *Elements of Computer Music* (page 429). I suggest you read those chapters, but I will also explain it here.

The way to describe the connection between two pitches is to have a chart of probable next pitches given the current pitch. Take for example the pitch G in the key of C. If you wanted to describe a tonal system in terms of probability you would say there is a greater chance that C follows G (resulting in a V-I relationship) than say F would follow G (a retrogressive V-IV). If the current pitch is F on the other hand, then there is a greater chance that the next pitch is E (resolution of the IV) than C (retrogressive). Markov Chains are not intended solely for tonal musics. In non-tonal musics you might likewise describe relationships by avoiding the connection G to C. So if the current pitch is G and avoiding a tonal relationship is the goal you might say there is a very small chance that the next pitch be C, but a greater chance that the next pitch is D-sharp, or A-flat.

You can describe any style of music using Markov Chains. You can even mimic an existing composers style based on an analysis of existing works. For example you could analyze all the tunes Stephen Foster wrote, examining the pitch G (or equivalent in that key) and the note that follows each G. You would then generate a chart with all the possible choices that might follow G. Count each occurrence of each of those subsequent pitches in his music and enter that number in the chart. This would be a probability chart describing precisely Stephen Foster's treatment of the pitch G, or the fifth step of the scale.

If we have determined the probabilities of one pitch based on our analysis, the next step is to compute similar probabilities for all possible current pitches and combine them in a chart. This is called a transition table. To create such an analysis of the tune "Frere Jacque" you would first create the chart with all the possible current pitches in the first column and all next pitches in the row above each column. The first pitch is C and it is followed by D. We would represent this single possible combination with a 1 in the C row under the D column.



	C	D	E	F	G
C	0	**1**	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0
F	0	0	0	0	0
G	0	0	0	0	0

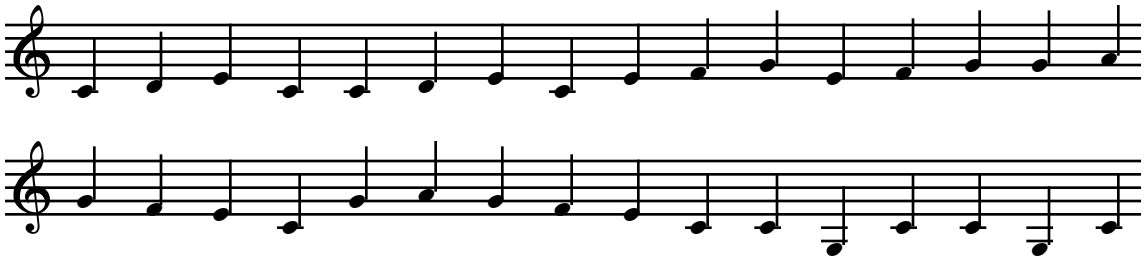
Next we count up all the times C is followed by D and enter that number (2) in that column. Next we examine all other Cs and count the number of times C is followed by C, E, F, G, and enter each of those totals into the table. We do the same for the pitch D, then E, and so on. This is the resulting chart or transition table:

	C	D	E	F	G	Total
C	1	2	1	0	0	4
D	0	0	2	0	0	2
E	2	0	0	2	0	4
F	0	0	0	0	2	2
G	0	0	1	0	0	1

For each row the total combinations are listed. The probability for each row is calculated by the number of actual occurrences in that column divided by the total possible outcomes, such that the C column values will be 1/4, 2/4, 1/4.

	C	D	E	F	G	Total
C	.25	.5	.25	0	0	4
D	0	0	1.0	0	0	2
E	.5	0	0	.5	0	4
F	0	0	0	0	1.0	2
G	0	0	1.0	0	0	1

This is a first order transition table[61]. Because we are using only the previous and next note (one connection) it will still lack a convincing sense of melodic progression. To imitate the melody we really need to look at patterns of two or three notes. This brings us to a second order Markov Chain. A second order adds one level to the sequence. That is to say, given the last two pitches, what is the probability of the next pitch being C, D, etc. Here is the same chart expanded to include all of "Frere Jacque" and a second order of probability. There are 36 combinations, but not all of them occur (e.g. C-A), and don't need to be included on the chart, so I've removed those combinations.



	C	D	E	F	G	A	Total
C-C		1			2		3
C-D			2				2
C-E				1			1
C-G	2					1	3
D-E	2						2
E-C	2		1		1		4
E-F					2		1
F-E	2						2
F-G			1		1		2
G-C	1						1
G-E				1			1
G-F			2				2
G-G						1	1
G-A					2		2
A-G				2			2
Total	9	1	6	4	8	2	30

Here are some guidelines: Note that I've totaled all the combinations at the bottom. This is a quick way to check if you have the correct number of total connections. The total should equal the number of notes in the piece minus two (because the first two don't have a connection of three items—or second order). The other thing you have to watch out for is a broken link. A broken link is a reference to a connection that doesn't have a probability row on the chart. Take for example the combination C-C. If you entered a probability for the F column in the C-C row, then the combination C, C, F could result. But there is no row of

[61] One could argue that a set of probabilities describing intervals rather than pitches is already a 1st order transition table, since even a single interval describes the relationship between two pitches.

probabilities for C-F, and the program would return a nil value and crash (or get stuck in a loop). I don't have a quick or clever way to check to make sure you don't have any bad leads. You just have to proof carefully.

Here is the chart with percentages.

	C	D	E	F	G	A	Total
C-C		.33			.66		3
C-D			1.0				2
C-E				1.0			1
C-G	.66					.33	3
D-E	1.0						2
E-C	.5		.25		.25		4
E-F					1.0		1
F-E	1.0						2
F-G			.5		.5		2
G-C	1.0						1
G-E				1.0			1
G-F			1.0				2
G-G						1.0	1
G-A					1.0		2
A-G				1.0			2
Total	9	1	6	4	8	2	30

The biggest problem with this type of system is the memory requirements. If, for example, you were to do a chart for the piano works of Webern, assuming a four octave range, 12 pitches for each octave, second order probability would require a matrix of 110,592 references for pitch alone. If you expanded the model to include rhythm and instrument choice, dynamic and articulation, you could be in the billions in no time. So there needs to be efficient ways of describing the matrix. That is why in the Foster example mentioned below I do a few confusing, but space saving short cuts. The chart above for "Frere Jacque" is demonstrated in the file Simple Markov. Following are some explanations of the code.

29.1. transTable

```
//A collection of the pitches used

legalPitches = [60, 62, 64, 65, 67, 69];

//An array of arrays, representing every possible previous pair.

transTable = [
    [0, 0], //C, C
    [0, 1], //C, D
    [0, 2], //C, E
    [0, 4], //C, G
    [1, 2], //D, E
    [2, 0], //E, C
    [2, 3], //E, F
    [3, 2], //F, E
    [3, 4], //F, G
    [4, 0], //G, C
```

```
    [4, 2], //G, E
    [4, 3], //G, F
    [4, 4], //G, G
    [4, 5], //G, A
    [5, 4]  //A, G
];
```

It would be inefficient to use actual midi values, since so many midi values are skipped in a tonal scheme. So *legalPitches* is used to describe all the pitches I will be working with and the actual code looks for and works around array positions, not midi values. (That is, array positions which contain the midi values.)

The variable *transTable* describe the first column of my transition table. Each of the possible previous pitches are stored in a two dimensional array (arrays inside an array).

The value I use to compare and store the current two pitches is *currentPair*. It is a single array holding two items, the first and second pitch in the pair I will use in the chain. At the beginning of the program they are set to 0, 0, or C, C.

Next I have to match the *currentPair* with the array *transTable*. This is done with a *do* iteration. In this function each of the two position arrays will be compared to the variable *currentPair*, which is also a two position array. When a match is found the index of that match (or position in the array where it was found) is stored in *nextIndex*. In other words, I have found the index position of *currenPair*. This is necessary because I have pared down the table to include only combinations I'm actually using.

```
    29.2.  Parsing the transTable

transTable.do({arg index, i; if(index == currentPair,
                {nextIndex = i; true;}, {false})});
```

Next I describe the index for each previous pair. If, for example, the current pair was D, E. Their values in the *transTable* would be [1, 2], and the lines of code above would find a match at array position 4 (remember to count from 0). That means I should use the probability array at position 4 in the chart below. In this chart it says that I have a 100% chance of following the D, E in *currentPair* with a C. I modified (where noted) some of the probabilities from the original chart in *Dodge*.

```
    29.3.  Probability chart

nPitchProb =
[
     //C     D     E     F     G     A
    [0.00, 0.33, 0.00, 0.00, 0.66, 0.00], //C, C
    [0.00, 0.00, 1.00, 0.00, 0.00, 0.00], //C, D
    [0.00, 0.00, 0.00, 1.00, 0.00, 0.00], //C, E
    [0.66, 0.00, 0.00, 0.00, 0.00, 0.33], //C, G
    [1.00, 0.00, 0.00, 0.00, 0.00, 0.00], //D, E
```

```
    [0.50, 0.00, 0.25, 0.00, 0.25, 0.00], //E, C
    [0.00, 0.00, 0.00, 0.00, 1.00, 0.00], //E, F
    [1.00, 0.00, 0.00, 0.00, 0.00, 0.00], //F, E
    [0.00, 0.00, 0.50, 0.00, 0.50, 0.00], //F, G
    [1.00, 0.00, 0.00, 0.00, 0.00, 0.00], //G, C
    [0.00, 0.00, 0.00, 1.00, 0.00, 0.00], //G, E
    [0.00, 0.00, 1.00, 0.00, 0.00, 0.00], //G, F
    [0.00, 0.00, 0.00, 0.00, 0.00, 1.00], //G, G
    [0.00, 0.00, 0.00, 0.00, 1.00, 0.00], //G, A
    [0.00, 0.00, 0.00, 1.00, 0.00, 0.00] //A, G

];
```

The choice is actually made using *windex*. The function windex (weighted index) takes an *array* of probabilities as its first argument. The array *nPitchProb* is an array of arrays, and I want to use one of those arrays as my probability array, say index 4. The way I identify the array within the array is *nPitchProb.at(4)*. Since it is a multi-dimensional array I have to reference it twice, e.g. *nPitchProb.at(4).at(5)*.

Using variables this becomes: *(nPitchProb.at(prevPitch).at(nextIndex)*. The array for *windex* has to total 1, and I don't remember why I entered values that total 16, but that is fixed with *normalizeSum*. The return from *windex* is an array position, which I will store in *nextIndex*, and the variable that tells me which array to use in *nextPitchProbility* is *nextIndex*.

The variable *nextPitch* is an array position that can then be used in conjunction with *legalPitches* to return the midi value for the correct pitch: *legalPitches.at(nextPitch)*. It is also used for some necessary bookkeeping. I need to rearrange *currentPair* to reflect my new choice. The value in the second position of the array *currentPair* needs to be moved to the first position, and the *nextPitch* value needs to be stored in the second position of the currenPair array. (In other words, *currentPair* was D, E, or array positions 1, 2, and I just picked a C, according to the table, or a 0. So what was [1, 2] needs to be changed to [2, 0] for the next pass through the function.)

currentPair.put(0, currentPair.at(1));
currentPair.put(1, nextPitch);

A more complex example of a transition table and Markov process is demonstrated in the file Foster Markov, which uses the chart for Stephen Foster tunes detailed in Dodge's *Computer Music* on page 287. I wrote this a while back, and I think there are more efficient ways to do the tables (and I make some confusing short cuts), but it does work.

For duration you can use the first line; 0.125. This is probably a more accurate realization since we haven't discussed rhythm. But to give it a more melodic feel I've added some random rhythmic cells. The system is not given any information about phrase structure or cadences, and you can hear that missing component. Even so, it is very Fosterish.

```
29.4.  Foster Markov
```

```
(

var wchoose, legalPitches, previousIndex, prevPitch,
currentPitch, nextIndex, nextPitch, nPitchProb,
pchoose, blipInst, envelope, pClass, count, resopluck;

prevPitch = 3;
currentPitch = 1;
count = 1;
pClass = #["A3", "B3", "C4", "D4", "E4", "F4", "F#4",
          "G4", "A4", "B4", "C5", "D5"];

//pchoose is the mechanism for picking the next value.

pchoose =
{
legalPitches = [57, 59, 60, 62, 64, 65, 66, 67, 69, 71, 72, 74];

//Both prevPitch and nextPitch are not pitches, but array positions.

previousIndex = [
   [2], //previous is 0 or A3
   [2], //1 or B3
   [0, 1, 2, 3, 4, 5, 7, 9, 10], //2: C4
   [1, 2, 3, 4, 7, 10], //3: D4
   [2, 3, 4, 5, 7, 8], //4: E4
   [4, 5, 7, 8], //5: F4
   [7], //6: F#4
   [2, 4, 5, 6, 7, 8, 10], //7: G4
   [2, 4, 5, 6, 7, 8, 10], //8: A4
   [8, 10], //9: B5
   [7, 8, 9, 10, 11], //10: C5
   [7, 9] //11: D5
];

previousIndex.at(prevPitch).do({arg index, i; if(index == currentPitch,
                {nextIndex = i; true;}, {false})});

nPitchProb =
[
// [00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11] array position
//      A,  B,  C,  D,  E,  F,  F#, G,  A,  B,  C,  D
[  //arrays for A3
   [00, 00, 16, 00, 00, 00, 00, 00, 00, 00, 00, 00] // one array: C4
],
[  //arrays for B3
   [00, 00, 05, 06, 00, 00, 00, 05, 00, 00, 00, 00]  // C4 only
],
[  //arrays for C4
   [00, 00, 16, 00, 00, 00, 00, 00, 00, 00, 00, 00], // A3
   [00, 00, 16, 00, 00, 00, 00, 00, 00, 00, 00, 00], // B3
// [00, 02, 02, 09, 02, 10, 00, 00, 00, 00, 00, 00], original C4
   [00, 06, 02, 09, 02, 06, 00, 00, 00, 00, 00, 00], // C4
```

```
    [00, 00, 03, 04, 08, 00, 00, 01, 00, 00, 00, 00], // D4
    [00, 00, 00, 07, 03, 02, 00, 04, 00, 00, 00, 00], // E4
    [00, 00, 00, 00, 11, 00, 00, 00, 05, 00, 00, 00], // F4
    [00, 00, 00, 00, 04, 00, 00, 12, 00, 00, 00, 00], // G4
    [00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 16, 00], // A4
    [00, 00, 00, 00, 00, 00, 00, 02, 11, 03, 00, 00]  // C5
],
//       A,  B,  C,  D,  E,  F,  F#, G,  A,  B,  C,  D
[  //arrays for D4
    [00, 00, 16, 00, 00, 00, 00, 00, 00, 00, 00, 00], // B4
// [01, 00, 01, 04, 05, 00, 00, 01, 00, 01, 03, 00], original C4
    [05, 00, 01, 04, 01, 00, 00, 01, 00, 01, 03, 00], // C4
// [00, 01, 12, 01, 02, 00, 00, 00, 00, 00, 00, 00], original D4
    [00, 06, 07, 01, 02, 00, 00, 00, 00, 00, 00, 00], // D4
    [00, 00, 01, 03, 06, 04, 00, 01, 01, 00, 00, 00], // E4
    [00, 00, 00, 00, 00, 00, 05, 08, 03, 00, 00, 00], // G4
    [00, 00, 00, 00, 00, 00, 00, 00, 00, 16, 00, 00]  // C5
],
[  //arrays for E4
    [00, 00, 00, 12, 03, 01, 00, 00, 00, 00, 00, 00], // C4
// [00, 02, 07, 03, 02, 00, 00, 01, 00, 01, 00, 00], original D4
    [00, 05, 04, 03, 02, 00, 00, 01, 00, 01, 00, 00], // D4
    [00, 00, 03, 04, 06, 02, 00, 01, 00, 00, 00, 00], // E4
    [00, 00, 00, 00, 04, 03, 00, 06, 03, 00, 00, 00], // F4
    [00, 00, 00, 00, 02, 00, 00, 10, 03, 00, 01, 00], // G4
    [00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00, 00]  // A4,
],
//       A,  B,  C,  D,  E,  F,  F#, G,  A,  B,  C,  D
[  //arrays for F4
    [00, 00, 00, 08, 00, 08, 00, 00, 00, 00, 00, 00], // E4
    [00, 00, 00, 00, 00, 08, 00, 08, 00, 00, 00, 00], // F4
    [00, 00, 02, 00, 00, 00, 00, 10, 00, 00, 04, 00], // G4
    [00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00, 00]  // A4,
],
[  //arrays for F#4
    [00, 00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00]  // G4,
],
[  //arrays for G4
    [00, 00, 00, 11, 05, 00, 00, 00, 00, 00, 00, 00], // C4
    [00, 00, 05, 04, 03, 01, 00, 02, 01, 00, 00, 00], // E4
    [00, 00, 00, 00, 16, 00, 00, 00, 00, 00, 00, 00], // F4
    [00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00, 00], // F#4
    [00, 00, 00, 00, 04, 01, 04, 04, 03, 00, 00, 00], // G4
    [00, 00, 01, 00, 01, 00, 05, 07, 01, 00, 01, 00], // A4
    [00, 00, 00, 00, 00, 00, 00, 06, 05, 03, 02, 00]  // C5
],
//       A,  B,  C,  D,  E,  F,  F#, G,  A,  B,  C,  D
[  //arrays for A4
    [00, 00, 16, 00, 00, 00, 00, 00, 00, 00, 00, 00], // C4
    [00, 00, 00, 11, 05, 00, 00, 00, 00, 00, 00, 00], // E4
    [00, 00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00], // F4
    [00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00, 00], // F#4
    [00, 00, 01, 00, 09, 01, 00, 02, 01, 00, 02, 00], // G4
```

```
    [00, 00, 00, 00, 02, 00, 00, 12, 00, 00, 02, 00], // A4
    [00, 00, 00, 00, 00, 00, 00, 09, 02, 05, 00, 00]  // C5
],
[  //arrays for B5
    [00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00, 00], // A4
    [00, 00, 00, 00, 00, 00, 00, 00, 06, 00, 00, 10]  // C5
],
//       A,  B,  C,  D,  E,  F,  F#, G,  A,  B,  C,  D
[  //arrays for C5
    [00, 00, 00, 00, 14, 00, 00, 02, 00, 00, 00, 00], // G4
    [00, 00, 00, 00, 00, 01, 00, 05, 06, 00, 04, 00], // A4
    [00, 00, 00, 00, 00, 00, 00, 00, 12, 00, 04, 00], // B4
    [00, 00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00], // C5
    [00, 00, 00, 00, 00, 00, 00, 05, 00, 11, 00, 00]  //D5
],
[  //arrays for D5
    [00, 00, 00, 00, 00, 00, 00, 16, 00, 00, 00, 00], // G4
    [00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 16, 00]  // B4
]
];

nextPitch = (nPitchProb.at(prevPitch).at(nextIndex).normalizeSum).windex;

//current is set to previous, next is current for next run. The actual pitch
//is returned from legal pitch at nextPitch.

[pClass.at(nextPitch), legalPitches.at(nextPitch)].post;
// if((count%10) == 0, {"".postln};);
count = count + 1;
prevPitch = currentPitch;
currentPitch = nextPitch;
legalPitches.at(nextPitch)
};

Pbind(

    \dur, 0.125,
    \dur,  Prand([
              Pseq(#[1]),
              Pseq(#[0.5, 0.5]),
              Pseq(#[0.5, 0.5]),
              Pseq(#[0.25, 0.25, 0.25, 0.25]),
              Pseq(#[0.5, 0.25, 0.25]),
              Pseq(#[0.25, 0.25, 0.5]),
              Pseq(#[0.25, 0.5, 0.25])
              ], inf),
    \midinote, Pfunc(pchoose),
    \db, -10,
// \instrument, "SimpleTone",
    \pan, 0.5
).play

)
```

Data Files, Data Types

As with the previous section, it might be useful to store the data for transition tables in a file so that the program can exist separate from the specific tables for each composition. The data files can then be used as a modular component with the basic Markov patch.

Text files, as discussed above, contain characters. But the computer understands them as integers (ascii numbers). The program you use to edit a text file converts the integers into characters. You could use SimpleText to create a file that contained integers representing a transition table, but the numbers are not really numbers, rather characters. To a cpu "102" is not the integer 102, but three characters (whose ascii integer equivalents are 49, 48, and 50) representing 102. The chart below shows the ascii numbers and their associated characters. Numbers below 32 are non-printing characters such as carriage returns, tabs, beeps, and paragraph marks. The ascii number for a space (32) is included here because it is so common. This chart stops at 127 (max for an 8 bit number) but there are ascii characters above 127. The corresponding characters are usually diacritical combinations and Latin letters.

032		033	!	034	"	035	#	036	$	037	%	038	&	039	'
040	(	041	)	042	*	043	+	044	,	045	-	046	.	047	/
048	0	049	1	050	2	051	3	052	4	053	5	054	6	055	7
056	8	057	9	058	:	059	;	060	<	061	=	062	>	063	?
064	@	065	A	066	B	067	C	068	D	069	E	070	F	071	G
072	H	073	I	074	J	075	K	076	L	077	M	078	N	079	O
080	P	081	Q	082	R	083	S	084	T	085	U	086	V	087	W
088	X	089	Y	090	Z	091	[	092	\	093	]	094	^	095	_
096	`	097	a	098	b	099	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	□

If you would like to test this, create a text file using BBEdit, SimpleText, or MS Word (save it as text only) and run these lines of code, which retrieves each 8 bit integer from the file then prints it as an integer, then the ascii equivalent. (Do a get info on the file you save to

check for a hidden extension such as ".rtf." Note that an rtf file has a lot more information than just the text you wrote.)

29.5. test ascii

```
var fp;
fp = File("Testascii.rtf", "r"); //open a text file
fp.length.do({a = fp.getInt8; [a, a.ascii].postln}); //read file as integers
```

Data appropriate for a transition table (integers or floats) could be opened in a text editor, but it would display gibberish, not the transition data. So the question is, how do you create a data file? It is not as simple as a text file. It must be done with a program that writes and reads data streams other than characters. SC can create such files. (But be sure to read ahead, there is a simpler method.)

The transition tables above used integers, but the probability table used floating points. It is important to distinguish between the two. Below are examples of code for writing, and reading files containing integers and floating point values. There are three messages for integers; *putInt8*, *putInt16*, and *putInt32* for 8 bits (a byte), 16 bits (two bytes), and 32 bits (four bytes). Each size has a limited capacity. An 8 bit integer can be as large as 128, 16 bit can be as large as 32,768, 32 bit has a 2 billion+ capacity. There are two messages for floats; *putFloat* and *putDouble*. A "Double" is larger and therefore more precise, but they take up twice the space, and floats should be sufficient for what we are doing. For characters the messages *putChar* and *putString* can be used.

It is important to understand these data types because you need to read the same type that you write. If you write data as 16 bit integers but read them as 8 bit integers the numbers will not be the same. Following are code segments that write and read floating-point values and integers.

29.6. data files

```
var fp, data;
fp = File("TestInt", "w"); //open a file
data = [65, 66, 67, 68, 69, 70, 71];
data.do({arg eachInt; fp.putInt16(eachInt)}); //place each int in file
fp.close;

var fp, data;
fp = File("TestInt", "r"); //open a file
data = fp.readAllInt16; //read all as Int array
data.postln;
fp.close;

var fp, data;
fp = File("TestFloat", "w"); //open a file
data = [6.5, 6.6, 6.7, 6.8, 6.9, 7.0, 7.1];
data.do({arg eachFloat; fp.putFloat(eachFloat)});
fp.close;
```

```
var fp, data;
fp = File("TestFloat", "r"); //open a file
data = fp.readAllFloat; //read all as array
data.postln;
fp.close;
```

I chose to write the integers 65 through 71 because they correspond with ASCII characters A through G. To confirm this, open the *TestInt* file, which supposedly has only integers, not characters, with SimpleText, MS Word, or SC to confirm that these programs convert them to text characters.

Interpreting Strings

Files containing integers and floats still present a problem. It is difficult to manage the data especially if they are grouped in arrays of arrays, as in the Markov patch. There is really no fast easy way to check the file to make sure all the data are correct. You can't read it as a text file. You have to read it as 8, 16, or 32 bit ints, or floating point values. If you get any of the data types incorrect, or if a value is in the wrong position, the structures and arrays will be off. (Don't misunderstand; it can be done, it's just a hassle and invites error.)

It would be easier if we could create the files using a text editor or SC, but read them as data files. Or read them as strings but parse them into the correct data.

For those of you who have worked with C you know that managing and parsing data represents a large amount of programming. True to form, it is very simple in SC. The *interpret* message translates a string into code that SC will understand. Strings can be read from files and interpreted as code. You can save entire functions, data structures, lists, error messages, and macros in files that can be edited as text with SC or any text editor, but used as code in an SC patch.

To test this, first open a new window (just text, not rich text) and type these lines that represent an array of arrays, containing different data types (note that I did not end with a semicolon):

```
[
	[1, 2, 3],
	[2.34, 5.12],
	["C4", "D4", "E4"],
	Array.fill(3, {rrand(1.0, 6.0)})
]
```

I've intentionally filled this array with several different data types, including strings[62], to illustrate how simple this is in SC as compared to more archaic languages. If I were managing this text with a C compiler I would have to keep close track of each data type, the number of items in each array, the total size and length of each line, etc. It was a hassle.

Now run this code and notice that while the first *postln* shows that the array is indeed a string when first read from the file, in the subsequent code it is being treated as code.

```
   29.7.  interpreting a string

var fp, array;
fp = File("arrayfile", "r");
array = fp.readAllString; //read the file into a string
array.postln; //print to confirm it is a string
array = array.interpret; //interpret it and store it again
array.at(0).postln; //confirm it is code not a string
array.at(1).sum.postln;
array.at(2).at(0).postln;
array.at(3).postln;
```

The advantage to this method is that I can open the *arrayfile* with BBEdit or any text editor and modify the data as text. Even more promising: I can import data from a database such as FileMaker Pro or Excel as a text file. Any data source that can be saved as text could be read into an SC program. Now it is possible to have a Markov patch with modular files containing data for different probabilities.

[Future discussion: genetic probabilities]

[62] Note that a string within a string will not work if typed directly into SC because of the duplicate quotation marks. For example, a = "[[1, 2, 3], ["C4", "D4"]]" is not allowed. The compiler reads the second quote, next to the C4, as the closing quote. You can get around this using the backslash character: "[[1, 2, 3], [\"C4\", \"D4\"]]". But when contained in a file and interpreted as it is here, the backslash is unnecessary.

29. Exercises

29.1. None at this writing.

30 - Concrète, Audio Files, Live Audio DSP

Music Concrète

The most common definition for concrète music, what you will hear in an electro-acoustic music course, is composition using recordings of real sounds as raw material. This definition takes its meaning partly as a distinction from analog electronic sounds which are purely electronic. The attraction of concrète is the richness and complexity of the source. The sounds have a depth that is difficult to reproduce with pure electronics.

The second perhaps more accurate definition, which I like the more I think of it, is any recorded sound. It is concrète because it will never change. Every performance will be exactly the same. A recorded symphony is analogous to a statue; set in stone. This definition reminds us that live music has a dimension recorded pieces do not; the potential for growth and change. A written work that has to be realized is not the performance, but the potential for a performance, or a failed performance, or a brilliant performance, or in the case of aleatory and improvisation, something completely new, never heard before. I believe too many people equate a recording with a performance. In a sense it has tainted our expectations: we attend a concert expecting what we heard on the CD.

This chapter could have been titled sound file manipulation but I use concrète in perhaps an ironic sense because the real time dynamic controls available in SC allow us to take manipulation of concrète audio out of the realm of classic concrète in the second sense; not set in stone, but always evolving.

Buffers

Before processing audio it needs to be loaded into a memory buffer on the server. After audio is loaded into a buffer, either from a sound file on disk or from an input source, it is then available for processing, quotation, or precise playback manipulation.

The first line below reads audio from a sound file. The arguments for *Buffer* are the server where the audio is loaded (think of it as a tape deck), and the sound file name (see the discussion on file pathnames above).

Check the post window after running the first line. If the file was buffered correctly it should display *Buffer(0, -1, 1, sounds/africa2)*. The information displayed is buffer number, number of frames, channels, and pathname. You can also retrieve these values using *bufNum, numChannels, path, and numFrames*.

The arguments for *PlayBuf* are number of channels and buffer number.

```
   30.1.  Loading Audio into and Playing From a Buffer

b = Buffer.read(s, "sounds/africa2");

c = Buffer.read(s, "sounds/africa1", numFrames: 44100); // one second
```

```
// check data:
[b.bufnum, b.numChannels, b.path, b.numFrames].postln;

[c.bufnum, c.numChannels, c.path, c.numFrames].postln;

{PlayBuf.ar(1, 0)}.play(s); // Your buffer number may differ

{PlayBuf.ar(1, 1)}.play(s);
```

A frame is a single sample from all channels. If your sound file is mono then a frame is a single sample. If it is stereo then a frame has two samples; from both the left and right channels. If your file has 8 channels then a frame contains 8 samples. The number of frames per second is the sample rate, e.g. 44,100 per second. If –1 is given for the number of frames it reads all the frames in the file.

You can also record to a buffer from any input. Assuming you have a two channel system, the inputs should be 2 and 3. (Check your sound settings in the system preferences.) You can record over an existing buffer, e.g. the ones we just created above, but it is more likely you will want a new one, which has to be created, or allocated before hand. The size of the buffer is the sample rate * number of seconds.

To record, set the input source (using *In.ar*) and designate the buffer number. You can either check the post window to get the correct buffer number, or use *d.bufnum* as shown below. If you are using an external mic mute your output or use headsets to avoid feedback.

It seems incorrect to use the *play* message to record, but think of it as hitting the record and play buttons on a cassette recorder.

```
   30.2.  Loading Audio into a Buffer from Live Audio Source

d = Buffer.alloc(s, 44100 * 4.0, 1); // a four second 1 channel Buffer

{RecordBuf.ar(In.ar(2), d.bufnum, loop: 0)}.play;
d
{PlayBuf.ar(1, d.bufnum)}.play(s);
```

I have to expose a few preferences for this next section. I like loops. Maybe not as foreground, but certainly as background. They add a coherent foundation for customarily more abstract ideas in the foreground of electronic composition. I also lean toward foreign languages for spoken text. I hope it's not just faux sophistication, but rather to disengage the listener's search for meaning, shifting focus to patterns and inflection. I should also reveal an aversion: pitch shift of voices. Pitch shift or modulated playback speed is wonderful with other sounds, but I've never liked it with spoken concrète examples.

PlayBuf and *RecordBuf* have a loop argument, set to 0 (no loop) by default. To loop playback, set this value to 1. In the case of *RecordBuf* a loop is applied to playback and record. There are two additional arguments that affect the recording process: *recLevel* and

preLevel. The first is the level of the source as it is recorded. The second is the level the existing loop (pre-recorded material) is mixed when repeated. If, for example, the *preLevel* is set to 0.5 it will slowly fade away with each repetition.

One strategy for looping material is to create files that are the length you want the loops to be and play them using *PlayBuf* with loop on. I think this is inefficient and less flexible; you would have to create a file for each loop. Also, once your loops are defined they won't change with each performance and it is harder to change the loop in real time, e.g. augmentation, diminution, (not just playing the loop slower or faster, but lengthening it by adding more material to the loop) or shifting in time. It is better to load an entire concrète example into a buffer then loop selections using *BufRd*. The arguments for *BufRd* are number of channels, buffer number, phase, and loop. Phase is the index position of the file for playback (index in frames) and can be controlled with an audio rate source.

Indulge me one more walking-to-school-in-the-snow story. Even in the days where concrète composition was well established we were limited to a tape, decks, and play heads. You could play back at speeds available on the deck, or you could rock the reels back and forth by hand. One clever composition called for the playback head to be removed from the deck and moved by hand over magnetic tape allowing the performer to "draw" the playback on a patch of magnetic tape. It seemed like an interesting idea but the precise tolerance of the head position was impossible to set up correctly and it was not very effective. Here is the SC version of that idea. This example assumes you have loaded a 4 second example into a buffer. The phase control has to be an audio rate, so I use *K2R* to convert *MouseX* to an audio rate.

```
   30.3.  Playback with Mouse

b.free; b = Buffer.read(s,"sounds/africa2", 0, 4*44100);

{BufRd.ar(1, b.bufnum, K2A.ar(MouseX.kr(0, 4*44100)))}.play
```

Not as cool as I thought it would be back then.

Our next goal is to loop sections of the file. So at 44100 kHz sampling rate a two second loop would have 88200 frames. If we wanted to begin the loop 1.5 seconds into the file then the start point of the loop would be 66150. The end point then would be 66150 + 88200, or 154350. Still using the mouse as a control we would just have to enter those two numbers for start and end. Better yet, use a variable for the sample rate and multiply it by the number of seconds.

To create a simple loop on the same section as above we use and *LFSaw* (but see also *Phasor*, which I also tried but didn't feel it was as flexible) which generates a smooth ramp from –1 to 1.

There are two critical values for controlling the playback position: First the scale (and offset) of the *LFSaw* have to be correct for the number of frames; second, the rate of the saw has to be correct for normal playback speed. The help file uses two utilities for this: *BufFrames*

which returns the number of frames in a buffer and *BufRateScale* which returns the correct rate in Hz for normal playback speed of a given buffer. But both of these assume you are playing the entire file. We are looping sections, so we have to do our own calculations.

It would be easier to define the loop points in seconds rather than samples. To convert seconds to frames multiply by the sample rate. Defining loops using time requires that you know the length of the entire file. Exceeding the length won't cause a crash, it will just loop around to the beginning. Not a bad thing, just perhaps not what you want, maybe a good thing, so I'll leave that option open. You can also define the loops as a percentage of the entire file. See below for an example.

(When first trying these examples I was terribly confused about why an *LFSaw* that had a scale equal to the total frames of the file, but had not yet been offset, would still play the file correctly. I was sure the *LFSaw* was generating negative values. It was, but it was just wrapping around back to the beginning of the file, playing it from start to finish while the *LFSaw* was moving from –1 to 0, then again as it moved from 0 to 1. So you could loop an entire file by only setting the scale to the total frames, but read on.)

The scale and offset of the LFSaw could be set in terms of seconds rather than frame number. Given a loop beginning at 2" and ending at 6" the offset would be 4 and the scale 2. Or you could say that offset is (length/2) + beginning, offset is length/2. With these settings the *LFSaw* will move between 2 and 4. Multiplying the entire *LFSaw* by the sample rate will return the correct frame positions.

I prefer using *LinLin* which converts any linear set of values to another range. The arguments are input (e.g. the ugen you want to convert), input low and high (i.e. the range of the input ugen by default), and output low and high (what you want it converted to). It does the same as an offset and scale, but is clearer if you are thinking in terms of a range. So these two lines of code have the same result, but with one important difference: with *LinLin* you can invert the signal (useful for reverse playback) with values such as *–1, 1, 1000, 700*. The last example won't play, but shows how the *LFSaw* needs to be setup to correctly sweep through a range of samples.

```
   30.4.  LinLin, LFSaw for Sweeping Through Audio File

// Same thing:

{SinOsc.ar(LinLin.kr(SinOsc.kr(5), -1, 1, 700, 1000))}.play

{SinOsc.ar(SinOsc.kr(5, mul: 150, add: 850))}.play

// Will sweep through 66150 110250, or 1.5" to 2.5" of audio file

LinLin.ar(LFSaw.ar(1), -1, 1, 1.5, 2.5)*44100
```

We have one more complicated calculation before doing a loop; playback rate. The rate of playback is going to be controlled by the frequency of the *LFSaw*. How fast do we want it to

sweep through the given samples? If the length of the loop is 1 second, then we would want the *LFSaw* to move through all the values in 1 second, so the frequency would be 1. But if the length of the loop is 2 seconds, we would want the saw move through the samples in 2 seconds. Remember that duration and frequency are reciprocal. So for an *LFSaw* duration of 2 seconds, frequency would be 1/2, or 1/duration.

We're not done yet. 1/duration is normal playback. If we want a faster or slower playback rate we would multiply the frequency by that ratio: ratio*(1/duration), or ratio/duration. So 2/duration would playback twice as fast. 0.5/duration half as fast.

Phew. Ok, on to the example. No, wait! one more thing.

When you are looping possibly random points in the file (which by now you know I'm heading for) you have potential for clicks. If the beginning point is a positive sample in the wave and the end is a negative the result is a sharp transition; the equivalent to a non-zero crossing edit. This problem is corrected with an envelope the length of the loop with a fast attack and decay, and a gate equal to the frequency of the loop. Ok, the example.

Oh yeah, and we also want the *LFSaw* to begin in the middle of its phase, which is where it reaches its lowest value, so the phase of the saw is set to 1 radian. This is not an issue with *Phasor*, but there was some other problem when I did an example with it.

30.5. Looping a Section of a File

```
(
{
var bufNum = 0, srate = 44100, start = 0, end = 3, duration, rate = 1;
// Use these lines for proportional lengths
// var bufNum = 0, srate = 44100, start = 0.21, end = 0.74,
// rate = 1, duration, total;
// total = BufFrames.kr(bufNum)/44100;
// end = end*total; start = start*total;
duration = abs(end - start);
BufRd.ar(1, bufNum, // Buffer 0
   LinLin.ar(
      LFSaw.ar(rate/duration, 1), -1, 1, start, end)*srate
   )*EnvGen.kr(Env.linen(0.01, 0.98, 0.01), timeScale: duration,
      gate: Impulse.kr(1/duration));
}.play
)
```

Magically, the same example works with proportional values or a percentage of the total length of a file (commented out above). In this case the ugen *BufFrames* is useful when divided by sample rate (to return total length of the file in seconds). Of course, when using this method length should not exceed 1.0 and start + length should not exceed 1.0 (but don't let me stop you—it wraps around).

30.6. Looper

```
(
{
var bufNum = 0, srate = 44100, start = 0.21, end = 0.74,
   rate = 1, totalDur = 20, pan = 0;
var out, duration, total;
start = [0.3, 0.2]; end = [0.2, 0.3];
total = BufFrames.kr(bufNum)/44100;
end = end*total; start = start*total;
duration = abs(end - start);
BufRd.ar(1, bufNum, // Buffer 0
   LinLin.ar(
      LFSaw.ar(rate/duration, 1), -1, 1, start, end)*srate
   )*EnvGen.kr(Env.linen(0.01, 0.98, 0.01), timeScale: duration,
      gate: Impulse.kr(1/duration));
}.play
)
```

Enough work, on to the fun. You might place this looper in a *SynthDef* with a task running different examples, but what I want to demonstrate would be more complicated as such, so the example above is just a stereo array for start and end. Here are some things to try (change the start and end values, remember these are proportional, i.e. percentage of total audio file time):

- forward and backward loops of the same material as shown above (*start = [0.2, 0.3]; end = [0.3, 0.2];*)
- loops with slightly different lengths so that they slowly move out, then eventually go back into phase, e.g. every tenth time (*start = [0.2, 0.2]; end = [0.31, 0.3];*)
- different lengths but a given ratio, such as 3/2 (start = 0.2 for both, end = 0.3 for one, so the length is 0.1, then the second length would be 0.15 so end is start + 0.15, or 0.35) so that one plays 3 times while the other plays 2, like an interval–this is a pretty cool effect with short examples (*end = [0.35, 0.3]*)
- expand the playback rate into an array achieving effects similar to different lengths
- combine different playback rates with lengths

Of course the audio file or live input is a signal and can be modulated in any of the ways previously discussed. Following are a few ideas. They all assume you are using buffer 0, and that you are recording live audio to it, or have loaded a sound file into it.

30.7. Modulating Audio Buffers

```
// FM Modulation

(
{
var bufNum = 0, srate = 44100, start = 0, end = 3, duration, rate = 1, signal;
duration = abs(end - start);
```

```
// or
// end = [2.3, 3.5];
signal = BufRd.ar(1, bufNum, // Buffer 0
    LinLin.ar(
        LFSaw.ar(rate/duration, 1), -1, 1, start, end)*srate
    )*EnvGen.kr(Env.linen(0.01, 0.98, 0.01), timeScale: duration,
        gate: Impulse.kr(1/duration));

SinOsc.ar(LFNoise1.kr([0.4, 0.43], mul: 200, add: 200))*signal;
// or
// SinOsc.ar(LFNoise0.kr([12, 15], mul: 300, add: 600))*signal;
// or
// SinOsc.ar(LFNoise1.kr([0.4, 0.43], mul: 500, add: 1000))*signal;

}.play(s)
)


// Pulsing in and out

(
{
var bufNum = 0, srate = 44100, start = 0, end = 3, duration, rate = 1;
var pulse;
pulse = [6, 10];
duration = abs(end - start);
BufRd.ar(1, bufNum, // Buffer 0
    LinLin.ar(
        LFSaw.ar(rate/duration, 1), -1, 1, start, end)*srate
    )*EnvGen.kr(Env.linen(0.01, 0.3, 0.01), timeScale: duration/pulse,
        gate: Impulse.kr(pulse/duration));
}.play(s)
)

// Filtered

(
{
var bufNum = 0, srate = 44100, start = 0, end = 3, duration, rate = 1, signal;
duration = abs(end - start);
signal = BufRd.ar(1, bufNum, // Buffer 0
    LinLin.ar(
        LFSaw.ar(rate/duration, 1), -1, 1, start, end)*srate
    )*EnvGen.kr(Env.linen(0.01, 0.98, 0.01), timeScale: duration,
        gate: Impulse.kr(1/duration));
RLPF.ar(signal, LFNoise1.kr([12, 5], mul: 700, add: 1000), rq: 0.05)*0.2;
}.play(s)
)

// Modulating one with another, dry in left, dry in right, modulated center
// Listen for a while for full effect

(
```

```
{
var bufNum = 0, srate = 44100, start = 0, end = 3, duration, rate = 1, signal;
end = [2.5, 2.9];
duration = abs(end - start);
signal = BufRd.ar(1, bufNum, // Buffer 0
    LinLin.ar(
        LFSaw.ar(rate/duration, 1), -1, 1, start, end)*srate
    )*EnvGen.kr(Env.linen(0.01, 0.98, 0.01), timeScale: duration,
        gate: Impulse.kr(1/duration));
(signal*0.1) + (signal.at(0) * signal.at(1))
}.play(s)
)
```

31 - Graphic User Interface Starter Kit

GUIs are rarely about the function of the patch, but rather connecting controls to a user or performer. When first building on an idea it is far more efficient for me to work with a non graphic (code based) synthesis program. For that reason (and also because I tend toward automation that excludes external control) GUIs are the last consideration for me. That's one of the reasons I prefer working with SC: GUIs are optional.

My second disclaimer is that since GUIs are not about the actual synthesis process I care even less about understanding the details of code and syntax. I usually have a few models or prototypes that I stuff my code into. That said, here are a few that I have found useful.

Display

Display is the quickest and easiest way I've seen to attach sliders, buttons, number boxes and names to a patch. Below is a prototype. The first argument is the display window itself. You use this argument to set attributes such as the name, GUI items, and patch to which it is attached. The next three arguments, a through c, are the sliders and numbers.

```
   31.1.  Display window
(
Display.make({arg thisWindow, a, b, c;
   thisWindow.name_("Example");
}).show;
)
```

The default values for the sliders are 0 to 1, starting point of 0.5, and an increment of 0.00001 (or so). If you want to change those defaults, use the *.sp* message, with arguments of default starting value, low value, high value, step value, and warp.

```
   31.2.  header
(
Display.make({arg thisWindow, a, b, c, d;
   a.sp(5, 0, 10, 1); //start, low, high, increment, warp
   b.sp(400, 0, 4000, 0.1, 1);
   c.sp(0, -500, 500);
   d.sp(0.3, 0, 0.9);
   thisWindow.name_("Example");
}).show;
)
```

Next you can define and attach a patch to the display, connecting the controls as arguments using *synthDef*.

```
   31.3.  Display with synthDef
(
Display.make({arg thisWindow, sawFreq, sawOffset, sawScale, volume;
   sawFreq.sp(5, 0, 10, 1);
   sawOffset.sp(400, 10, 1000, 0.1, 1);
```

```
    sawScale.sp(0, -500, 500);
    volume.sp(0.3, 0, 0.9);
    thisWindow.name_("Example");
    thisWindow.synthDef_({arg sawFreq, sawOffset, sawScale, volume;
        Out.ar(0,
            SinOsc.ar(
                abs(LFSaw.kr(sawFreq, mul: sawScale, add: sawOffset)),
                mul: volume
            )
        )},
        [\sawFreq, sawFreq, \sawOffset, sawOffset, \sawScale,
            sawScale, \volume, volume]
    );
}).show;
)
```

There is a way to link all these items together using an environment, but I'll leave that to you. I mostly use this as a quick easy interface for examples and experiments. You can declare extra arguments even if they aren't used, so I keep this model handy and just insert the synth as needed.

```
    31.4.  Display shell
(
Display.make({arg thisWindow, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p;
    // [Your variable defaults go here], e.g.
    a.sp(440, 100, 1000, 1, 1);
    thisWindow.synthDef_({arg a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p;
        Out.ar(0,
            // [Your patch goes here with variables a, b, c, etc.], e.g.
            SinOsc.ar(a);
        )},
        [\a, a, \b, b, \c, c, \d, d, \e, e, \f, f, \g, g, \h, h,
            \i, i, \j, j, \k, k, \l, l, \m, m, \n, n, \o, o, \p, p]
    );
    thisWindow.name_("Example");
}).show;
)
```

Document

The next is *Document*. This object manages documents (the files represented by windows where you type code) in SC3. You can open, close, change the size, background color and visibility of any window of any document.

All objects in SC have attributes. These are stored in variables that you can (if they are coded as such) retrieve or change (get or set). Any open document has a title, background color, and a boundary. You can get them, or change them, with the following lines. Run each line below and follow the post window. The second to last line makes the window nearly invisible, so I've added a last line that returns it to a more opaque state. Seeing the window disappear is a little disconcerting. Just hit enter one more time to run the last line.

31.5. This Document

```
d = Document.current;

d.title; // get the title
d.bounds; // get the bounds

d.title_("new title"); // set the title
d.bounds_(Rect(100, 100, 500, 500)); // set the bounds
d.background_(Color(0.5, 0.2, 0.7, 0.3)); // set color and visibility
d.background_(Color(0.9, 0.7, 0.7, 1)); // set color and visibility
```

The Rect arguments are points (72 to an inch?) and describe the position and size of the document window starting from the left of the screen in, then from the bottom up (used to be top to bottom). That's where the window starts, then the size of the document window is left across, then bottom up.

The background *Color* arguments are red, green, blue, and alpha (transparency)[63]. Try changing the alpha to 0.1. (Save it first.) It disappears not only from view but is also invisible to the mouse. You will no longer be able to bring the document to the front by clicking the window. You have to click the top of the scroll bars.

It's a good idea to use global variables for the document names and the action inside the functions. Local variables will not be understood by subsequent commands.

31.6. Flashing alpha

```
~doc = Document.current;

Task({
100.do({
   ~doc.background_(Color(
      0.2, // red
      0.1, // green
      0.9, // blue
      rrand(0.3, 0.9) // alpha, or transparancy
   ));
   0.1.wait;
})
}).play(AppClock)

// Each open document has an index number:

Document.allDocuments.at(1).front;

(
```

[63] Look in the help file for a collection of colors by name. These have apparently been disabled in the .sc file, but you can copy and past them.

```
Document.new("Test one", "test text");
Document.new("Test two", "more text");
Document.new("Test three", "three");
Document.new("Four", "four");
 Task({
    Document.allDocuments.do({arg thisOne;
    // random
    thisOne.front;
    thisOne.background_(
        Color(rrand(0.1, 0.9), rrand(0.1, 0.9), rrand(0.1, 0.9)));
     0.4.wait;
    });

 }).play(AppClock)
)
```

Pretty flashy, but just flashy.

A more practical application is being able to attach functions to the actions of keys, mouse clicks of a document or when a document is brought to the front. Take close note of how to turn these actions off. They will engage every time you click the mouse or type or bring the document to the front, and can get annoying pretty fast.

First identify the current document and assign it to a variable. The current document is the one you are typing the code into. Then set the *toFrontAction* with a function to be evaluated. After running the first two lines below, switch to and from that document and watch the post window.

```
    31.7.  This Document to Front Action

(
~doc.toFrontAction_({"bong".postln});
~doc.endFrontAction_({"bing".postln});
)

~doc.toFrontAction_({nil}); //turn action off
~doc.endFrontAction_({nil});
```

The most obvious application of this feature is to start or stop a process. Again, use global variables. (Or use a global synth definition.) Any code that you can evaluate from an open window can be placed inside a *toFrontAction*.

```
    31.8.  This Document to front

~randSineDoc = Document.current;
~randSineDoc.toFrontAction_({
    ~sine = {Mix.ar({SinOsc.ar(rrand(200, 900))}.dup(20))*0.01}.play
})
```

```
~randSineDoc.endFrontAction_({~sine.free})
```

See what I mean about being able to turn it off? Now every time you try to edit the stupid thing you get random sine waves!

You can create similar actions linked to functions using key down or mouse down. The arguments for the *keyDownAction* function are the document itself (so you could insert *thisDoc.title.postln*) the key pressed, the modifier, and the ascii number of the key pressed. Use an if statement to test for each key using either *key == $j* or *num == 106* where 106 is the ascii equivalent for j. In the example below there are two ways to turn the events off; explicitly with *key = $k* and a *free* message, or globally with the *s.freeAll*, which is assigned to ascii 32, or the space bar. The last example shows how to assign a *freeAll* (or any other action) to a mouse down.

I use global variables for consistency in this example. It would be better to use *Synth* or OSC commands to start and stop instruments you've already defined.

```
   31.9.  Mouse and Key Down

(
~doc.keyDownAction_({arg thisDoc, key, mod, num;
   if(key == $j, /* or num == 106 */ {
      ~mySine = {SinOsc.ar(
         LFNoise0.kr(rrand(8, 15), mul: 500, add: 1000),
         mul: 0.2)*
         EnvGen.kr(Env.perc(0, 3))
      }.play;
   });
   if(key == $h, {
      ~randSine = {
         Mix.ar({SinOsc.ar(rrand(200, 900))}.dup(20))*0.01
      }.play
   });
   if(key == $k, {~randSine.free});
   if(num == 32, {s.freeAll}); // space bar
});
)

~doc.mouseDownAction_({s.freeAll});
```

Keyboard Window

The next very clever interface is the keyboard window. It is currently in the examples folder of the SC3 folder, reproduced without modification below. Run the top part to bring up the window then drag functions onto any key. That key (either typed on the keyboard or clicking the keyboard window) will then execute the dragged function. Note the method for using a single key to turn the same function on and off.

```
   31.10. Keyboard Window From Examples (by JM?)
```

```
(
var w; // window object
var courier; // font object

// an Array of Strings representing the key layout.
var keyboard = #["`1234567890-=", "QWERTYUIOP[]\\",
                        "ASDFGHJKL;'", "ZXCVBNM,./"];

// horizontal offsets for keys.
var offsets = #[42, 48, 57, 117];

var actions; // an IdentityDictionary mapping keys to action functions.
var makeKey; // function to create an SCDragSink for a key.

courier = Font("Courier-Bold", 14);

// an IdentityDictionary is used to map keys to functions so that
// we can look up the action for a key
actions = IdentityDictionary.new; // create actions dictionary

// define a function that will create an SCDragSink for a key.
makeKey = {|char, keyname, bounds|
    var v;

    keyname = keyname ? char.asString;
    bounds = bounds ? (24 @ 24);

    v = SCDragBoth(w, bounds);
    v.font = courier;
    v.string = keyname;
    v.align = \center;
    v.setBoth = false;
    v.acceptDrag = {
        SCView.currentDrag.isKindOf(Function)
    };
    v.action = {
        ("added key action : " ++ keyname).postln;
        if (char.isAlpha) {
            actions[char.toUpper] = v.object;
            actions[char.toLower] = v.object;
        }{
            actions[char] = v.object;
        };
        w.front;
    };
};

w = SCWindow("keyboard", Rect(128, 320, 420, 150));

w.view.decorator = FlowLayout(w.view.bounds);

// define a function to handle key downs.
```

```
w.view.keyDownAction = {|view, char, modifiers, unicode, keycode|
     var result;

    // call the function
    result = actions[char].value(char, modifiers);

    // if the result is a function, that function becomes the
    // new action for the key
    if (result.isKindOf(Function)) {
        actions[char] = result;
    };
};

// make the rows of the keyboard
keyboard.do {|row, i|
    row.do {|key| makeKey.(key) };
    if (i==0) { makeKey.(127.asAscii, "del", 38 @ 24) };
    if (i==2) { makeKey.($\r, "retrn", 46 @ 24) };
    w.view.decorator.nextLine;
    w.view.decorator.shift(offsets[i]);
};

// make the last row
makeKey.($ , "space", 150 @ 24);
makeKey.(3.asAscii, "enter", 48 @ 24);

w.front;
)




//////////////////

// Drag these things to the keyboard to test it.

(
{
    var synth, original;
    original = thisFunction;
    synth = { SinOsc.ar(exprand(500,1200),0,0.2) }.play;
    { synth.free; original }
}
)


(
{
    {
        Pan2.ar(
            SinOsc.ar(
                ExpRand(300,3000),
                0,
```

```
            SinOsc.kr(ExpRand(1,15),0,0.05).max(0)),
          Rand(-1,1))
   }.play;
}
)

{ s.sendMsg(\n_free, \h, 0); } // kill head

{ s.sendMsg(\n_free, \t, 0); } // kill tail

(
{{
   var eg, o, freq, noise;
   eg = EnvGen.kr(Env.linen(0.1,2,0.4,0.2), doneAction: 2);
   freq = Rand(600,1200);
   noise = {LFNoise2.ar(freq*0.1, eg)}.dup;
   o = SinOsc.ar(freq,0,noise);
   Out.ar(0, o);
}.play})


(
{{
   var in, sr;
    in = LFSaw.ar([21000,21001], 0, LFPulse.kr(ExpRand(0.1,1),0,0.3,0.2,0.02));
    sr = ExpRand(300,3000) + [-0.6,0.6];
   Out.ar(0,  RLPF.ar(in * LFPulse.ar(sr, 0, MouseY.kr(0.01, 0.99)), sr *
(LFPulse.kr(ExpRand(0.1,12),0,0.4,0.2,0.2) +
LFPulse.kr(ExpRand(0.1,12),0,0.7,0.2)), 0.1));
}.play;})

(
{{ var in;
   in = In.ar(0,2);
   ReplaceOut.ar(0, CombN.ar(in, 0.24, 0.24, 8, 1, in.reverse).distort);
}.play})

(
{{ var in;
   in = In.ar(0,2);
   ReplaceOut.ar(0, in * SinOsc.ar(MouseX.kr(2,2000,1)));
}.play})
```

Windows and Buttons

When creating an interface the first thing you need is a window. (See also *Sheet* and *ModalDialog*.) It is traditionally assigned to the variable w, but you can use any name you want. We will create two, *v* and *w*. Once created, and brought to the front, buttons can be added. The first argument for *SCButton* is the window where it is placed, the second is its

size. The *states* message describes an array of button states which include name, text color, and background color. Button *b* has only one state and is placed in window *v*, using defaults for color while button c has two states with specified colors for state one.

It's a good idea to close windows when you're done. Otherwise they pile up.

```
    31.11. Windows and Buttons

(
v = SCWindow("Window v", Rect(20, 400, 400, 100));
v.front;
w = SCWindow("Window w", Rect(460, 400, 400, 100));
w.front;

b = SCButton(v, Rect(20, 20, 340, 30));
b.states = [["Button b"]];
c = SCButton(w, Rect(20, 20, 340, 30));
c.states = [["Button c on", Color.black, Color.red], ["Button c off"]];
)

// When finished experimenting, close both:

v.close; w.close;
```

Next we assign some type of action to each button. Pressing the button evaluates the function and passes the "state" argument (which state the button is after this click). The state is 0 for button *b*, since there is only one state, 0 or 1 for button *c* since there are two states.

Note that the button is created at state 0, so state 1 will be the first evaluated state when the button is pushed, 0 will be evaluated at next button press. So I label the 0 state "start" because that is what will happen when the button goes to state 1, but state 1 is labeled "stop" because that's what state 0 will do.

```
    31.12. States and Actions of Buttons

(
v = SCWindow("Window v", Rect(20, 400, 400, 100));
v.front;
w = SCWindow("Window w", Rect(460, 400, 400, 100));
w.front;

b = SCButton(v, Rect(20, 20, 340, 30));
b.states = [["Button b"]];
c = SCButton(w, Rect(20, 20, 340, 30));
c.states =
    [["Start (State 0)", Color.black, Color.red], ["Stop (State 1)"]];

b.action = {w.view.background = Color(0.8, 0.2, rrand(0.2, 0.9))};
c.action = { | state | // shorthand for arg state;
    if(state.value == 0, {s.freeAll});
    if(state.value == 1, {{SinOsc.ar}.play})
};
```

```
)

// When finished experimenting, close both:

v.close; w.close;
```

Slider

Next we add a slider to control frequency. In SC2 when you added new items to a window you had to calculate carefully the layout so you knew where to place each new button or slider, but SC3 has a handy feature, *FlowLayout* which understands the *nextLine* message and will position the next button, slider, text box, etc. on the next line.

The sliders are going to control frequency and amplitude, so this example starts out with a *SynthDef* with arguments for *freq* and *amp*. You can run this entire section at once, or run the expressions one at a time to see each new button and slider appear in the window.

The arguments for easy slider are destination window, bounds (I use another shorthand; 500 @ 24 = width and height of slider or button), name, control specifications, and function to be evaluated with the argument of the control value. Control specifications arguments are minimum, maximum, warp, and step value.

```
    31.13. Slider

(

SynthDef("WindowSine",
    {arg freq = 440, amp = 0.9;
    Out.ar(0, SinOsc.ar(freq, mul: amp))
    }).load(s);

w = SCWindow("Window w", Rect(460, 400, 600, 200));
w.front;

w.view.decorator = FlowLayout(w.view.bounds);

c = SCButton(w, 500 @ 24);
c.states = [["Start (State 0)"], ["Stop (State 1)"]];

c.action = { | state | // shorthand for arg state;
    if(state.value == 0, {a.free});
    if(state.value == 1, {a = Synth("WindowSine")})
};
w.view.decorator.nextLine;
EZSlider(w, 500 @ 24,
    "Frequency", ControlSpec(200, 1000, \exponential, 1),
    {|ez| a.set(\freq, ez.value) });
w.view.decorator.nextLine;
EZSlider(w, 500 @ 24,
    "Volume", ControlSpec(0.1, 1.0, \exponential, 0.01),
    {|ez| a.set(\amp, ez.value) });
```

```
)

// When finished experimenting, close both:

v.close; w.close;
```

Complicated? You're not done yet: slider settings aren't remembered, com-. stops the sound but the button doesn't change, closing the window doesn't stop the sound. I'll refer you to the GUI demonstration in the examples folder for solutions to those problems. There are many situations where I use a GUI. But I think often too much programming time is spent on user interface. Wouldn't you rather be working on the music?

30. Exercises

30.1. None at this writing.

APPENDIX

A. Converting SC2 Patches to SC3

Converting a simple patch

For years I put off learning SC3 thinking I had to start from scratch, then a student–to whom I am very grateful–took the plunge and discovered the criterion that finally pushed me forward. For the most part, your code will run in SC3. The principle things that won't run are spawners. The other good news is that the language hasn't changed that much. An object is still an object, messages still have argument lists, etc.

There are several good help files that explain the conceptual differences. This section is intended to give you the practical details; what changes you need to make to get older code to run.

The biggest change is SC now has three components: the program that edits and evaluates code, an internal server, and a local server. The servers are synthesizers. They take the definitions and instruments you design and play them. They know nothing about the code. They just play. The server is the performer. The language is the score. The server can't see the score. It just knows how to play.

The two synths are sometimes a source of confusion. You can use either one, but the local server is usually default. You have to send definitions to the one that is running. Not specifying which server may send a definition to the wrong one. In a nutshell the local is more stable, scope only works on internal. Otherwise you can use either one.

To make SC3 backward compatible the programmers have added some utilities that are less efficient and have fewer control options than doing it the new way. All of the details, like creating a synth, naming it, and assigning a node are done automatically, so *Synth.play({ SinOsc.ar })* and *{SinOsc.ar}.play* both work. But it is a good idea to designate the server so I usually use *{SinOsc.ar}.play(s)*, where s is the server.

```
   31.14. Converting a simple patch to SC3

// SC2 patch

(
Synth.play(
   {
      SinOsc.ar(LFNoise0.kr(12, mul: 500, add: 600))
   }
)
)

// SC3 backward compatible version
```

```
s = Server.internal.boot;

(
   {
      SinOsc.ar(LFNoise0.kr(12, mul: 500, add: 600})
   }.play(s)
)
```

When you run this code SC3 creates a synth, names it, chooses a bus out, sends it to the server with a node number and plays it. I was pleasantly surprised to find most of my code still worked using these backward compatible features.

The problem with letting SC do all that behind the scenes is that you have no control over the synth once it's playing. If you do those things explicitly you can manipulate the synth once it is on the server. Here is the same code done the SC3 way. The *UgenGraphFunc* is the same, you just place it in a *SynthDef*.

```
   31.15. SynthDef

// SC2 patch

(
Synth.play(
   {
      SinOsc.ar(LFNoise0.kr(12, mul: 500, add: 600))
   }
)
)

// SC3 method

s = Server.internal.boot;

(
SynthDef("MySine",
   {
   var out;
      out = SinOsc.ar(LFNoise0.kr(12, mul: 500, add: 600));
      Out.ar(0, out);
   }).load(s) // or .play(s) or .send(s)
)

// Prototype

Synth.play({oldUgenFunc})

SynthDef("Name",
{
var out;
out = oldUgenFunc;
Out.ar(0, out);
```

```
}).load(s)
```

Look in the help files for the differences between play, load, and send. Now the synth exists on the server but it isn't playing. To play it, run *Synth("name")*. It is a good idea to assign it to a variable because that allows us to send it commands.

```
    31.16. SynthDef

// After the synth has been sent to the server, play it

a = Synth("MySine")

// And stop it

a.free;
```

If you add arguments to the UgenGraphFunc then you can send commands to change them while the synth is running.

```
    31.17. SynthDef with arguments

(
SynthDef("MySine",
    { arg rate = 12;
    var out;
     out = SinOsc.ar(LFNoise0.kr(rate, mul: 500, add: 600));
    Out.ar(0, out);
    }).load(s) // or .play(s) or .send(s)
)


a = Synth("MySine") // or a = Synth("MySine", [\rate, 15]);

// And stop it

a.set(\rate, 22);
a.set(\rate, 3);
a.free;
```

All the other details about creating and managing synths, routing to busses, managing node numbers, etc. are completely different from SC2, so you just need to read those sections. Those methods often require you to rework the patches entirely. For example, the patch above could be broken into two synths: the *LFNoise0* and the *SinOsc*, which would be patched together using busses.

iphase

In several Ugens, most notably *LFSaw* and *LFPulse*, there is an additional argument; iphase. If you had a patch that used *LFSaw(3, 500, 900)* for freq, mul, and add, you will now have to add a 0 for iphase or use keyword assignments. These are the only two I've encountered. There might be more.

rrand, rand, choose, Rand, TRand, TChoose

Once you send a definition to the server all the parameters are hard wired unless you change them using *set*. For example, if you use a *rrand(200, 800)* or *[100, 300, 500, 700].choose* for a frequency argument the program will choose one value that it will use when the synth is defined. That value—not the *rrand* function—is hard wired into the synth definition and will stay the same each time the synth is played. The Ugens Rand, TRand, TWChoose, and TChoose, replace the functionality of a rand or rrand on the server side. These will choose a new value each time the synth is played, or when they are triggered.

Spawning Events

There are no spawners in SC3. The servers don't understand how to spawn events on their own. The way you create multiple events is by sending commands to the server to start several synths or several copies of the same synth. The language side of the program manages when the events occur, how many there are, and what attributes they have.

Presumably these events have a beginning and end. The spawners from SC2 recouped the processor power for each of these events when thy were completed. With SC3 we need to do a similar action. There is an argument called *doneAction* in *EnvGen* which tells the synth what to do when it is finished. The default is to do nothing. We need to set it to 2, or turn yourself off. If not our spawned events would build up and crash the processor.

```
   31.18. Spawn

(
SynthDef("MySine",
   { arg rate = 12, offset = 1000, scale = 900;
      var out;
   out = SinOsc.ar(LFNoise0.kr(rate, mul: scale, add: offset}
      *
   EnvGen.kr(Env.perc(0, 2), doneAction: 2);
   Out.ar(0, out)
   }).load(s) // or .play(s) or .send(s)
)

// Run this line several times.
// Each one is equivalent to a spawned event.
Synth("MySine", [\rate, rrand(4, 16), \offset, 600, \scale, 400])
```

Now that the synth understands how to play an instrument we need to set up a routine to send start event signals to the server. This would replace the spawn. We can either start new events each time, or modify events that are still sounding. Since there is a rather short decay on our instrument we'll send multiple events. How is this different from just building a trigger into the synth? This way we can have several events happening at once. Note that the language side of the program understands rrand so we can use it here.

```
   31.19. Spawning events with Task

(
//run a task to play the synth
r = Task({
   {Synth("MySine",
      [\rate, rrand(5, 20),
      \offset, rrand(400, 800),
      \scale, rrand(100, 300)]);
   //Choose a wait time before next event
   rrand(0, 3.0).wait;
   }.loop;
}).play
)
```

B. Cast of Characters, in Order of Appearance

[Maybe next time]

C. OSC

[See Mark Polishook's tutorial for information on OSC]

D. Step by Step (Reverse Engineered) Patches

Here are some patches that are broken down into steps. Many of them are taken out of the SC2 examples folder (written by James).

```
Server.default = s = Server.internal.boot
```

// *Rising Sine Waves*

```
(
{
SinOsc.ar(440, mul: 0.4) //SinOsc
}.play(s)
)

(
{
var cont2;
//control frequency with LFSaw
//midi values are converted to frequency
cont2 = LFSaw.kr(2/5, 0, 24, 80).midicps;
SinOsc.ar(cont2, mul: 0.4)
}.play(s)
)

(
{
var cont1;
cont1 = LFSaw.kr(8, 0, 3, 80).midicps; //second control for offset
SinOsc.ar(cont1, mul: 0.4)
}.play(s)
)

(
{ //combine the two, but do the midicps only once
var cont1, cont2;
cont1 = LFSaw.kr(8, 0, 3, 80);
cont2 = LFSaw.kr(2/5, 0, 24, cont1).midicps;
SinOsc.ar(cont2, mul: 0.4)
}.play(s)
)

(
{
var cont1, cont2; //add random values and stereo
cont1 = LFSaw.kr([rrand(6.0, 8.0), rrand(6.0, 8.0)], 0, 3, 80);
cont2 = LFSaw.kr(2/5, 0, 24, cont1).midicps;
SinOsc.ar(cont2, mul: 0.4)
}.play(s)
)
```

```
(
{
var cont1, cont2, out;
cont1 = LFSaw.kr([rrand(6.0, 8.0), rrand(6.0, 8.0)], 0, 3, 80);
cont2 = LFSaw.kr(2/5, 0, 24, cont1).midicps;
out = SinOsc.ar(cont2, mul: 0.1);
out = CombN.ar(out, 0.2, 0.2, 4); //add echo
out
}.play(s)
)

(
SynthDef("RisingSines",
{  arg busOut = 3, sweep = 3, rate = 0.4, offset = 80, range = 24;
   var cont1, cont2, out;
   cont1 = LFSaw.kr([rrand(6.0, 8.0), rrand(6.0, 8.0)], 0, sweep, offset);
   cont2 = LFSaw.kr(rate, 0, range, cont1).midicps;
   out = SinOsc.ar(cont2, mul: 0.1);
   Out.ar(busOut, out);
}).send(s);
SynthDef("Echo",
{ arg busIn = 3, delay = 0.2;
   var out;
   out = CombN.ar(In.ar(busIn, 2), delay, delay, 4);
   Out.ar(0, out);
}).send(s)
)

Synth("Echo");
Synth("RisingSines");
Synth("RisingSines", [\rate, 1/7, \offset, 60, \range, 32]);
```

// Random Sine Waves

```
(
{
FSinOsc.ar(exprand(700, 2000)) //single random Sine
}.play(s)
)

(
{
FSinOsc.ar(exprand(700, 2000), 0,
   //Random envelopes using LFNoise1
   //Let it run for a while
   max(0, LFNoise1.kr(3/5, 0.9)))
}.play(s)
)

(
```

```
{

Pan2.ar( //pan position
    FSinOsc.ar(exprand(700, 2000), 0,
    max(0, LFNoise1.kr(3/5, 0.9))),
    //random moving pan, let it run for a while
    LFNoise1.kr(1/3))
}.play(s)
)

(
{
var sines;
sines = 60;
//Mix a bunch of them down and decrease frequency of LF env
Mix.ar({Pan2.ar(
    FSinOsc.ar(exprand(700, 2000), 0,
    max(0, LFNoise1.kr(1/9, 0.7))),
    LFNoise1.kr(1/3))}.dup(sines))*0.2
}.play(s)
)

(
{
var sines;
sines = 60;
//Increase frequency of env
Mix.ar({Pan2.ar(
    FSinOsc.ar(exprand(700, 2000), 0,
    max(0, LFNoise1.kr(9, 0.7))),
    LFNoise1.kr(1/3))}.dup(sines))*0.2
}.play(s)
)

Sync Saw

(
{
SyncSaw.ar(440, mul: 0.2) //simple Saw
}.play(s)
)

(
{
SyncSaw.ar(
    100, //Saw frequency
    MouseX.kr(50, 1000), //Sync frequency
    mul: 0.2)
}.scope(1)
)

(
{
```

```
SyncSaw.ar(
    100, //Saw frequency
    //Sync controlled by SinOsc
    SinOsc.ar(1/5, 0, mul: 200, add: 300),
    mul: 0.2)
}.scope(1)
)

(
{
SyncSaw.ar(
    100, //Saw frequency
    //Separate phase for left and right channel
    SinOsc.ar(1/5, [0, 3.0.rand], mul: 200, add: 300),
    mul: 0.2)
}.scope(2)
)

(
{
SyncSaw.ar(
    [100, 100*1.02], //Separate freq for L, R
    SinOsc.ar(1/5, [0, 3.0.rand], mul: 200, add: 300),
    mul: 0.2)
}.scope(2)
)

(
{
var freq;
freq = rrand(30, 80).midicps; //choose freq
SyncSaw.ar(
    [freq, freq*1.02], //freq variable replaces static values
    SinOsc.ar(1/5, [0, 3.0.rand], mul: freq*2, add: freq*3),
    mul: 0.2)
}.scope(2)
)


(//add an envelope
{
var freq, sig, env;
freq = rrand(30, 80).midicps;
env = EnvGen.kr(Env.linen(rrand(1.0, 3.0), rrand(4.0, 7.0), rrand(2.0, 3.0)));
sig = SyncSaw.ar(
    [freq, freq*1.002], //Saw frequency
    SinOsc.ar(1/5, [0, 3.0.rand], mul: freq*2, add: freq*3),
    mul: 0.1);
sig = CombN.ar(sig, 0.3, 0.3, 4, 1); //Add echo
sig*env
}.scope(2)
)
```

```
(//Send synth def to server with freq argument
SynthDef("SyncSaw-Ex",
{
arg freq;
var sig, env;
env = EnvGen.kr(Env.linen(rrand(1.0, 3.0), rrand(4.0, 7.0), rrand(2.0, 3.0)),
doneAction: 2);
sig = SyncSaw.ar(
    [freq, freq*1.002], //Saw frequency
    SinOsc.ar(1/5, [0, 3.0.rand], mul: freq*2, add: freq*3),
    mul: 0.1);
sig = CombN.ar(sig, 0.3, 0.3, 4, 1); //Add echo
sig = sig*env;
Out.ar(0, sig*0.8)
}).play(s)
)

(
//run a task to play the synth
r = Task({
    {Synth("SyncSaw-Ex", [\freq, rrand(30, 80).midicps]);
    //Choose a wait time before next event
    rrand(2.0, 5.0).wait;
    }.loop;
}).play
)
```

// *Uplink*

```
(
{
LFPulse.ar(200, 0, 0.5, 0.4) //simple pulse
}.play(s)
)

(
{
var freq;
freq = LFPulse.kr(10, 0, 0.3, 2000, 200); //freq control
LFPulse.ar(freq, 0, 0.5, 0.4)
}.play(s)
)

(
{
var freq;
//add random values and additional control for add
freq = LFPulse.kr(rrand(10, 20), 0, rrand(0.1, 0.8),
       LFPulse.kr(1, 0, 0.5, 4000, 700));
LFPulse.ar(freq, 0, 0.5, 0.4)
}.play(s)
)
```

```
(
{
var freq;
//duplicate and add the two together
freq = LFPulse.kr(20.rand, 0, rrand(0.1, 0.8),
       LFPulse.kr(rrand(1.0, 5.0), 0,
              rrand(0.1, 0.8),
              8000.rand,
              2000.rand));
freq = freq + LFPulse.kr(20.rand, 0, rrand(0.1, 0.8),
       LFPulse.kr(rrand(1.0, 5.0), 0,
              rrand(0.1, 0.8),
              8000.rand,
              2000.rand));
LFPulse.ar(freq, 0.5, 0.1)

}.play(s)
)

(
{
var freq, out, env;
//add an envelope
env = EnvGen.kr(Env.linen(rrand(4.0, 7.0), 5.0, rrand(2.0, 5.0)));
freq = LFPulse.kr(20.rand, 0, rrand(0.1, 0.8),
       LFPulse.kr(rrand(1.0, 5.0), 0,
              rrand(0.1, 0.8),
              8000.rand,
              2000.rand));
freq = freq + LFPulse.kr(20.rand, 0, rrand(0.1, 0.8),
       LFPulse.kr(rrand(1.0, 5.0), 0,
              rrand(0.1, 0.8),
              8000.rand,
              2000.rand));
//pan and echo
out = Pan2.ar(LFPulse.ar(freq, 0.5, 0.1), 1.0.rand2);
2.do(out = AllpassN.ar(out,
                [rrand(0.1, 0.01), rrand(0.1, 0.01)]));
out*env
}.play(s)
)

(
//Send synth def to server with freq argument
SynthDef("Uplink-Ex",
{
var freq, out, env;
//add an envelope
env = EnvGen.kr(Env.linen(rrand(4.0, 7.0), 5.0, rrand(2.0, 5.0)), doneAction: 2);
freq = LFPulse.kr(20.rand, 0, rrand(0.1, 0.8),
       LFPulse.kr(rrand(1.0, 5.0), 0,
              rrand(0.1, 0.8),
```

```
            8000.rand,
            2000.rand));
freq = freq + LFPulse.kr(20.rand, 0, rrand(0.1, 0.8),
       LFPulse.kr(rrand(1.0, 5.0), 0,
            rrand(0.1, 0.8),
            8000.rand,
            2000.rand));
//pan and echo
out = Pan2.ar(LFPulse.ar(freq, 0.5, 0.1), 1.0.rand2);
2.do(out = AllpassN.ar(out,
                [rrand(0.1, 0.01), rrand(0.1, 0.01)]));
out*env;
Out.ar(0, out*0.8)
}).play(s)
)

(
//run a task to play the synth
r = Task({
   {Synth("Uplink-Ex");
   //Choose a wait time before next event
   rrand(4.0, 9.0).wait;
   }.loop;
}).play
)
```

// *Ring and Klank*

```
(
{
   Dust.ar(20, 0.2) //noise bursts
}.play(s)
)

(
{
var partials;
partials = 8;
Klank.ar( //fill klank with random partials and amplitudes
   `[Array.rand(partials, 100, 10000), nil,
     Array.rand(partials, 0.2, 0.9)],
   Dust.ar(20, 0.2))
}.play(s)
)

(
{ //ring element
SinOsc.ar(LFNoise2.kr(1.0, 200, 300), mul: 0.5)
}.play(s)
)
```

```
(
{
var partials, out, filter, bell;
partials = 8;
filter = SinOsc.ar(LFNoise2.kr(1.0, 200, 300), mul: 0.3);
bell = Klank.ar(
    `[Array.rand(partials, 100, 10000), nil,
      Array.rand(partials, 0.2, 0.9)],
    Dust.ar(20, 0.2))*filter; //ring klank with filter
bell
}.play(s)
)

(
{
var partials, out, filter, bell;
partials = 8;
filter = SinOsc.ar(
       LFNoise2.kr(rrand(0.7, 1.3),
           rrand(200, 400), //add random choices
           rrand(500, 1000)),
       mul: 0.2);
Mix.ar({ //insert inside Mix
    bell = Klank.ar(
        `[Array.rand(partials, 100, 10000), nil,
          Array.rand(partials, 0.2, 0.9)],
        Dust.ar(12, 0.2))*filter;
    bell = Pan2.ar(bell, LFNoise2.kr(2/3));
    bell}.dup(4))*0.4
}.play(s)
)
```

// *Tremulate*

```
{FSinOsc.ar(440)}.play //Sine oscillator

//Amp control begins with LFNoise2
{LFNoise2.ar(20, 0.9)}.scope(1)

//Max removes negative values (makes them 0)
{max(0, LFNoise2.ar(20, 0.9))}.scope(1)

(
{
var ampCont;
//Amp controlled by LFNoise
ampCont = max(0, LFNoise2.ar(20, 0.4));
FSinOsc.ar(440, mul: ampCont)}.play
)
```

```
(
{
var ampCont;
ampCont = max(0, LFNoise2.ar([20, 30], 0.1));
FSinOsc.ar([400, 500], mul: ampCont)}.play
)

(
{
var ampCont, rate, freq, chord;
rate = rrand(30, 70);
freq = 500;
chord = [1, 5/4, 3/2, 15/8];
ampCont = max(0, LFNoise2.ar([rate, rate, rate, rate], 0.1));
//create a bunch of these then mix them down
Mix.ar(FSinOsc.ar(freq*chord, mul: ampCont))}.play
)

(
({
var ampCont, rate, freq, chord;
rate = rrand(30, 70);
freq = rrand(300, 1000);
chord = [
    [1, 5/4, 3/2, 15/8],
    [1, 6/5, 3/2, 9/5],
    [1, 4/3, 3/2, 9/5],
    [1, 9/8, 3/2, 5/3]];
ampCont = max(0, LFNoise2.ar([rate, rate, rate, rate], 0.1));
//choose a chord
Mix.ar(FSinOsc.ar(freq*chord.choose, mul: ampCont))}).play
)

(
{ //Add pan and env
var ampCont, rate, freq, chord, env, panp, out;
rate = rrand(30, 70);
freq = rrand(300, 1000);
panp = 1.0.rand2;
env = EnvGen.kr(Env.linen(0.1, 2.0, 5.0));
chord = [
    [1, 5/4, 3/2, 15/8],
    [1, 6/5, 3/2, 9/5],
    [1, 4/3, 3/2, 9/5],
    [1, 9/8, 3/2, 5/3]];
ampCont = max(0, LFNoise2.ar([rate, rate, rate, rate], 0.1));
//choose a chord
out = Mix.ar(
    Pan2.ar(FSinOsc.ar(freq*chord.choose, mul: ampCont), panp)
);
out*env;
}.play
)
```

Harmonic Swimming and Tumbling

```
(
{
   FSinOsc.ar(500, mul: 0.3) //Sine oscillator
}.play(s)
)

(
{
   FSinOsc.ar(500,
      //amp control same as tremulate
      mul: max(0, LFNoise1.kr(rrand(6.0, 12.0), mul: 0.6)))
}.play(s)
)

(
{
   FSinOsc.ar(500,
      mul: max(0, LFNoise1.kr(rrand(6.0, 12.0), mul: 0.6,
                add: Line.kr(0, -0.2, 20)))) //slow fade
}.play(s)
)

(
{
var freq;
freq = 500;
   //two frequencies a fifth apart
   FSinOsc.ar(freq*[1, 3/2],
      mul: max(0, LFNoise1.kr(rrand([6.0, 6.0], 12.0), mul: 0.6,
                add: Line.kr(0, -0.2, 20))))
}.play(s)
)

(
{
var signal, partials, freq;
signal = 0;
partials = 8;
//Begin with low fundamental
freq = 50;

//duplicate and sum frequencies at harmonic intervals
partials.do({arg harm;
harm = harm + 1;
signal = signal +
   FSinOsc.ar(freq * [harm, harm*3/2],
      mul: max(0, LFNoise1.kr(rrand([6.0, 6.0], 12.0),
                mul: 1/(harm + 1) * 0.6,
                add: Line.kr(0, -0.2, 20))))
   });
signal
```

```
}.play(s)
)

(
SynthDef("Tumbling",
{arg freq = 50;
var signal, partials;
signal = 0;
partials = 8;
partials.do({arg harm;
harm = harm + 1;
signal = signal +
    FSinOsc.ar(freq * [harm, harm*3/2],
        mul: max(0, LFNoise1.kr(Rand([6.0, 6.0], 12.0), mul: 1/(harm + 1) * 0.6)
        ))
    });
signal = signal*EnvGen.kr(Env.perc(0.2,20.0), doneAction: 2);
Out.ar(0, signal*0.8)
}
).send(s)
)

(
//run a task to play the synth
r = Task({
    {Synth("Tumbling", [\freq, rrand(30, 80)]);
    //Choose a wait time before next event
    rrand(12.0, 20.0).wait;
    }.loop;
}).play
)
```

// *Police State*

```
(
{
//single siren
SinOsc.ar(
    SinOsc.kr(0.1, 0, 600, 1000),
    0,
    0.2)

}.play(s)
)

(
{
SinOsc.ar(//random frequencies and phase
    SinOsc.kr(Rand(0.1, 0.12),
        2pi.rand, Rand(200, 600), Rand(1000, 1300)),
    mul: 0.2)
```

```
}.play(s)
)

(
{
SinOsc.ar(
    SinOsc.kr(Rand(0.1, 0.12),
        6.0.rand, Rand(200, 600), Rand(1000, 1300)),
    //conrol scale
    mul: LFNoise2.ar(Rand(100, 120), 0.2))

}.play(s)
)

(
{
//pan and mix several
Mix.arFill(4, {
    Pan2.ar(
        SinOsc.ar(
            SinOsc.kr(Rand(0.1, 0.12),
                6.0.rand, Rand(200, 600), Rand(1000, 1300)),
            mul: LFNoise2.ar(Rand(100, 120), 0.1)),
        1.0.rand2)
})

}.play(s)
)

(
{
LFNoise2.ar(600, 0.1) //second component
}.play(s)
)

(
{
//ring modulate?
LFNoise2.ar(LFNoise2.kr(2/5, 100, 600), LFNoise2.kr(1/3, 0.1, 0.06))
}.play(s)
)

(
{
//stereo
LFNoise2.ar(LFNoise2.kr([2/5, 2/5], 100, 600), LFNoise2.kr([1/3, 1/3], 0.1, 0.06))
}.play(s)
)

(
{
//add the two and add echo
CombL.ar(
```

```
    Mix.arFill(4, {
        Pan2.ar(
            SinOsc.ar(
                SinOsc.kr(Rand(0.1, 0.12),
                    6.0.rand, Rand(200, 600), Rand(1000, 1300)),
                mul: LFNoise2.ar(Rand(100, 120), 0.1)),
            1.0.rand2)
    }) + LFNoise2.ar(
            LFNoise2.kr([2/5, 2/5], 90, 620),
            LFNoise2.kr([1/3, 1/3], 0.15, 0.18)),
    0.3, 0.3, 3)
}.play(s)
)
```

Latch or Sample and Hold

```
//Simple Oscillator
(
{
    SinOsc.ar(
        freq: 440,
        mul: 0.5
    );
}.play(s)
)

//Add a frequency control using a Saw
(
{
    SinOsc.ar(
        freq: LFSaw.ar(freq: 1, mul: 200, add: 600), //Saw controlled freq
        mul: 0.5
    );
}.play(s)
)

//Place the LFSaw inside a latch, add a trigger
(
{
    SinOsc.ar(
        freq: Latch.ar( //Using a latch to sample the LFSaw
            LFSaw.ar(1, 0, 200, 600),  //Input wave
            Impulse.ar(10) //Trigger (rate of sample)
        ),
        mul: 0.5
    );
}.play(s)
)

//SinOsc is replaced by Blip, try replacing
//the 1.1 with a MouseX
(
{
```

```
    Blip.ar( //Audio Ugen
        Latch.kr( //Freq control Ugen
            LFSaw.kr(1.1, 0, 500, 700), //Input for Latch
            Impulse.kr(10)), //Sample trigger rate
        3, //Number of harmonics in Blip
        mul: 0.3 //Volume of Blip
    )
}.play(s)
)

//Freq of the Saw is controlled by a Saw
(
{
    Blip.ar( //Audio Ugen
        Latch.kr( //Freq control Ugen
            LFSaw.kr( //input for Latch
                Line.kr(0.01, 10, 100), //Freq of input wave, was 1.1
                0, 300, 500), //Mul. and Add for input wave
            Impulse.kr(10)), //Sample trigger rate
        3, //Number of harmonics in Blip
        mul: 0.3 //Volume of Blip
    )
}.play(s)
)

//A variable is added for clarity.
(
{
    var signal;
    signal = Blip.ar( //Audio Ugen
        Latch.kr( //Freq control Ugen
            LFSaw.kr( 6.18, 0,//Freq of input wave (Golden Mean)
                300, 500), //Mul. and Add for input wave
            Impulse.kr(10)), //Sample trigger rate
        3, //Number of harmonics in Blip
        mul: 0.3 //Volume of Blip
    );
    //reverb
    2.do({ signal = AllpassN.ar(signal, 0.05, [0.05.rand, 0.05.rand], 4) });
    signal //return the variable signal
}.play(s)
)


//Add a Pan2
(
{
    var signal;
    signal = Blip.ar( //Audio Ugen
        Latch.kr( //Freq control Ugen
            LFSaw.kr( 6.18, 0,//Freq of input wave
                300, 500), //Mul. and Add for input wave
            Impulse.kr(10)), //Sample trigger rate
```

```
        3, //Number of harmonics in Blip
        mul: 0.3 //Volume of Blip
    );
    signal = Pan2.ar(
        signal, //input for the pan,
        LFNoise1.kr(1) //Pan position. -1 and 1, of 1 time per second
    );
    //reverb
    4.do({ signal = AllpassN.ar(signal, 0.05, [0.05.rand, 0.05.rand], 4,
        mul: 0.3, add: signal) });
    signal //return the variable signal
}.play(s)
)


//Control the number of harmonics
(
{
    var signal;
    signal = Blip.ar( //Audio Ugen
        Latch.kr( //Freq control Ugen
            LFSaw.kr( 6.18, 0, //Freq of input wave
                300, 500), //Mul. and Add for input wave
            Impulse.kr(10)), //Sample trigger rate
        LFNoise1.kr(0.3, 13, 14), //Number of harmonics in Blip
        mul: 0.3 //Volume of Blip
    );
    signal = Pan2.ar(
        signal, //input for the pan
        LFNoise1.kr(1) //Pan position.
    );
    //reverb
    4.do({ signal = AllpassN.ar(signal, 0.05, [0.05.rand, 0.05.rand], 4,
        mul: 0.3, add: signal) });
    signal //return the variable signal
}.play(s)
)


//Add an envelope
(
{
    var signal, env1;
    env1 = Env.perc(
            0.001, //attack of envelope
            2.0 //decay of envelope
        );
    signal = Blip.ar( //Audio Ugen
        Latch.kr( //Freq control Ugen
            LFSaw.kr( 6.18, 0,//Freq of input wave
                300, 500), //Mul. and Add for input wave
            Impulse.kr(10)), //Sample trigger rate
        LFNoise1.kr(0.3, 13, 14), //Number of harmonics in Blip
```

```
       mul: 0.3 //Volume of Blip
    );
    signal = Pan2.ar(
       signal, //input for the pan
       LFNoise1.kr(1) //Pan position.
    );
    //reverb
    4.do({ signal = AllpassN.ar(signal, 0.05, [0.05.rand, 0.05.rand], 4,
       mul: 0.3, add: signal) });
    signal*EnvGen.kr(env1) //return the variable signal
}.play(s)
)

//Place it in a Pbind
(
SynthDef("S_H",
{

    var signal, env1;
    env1 = Env.perc(
           0.001, //attack of envelope
           2.0 //decay of envelope
       );
    signal = Blip.ar( //Audio Ugen
       Latch.kr( //Freq control Ugen
          LFSaw.kr( Rand(6.0, 7.0), 0,//Freq of input wave
             Rand(300, 600), Rand(650, 800)), //Mul. and Add for input wave
          Impulse.kr(Rand(10, 12))), //Sample trigger rate
       LFNoise1.kr(0.3, 13, 14), //Number of harmonics in Blip
       mul: 0.3 //Volume of Blip
    );
    signal = Pan2.ar(
       signal, //input for the pan
       LFNoise1.kr(1) //Pan position.
    );
    //reverb
    4.do({ signal = AllpassN.ar(signal, 0.05, [0.05.rand, 0.05.rand], 4,
       mul: 0.3, add: signal) });
    signal = signal*EnvGen.kr(env1, doneAction:2); //return the variable signal

Out.ar(0, signal*0.9)

}).load(s);

SynthDescLib.global.read;

e = Pbind(
    \server, Server.internal,
    \dur, 0.3,
    \instrument, "S_H"
).play;
)
```

```
e.mute;
e.reset;
e.pause;
e.play;
e.stop;

//Add random values for each event

e = Pbind(
   \server, Server.internal,
   \dur, Prand([0, 0.1, 0.25, 0.5, 0.75, 1], inf),
   \instrument, "S_H"
).play;

e.stop;
```

// *Pulse*

```
(
{
   var out;
   out = Pulse.ar(
      200, //Frequency.
      0.5, //Pulse width. Change with MouseX
      0.5
   );
   out
}.play(s)
)


//Add a control for frequency
(
{
   var out;
   out = Pulse.ar(
      LFNoise1.kr(
         0.1, //Freq of LFNoise change
         mul: 20, //mul = (-20, to 20)
         add: 60 //add = (40, 80)
      ),
      0.5, 0.5);
   out
}.play(s)
)

//Control pulse
(
{
   var out;
   out = Pulse.ar(
```

```
       LFNoise1.kr(0.1, 20, 60),
       SinOsc.kr(
          0.2, //Freq of SinOsc control
          mul: 0.45,
          add: 0.46
       ),
        0.5);
   out
}.play(s)
)

//Expand to Stereo
(
{
   var out;
   out = Pulse.ar(
      LFNoise1.kr([0.1, 0.15], 20, 60),
      SinOsc.kr( 0.2,  mul: 0.45, add: 0.46),
       0.5);
   out
}.play(s)
)

//Add reverb
(
{
   var out;
   out = Pulse.ar(LFNoise1.kr([0.1, 0.12],  20, 60),
               SinOsc.kr( 0.2, mul: 0.45, add: 0.46),0.5);
   4.do({out = AllpassN.ar(out, 0.05, [0.05.rand, 0.05.rand], 4,
      mul: 0.4, add: out)});
   out
}.play(s)
)

//Smaller pulse widths
(
{
   var out;
   out = Pulse.ar(LFNoise1.kr([0.1, 0.12], 20, 60),
            SinOsc.kr( 0.2, mul: 0.05, add: 0.051),0.5);
   4.do({out = AllpassN.ar(out, 0.05, [0.05.rand, 0.05.rand], 4,
      mul: 0.4, add: out)});
   out
}.play(s)
)

//Add an envelope
(
{
   var out, env;
   env = Env.linen([0.0001, 1.0].choose, 2.0.rand, [0.0001, 1.0].choose);
   out = Pulse.ar(LFNoise1.kr([0.1, 0.12], 20, 60),
```

```
            SinOsc.kr( 0.2, mul: 0.05, add: 0.051),0.5);
   4.do({out = AllpassN.ar(out, 0.05, [0.05.rand, 0.05.rand], 4,
      mul: 0.4, add: out)});
   out*EnvGen.kr(env)
}.play(s)
)


//Define an instrument
(
SynthDef("Pulse1",
{arg att = 0.4, decay = 0.4;
   var out, env;
   env = Env.linen(att, Rand(0.1, 2.0), decay);
   out = Pulse.ar(LFNoise1.kr([0.1, 0.12], 20, 60),
            SinOsc.kr( 0.2, mul: 0.05, add: 0.051),0.5);
   4.do({out = AllpassN.ar(out, 0.05, [Rand(0.01, 0.05), Rand(0.01, 0.05)], 4,
      mul: 0.4, add: out)});
   out = out*EnvGen.kr(env, doneAction:2);
   Out.ar(0, out*0.4);
}).load(s);

SynthDescLib.global.read;

e = Pbind(
   \server, Server.internal,
   \dur, 3,
   \instrument, "Pulse1"
).play;
)

e.stop;

//Add another instrument and random values
(

e = Pbind(
   \server, Server.internal,
   \att, Pfunc({rrand(2.0, 5.0)}),
   \decay, Pfunc({rrand(4.0, 6.0)}),
   \dur, Prand([0, 1.0, 2.0, 2.5, 5], inf),
   \instrument, Prand(["S_H", "Pulse1"], inf)
).play;
)

e.stop;

//Add more structure, more instruments, nest Pseq, Prand, Pfunc, etc.
```

// *FM*

```
//Begin with LFO control
```

```
(
{
    var out;
    out = SinOsc.ar(
        SinOsc.ar( //control Osc
            5, //freq of control
            mul: 10, //amp of contrul
            add: 800), //add of control
        mul: 0.3 //amp of audio SinOsc
    );
    out
}.play(s)
)

//Add a control to move into audio range. The MouseX represents
//the control frequency, the add is the carrier. Mul is the index.
(
{
    var out;
    out = SinOsc.ar(
        SinOsc.ar( //control Osc
            MouseX.kr(5, 240), //freq of control
            mul: 10, //amp of contrul
            add: 800), //add of control
        mul: 0.3 //amp of audio SinOsc
    );
    out
}.play(s)
)


//Control of amp, or index.
(
{
    var out;
    out = SinOsc.ar(
        SinOsc.ar( //control Osc
            131, //freq of control
            mul: MouseX.kr(10, 700), //amp of contrul
            add: 800), //add of control
        mul: 0.3 //amp of audio SinOsc
    );
    out
}.play(s)
)

//Both
(
{
    var out;
    out = SinOsc.ar(
        SinOsc.ar( //control Osc
            MouseY.kr(10, 230), //freq of control
```

```
            mul: MouseX.kr(10, 700), //amp of contrul
            add: 800), //add of control
        mul: 0.3 //amp of audio SinOsc
    );
    out
}.play(s)
)


//Add must be higher than mul, so a variable is added to
//make sure it changes in relation to mul.
(
{
    var out, mulControl;
    mulControl = MouseX.kr(10, 700);
    out = SinOsc.ar(
        SinOsc.ar( //control Osc
            MouseY.kr(10, 230), //freq of control
            mul: mulControl, //amp of control
            add: mulControl + 100), //add will be 100 greater than mulControl
        mul: 0.3 //amp of audio SinOsc
    );
    out
}.play(s)
)


//Replace Mouse with LFNoise control
(
{
    var out, mulControl;
    mulControl = LFNoise1.kr(0.2, 300, 600); //store control in variable
    out = SinOsc.ar(
        SinOsc.ar( //control Osc
            LFNoise1.kr(0.4, 120, 130), //freq of control
            mul: mulControl, //amp of contrul
            add: mulControl + 100), //add will be 100 greater than mulControl
        mul: 0.3 //amp of audio SinOsc
    );
    out
}.play(s)
)

//Another control
(
{
    var out, mulControl;
    mulControl = LFNoise1.kr(0.2, 300, 600);
    out = SinOsc.ar(
        SinOsc.ar( //control Osc
            LFNoise1.kr(0.4, 120, 130), //freq of control
            mul: mulControl, //amp of contrul
            add: mulControl + LFNoise1.kr(0.1, 500, 600)), //add of control
```

```
      mul: 0.3 //amp of audio SinOsc
   );
   out
}.play
)

//Multichannel expansion
(
{
   var out, mulControl;
   mulControl = LFNoise1.kr([0.2, 0.5], 300, 600);
   out = SinOsc.ar(
      SinOsc.ar( //control Osc
         LFNoise1.kr(0.4, 120, 130), //freq of control
         mul: mulControl, //amp of contrul
         add: mulControl + LFNoise1.kr(0.1, 500, 600)), //add of control
      mul: 0.3 //amp of audio SinOsc
   );
   out
}.play
)


//Reverb and envelope
(
{
   var out, mulControl, env, effectEnv;
//  effectEnv = Env.perc(0.001, 3);
   env = Env.linen(0.01.rand, 0.3.rand, rrand(0.1, 3.0));
   mulControl = LFNoise1.kr([0.2, 0.5], 300, 600);
   out = SinOsc.ar(
      SinOsc.ar( //control Osc
         LFNoise1.kr(0.4, 120, 130), //freq of control
         mul: mulControl, //amp of contrul
         add: mulControl + LFNoise1.kr(0.1, 500, 600)), //add of control
      mul: 0.3 //amp of audio SinOsc
   );
   out*EnvGen.kr(env, doneAction:2);
}.play
)

(
SynthDef("FMinst",
{
   var out, mulControl, env, effectEnv;
   env = Env.linen(Rand(0.01, 1.0), Rand(0.03, 0.09), Rand(0.01, 1.0));
   mulControl = LFNoise1.kr([0.2, 0.5], 300, 600);
   out = SinOsc.ar(
      SinOsc.ar( //control Osc
         LFNoise1.kr(0.4, 120, 130), //freq of control
         mul: mulControl, //amp of contrul
         add: mulControl + LFNoise1.kr(0.1, 500, 600)), //add of control
      mul: 0.3 //amp of audio SinOsc
```

```
    );
    out = out*EnvGen.kr(env, doneAction:2);
    Out.ar(0, out)
}).load(s)
)
SynthDescLib.global.read;

//Note that this is not a very interesting composition. But you get the idea. Also
be aware that there
//are probably more efficient ways to do these using busses. For now I'm just
trying to get them
//to work.

(

e = Pbind(
    \server, Server.internal,
    \att, Pfunc({rrand(2.0, 5.0)}),
    \decay, Pfunc({rrand(4.0, 6.0)}),
    \dur, Prand([0, 1.0, 2.0, 2.5, 5], inf),
    \instrument, Prand(["S_H", "Pulse1", "FMinst"], inf)
).play;
)

e.stop;
```

// *Filter*

```
//Saw and filter
(
{
    RLPF.ar( //resonant low pass filter
        Saw.ar(100, 0.2), //input wave at 100 Hz
        MouseX.kr(100, 10000) //cutoff frequency
    )}.play
)

//Control with SinOsc
(
{
    RLPF.ar(
        Saw.ar(100, 0.2),
        SinOsc.ar(0.2, 0, 900, 1100)
    )
}.play
)

//Control resonance
(
{
    RLPF.ar(
        Saw.ar(100, 0.2),
```

```
        SinOsc.kr(0.2, 0, 900, 1100),
        MouseX.kr(1.0, 0.001) //resonance, or "Q"
    )}.play(s)
)

//Two controls
(
{
    RLPF.ar(
        Saw.ar(//input wave
            LFNoise1.kr(0.3, 50, 100),//freq of input
            0.1
        ),
        LFNoise1.kr(0.1, 4000, 4400), //cutoff freq
        0.04 //resonance
    )}.play(s)
)


//Add a pulse
(

{
var freq;
freq = LFNoise1.kr(0.3, 50, 100);
    RLPF.ar(
        Pulse.ar( //input wave
            freq,//freq of input
            0.1, //pulse width
            0.1 //add, or volume of pulse
        ),
        LFNoise1.kr(0.1, 4000, 4400), //cutoff freq
        0.04 //resonance
    )}.play(s)
)
```

// *Wind and Metal*

```
{LFNoise1.ar}.scope // random wave

{max(0, LFNoise1.ar)}.scope // random wave with max

{min(0, LFNoise1.ar)}.scope // random wave with min

{PinkNoise.ar(max(0, LFNoise1.ar(10)))}.scope // used as amp control

{PinkNoise.ar(max(0, LFNoise1.ar(1)))}.play // let this one run a while

{PinkNoise.ar * max(0, LFNoise1.ar([10, 1]))}.play //expanded to two channels

{PinkNoise.ar * max(0, LFNoise1.ar([10, 10]))}.play
```

```
// Scale and offest controls how often LFNoise moves to positive values
// Use the mouse to experiment:

{max(0, LFNoise1.ar(100, 0.75, MouseX.kr(-0.5, 0.5)))}.scope(zoom: 10)

(
{
PinkNoise.ar *
max(0, LFNoise1.ar([10, 10], 0.75, 0.25))
}.play
)

//Klank with one frequency.

{Klank.ar(`[[500], [1], [1]], PinkNoise.ar(0.05))}.play

//An array of freqs

{Klank.ar(`[[100, 200, 300, 400, 500, 600, 700, 800]], PinkNoise.ar(0.01))}.play

//Add amplitudes. Try each of these and notice the difference.

(
{Klank.ar(`[
    [100, 200, 300, 400, 500, 600, 700, 800], //freq
    [0.1, 0.54, 0.2, 0.9, 0.76, 0.3, 0.5, 0.1] //amp
    ], PinkNoise.ar(0.01))}.play
)

(
{Klank.ar(`[
    [100, 200, 300, 400, 500, 600, 700, 800], //freq
    [0.54, 0.2, 0.9, 0.76, 0.3, 0.5, 0.1, 0.3] //amp
    ], PinkNoise.ar(0.01))}.play
)

(
{Klank.ar(`[
    [100, 200, 300, 400, 500, 600, 700, 800], //freq
    [0.9, 0.76, 0.3, 0.5, 0.1, 0.3, 0.6, 0.2] //amp
    ], PinkNoise.ar(0.01))}.play
)

//Using enharmonic frequencies.

{Klank.ar(`[[111, 167, 367, 492, 543, 657, 782, 899]], PinkNoise.ar(0.01))}.play

//Use Array.fill to fill an array with exponential values. (biased toward 100)

Array.fill(20, {exprand(100, 1000).round(0.1)})

//compare with (even distribution)
```

```
Array.fill(20, {rrand(100.0, 1000).round(0.1)})

//Added to the patch. Run this several times. The postln will print
//the freq array.

(
{Klank.ar(
    `[Array.fill(10, {exprand(100, 1000)}).round(0.1).postln],
    PinkNoise.ar(0.01))}.play
)

//Add LFNoise for amp control.

(
{Klank.ar(
    `[Array.fill(10, {exprand(100, 1000)}).round(0.1).postln],
    PinkNoise.ar(0.01) * max(0, LFNoise1.ar([10, 10], 0.75, 0.25)))}.play
)

//Same thing with variables.

(
{
var excitation, speed, filters, range;
range = {exprand(100, 1000)};
filters = 10;
excitation = PinkNoise.ar(0.01) * max(0, LFNoise1.ar([10, 10], 0.75, 0.25));

Klank.ar(`[Array.fill(filters, range).round(0.1).postln], excitation)}.play
)

//With ring times and amplitudes.

(
{
var excitation, speed, filters, range, freqBank, ampBank, ringBank;
range = {exprand(100, 1000)};
filters = 10;
excitation = PinkNoise.ar(0.01) * max(0, LFNoise1.ar([10, 10], 0.75, 0.25));
freqBank = Array.fill(filters, range).round(0.1).postln;
ampBank = Array.fill(filters, {rrand(0.1, 0.9)}).round(0.1).postln;
ringBank = Array.fill(filters, {rrand(1.0, 4.0)}).round(0.1).postln;
Klank.ar(`[freqBank, ampBank, ringBank], excitation)
}.play
)

//Finally, slow down the excitation:

(
{
var excitation, speed, filters, range, freqBank, ampBank, ringBank;
range = {exprand(100, 1000)};
filters = 10;
```

```
excitation = PinkNoise.ar(0.01) * max(0, LFNoise1.ar([0.1, 0.1], 0.75, 0.25));
freqBank = Array.fill(filters, range).round(0.1).postln;
ampBank = Array.fill(filters, {rrand(0.1, 0.9)}).round(0.1).postln;
ringBank = Array.fill(filters, {rrand(1.0, 4.0)}).round(0.1).postln;
Klank.ar(`[freqBank, ampBank, ringBank], excitation)
}.play
)
```

// *Sci-Fi Computer*

```
(
{
PMOsc.ar(
   MouseX.kr(700, 1300),
   MouseY.kr(700, 1300),
   3)
}.play
)

(
{
PMOsc.ar(
   MouseX.kr(700, 1300),
   LFNoise0.kr(10, 1000, 1000),
   MouseY.kr(0.1, 5.0),
   mul: 0.3)
}.play
)

(
{
PMOsc.ar(
   LFNoise1.kr(10, 1000, 1000),
   LFNoise0.kr(10, 1000, 1000),
   MouseY.kr(0.1, 5.0),
   mul: 0.3)
}.play
)

(
{
PMOsc.ar(
   LFNoise1.kr([10, 10], 1000, 1000),
   LFNoise0.kr([10, 10], 1000, 1000),
   MouseY.kr(0.1, 5.0),
   mul: 0.3)
}.play
)

(
{
```

```
PMOsc.ar(
   LFNoise1.kr(
      MouseX.kr([1, 1], 12),
      mul: 1000,
      add: 1000),
   LFNoise0.kr(
      MouseX.kr([1, 1], 12),
      mul: 1000,
      add: 1000),
   MouseY.kr(0.1, 5.0),
   mul: 0.3)
}.play
)

(
{
PMOsc.ar(
   LFNoise1.kr(
      MouseX.kr([1, 1], 12),
      mul: MouseY.kr(10, 1000),
      add: 1000),
   LFNoise0.kr(
      MouseX.kr([1, 1], 12),
      mul: MouseY.kr(30, 1000),
      add: 1000),
   MouseY.kr(0.1, 5.0),
   mul: 0.3)
}.play
)
```

// Harmonic Swimming

```
(
// harmonic swimming
play({
   var fundamental, partials, out, offset;
   fundamental = 50;      // fundamental frequency
   partials = 20;      // number of partials per channel
   out = 0.0;       // start of oscil daisy chain
   offset = Line.kr(0, -0.02, 60); // causes sound to separate and fade
   partials.do({ arg i;
      out = FSinOsc.ar(
            fundamental * (i+1),       // freq of partial
            0,
            max(0,           // clip negative amplitudes to zero
               LFNoise1.kr(
                  6 + [4.0.rand2, 4.0.rand2],      // amplitude rate
                  0.02,               // amplitude scale
                  offset              // amplitude offset
```

```
                )
            ),
            out
        )
    });
    out
})
)


(
{
var out = 0;
2.do({ |i|
    out = out + FSinOsc.ar(400 * (i + 1),
        mul: max(0, LFNoise1.kr(rrand(6.0, 10.0))))
});
out
}.play
)


(
{
var out = 0;
4.do({ |i|
    out = out + FSinOsc.ar(400 * (i + 1),
        mul: max(0,
        LFNoise1.kr(
            rrand(6.0, 10.0),
            0.2
        ))
    )
});
out
}.play
)

(
{
var out = 0;
20.do({ |i|
    out = out + FSinOsc.ar(400 * (i + 1),
        mul: max(0,
        LFNoise1.kr(
            rrand(6.0, 10.0),
            0.2
        ))
    )
});
out
}.play
)
```

```
(
{
var out = 0, fundamental = 50, partials = 20;
partials.do({ |i|
    out = out + FSinOsc.ar(fundamental * (i + 1),
        mul: max(0,
        LFNoise1.kr(
            rrand(6.0, 10.0),
            0.2
        ))
    )
});
out
}.play
)

(
{
var out = 0, fundamental = 50, partials = 20;
partials.do({ |i|
    out = out + FSinOsc.ar(fundamental * (i + 1),
        mul: max(0,
        LFNoise1.kr(
            rrand(6.0, 10.0),
            0.2,
            MouseX.kr(0, -0.2)
        ))
    )
});
out
}.play
)

(
{
var out = 0, fundamental = 50, partials = 20;
partials.do({ |i|
    out = out + FSinOsc.ar(fundamental * (i + 1),
        mul: max(0,
        LFNoise1.kr(
            rrand(6.0, [10.0, 10.0]),
            0.2,
            Line.kr(0, -0.2, 60)
        ))
    )
});
out
}.play
)
```

// Variable decay bell

```
{SinOsc.ar(400 * LFNoise1.kr(1/6, 0.4, 1))}.play


(
{
SinOsc.ar(
    400 * LFNoise1.kr(1/6, 0.4, 1),
    mul: EnvGen.kr(Env.perc(0, 0.5), Dust.kr(1))
)
}.play
)

// add formula so that low has long decay, high has short
(
{
SinOsc.ar(
    100 * LFNoise1.kr(1/6, 0.4, 1),
    mul: EnvGen.kr(
        Env.perc(0, (100**(-0.7))*100), Dust.kr(1))
)
}.play
)


(
{
SinOsc.ar(
    3000 * LFNoise1.kr(1/6, 0.4, 1),
    mul: EnvGen.kr(
        Env.perc(0, (3000**(-0.7))*100), Dust.kr(1))
)
}.play
)

(
{
Pan2.ar(
    SinOsc.ar(
        3000 * LFNoise1.kr(1/6, 0.4, 1),
        mul: EnvGen.kr(
            Env.perc(0, (3000**(-0.7))*100), Dust.kr(1))
    ), LFNoise1.kr(1/8)
)
}.play
)

(
{
Mix.fill(15,
```

```
{
var freq;
freq = exprand(100, 3000);
    Pan2.ar(
        SinOsc.ar(
            freq * LFNoise1.kr(1/6, 0.4, 1),
            mul: EnvGen.kr(
                Env.perc(0, (freq**(-0.7))*100), Dust.kr(1/5))
        ), LFNoise1.kr(1/8)
    )*0.2
})
}.play
)
```

// Gaggle of sine variation

```
{SinOsc.ar(400, mul: max(0, FSinOsc.kr(2)))}.play

{SinOsc.ar(400, mul: max(0, FSinOsc.kr([2, 4])))}.play

{SinOsc.ar([400, 800], mul: max(0, FSinOsc.kr([2, 3])))}.play

(
{Mix.ar(SinOsc.ar([400, 800, 1200],
    mul: max(0, FSinOsc.kr([1, 2, 3]))))*0.1}.play
)

(
{
var harmonics = 4, fund = 400;

Mix.fill(harmonics,
    {arg count;
        SinOsc.ar(fund * (count+1),
            mul: max(0, FSinOsc.kr(count))
            )
    }
)*0.1}.play
)

(
{
var harmonics = 4, fund = 400;

Mix.fill(harmonics,
    {arg count;
        SinOsc.ar(fund * (count+1),
            mul: max(0, FSinOsc.kr(count/5))
            )
    }
```

```
)*0.1}.play
)


(
{
var harmonics = 16, fund = 400;

Mix.fill(harmonics,
    {arg count;
        SinOsc.ar(fund * (count+1),
            mul: max(0, FSinOsc.kr(count/5))
            )
    }
)*0.1}.play
)

(
{
var harmonics = 16, fund = 50;

Mix.fill(harmonics,
    {arg count;
        Pan2.ar(
            SinOsc.ar(fund * (count+1),
                mul: max(0, FSinOsc.kr(count/5))
                ),
            1.0.rand2
        )
    }
)*0.07}.play
)
```

// *KSPluck*

// *More*

```
{max(0, LFNoise1.ar)}.scope // random wave with max
```

E. Pitch Chart, MIDI, Pitch Class, Frequency, Hex, Binary Converter:

Notes for the pitch chart:

PC-Pitch Class, MN-Midi Number, Int-Interval, MI-Midi Interval, ETR-Equal Tempered Ratio, ETF-ET Frequency, JR-Just Ratio, JC-Just Cents, JF-Just Frequency, PR-Pythagorean Ratio, PC-Pyth. Cents, PF-Pyth. Freq., MR-Mean Tone Ratio, MC-MT Cents, MF-MT Freq.

The shaded columns are chromatic pitches and correspond with the black keys of the piano. Italicized values are negative.

Some scales show intervals, ratios, and cents from middle C. The Pythagorean scale is mostly positive numbers showing intervals and ratios from C1. I did this to show the overtone series. All ratios with a 1 as the denominator are overtones. They are bold. To invert the ratios just invert the fraction; 3:2 becomes 2:3.

I restart the cents chart at 000 for C4 in the PC column because I just don't think numbers higher than that are very useful.

Also, here is a music number converter. The Hex and Binary refer to the MIDI number.

```
   31.20. Pitch class, MIDI number, Frequency, Hex, Binary conversion GUI
 (
Sheet({ arg l; var pcstring;
pcstring = ["C", "C#", "D", "Eb", "E", "F", "F#", "G", "Ab", "A", "Bb", "B"];
SCStaticText(l, l.layRight(70, 30)).string_("MIDI");
SCStaticText(l, l.layRight(70, 30)).string_("Pitch");
SCStaticText(l, l.layRight(70, 30)).string_("Frequency");
SCStaticText(l, l.layRight(70, 30)).string_("Hex");
SCStaticText(l, l.layRight(70, 30)).string_("Binary");
l.view.decorator.nextLine;
m = SCNumberBox(l,l.layRight(70,30)); p = SCTextField(l,l.layRight(70,30));
f = SCNumberBox(l,l.layRight(70,30)); h = SCTextField(l,l.layRight(70,30));
b = SCTextField(l, l.layRight(70, 30));
p.value = "C4"; f.value = 60.midicps.round(0.01);
m.value = 60; h.value = "0000003C"; b.value = "00111100";
m.action = {
   arg numb;  var array;
   numb.value.asInteger.asBinaryDigits.do({arg e, i; array = array ++
e.asString});
   p.value = pcstring.wrapAt(numb.value) ++ (numb.value/12 - 1).round(1).asString;
   f.value = numb.value.midicps.round(0.001);
   h.value = numb.value.asInteger.asHexString;
   b.value = array;
};
//p.defaultKeyDownAction = {arg a, b, c, d, e; [a, b, c, d, e].value.postln;};
}, "Conversions");
)
```

P	MN	Int	MI	ETR	ETF	JR	JR	JF	PR	PF	MR	MF
C2	36	*P15*	*24*	0.250	65.41	**1:4**	0.250	65.406	0.500	65.406	0.250	65.406
Db	37	*M14*	*23*	0.265	69.30	4:15	0.266	69.767	0.527	68.906	0.268	70.116
D	38	*m14*	*22*	0.281	73.42	5:18	0.278	72.674	0.562	73.582	0.280	73.255
Eb	39	*M13*	*21*	0.298	77.78	3:10	0.300	78.488	0.592	77.507	0.299	78.226
E	40	*m13*	*20*	0.315	82.41	5:16	0.312	81.758	0.633	82.772	0.313	81.889
F	41	*P12*	*19*	0.334	87.31	**1:3**	0.333	87.209	0.667	87.219	0.334	87.383
F#	42	*A11*	*18*	0.354	92.50	16:45	0.355	93.023	0.702	93.139	0.349	91.307
G	43	*P11*	*16*	0.375	98.00	3:8	0.375	98.110	0.750	98.110	0.374	97.848
Ab	44	*M10*	*16*	0.397	103.8	2:5	0.400	104.65	0.790	95.297	0.400	104.65
A	45	*m10*	*15*	0.421	110.0	5:12	0.416	109.01	0.844	110.37	0.418	109.36
Bb	46	*M9*	*14*	0.446	116.5	4:9	0.444	116.28	0.889	116.29	0.447	116.95
B	47	*m9*	*13*	0.472	123.5	15:32	0.469	122.64	0.950	124.17	0.467	122.18
C3	48	*P8*	*12*	0.500	130.8	**1:2**	0.500	130.81	1.000	130.81	0.500	130.81
Db	49	*M7*	*11*	0.530	138.6	8:15	0.533	139.53	1.053	137.81	0.535	139.97
D	50	*m7*	*10*	0.561	146.8	5:9	0.556	145.35	1.125	147.16	0.559	146.25
Eb	51	*M6*	*9*	0.595	155.6	3:5	0.600	156.98	1.185	155.05	0.598	156.45
E	52	*m6*	*8*	0.630	164.8	5:8	0.625	163.52	1.266	165.58	0.625	163.52
F	53	*P5*	*7*	0.668	174.6	2:3	0.667	174.42	1.333	174.41	0.669	175.03
F#	54	*A4*	*6*	0.707	185.0	32:45	0.711	186.05	1.424	186.24	0.699	182.88
G	55	*P4*	*5*	0.749	196.0	3:4	0.750	196.22	1.500	196.22	0.748	195.70
Ab	56	*M3*	*4*	0.794	207.7	4:5	0.800	209.30	1.580	190.56	0.800	209.30
A	57	*m3*	*3*	0.841	220.0	5:6	0.833	218.02	1.688	220.75	0.836	218.72
Bb	58	*M2*	*2*	0.891	233.1	8:9	0.889	232.56	1.778	232.55	0.895	234.16
B	59	*m2*	*1*	0.944	246.9	15:16	0.938	245.27	1.898	248.35	0.935	244.62
C4	60	P1	0	1.000	261.6	**1:1**	1.000	261.63	2.000	261.63	1.000	261.63
Db	61	m2	1	1.059	277.2	16:15	1.067	279.07	2.107	275.62	1.070	279.94
D	62	M2	2	1.122	293.7	9:8	1.125	294.33	2.250	294.33	1.118	292.50
Eb	63	m3	3	1.189	311.1	6:5	1.200	313.95	2.370	310.06	1.196	312.90
E	64	M3	4	1.260	329.6	5:4	1.250	327.03	2.531	331.12	1.250	327.03
F	65	P4	5	1.335	349.2	4:3	1.333	348.83	2.667	348.85	1.337	349.79
F#	66	A4	6	1.414	370.0	45:32	1.406	367.91	2.848	372.52	1.398	365.75
G	67	P5	7	1.498	392.0	3:2	1.500	392.44	3.000	392.44	1.496	391.39
Ab	68	m6	8	1.587	415.3	8:5	1.600	418.60	2.914	381.12	1.600	418.60
A	69	M6	9	1.682	440.0	5:3	1.667	436.04	3.375	441.49	1.672	437.44
Bb	70	m7	10	1.782	466.2	9:5	1.800	470.93	3.556	465.10	1.789	468.05
B	71	M7	11	1.888	493.9	15:8	1.875	490.55	3.797	496.70	1.869	488.98
C5	72	P8	12	2.000	523.3	**2:1**	2.000	523.25	4.000	523.25	2.000	523.25
Db	73	m9	13	2.118	554.4	32:15	2.133	558.14	4.214	551.25	2.140	559.88
D	74	M9	14	2.244	587.3	9:4	2.250	588.66	4.500	588.66	2.236	585.00
Eb	75	m10	15	2.378	622.3	12:5	2.400	627.90	4.741	620.15	2.392	625.81
E	76	M10	16	2.520	659.3	5:2	2.500	654.06	5.063	662.24	2.500	654.06
F	77	P11	17	2.670	698.5	8:3	2.667	697.67	5.333	697.66	2.674	699.59
F#	78	A11	18	2.828	740.0	45:16	2.813	735.82	5.695	745.01	2.796	731.51
G	79	P12	19	2.996	784.0	**3:1**	3.000	784.88	6.000	784.88	2.992	782.78
Ab	80	m13	20	3.174	830.6	16:5	3.200	837.20	5.827	762.28	3.200	837.20
A	81	M13	21	3.364	880.0	10:3	3.333	872.09	6.750	882.99	3.344	874.88
Bb	82	m14	22	3.564	932.3	18:5	3.600	941.85	7.111	930.21	3.578	936.10
B	83	M14	23	3.776	987.8	15:4	3.750	981.10	7.594	993.36	3.738	977.96
C6	84	P15	24	4.000	1047	**4:1**	4.000	1046.5	8.000	1046.5	4.000	1046.5
C#	61	A1	1	1.059	277.2	25:24	1.042	272.53	1.068	279.38	1.045	279.94
Gb	66	d5	6	1.414	370.0	64:45	1.422	372.09	1.405	367.50	**	**
G#	80	A5	8	3.174	830.6	25:16	1.563	408.79	1.602	419.07	**	**
A#	70	m7	10	1.782	466.2	45:16	1.758	459.89	**	**	1.869	488.98

Answers to Exercises

Answers.

Open this file in SC to run examples, since many of the answers are given as code or formulas to calculate the answer.

4.1. Of the wave examples below (each is exactly 1/100th of a second), which is the loudest? softest? highest pitch? brightest sound? is aperiodic? is periodic?

A, B, B, B (C?), D, ABC

4.2. What is the length of each of these waves: 10 Hz, 250 Hz, 440 Hz, 1000 Hz?

```
1000/[10, 250, 440, 1000]
```

4.3. What is the lowest pitch humans can hear? What is the highest? What is the lowest frequency humans can hear?

20 to 20 kHz, infinite

4.4. The Libby performance hall is about 200 feet deep. If you are at one end of the hall, how long would it take sound to get back to you after bouncing off of the far wall?

400/1000

5.2. Beginning at C4 (261.6) calculate the frequencies for F#4, G3, B4, A3, and E4, using lowest possible ratios (only M2, m3, M3, 4, 5). 44

```
261.6 * 9/8 * 9/8 * 9/8
261.6 * 3/2
261.6 * 3/2 * 5/4
261.6 * 3/2 * 9/8
261.6 * 5/4
```

5.3. Prove the Pythagorean comma (show your work). 44

```
A2, A3, A4, A5, A6, A7, A8, A9 = 7 octaves
A2, E3, B3, F#4, C#5, G#5, D#6, A#6, F7, C8, G8, D9, A9 = 11 fifths,

(110 * pow(3/2, 11) ) * pow(1/2, 7)

(110 * pow(2, 7)) - (110 * pow(3/2, 11))

110*3/2*3/2*3/2*3/2*3/2*3/2*3/2*3/2*3/2*3/2*3/2
110*2*2*2*2*2*2*2
```

5.4. Story problem! Two trains leave the station, . . . No, wait, two microphones are set up to record a piano. One is 10' away, the other is 11 feet away (a difference of one foot). What frequency will be cancelled out? 44

A wave with a length of 2': 500 Hz

1000/500 = 2

6.1. Calculate the sizes of a 10 minute recording in these formats: mono, 22k sampling rate, 8 bit; stereo, 88.2k, 24 bit; mono, 44.1k, 16 bit; and mono, 11k, 8 bit. 52

10 minutes at 44.1, stereo, 16 bit = 100M

12.5, 300M, 50M, 6.25

8.1. Identify all objects, functions, messages, arrays, and argument lists. {RLPF.ar(LFSaw.ar([8, 12], 0, 0.2), LFNoise1.ar([2, 3].choose, 1500, 1600), 0.05, 0.4)}.play 69

Objects are RLPF, LFSaw, all numbers, LFNoise1, and the Function. The function is everything enclosed in { and }. The messages are .ar, .choose and .play. The arrays are [8, 12]. The keywords are mul. The argument list for RLPF is LFSaw.ar([8, 12], 0, 0.2),   LFNoise1.ar([2, 3].choose, 1500, 1600),   0.05,   mul: 0.4. The argument list for LFSaw is [8, 12], 0, 0.2. The argument list for LFNoise1 is [2, 3].choose, 1500, 1600.

8.2. Identify the two errors in this example. {SinOsc.ar(LFNoise0.ar([10, 15], 400 800), 0, 0.3}.play 69

No comma after 400, missing ")"

8.3. Explain what each of the numbers in the argument list mean (using help). MIDIOut.noteOn(0, 60, 100) 69

channel, note, velocity

8.4. Modify the CombN code below so that the contents of each enclosure are indented successive levels. For example, SinOsc.ar(rrand(100, 200)) would become: SinOsc.ar(rrand(100,   200))   CombN.ar(WhiteNoise.ar(0.01), [1, 2, 3, 4], XLine.kr(0.0001, 0.01, 20), -0.2) 69

CombN.ar(in: WhiteNoise.ar(mul: 0.01), maxdelaytime: 0.01, delaytime: XLine.kr(start: 0.0001, end: 0.01, duration: 20), decaytime: -0.2)

8.5. In the above example, which Ugens are nested? 69

WhiteNoise.ar, XLine.kr

8.6. Which of these are not legal variable names? lfNoise0ar, 6out, array6, InternalBus, next pitch, midi001, midi.01, attOne 69

6out (begins with number), InternalBus (first letter cap), next pitch (not contiguous), midi.01 (period)

9.3. Write a small patch with a SinOsc with a frequency of 670, a phase of 1pi, and an amplitude of 0.5 77

```
{SinOsc.ar(670, 1pi, 0.5)}.play
```

11.1. Rewrite this patch replacing all numbers with variables. {SinOsc.ar(freq: SinOsc.ar(512, mul: 673, add: LFNoise0.kr(7.5, 768, 600)), mul: 0.6)}.play 89

```
var modFreq, carrierFreq, indexFreq, indexScale, indexOffset, amp, modulator,
indexControl, signal;
modFreq = 512;
carrierFreq = 673;
indexFreq = 7.5;
indexScale = 768;
indexOffset = 600;
amp = 0.6;
indexControl = LFNoise0.kr(indexFreq, indexScale, indexOffset);
modulator = SinOsc.ar(indexControl);
signal = SinOsc.ar(modulator);
{signal}.play;
```

11.2. Rewrite the patch above with variables so that the scale of the SinOsc (673) is twice the frequency (512), and the offset of LFNoise0 (600) is 200 greater than the scale (768). 89

```
var modFreq, carrierFreq, indexFreq, indexScale, indexOffset, amp, modulator,
indexControl, signal;
modFreq = 512;
carrierFreq = modFreq*2;
indexFreq = 7.5;
indexScale = 768;
indexOffset = 100 + indexScale;
amp = 0.6;
indexControl = LFNoise0.kr(indexFreq, indexScale, indexOffset);
modulator = SinOsc.ar(indexControl);
signal = SinOsc.ar(modulator);
{signal}.play;
```

11.3. How would you offset and scale an LFNoise0 so that it returned values between the range of C2 and C6 (C4 is 60). 89

four octaves above C2 is 36 to 84

scale: (84 - 36)/2, offset: 36 + (84 - 36)/2

11.4. An envelope is generating values between 400 and 1000. What is the offset and scale? 89

Range of 600. Envelopes are default to 0 and 1. Offset = 400, scale = 600.

12.1. In this patch, what is the speed of the vibrato? {SinOsc.ar(400 + SinOsc.ar(7, mul: 5))}.play 104

7 Hz

12.2. Rewrite the patch above with a MouseX and MouseY to control the depth and rate of vibrato. 104

```
{SinOsc.ar(400 + SinOsc.ar(MouseX.kr(2, 7), mul: MouseY.kr(2, 8)))}.play
```

12.3. Rewrite the patch above using an LFPulse rather than a SinOsc to control frequency deviation. 104

```
{SinOsc.ar(400 + LFPulse.ar(MouseX.kr(2, 7), mul: MouseY.kr(2, 8)))}.play
```

12.4. Rewrite the Theremin patch so that vibrato increases with amplitude. 104

```
({
var amp;
amp = MouseX.kr(0.02, 1);
   SinOsc.ar(
      freq: MouseY.kr([3200, 1600], [200, 100], lagTime: 0.5, warp: 1) *
      (1 + SinOsc.kr(amp*6, mul: 0.02)), //Vibrato
      mul: abs(amp) //Amplitude
   )
}.play)
```

12.5. Write a patch using a Pulse and control the width with a Line moving from 0.1 to 0.9 in 20 seconds. 104

```
{Pulse.ar(50, Line.kr(0.1, 0.9, 20))}.play
```

12.6. Write a patch using a SinOsc and EnvGen. Use the envelope generator to control frequency of the SinOsc and trigger it with Dust. 104

```
{SinOsc.ar(300 + EnvGen.kr(Env.linen(0.1, 0.1, 0.1), gate:
   Dust.kr(2), levelScale: 1000))}.play
```

12.7. Start with this patch: {SinOsc.ar(200)}.play. Replace the 200 with another SinOsc offset and scaled so that it controls pitch, moving between 400 and 800 3 times per second. Then insert another SinOsc to control the freq of that sine so that it moves between 10 and 20 once every 4 seconds. Then another to control the frequency of that sine so that it moves between once every 4 seconds (a frequency of 1/4) to 1 every 30 seconds (a frequency of 1/30), once every minute (a frequency of 1/60). 104

Wow. This one was tough.

```
{
var lastRange;
lastRange = (1/4) - (1/30)
SinOsc.ar(
   SinOsc.ar(
      SinOsc.ar(
         SinOsc.ar(1/60, mul: lastRange/2, add: 1/30 + (lastRange/2))),
         mul: 5, add: 15),
      mul: 200, add: 600)
```

```
   )
}.play
```

12.8. What is the duration of a single wave with a frequency of 500 Hz? 104

1/500 (0.002)

12.9. What is the frequency, in Hz (times per second), of two and a half minutes? 104

1/150 or 0.00667

13.1. What is the 6th harmonic of a pitch at 200 Hz? 113

1200

13.2. What is the phase of the sine wave in channel 4? SinOsc.ar([10, 15, 20, 25], [0, 1.25], 500, 600) 113

1.25 (the second array wraps for the first array: [0, 1.25, 0, 1.25]

13.3. What value is returned from this function? {a = 10; b = 15; c = 25; a = c + b; b = b + a; b} 113

Just run it:

```
{a = 10; b = 15; c = 25; a = c + b; b = b + a; b}.value
```

13.4. Arrange these frequencies from most consonant to most dissonant. 180:160, 450:320, 600:400, 1200:600 113

1200:600 = 2/1, 600:400 = 3/2, 180:160 = 9/8, 450:320 = 45/32

14.1. Using any of the additive patches above as a model, create a harmonic collection of only odd harmonics, with decreasing amplitudes for each partial. What is the resulting wave shape? 134

```
{
   Mix.fill(20, {arg i;
      SinOsc.ar(500*(i*2+1).postln, mul: 1/(i+1))
   })*0.5
}.scope
```

Pulse

14.2. Begin with the "additive saw with modulation" patch above. Replace all the LFNoise1 ugens with any periodic wave (e.g. SinOsc), offset and scaled to control amplitude, with synchronized frequencies (3/10, 5/10, 7/10). 134

```
(
{
var speed = 14;
```

```
f = 300;
t = Impulse.kr(1/3);
Mix.fill(12, {arg i;
   SinOsc.ar(f*(i+1), mul: SinOsc.ar((i+1)/1, 0.5, 0.5)/(i+1))})*0.5
}.scope(1)
)
```

14.3. Modify the "additive saw with independent envelopes" patch so that all the decays are 0, but the attacks are different (e.g. between 0.2 and 2 seconds). Use a random function if you'd like. 134

```
(
{
f = 300;
t = Impulse.kr(1/3);
Mix.fill(12, {arg i;
   SinOsc.ar(f*(i+1), mul: EnvGen.kr(Env.perc(rrand(0.2, 2.0), 0), t)/(i+1))
})*0.5
}.scope(1)
)
```

14.4. Use Array.fill to construct an array with 20 random values chosen from successive ranges of 100, such that the first value is between 0 and 100, the next 100 and 200, the next 200 and 300, etc. The result should be something like [67, 182, 267, 344, 463, 511, etc.]. 134

```
Array.fill(20, {arg i; rrand(i*100, i+1*100)})
```

14.5. In the patch below, replace 400 and 1/3 with arrays of frequencies, and 0.3 with an array of pan positions. You can either write them out, or use {rrand(?, ?)}.dup(?). 134

```
(
{Mix.ar(
   Pan2.ar(
      SinOsc.ar({rrand(100, 1000)}.dup(8), // freq
      mul: EnvGen.kr(Env.perc(0, 2),
      Dust.kr({rrand(5, 1/9)}.dup(8)) // trigger density
       )*0.1),
       {1.0.rand}.dup(8) // pan position
)
)}.play )
```

14.6. Devise a method of ordering numbers and/or letters that would appear random to someone else, but is not random to you. 134

3, 9, 12, 36, 39, 17, 20, 60, 63, 89, 92, 76, 79, 37, 40, 20, 23, 69, 72, 16, 19, 57, 60, 80, 83, 49, 52, 56, 59, 77, 80, 40, 43, 29, 32, 96, 99, 97, 0, 0, 20

```
i = 0; 20.do({i = (i + 3)%100; i.postln; i = (i*3)%100; i.postln;})
```

15.1. Write a patch using an RLPF with a 50 Hz Saw as its signal source. Insert an LFNoise0 with frequency of 12 to control the frequency cutoff of the filter with a range of 200, 2000. 148

```
{RLPF.ar(Saw.ar(50), LFNoise0.kr(12, 900, 1100), 0.05)*0.2}.play
```

15.2. Write a patch similar to "tuned chimes" using Klank with PinkNoise as its input source. Fill the frequency array with harmonic partials but randomly detuned by + or – 2%. In other words, rather than 100, 200, 300, etc., 105, 197, 311, 401, etc. (Multiply each one by rrand(0.98, 1.02)). 148

```
{
var fund = 200;
Mix.fill( 6, {
var harm;
harm = rrand(5, 20);
    Pan2.ar(
        Klank.ar(
            `[Array.fill(harm, {arg i; (i + 1)*rrand(0.8, 1.2)*fund}),
            1, Array.fill(harm, {rrand(1.0, 3.0)})],
                PinkNoise.ar*Decay.ar(Dust.ar(1))
        )*0.01,
        1.0.rand2)
})
}.play
```

15.3. Without deleting any existing code, rewrite the patch below with "safe" values to replace the existing trigger with a 1 and the fund with 100. 148

```
({var trigger, fund;
trigger = Dust.kr(3/7); fund = rrand(100, 400); trigger = 1; fund = 400;
Mix.ar(
    Array.fill(16,
    {arg counter; var partial;
    partial = counter + 1;
    Pan2.ar(
        SinOsc.ar(fund*partial) *
        EnvGen.kr(Env.adsr(0, 0, 1.0, 5.0),
            trigger, 1/partial
        ) * max(0, LFNoise1.kr(rrand(5.0, 12.0))),
        1.0.rand2)
    }))*0.5}.play)
```

15.4. In the patch above monitor the frequency arguments only of all the SinOscs, LFNoise1s, and the pan positions by only adding postlns. 148

```
({var trigger, fund;
trigger = Dust.kr(3/7); fund = rrand(100, 400); trigger = 1; fund = 400;
Mix.ar(
    Array.fill(16,
    {arg counter; var partial;
    partial = counter + 1;
    Pan2.ar(
        SinOsc.ar((fund*partial).postln) *
        EnvGen.kr(Env.adsr(0, 0, 1.0, 5.0),
            trigger, 1/partial
        ) * max(0, LFNoise1.kr(rrand(5.0, 12.0))),
        1.0.rand2)
    }))*0.5}.play)
```

15.5. In the patch above comment out the EnvGen with all its arguments, but not the max(). Be careful not to get two multiplication signes in a row (* *), which means square. Check using syntax colorize under the format menu. 148

```
({var trigger, fund;
trigger = Dust.kr(3/7); fund = rrand(100, 400); trigger = 1; fund = 400;
Mix.ar(
    Array.fill(16,
    {arg counter; var partial;
    partial = counter + 1;
    Pan2.ar(
        SinOsc.ar((fund*partial).postln) *
        /* EnvGen.kr(Env.adsr(0, 0, 1.0, 5.0),
            trigger, 1/partial
        ) * */ max(0, LFNoise1.kr(rrand(5.0, 12.0))),
        1.0.rand2)
    }))*0.5}.play)
```

15.6. Without deleting, replacing any existing code, or commenting out, replace the entire Mix portion with a "safe" SinOsc.ar(400). 148

```
({var trigger, fund;
trigger = Dust.kr(3/7); fund = rrand(100, 400); trigger = 1; fund = 400;
Mix.ar(
    Array.fill(16,
    {arg counter; var partial;
    partial = counter + 1;
    Pan2.ar(
        SinOsc.ar((fund*partial).postln) *
        EnvGen.kr(Env.adsr(0, 0, 1.0, 5.0),
            trigger, 1/partial
        ) * max(0, LFNoise1.kr(rrand(5.0, 12.0))),
        1.0.rand2)
    }))*0.5; SinOsc.ar(400)}.play)
```

16.1. In a KS patch, what delay time would you use to produce these frequencies: 660, 271, 1000, 30 Hz? 161

Easy: 1/660, 1/271, 1/1000, 1/30

16.2. Rewrite three patches from previous chapters as synth definitions written to the hard disk with arguments replacing some key variables. Then write several Synth() lines to launch each instrument with different arguments. Don't forget the Out.ar(0, inst) at the end. For example: {SinOsc.ar(400)}.play Would be: SynthDef("MySine", {arg freq = 200; Out.ar(0, SinOsc.ar(freq))}).play Then: Synth("MySine", [\freq, 600]) 161

```
SynthDef("Chime", {
arg baseFreq = 100;
var totalInst = 7, totalPartials = 12, out;
out = Mix.ar(etc.);
Out.ar(0, out)
}).load(s)
```

```
SynthDef("Cavern",
{
arg base = 100;
var out, totalPartials = 12;
out = Mix.ar(etc.);
Out.ar(0, out);
}).play(s)
```

Etc.

```
Synth("Cavern", [\base, 500])
Synth("Chime", [\baseFreq, 300])
```

Etc.

17.1. What are the AM sidebands of a carrier at 440 Hz and a modulator of 130? 175

570 and 310

17.2. What are the PM sidebands of a carrier at 400 Hz and a modulator at 50? 175

potentially infinite sidebands, but assuming 6:

```
(-6..6)*50 + 400
```

17.3. Create a PM patch where the modulator frequency is controlled by an LFNoise1 and the modulation index is controlled by an LFSaw ugen. 175

```
{
PMOsc.ar(
   LFNoise1.kr(1/3, 800, 1000),
   LFNoise1.kr(1/3, 800, 1000),
   LFSaw.kr(0.2, mul: 2, add: 2)
)
}.play
```

17.4. Create a PM patch where the carrier frequency, modulator frequency, and modulation index are controlled by separate sequencers (with arrays of different lengths) but all using the same trigger. 175

```
{
var carSeq, modSeq, indexSeq, trigger;
trigger = Impulse.kr(8);
carSeq = Select.kr(
   Stepper.kr(trigger, max: 20), Array.fill(20, {rrand(200, 1000)}).postln;);
modSeq = Select.kr(
   Stepper.kr(trigger, max: 10), Array.fill(10, {rrand(200, 1000)}).postln;);
indexSeq = Select.kr(
   Stepper.kr(trigger, max: 15), Array.fill(15, {rrand(0.1, 4.0)})
   .round(0.1).postln;);
PMOsc.ar(
   carSeq,
   modSeq,
   indexSeq
)
}.play
```

17.5. Create a PM patch where the carrier frequency is controlled by a TRand, the modulator is controlled by an S&H, and the index a sequencer, all using the same trigger. 175

```
{
var triggerRate, carSeq, modSeq, indexSeq, trigger;
trigger = Impulse.kr([6, 9]);
carSeq = TRand.kr(200, 1200, trigger);modSeq = Latch.kr(LFSaw.kr(Line.kr(0.1, 10,
70), 0, 500, 600),trigger);indexSeq = Select.kr(
   Stepper.kr(trigger, max: 15), Array.fill(15, {rrand(0.2, 4.0)}));
PMOsc.ar(
   carSeq,
   modSeq,
   indexSeq
)
}.play
```

18.1. Modify this patch so that the LFSaw is routed to a control bus and returned to the SinOsc using In.kr. {Out.ar(0, SinOsc.ar(LFSaw.kr(12, mul: 300, add: 600)))}.play 195

```
{Out.kr(0, LFSaw.kr(12, mul: 300, add: 600))}.play
{Out.ar(0, SinOsc.ar(In.kr(0)))}.play
```

18.2. Create an additional control (perhaps, SinOsc) and route it to another control bus. Add an argument for the input control bus on original SinOsc. Change between the two controls using Synth or set. 195

```
{Out.kr(0, LFSaw.kr(12, mul: 300, add: 600))}.play
{Out.kr(1, SinOsc.kr(120, mul: 300, add: 1000))}.play
a = SynthDef("Sine", {arg bus = 0; Out.ar(0, SinOsc.ar(In.kr(bus)))}).play
a.set(\bus, 1);
```

18.3. Create a delay with outputs routed to bus 0 and 1. For the delay input use an In.ar with an argument for bus number. Create another stereo signal and assign it to 4/5. Set your sound input to either mic or line and connect a signal to the line. Use set to switch the In bus from 2 (your mic or line) to 4 (the patch you wrote). 195

```
a = SynthDef("Echo", {arg busIn = 4;
   Out.ar(0, In.kr(busIn, 1) + CombN.ar(In.ar(busIn, 1), [0.3, 0.5], 2,
4))}).play;
{Out.ar(4, SinOsc.ar(LFNoise1.kr(0.1, 500, 1000), mul: LFPulse.kr(3, width:
0.1)))}.play
a.set(\busIn, 2);
```

18.4. Assuming the first four bus indexes are being used by the computer's out and in hardware, and you run these lines: a = Bus.ar(s, 2); b = Bus.kr(s, 2); c = Bus.ar(s, 1); c.free; d = Out.ar(Bus.ar(s), SinOsc.ar([300, 600])); -Which bus or bus pair has a SinOsc at 600 Hz? -What variable is assigned to audio bus 6? -What variable is assigned to control bus 3? -What variable is assigned to audio bus 4? 195

I think bus 7 will have a sine at 600 Hz. There is no variable assigned to bus 6. There is no variable assigned to bus 3. Variable b is assigned to bus 4.

18.5. Assuming busses 0 and 1 are connected to your audio hardware, in which of these examples will we hear the SinOsc? ({Out.ar(0, In.ar(5))}.play; {Out.ar(5, SinOsc.ar(500))}.play) ({Out.ar(5, SinOsc.ar(500))}.play; {Out.ar(0, In.ar(5))}.play) ({Out.ar(0, In.ar(5))}.play; {Out.kr(5, SinOsc.ar(500))}.play) ({Out.ar(5, In.ar(0))}.play; {Out.ar(0, SinOsc.ar(500))}.play) ({Out.ar(0, SinOsc.ar(500))}.play; {Out.ar(5, In.ar(0))}.play) 195

I think this is right:
({Out.ar(0, In.ar(5))}.play; {Out.ar(5, SinOsc.ar(500))}.play) Yes
({Out.ar(5, SinOsc.ar(500))}.play; {Out.ar(0, In.ar(5))}.play) No, destination before source
({Out.ar(0, In.ar(5))}.play; {Out.kr(5, SinOsc.ar(500))}.play) No, control bus
({Out.ar(5, In.ar(0))}.play; {Out.ar(0, SinOsc.ar(500))}.play) Yes, though bus 5 has audio too
({Out.ar(0, SinOsc.ar(500))}.play; {Out.ar(5, In.ar(0))}.play) Yes

19.1. Write a function with two arguments (including default values); low and high midi numbers. The function chooses a MIDI number within that range and returns the frequency of the number chosen. 205

```
~myfunc = {
arg low = 0, high = 100;
rrand(low, high).midicps;
};
~myfunc.value;
~myfunc.value(40, 80);
```

19.2. Write a function with one argument; root. The function picks between minor, major, or augmented chords and returns that chord built on the supplied root. Call the function using keywords. 205

```
~chord = {arg root = 0; ([[0, 3, 7], [0, 4, 7], [0, 4, 8]].choose + root)%12;};
~chord.value(4);
~chord.value(7);
~chord.value(2);
```

20.1. Write an example using two do functions, nested so that it prints a multiplication table for values 1 through 5 (1*1 = 1; 1*2 = 2; . . . 5*4 = 20; 5*5 = 25). 212

```
5.do({arg i;
   i = i + 1;
   5.do({arg j;
       j = j + 1;
       [i, " times ",  j, " eguals ", (i*j)].do({arg e; e.post});
       "".postln;
})})
```

20.2. Write another nested do that will print each of these arrays on a separate line with colons between each number and dashes between each line: [[1, 4, 6, 9], [100, 345, 980, 722], [1.5, 1.67, 4.56, 4.87]]. It should look like this: 1 : 4 : 6 : 9 : ---------- 100 : 345 : etc. 212

```
[[1, 4, 6, 9], [100, 345, 980, 722], [1.5, 1.67, 4.56, 4.87]].do({arg next;
   next.do({arg number;
       number.post; " : ".post;
   });
   "\n---------------------------\n".post;
});
```

21.1. True or false? if(true.and(false.or(true.and(false))).or(false.or(false)), {true}, {false}) 220

False

21.2. Write a do function that returns a frequency between 200 and 800. At each iteration multiply the previous frequency by one of these intervals: [3/2, 2/3, 4/3, 3/4]. If the pitch is too high or too low, reduce it or increase it by octaves. 220

```
var freq;
freq = rrand(200, 800);
```

```
100.do({freq =( freq * [3/2, 2/3, 4/3, 3/4].choose).round(0.1);
   if(freq > 800, {freq.postln; "too high".postln; freq = freq/2});
   if(freq < 200, {freq.postln; "too low".postln; freq = freq*2});
   freq.postln;})
```

21.3. Write a 100.do that picks random numbers. If the number is odd, print it, if even, print an error. (Look in the SimpleNumber help file for odd or even.) 220

```
100.do({var n; n = 1000.rand; if(n.odd, {n.postln}, {"Error: number is even."})})
```

21.4. In the patch below, add 10 more SinOsc ugens with other frequencies, perhaps a diatonic scale. Add *if* statements delineating 12 vertical regions of the screen (columns) using *MouseX*, and triggers for envelopes in the ugens, such that motion across the computer screen will play a virtual keyboard. 220

There are certainly better methods to acheive this effect, but for the sake of the "if" assignment, here is how it's done:

```
(
{
var  mx, mgate1, mgate2, mgate3, mgate4;
mx = MouseX.kr(0, 1);
mgate1 = if((mx>0) * (mx<(1/12)), 1, 0);
mgate2 = if((mx>(1/12)) * (mx<(2/12)), 1, 0);
mgate3 = if((mx>(2/12)) * (mx<(3/12)), 1, 0);
mgate4 = if((mx>(3/12)) * (mx<(4/12)), 1, 0);
//etc.
Mix.ar(
SinOsc.ar([400, 600, 700, 900 /* etc */],
   mul: EnvGen.kr(Env.perc(0, 1),
       [mgate1, mgate2, mgate3, mgate4 /* etc */]))
)*0.3}.play;
)
```

22.1. Write a function that returns 50 random midi values, 10 at a time from these scales; 10 from a whole tone scale, then 10 from minor, chromatic, and pentatonic. 226

```
[[0, 2, 4, 6, 8, 10], [0, 2, 3, 5, 7, 8, 10],
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], [0, 2, 4, 7, 9]].do({arg scale;
   10.do({scale.choose.postln})
   })
```

22.2. Write a function that scrambles 12 MIDI pitches 0 through 11 then prints out that array transposed to random values between 0 and 11, but kept within the range 0 to 11 (if transposition is 10 then an original pitch 8 would become 6). Print the transposition interval and the transposed array. 226

```
var trans, row;
row = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].scramble.postln;
10.do({
trans = 12.rand;
```

```
"Transposition ".post; trans.post; " : ".post;
((row + trans)%12).postln;
})
```

23.1. Using this sentence set up an iteration that chooses one of the characters converts it to an ascii number, reduces that to below an octave (12) and prints that value. 232

```
100.do({
(("Using this sentence etc.".choose.ascii)%12).postln
})
```

23.2. With the random study above, add lines of code that print out all the information you would need to score the choices made. They include instrument, pitch, amplitude, start time, and duration. You will have to add a "time" variable that sums up each "next" value as it is chosen.

```
// Repeat this in each voice:

(
a = Task({
    var totalTime = 0;
    inf.do({arg i;
    var note, dur, next, amp, inst;
    note = rrand(24, 84);
    dur = rrand(0.1, 0.5);
    amp = rrand(30, 127);
    next = rrand(0.1, 0.5);
// m.noteOn(1, note, amp);
// thisThread.clock.sched(dur, {m.noteOff(1, note); nil});
    postf("inst: violin, pitch: %, time: %,  \tdur: %,  \tamp: %\n",
        note, totalTime.round(0.1), dur.round(0.1), amp);
    totalTime = totalTime + next;
    next.wait
    })
});

)
```

23.3. Use an array of pitch class strings, and an array of numbers 0 through 11 to generate a random 12-tone row and print a pitch class matrix showing all transpositions of original, retrograde, inversion, and retrograde-inversion. Here's a hint: begin with numbers only (from 0 to 11), it's easier to see the transposition and inversions. Try inverting just one row. Then do just the transpositions, then the inversions, then scramble the original row. Add the pitch class string last. User interface always comes last because it's the hardest. It is deceptively simple; about six lines of code. The printout should look like something like this. 232

```
(
//dodecaphonic matrix
```

```
var inv, pitchClass;
// traditionally the first item must be 0, so I scramble 11 then insert 0
inv = Array.series(11, 1).scramble.insert(0, 0);
pitchClass = ["C  ", "C# ", "D  ", "Eb ",
   "E  ", "F  ", "F# ", "G  ", "Ab ", "A  ", "Bb ", "B  "];
inv.do({arg trans;
   var row;
   row = ((12 - inv) + trans)%12; // un-invert
   // not: thisRow = (12 - (inv + trans))%12;
   row.do({arg index; pitchClass.at(index).post});
   "".postln;
   });
"".postln;
)
```

24.1. What is the most random event you can think of? What is the most predictable? Given enough information about the most random, could you predict it? Is there any chance the predictable event would not have the outcome you expected? 240

Other people, Me, No, and No. (But I like it that way.)

24.2. Draw a probability graph for this code: [1, 2, 3, 1, 3, 4, 5, 2, 3, 1].choose. 240

1 is 30%, 2 is 20%, 3 is 30%, 4 is 10%, 5 is 10%

24.3. Monkeys at typewriters: Will a random process result in recognizable melodies? Devise a system that generates random 7 note melodies and tests for matches with a collection of known melodies (e.g. Mary Had a Little Lamb). How long before the system generates one of 7 or so recognizable melodies? (Use arrays.) 240

```
var known, random, match, count;
match = 0; count = 0;
known = [
[3, 2, 1, 2, 3, 3, 3], //mary had . .
[8, 7, 8, 5, 3, 5, 1], //bach partita in E
[5, 3, 2, 1, 2, 1, 6], //my bonnie
[5, 6, 7, 8, 5, 3, 2], //seventy six trombones . . .
[1, 8, 7, 5, 6, 7, 8], //somewhere . .
[5, 3, 1, 3, 5, 8, 8], //oh say . .
[1, 1, 1, 2, 3, 3, 2], //row your boat
[1, 2, 3, 1, 1, 2, 3], //frere jacques
[5, 6, 5, 4, 3, 1, 1], //??
[8, 7, 6, 5, 6, 6, 6], //we will rock you (modal)
[5, 6, 5, 7, 8, 7, 5] //my three sons
];
random = [0, 0, 0, 0, 0, 0, 0];
while({(match == 0).and(count < 100000)},
{random = Array.rand(7, 1, 8);
count = count + 1;
known.do({arg each; if((random == each), {match = 1;})})
};
```

);
[count, random]

25.1. Name five microtonal composers (without looking it up). 247

I can only do four, including myself.

25.2. Write out and play every possible combination of the pitches B, and F resolving to C and E. 247

Did you include melodic intervals as well as harmonic? pitches outside our range of hearing? A440 and A400? equal tempered, just, mean tone tunings? compound intervals (B2 and F7 to C4 and E1)? What else have you excluded because of your bias?