{theseanco}

/ howto_co34pt_liveCode /

The First Edition

adapted to E-Book by {shamansir}

27. Feb 2018

About

howto_co34pt_liveCode is an attempt to extensively document my live-coding music practice. I live code as **coï¿¥ï³⁄₄ipt**, mostly making dance music (including for Algoraves), but I have also employed live coding as part of some other projects. This repo contains a number of articles and essays listed in this contents page which cover various aspects of my live coding practice from the ground up, and also contains a number of files to support your use of SuperCollider in the way that I use it. More info about this in ‘*What This Repo Is*’.

This resource is hosted both on GitHub and on GitHub Pages (you will likely already be on one of these). I'd recommend browsing articles on GitHub pages, and if you want to use any of the examples, see any SuperCollider code and use my setup I'd recommend downloading the repo, this will be covered in ‘*How To Use This Repo*’.

This repository is currently mostly finished, but does need proofreading, which I will be doing soon. It will be continually updated as my live coding practice develops.

I'm always keen to know the ways in which this has been helpful to anyone, or any comments you have at all. Drop me a line on Twitter, GitHub or via Email.

> The examples have been tested on *ubuntu 16.10 only thus far.

1. Introduction

What this Repo Is

Hello!

I live code under the name **coï¿¥ï¾¡pt** (bandcamp, soundcloud), and have been performing since 2014, including for Algoraves, gigs, theatre and dance.

This repo is an ongoing collection of materials about my live coding practice using SuperCollider. I'm hoping to use it to post writing, how-tos and guides, helpful code, techniques, frameworks and whatever else might be useful to anyone wanting to learn to live code using SuperCollider, or anyone who already does. It's a clone/version of and companion to my co34pt_livecode repo, which contains finished sets from gigs alongside my setup code and samples.

I'm putting together this repo because I wish there'd been such a thing when I had been learning to live code in SuperCollider, (or at least I wish there'd been one i'd been able to find). There are a bunch of great tutorials for SuperCollider out there (I'd particularly recommend this set by Eli Fieldsteel), but I found advice on live coding in SuperCollider reasonably hard to come by, and was fortunate enough to have an experienced live coder as a teacher. An awful lot of the examples I give and techniques I talk about can be found in the documentation of SuperCollider, but finding it can be a little frustrating and examples can often be written in quite different styles. I'm aiming for this repo to be a curated set of resources following a central style, with various parts of the language explained in the context of the kind of live coding I do, with musical examples of how I would use these techniques in actual live sets.

For the resources here i'm assuming a basic knowledge of how to use SuperCollider and programming concepts generally, because i'm not a great person to explain this. If you're totally new to SuperCollider and programming i'd either recommend the Eli Fieldsteel videos previously mentioned, Supercollider's inbuilt turotials, or Nick Collins's tutorials. The more advanced tutorials won't be necessary for the most part, but a familiarity with executing SuperCollider code, what UGens are and basic syntax will be super helpful.

The materials in this repo are by no means the *best* way to live code using SuperCollider, or the most efficient way to solve any problems, but they are the ways that have worked for me personally. If you spot any glaring issues in this repo, please fork and change/correct! I'd also welcome guest contributions, although I maybe should rename this repo if i'm going to do that so as not to have it under my 'umbrella' name.

I'm also continually looking for a better way to manage this repo as a learning resource. I've figured GitHub is probably the best way to do it as there is version control, all resources can be hosted here along with details on how to use them, and it can easily be packaged and downloaded, the only thing missing being a 'comments' section.

If you want to get in touch with me about this repo, please do so on Github, Twitter (same username) or via theseancotterill atsymbol live period com. If there is anything not working, please open an issue/make a pull request and I will look at it as soon as I can.

A note about formats: The examples and code in this repo will be tested using Linux, in my case Kubuntu 16.04+, but as SuperCollider is cross-platform this shouldn't matter too much for the most part.

Why I Live Code

As an exiled classical violinist, dormant guitarist, habitual electronic tinkerer and (as of 2014) live coder, I got interested in making electronic music when I listened to Portishead's 'Dummy' and Boards of Canada's 'Geogaddi' (among others) in my early teens. I began learning how to produce it as soon as I could by experimenting with FL Studio alongside early YouTube tutorials, the first milestone of this being the release of my first 'album' of 'Ambient music' as a .zip of 128kbps .mp3 files on MediaFire.

These Digital Audio Workstation (hereafter DAW) compositions and arrangements were a lot of fun to make, and enabled me to experiment with many techniques and genres, but I couldn't 'perform' them. This was of course until I discovered Ableton Live. As someone who had been confined to static DAW arrangements for some time, Ableton with its emphasis on live performance through alternative interfaces/controller mechanisms was my platform of choice for around five years. Ableton's emphasis on performance initially allowed me to compose music in a performative manner by using loops, triggers and controllers, and eventually gave me the confidence to take specific compositions to a stage, with varying degrees of success. I then began composing and performing music using a mix few proprietary DAWs and programs.

Here's an old performance of mine.

After a while I had some reservations about my continuing use of proprietary DAWs, for a few reasons.

First was the inflexible nature of the kinds of performances I was delivering. I had a set of compositions (or 'songs', if you will), which were arranged into a set of loops which could be triggered in *theoretically* any combination, but in order for the songs to make sense as pieces of music, the order had to be reasonably strictly obeyed. I had some flexibility in the way I applied effects to individual channels, but this to me did not translate to directly 'performing' tracks in the way I would 'perform' with a traditional instrument—I felt as if my performances had become glorified button pushing ceremonies. I am very aware that there are much more 'live' ways to play with various DAWs than the methods I used, but this was not how I had ended up performing. Around the time I decided to give up on proprietary DAWs I was pretty immersed in playing improvised music with guitar/violin/electronics/various media during my Music degree, and

I wanted to be able to bring an improvisatory instrumental spirit to my performances of electronic music. In performing with proprietary DAWs however I personally fell far short.

Second was the fact that the software was --h u g e--, and *!DEMANDING!*. My performance DAW suite of choice took up around 54GB of hard disk space, and became very difficult for my laptop to handle if I used any external software instruments at all. As a result of this, each individual track was an unwieldy bundle of samples and instruments, which would take a large amount of processing power to render. If I then wanted to perform a set of these tracks, I'd often have to combine a number of live 'projects' together and save them as one large project, as having to load each individual song before I played it would take minutes, breaking the flow of performance. What resulted were metaprojects which would be utterly enormous, unresponsive and would sometimes crash on loading. They could also be quite buggy, and performances felt 'risky' in the sense that any movement could topple them and bring my entire performance with it. While i'm all for embracing the possibility of a crash, this possibility being a structural feature of a performance without that being my intention was not an enjoyable way to perform.

Third is that the software is proprietary, and I was unhappy with what that represents. Leading up to the time I eventually gave up with proprietary DAWs (and subsequently proprietary software in general, where possible) I had been watching a number of lectures by Richard Stallman discussing proprietary software and user freedom. This, coupled with the work of glitch artists (particularly Rosa Menkman and Nick Briz) focusing on the role of platforms and softwares as often unacknowledged intermediaries in our material experiences of technology presented me with a set of issues I could not personally resolve. While I released all of my music under creative commons in disagreement with copyright legislation, I was producing music using tools that were not only bound by the legislation I disagreed with, but tools that were purposefully restricted the way that I could use them. In the words of Richard Stallman:

> With software, either the users control the program (free software) or the program controls the users (proprietary or non-free software).

The proprietary nature of the software also means that it can only be run on certain systems by those with the financial ability to run it (or willingness to break various laws), on top of having to have access to a computer. The copyleft approach I had to the works I produced were very difficult (if not impossible) to apply to the materials used to make the works themselves.

Fourth was my relationship to traditions of performance in 'laptop music'. Even with controllers, performances I would deliver would always be me staring into a black box in the form of a laptop, occasionally triggering things on a controller. While I attempted to get around this in some ways by projecting a video of my controller during sets as part of the visuals during sets, this didn't alleviate the problem of obfuscation. I was very used to a direct cause-and-effect relationship between actions and sounds, and for that relationship to be apparent to an audience. Whether I was bowing a violin, chugging away at 12/8 swing, or playing guitar with a handheld fan and a wood file (actually happened), the cause-effect relationship between myself and any potential audience was pretty clear. I felt as if my performances of electronic music did not have this kind of immediacy, and I didn't like that at all†. I'm very aware that this kind of immediacy isn't something that everyone strives for in laptop performance, but I missed it dearly. In addition to this, performances of electronic music of this type offered no opportunities for me as an audience member to learn about its construction besides how it sounded. I've always been fascinated by the construction of music and art, and the ability to deconstruct this in real-time is something I really value, much like the YouTube FL Studio tutorials I followed to learn how to make electronic music in the first place (I did this because I didn't realise the software actually had a manual, and I didn't realise my performance DAW even had a manual until I had been using it for three years). With this 'black boxing' of the performance setup, I had no layers to peel back - if a performer did something cool and I wanted to do it, tough luck, time to go home and reverse-engineer it without any idea what tools were used in its construction! I've never been enamoured of obfuscation or secrecy around technique. Why should techniques be a big secret? Much like the copyrighting and locking-down of the software, performance traditions that obscure the mechanisms one can use to do 'cool things' are pretty frustrating for me, whether or not that is the intention of the performer.

With these issues in mind, what was the answer to my problems with digital music performance? The best answer I have found is live coding, but it took me a while to get there.

Until around 2014, I had been dead-set against 'music-programming' (at the time I meant Pure Data and Max/MSP), as I was convinced that the integration of programming and music would take the 'human element' out of the music I was performing. Needless to say this was short-sighted and incorrect, and was probably a hangover from my education in the classical music tradition through the British schooling system, in which electronic music was often derided as a something not to be taken seriously, and not as 'real music'. I had overcome this once I learned that my university took electronic music pretty seriously, however the idea of programming still stuck around as ‘non-musical’. As was reasonably common among my peers, I found programming to be an alienating concept, with its syntax, language, args/ints/strings/longs and so on, it seemed the exact opposite of what I considered music creation to be—intuitive, tactile, ‘musical’. How could…

```
{SinOsc.ar(LFSaw.ar(XLine.kr([0.01,0.02],[400,500],
100)).range(1,2000).round(200))}.play;
```

…be music if it didn't look like any music I had ever played before?

Around the time I was considering these issues and starting to look for alternatives I was fortunate enough to audit some classes by John Bowers where I learned how to use Pure Data and Arduino for multimedia performance and installation work. As a result, I actually learned how programming worked and what it was capable of, and began producing interactive digital works and performances. In addition, I was using free and open soure software almost exclusively to create these works (with the exception of Max/MSP for video). It turned out that by using programming I could not only escape the trappings of limited systems for artistic expression by creating my own, but could extend outside of audio and into video, graphics and electronics through the use of open standards. I had overcome my fear of code!

While this was great for developing artworks, and provided a way out of using proprietary software (again, with the exception of Max/MSP), it didn't provide me with a solution for the music performance problem.

However, a housemate of mine at the time had been teaching me a little SuperCollider, a platform for audio synthesis and algorithmic composition. SuperCollider seemed to be the best platform for applying my newfound programming enthusiasm to electronic music, with the ability to operate outside of proprietary software, and the ability to choose the terms on which I would interact with the music I created (what DAW environment will let you play 1,000 copies of a three minute sound at random speeds with one action?). Around the time that I learned basic SuperCollider skills I had to complete my final year of my undergraduate music course, where I elected to do a 40-minute performance in place of a formal written dissertation. I figured the best thing to do would be to put my money where my mouth is (so to speak) and take the plunge away from proprietary DAWs into performing music with code. When I decided to do this Algorave had been in my periphery for a little while as live-coding's answer to electronic music performance. The TOPLAP Draft Manifesto alongside some events I had attended in Newcastle and Sheffield featuring live coding musicians piqued my interest in Algorave and what it could offer me by way of an approach to electronic music performance, and it turned out to be a great working answer to my main gripes with performing electronic music with proprietary DAWs.

"First was the inflexible nature of the kinds of performances I was delivering"—Live Coding tends to revolve around wholly or partly improvised performances, and the ability to write code in a non-linear way and execute it in real time and have the results instantly rendered as audio opened the playing field for me hugely. While it is possible to have live coding performances with a very set trajectory which evolve in the manner of a meticulous composition, it's equally possible to start from literally nothing except a running synthesis server. With a language as broad as SuperCollider, I could integrate anything from blistering noise based on non-linear maps through to 5/4 kick drums through to complex sample manipulation through to 4/4 kick-snare-clap patterns within one performance. While of course it's not always productive (or possible) to draw on such wildly disparate techniques during performances, the fact that the possibilities exist is very important. In addition to this, there are a plethora of live coding languages that can all be networked to one degree or another (although I usually stick to SuperCollider for reasons I'll detail in a later post).

"Second was the fact that the software was --h u g e--, and *!DEMANDING!*"—In switching to a programming platform like SuperCollider to make music, one is presented with the ability to start from basically zero. The SuperCollider source code is currently (as of March 2017) an 14.6 MB download from GitHub, and runs without any GUI by default, meaning that system load is very low out of the box (SuperCollider comfortably runs on Raspberry Pi), with the loading of extended functionality and libraries at the discretion of the user. In addition, projects are written and loaded as text files, which take up very little disk space and can be loaded near-instantly. By switching out my proprietary DAW for a live coding setup, I wouldn't have to wait minutes for projects to load (or have them crash outright after loading), and the separation of editor/server/interpreter in SuperCollider makes the management of any crashes much easier. If i need to, I can also perform on low-cost, low-power hardware, or use SuperCollider to create embedded installation works.

As it is a programming language, SuperCollider can be (and has been) built up to a fully-functioning DAW-type environment if necessary. With this I could try to like-for-like replace a proprietary DAW environment if I wanted, but doing so would, for me, partially defeat the point of learning how to live code in the first place. In live coding I can build and maintain an environment that suits me as a performer, keeping a simple, effective workflow to articulate my ideas within.

“Third is that the software is proprietary”—With a few exceptions (notably Max/MSP), live coding draws from rich ecosystem of free and open source tools, often with practitioners being active contributors to the software packages that they use (a good example being Alex McLean and TidalCycles). In adopting Live Coding as a method for electronic music performance I could finally leave the Apple ecosystem and the proprietary DAW paradigm in favour of using GNU/ Linux and open source tools. I could now have full access to the tools I would be using to create music and the ability to modify these tools as I wished. In addition, so can anyone else! I can happily write a set of tutorials on how I live code electronic music knowing that anyone who has access to a computer running a compatible operating system should have the ability to follow that tutorial without them having to have access to hundreds of pounds worth of software and a license for Windows or an Apple machine. Live Coding was the last piece of the puzzle in my transition to a fully open source art practice, both in the tools I use and the work I create, which is now the focus of my PhD research. I try to keep an updated GitHub repo containing my live coding setup and sets, and I am going to be writing some docs/guides on how I live code dance music using SuperCollider and my own custom boilerplate code. The repo can be found hereand a set of resources on how to live code in SuperCollider can be found here.

“Fourth was my relationship to traditions of performance in ‘laptop music’”—I'm *far* from the first person to pick up on this, but the TOPLAP manifesto's 'Obscurantism is dangerous. Show us your screens.' seemed like a beautiful answer to the kinds of indecipherable laptop performances that frustrated me as a concert-goer. Important to ‘Show us your screens’ too is its corollary:

> It is not necessary for a lay audience to understand the code to appreciate it, much as it is not necessary to know how to play guitar in order to appreciate watching a guitar performance.

By adopting a text-based interface to perform and also projecting that text-based interface for an audience to see during a performance, a number of things are achieved.

First, for anybody interested the text makeup of a performance is shown, showcasing the inner workings of a performance as it comes together, live on stage. This is useful for me as a live coder myself because I can see how 'cool things' are done as the 'black box' of the performance laptop is removed to some degree - I've learned a whole bunch of techniques by going along to algoraves and following the projections to see what is being done by the performer (this also includes live streaming one's sets, which I have done a decent amount of). In addition to this, for anyone who doesn't understand the specifics of the language being used (or isn't interested) this opening of the laptop performance ecology serves the purpose of exposing the materiality of the performance - in watching a performer type and execute code you are seeing the performer at work, how they respond to various stimuli during performance, and how their thoughts are translated to text. In addition to this, through the selective writing of, navigation through, and execution of text, the kinetic intent of the music is demonstrated. Much as an instrumentalist stamping their foot to a beat more than likely shows the path of their playing, a live coder hurriedly typing `~kickdrum.play` (or equivalent) shows their vision of the music in real time.

More significantly though, I'd argue this projection of text is more than the fleeting glimpse one can see when observing a traditional instrumentalist at work. In watching a performer articulate their music as a text file on screen, I feel as if I am watching a performer build and manipulate a sculpture over the course of a performance, with the form of that sculpture being mirrored in the changes in the music heard throughout the performer's set. Whether that involves a performer starting from absolutely nothing and building a performance from minimal roots, regularly deleting their entire text and starting again, or a performer loading a pre-written text and selectively executing/modifying it, drawing on an extensive codebase to craft a detailed performance (both of which I've seen Yaxu alone do), or anything in between. As I perform using SuperCollider, the level of verbosity required means I often type and navigate through text a *lot*, however I am always shocked at how little code I actually have at the end of a performance. My performances are usually composed of a select few carefully-maintained symbiotic micro-structures which I edit extensively. I don't write an awful lot from scratch, but I fairly meticulously edit and re-edit what I do write, executing the same piece of code many times in one performance with slight changes to fit the other few running pieces of code.

In watching a live coding performance, you can see the performer not only deal with the environment of performance in real time in a way that is potentially useful to practitioners and (relatively) transparent to “lay-persons”, but see them dealing with both the history of, and potential futures of their performance in an engaging way.

It's also undeniably eye-catching.

So with all of this in mind I decided to take the plunge and learn to live code. I was fortunate enough to have a great opportunity to uproot everything I knew about performing electronic music in the form of my final-year undergraduate dissertation, which I used as an opportunity to deliver a 40-minute live coding performance. I was also fortunate enough to have some teaching on how to live coding using SuperCollider from Data Musician and Algobabe Shelly Knotts. I've since played a bunch of Algoraves and live shows (a lot of which can be found here), streamed a whole bunch of sets, and applied live coding approaches to other projects.

Reasonably quickly Live Coding became “how I made music”, and a few realisations followed:

In live coding I could not only embrace alternate traditions of laptop performance, but also paradigms of laptop music. The way I had worked in DAW software had always been dominated by audio loops, MIDI data and VST plugins, and these methods are much less immediately accessible in live coding performance with SuperCollider. Much is made in the live coding community of the role of the algorithm in performance, and I've only recently realised what that *actually meant*, after initially being quite scared by the 'maths-ness' of the term. In creating a drum pattern in a DAW environment, I would layer together drum loops and play instrumental lines using a keyboard to achieve the desired rhythms, but in a live coding environment I specify a bunch of behaviours to determine how drums are 'played', and similarly with melodies, textures and bass. In performing I am creating multiple rule-governed self-managing instrumental 'players', and shepherding them around to create a performance, rather than 'playing' the music in a traditional sense - this is something that is intuitively quite easy to achieve through live coding in SuperCollider, but something I found quite difficult to achieve in a DAW environment. Incidentally I find this method of performance much more tactile and 'instrumental' than the DAW paradigm, after this method of performance was the very thing I was afraid would take the 'human element' out of music!

Aspects of music as fundamental as pitch and rhythm organisation are easy to experiment with too. I'm a big fan of using Euclidean rhythms and some constrained randomness to generate compound rhythmic patterns, as well as using the Harmonic Series to determine pitch for melodies and textures, and the bare-bones 'do it yourself' nature of live coding in SuperCollider means that I can fairly easily build performance systems based around non-standard musical techniques.

Electronic Music also has problems with diversity, and there are a number of facets of the live coding community that are actively addressing this. There are groups such as SoNA and YSWN encouraging the involvement of women in the live coding community, and socially-concerned organisations such as Access Space are also actively involved. My experience both attending and taking part in live coding events shows commitment to addressing these issues too - while there is no formal code of conduct, a general commitment to inclusivity in participation (no all-male bills at Algoraves), attitudes and language are commonplace. With the recent #Algofive stream showcasing not only a diverse global network of artists but a diversity of approaches to live coding too, it's a community I'm very proud to be a part of.

Like everything, Live Coding does have its problems. I've realised that all of the freedom that live coding in SuperCollider offers also comes with the drawback that I have to build my own frameworks to perform with, starting from the basics, which is sometimes pretty paralysing. If I'm stuck for inspiration, it's actually quite hard to get myself out of a rut, and discovering how to use different features is actually quite difficult without having the software having a 'manual'. Further to this, Open Source software and libraries can sometimes be scantily documented, with incredibly useful tools remaining difficult to access because only the creator of those tools knows how to use them properly. In addition, the issue of performative transparency isn't quite as clear cut as 'I'm projecting code, therefore my intent, action and gesture in performance are immediately and clearly articulated' - in '[showing] your screens', the black box has just been shifted to the processes underlying the code itself. There's also the issue of 'code literacy' presenting a barrier to entry to live coding, however this is addressed both through the publishing of learning tools by the community and languages that require less specialist knowledge to use effectively, as well as workshops by the community to engage those unfamiliar with live coding and programming in general. I am also very aware that my somewhat idealistic notions of what *I* want to demonstrate through performance may well not matter to other performers, and this is fine too.

All things considered, I live code because it allows me to use free/libre/open source tools to create flexible musical environments that allow me to perform electronic music in a way that I feel gives me the ability to think and play like an improviser. My initial fears that coding music would lead me to academic 'maths music' turned out to be completely the opposite - performing with live coding is by far and away the closest I have come to an 'instrumental' way of performing electronic music. Let's keep going with those repetitive conditionals!

I have written (and am continuing to write) resources/guides/tutorials/docs etc on live coding with SuperCollider here. My website is here.

† As a caveat to this, the closest I probably came to this cause-effect relationship becoming clear while using DAW software was with Mutual Process, an improvised music project with Adam Denton of Trans/Human. For Mutual Process I performed manipulations of live-recorded samples of Denton's guitar, which were fed back to him—and I used a number of controllers to live-patch effects and record/process samples. I had a huge amount of control over this setup to the point where I felt as if I could impact upon the performance with physical control gestures, and embody my action within the music somewhat. Interestingly enough this performance setup was a complete 'hack' of Ableton's core functionality.

Why SuperCollider?

/ Why I Live Code in SuperCollider /

You're brave to use SuperCollider!

— Anonymous, after a performance of mine, also probably slightly misremembered.

Looking at lvm's awesome-livecoding list, there are currently a whole bunch of live coding languages and platforms built around a whole bunch of paradigms, suited to many different users with varying aims, mediums, skillsets and abilities.

SuperCollider sits on the back-end of a few live coding-specific languages, including FoxDot, TidalCycles (with SuperDirt), Overtone, ixi lang and probably some I've forgotten, but within SuperCollider there is ample support for live coding in the form of various libraries and techniques (I use JITLib), and I've been using it since 2014 for performances, composition and for building other projects.

I've tried (and performed with) a bunch of other live coding platforms (mostly TidalCycles and FoxDot), and have repeatedly settled on SuperCollider over and over for live coding. As someone mostly performing metre-driven beat-based dance music, this can seem like an odd choice. TidalCycles, for example, is specifically built around rhythmic cycles, and is a fast, efficient way to create complex rhythmic units.

SuperCollider on the other hand has no one central method to produce rhythmic patterns or loops - instead there are a number of different ways to leverage pattern classes, some of which are really quite unwieldy and not at all suited to live coding and rely on a lot of pretty complicated nesting. SuperCollider is also *really* verbose—when creating patterns basic arguments need to be manually specified, which requires a lot of typing. In addition to this, SuperCollider has no real 'built-in' mechanisms for live performance—these often have to be built by the user and imported as libraries. This repo contains a number of SynthDefs, or 'instruments' that I have had to build myself or copy from elsewhere in order to perform basic functions within patterns - want to play a kickdrum sample? Better build a way to do that yourself! Want a square wave you can trigger as part of a pattern? Better go write that synth!

It's also full of strange undocumented methods and classes, which can hold keys to performance techniques that I'll never find because I don't know what they are—I had to catch someone using the method `.stutter` during a live set to figure out its potential uses for me. People who live code in SuperCollider also often do it very differently from each other, using different, sometimes not transferable sets of techniques - this is a result of SuperCollider being a comparatively enormous language, but as a result I had quite a bit of difficulty learning how to use it, especially for a musician who hadn't been coding very long (myself, when I first learned SuperCollider). *a mess*

SuperCollider can also be pretty unforgiving. With no built-in limiter, one incorrect argument can be absolutely devastating—the main perceptual difference between `SinOsc.ar(400,0,1)` and `SinOsc.ar(400,0,10)`, is pain. Especially when you're wearing headphones. It's also pretty easy to bring the whole server to a halt with a mis-typed `\dur` argument.

The results of this?

From absolutely nothing,

```
d1 $ sound "bd sn"
```

...in TidalCycles produces a kick-snare pattern, which can very easily be extended to...

```
d1 $ sound "bd sn cp"
```

...to produce a kick-snare-clap pattern.

In SuperCollider however, producing a kick-snare pattern can take a number of forms, but this is how I would end up doing it from boot-up (without any of the setup code in this repo).

```
a = Buffer.read(s,"/path/to/kick/kick.wav");
b = Buffer.read(s,"/path/to/snare/snare.wav");
SynthDef(\bplay,
    {arg out = 0, buf = 0, rate = 1, amp = 0.5, pan = 0, pos = 0, rel=15;
    var sig,env ;
    sig = Pan2.ar(PlayBuf.ar(2,buf,BufRateScale.ir(buf) * rate,
1,BufDur.kr(buf)*pos*44100,doneAction:2),pan);
    env = EnvGen.ar(Env.linen(0.0,rel,0),doneAction:2);
    sig = sig * env;
    sig = sig * amp;
    Out.ar(out,sig);
    }).add;
p = ProxySpace.push(s);
p.makeTempoClock;
~k = Pbind(\instrument,\bplay,\buf,a,\dur,0.5,\amp,1);
~s = Pbind(\instrument,\bplay,\buf,b,\dur,1,\amp,1);
~k.play;
~s.play;
```

And in order to do the kick-snare-clap pattern I would have to add:

```
c = Buffer.read(s,"/path/to/clap/clap.wav");
~k = Pbind(\instrument,\bplay,\buf,a,\dur,1/3,\amp,1);
~s = Pbind(\instrument,\bplay,\buf,b,\dur,1/3,\amp,1);
~c = Pbind(\instrument,\bplay,\buf,c,\dur,1/3,\amp,1);
~c.play;
```

So **why** would I choose to use a system like this, when there are some that are much more efficient for the kinds of things I am doing? (I am being a *little* obtuse in the code example above for the sake of argument).

The answer is primarily, of course, *because it works for me*, but here's why.

SuperCollider is a huge language, containing not only a really great set of pattern libraries and live coding functionality, but some of the best synthesis capabilities of any program I have ever used, and with extensions, the possible functionality I can draw upon is absolutely enormous. In this repo I'll be talking about how I use Euclidean Rhythms, Nonlinear Maps, Common fundamental frequencies 53 tone scales, and many other techniques to make parts of music. SuperCollider's amazing array of native and extended functionality is not immediately usable for live coding from the time of installation, but with some reusable scaffolding in place these features can be relatively easily leveraged. The issue of the verbosity of SuperCollider compared to Tidalcycles can be mitigated with setup code and extensions - it's taken me a while to build and work with structures to make using SuperCollider as a performer more effective, but once the framework is in place things get much easier, and can be tuned to suit any particular performance needs.

The lack of pre-built foundations is also liberating in some respects because if I want to get down to a 'lower level' during a performance it's trivial to do so. If I am hitting a wall during a performance of some heavy beats, the same library that allows me to change high-level pattern structures on the fly will also allow me to start multiplying bare sine waves and performing brutal additive synthesis live alongside these patterns. The code shown earlier of a kick, snare and clap all being run as separate 'instruments' is how I usually do my live coding, and while this seems very text-heavy and verbose, it allows me to create a number of small, relatively self-governing processes which will compute and play of their own accord, until I change them. Through this method, my performances usually involve building up musical textures and patterns through allowing each 'instrument' a small amount of its own variability - together each small amount of variability comes together to form a kind of emergent complexity, the sum of all of its (relatively) simple parts. Through the performance I'll then manage these units, building new ones as old ones become fatiguing, and injecting new life into stalwart units (such as kick/snare drums) by modifying their patterns/pitch/effects/etc. I like to think of this live coding performance setup as a kind of ecology of small units being constructed, managed, decommissioned, revamped and destroyed throughout a performance.

This kind of “ecological” approach means that once the basis of a 'sound' during a performance are established (eg. hi-hat pattern, kick and two melody lines) I can spend some time building the next set of sounds, while the other sounds manage themselves and stay sonically interesting through some well(/poorly)-placed algorithmic transformations. The verbosity of the pattern language also helps in some respects too, having to type the names of individual parameters means I am forced to consider the nature of the sound I'm about to throw into the mix while I'm typing it. This is one of the reasons why I don't think I got on with Tidalcycles when I tried to perform solo with it - it's powerful enough to change the entire dynamic of a performance using a few characters, and I'm not responsible enough to wield that power.

My biggest gripe with SuperCollider is the pretty verbose Pattern syntax, as patterns are a huge part of my live performances. I think the pattern classes in SuperCollider are very powerful, but a lot of typing *does* need to be done. Fortunately the ddwSnippets quark has finally arrived, delivering some snippets to the SuperCollider IDE! Before that, I would keep a bunch of 'default' patterns on hand in another document during performance to copy-paste and change. I've also heard that scvim is currently in active development, and as a vim user I'd love to integrate it as my SuperCollider editor.

All things considered, while not the most intuitive live coding platform, it's the one that works for me, and will probably continue to be so.

How to Use this Repo

This repo is designed to be an interactive and explorable set of guides, as well as being browsable online and outside of SuperCollider. It's also designed to be as platform-agnostic as possible, and easily accessible with GitHub's basic tools.

If you are reading this online, and want to use any examples contained in these documents, or see any code I've written "in the flesh", there's a few things you'll need to do.

1. Install SuperCollider;
2. Install SC3-Plugins (optional but recommended, reasons for this are documented in 2.1)
   NOTE: Steps 1 and 2 on *buntu (and probably other types of Linux too) can be performed with the `install_supercollider_linux.sh` script in the `scripts` folder. This installs both SuperCollider 3.8 and sc3-plugins as well as all relevant dependencies for *buntu.
3. Install the recommended Quarks by evaluating the following code in SuperCollider (optional but recommended, reasons for this are documented in 2.1):

```
(
Quarks.install("Bjorklund");
Quarks.install("BatLib");
Quarks.install("ddwSnippets");
)
```

4. Clone this repo, which can be done from the repo page, using the "Clone or Download" page.

The articles in this repo are grouped into category folders, which are then contained in subfolders with the name of the article. Each subfolder will have in it a `.md` file, which contains the text of the article containing examples, and usually a separate `.scd` file, which will be the examples from the `.md` file extracted and packaged for direct use within SuperCollider. The `.scd` files are designed to be loaded and run directly, and will usually contain a `.loadRelative` which will Setup that you will need to get started. You should then be able to run all examples.

If you want to read online while playing with examples in SuperCollider, run the `Setup.scd` file within this repository and then copy-paste the examples on the site into a file in SuperCollider and it *should* work. I need people to test these examples, so if it doesn't work, please raise an issue on GitHub with exactly what you've done and I'll look into it when I can, or get in touch with me.

2. Basics

Recommended Addons

/ SuperCollider Addons I'd recommend /

Here is a list of Extensions and Quarks that are crucial to my live performances. If you want to be able to use all of the resources in this repo, you should install them.

Extensions

Extensions have to be inserted into SuperCollider manually. See this document for more information. Note sc3-plugins have to be compiled on Linux. See the sc3-plugins readme on GitHub for more information.

sc3-Plugins

This repository contains the community collection of unit generator plugins for SuperCollider. An installation extends the functionality of SuperCollider by additional UGens that run on scsynth, the SuperCollider audio synthesis server.

sc3-plugins is a mixed bag of tools, and contains a lot of things I don't use, but it's pretty essential for getting the most out of SuperCollider. Some of the sc3-plugins are fairly scantily-documented, and fall into the “sounds cool, but no idea what it does or how it works” category.

Particular tools from sc3-plugins I use regularly:

- Concat and Concat2 Tools for concatenative synthesis. Particularly useful when dealing with speech and sampling - I've used them to “reconstruct” speech using existing samples.
- Decimator and SmoothDecimator Bitcrushing effect Ugens for that classic digital destruction sound. SmoothDecimator has a smoothing option to take some of the digital bite out of the bitcrushing sound.
- SawDPW (and PulseDPW) Alternatives to SuperCollider's native Saw and Pulse Ugens, which alias much less, use less CPU and sound an awful lot better especially during additive synthesis. Can also get really wild at unusual frequencies.
- DFM1 A really fantastic sounding digitally-modelled analog filter. Great both as a scuzzy-sounding filter on existing sounds and when pushed into self oscillation to make rich drones. Sounds good both in moderation and absurdity.
- CrossoverDistortion A savage distortion. I don't really have a lot more to say about it.
- WaveLoss An effect for dropping sections of waveforms in either a deterministic or random fashion. Produces a 'degradation' effect from slight dropouts all the way to isolated spluttering.

BenoitLib

A set of SuperCollider extensions used by Benoît and the Mandelbrots.

The main tool I install this for is Pkr, a pattern proxy for synchronising control rate Ugens inside of patterns, which is a technique I will be covering in this repo. It's a small part of the extension but is totally invaluable for my performances.

There's also some super useful stuff in BenoitLib for collaborative performance which I have used before in a performance with Shelly Knotts, including MandelHub and MandelClock.

Quarks

Quarks can be installed from within SuperCollider, either by installing them manually (`Quarks.install('BatLib')` for example), or using `Quarks.gui` to bring up a gui install them there.

Bjorklund Quark

The Bjorklund quark implements Euclidean Rhythms, a concept outlined in this paper, involving taking a number of onsets and a number of possible steps, and spaces out the onsets as equally as possible in the given number of steps. A verbal explanation of this doesn't really do it any justice, so I'd encourage you to use this cool web app which visually and aurally explains what these rhythms are. I've found Euclidean rhythms a great way to program rhythm that is dynamic and interesting, but also sits well within a set of metric dance music. The class I use from this quark is Pbjorklund2, which gives an array of durations for euclidean rhythms.

BatLib Quark

BatLib contains StageLimiter, a class that puts a basic limiter across all sounds in the SuperCollider server. StageLimiter doesn't really have any effect on the sound the server makes unless you exceed an amplitude of +/- 1 (the top and bottom of the default SuperCollider scope), and when you do push harder than that, you can use it creatively to get 'side-chaining' type effects. I'd recommend always running StageLimiter unless you have a specific reason not to anyway, as an amplitude value that is accidentally out by a factor of ten can be *really* painful.

ddwSnippets Quark

ddwSnippets is a 'Rudimentary snippets facility for ScIDE, implemented in sclang'. I've found snippets are very useful for any piece of text that will be typed multiple times during a performance, or to lay the groundwork for 'basic' musical patterns without having to write them from scratch (see my comments in 0-2 about SuperCollider's verbosity). I use ddwSnippets to realise musical ideas more quickly when performing, especially using Ugens or patterns that have a lot of arguments, without having to copy-paste from a 'template' file containing the snippets.

sc3-plugins and BenoitLib have to be installed manually. a note for compiling sc3-plugins on Linux is that my /path/to/scsource is /usr/local/include/SuperCollider, and I would assume that would be a typical path for most users To install all quarks listed in this document, execute the following in SuperCollider:

```
(
Quarks.install("Bjorklund");
Quarks.install("BatLib");
Quarks.install("ddwSnippets");
)
```

If you have trouble installing `ddwSnippets`, execute this too:

```
(
Quarks.install("https://github.com/jamshark70/ddwSnippets.git");
)
```

ProxySpace

/ My Foundation for Live Coding in SuperCollider /

Why ProxySpace?

If you haven't heard of or used it before, ProxySpace and it's associated JITLib are *well* worth knowing about, and are without exception what I use to live code in SuperCollider.

According to the docs (see link above):

> Generally a proxy is a placeholder for something. A node proxy is a placeholder for something playing on a server that writes to a limited number of busses (e.g. a synth or an event stream). NodeProxy objects can be replaced and recombined while they play. Also they can be used to build a larger structure which is used and modified later on.

In other words, ProxySpace opens up SuperCollider's language into a powerful performance tool by allowing individual functions/patterns/etc to become flexible and modifiable, as well as to make these patterns interact. When using ProxySpace, the traditional…

```
{SinOsc.ar(440,0,0.2)!2}.play
```

…is turned into an "instrument" when given an arbitrary name and edited on the fly. It can also be used within other "instruments", for example:

```
(
~sine1 = {SinOsc.ar(440,0,0.2)!2};
~modulation = {Saw.ar(10,0,1)!2};
~sine2 = {~sine1 * ~modulation};
~sine2.play;
)
```

ProxySpace can also be used for synchronising together patterns (including percussion, melodies, basses etc) in a quick and easy way, while allowing them to be edited and combined on-the-fly. Most of my live sets revolve around the creation (and destruction) of patterns, and ProxySpace makes this really quite easy. With ProxySpace I can build a performance using multiple self-managing 'instruments' and play them as I build them. By doing this I can think reasonably laterally about the performance, building up and packing down individual "instruments" as I need them, while all of the existing 'instruments' continue playing. It also has some functionality such as automatic crossfading which is very useful for creating smooth performances.

In addition, while I don't use it very much, the `ProxyMixer` class uses SuperCollider's GUI to automatically create a visual mixer to change the levels of all "instruments" created.

I've written two extended examples of how I use ProxySpace which are in this folder. They are musical examples that I would use in live performances I deliver. Open them up in your SuperCollider IDE and follow along. If you are browsing via GitHub Pages, the tutorials can be found here for basics and here for patterns.

ProxySpace (and JITLib in general) also have *great* documentation, which i'd recommend: - ProxySpace Examples - The JITLib Basic Concepts series - JITLib Overview.

Setup Code

/ Making Performance Easier /

In the root directory of this repo, there is a Setup folder, which contains some files, including `Setup.scd`, `SynthDefs.scd` and `Snippets.scd`.

As I mentioned in 'Why SuperCollider', one of my big gripes with SuperCollider and performing with it is the amount of pre-building that needs to be done in order to incorporate any higher level structures, such as playing samples, triggering instruments, and suchlike. This setup folder addresses that problem, and contains my personal SuperCollider performance setup, and can be loaded entirely by either running the `Setup.scd` file, or calling it from somewhere else (for example in line 14 of the second ProxySpace tutorial by specifying the relative filepath to the setup file and using the `.loadRelative` method on it. I can (and have) performed without this setup file, but for the most part I run this setup file before any performance I do.

The `Setup.scd` file does the following things:

1. Increase the number of buffers available for SuperCollider to load;
2. Increase the amount of memory size available to the Server, to allow for more CPU-heavy work;
3. Boot the server Display the server Oscilloscope (which I regularly use as visuals in my set);
4. Start ProxySpace, and make a 60BPM proxy tempo clock;
5. Lines 20-27:
   - Creates a `Dictionary`, `d`, to hold samples;
   - Recursively loads all samples of the correct set in the `samples` folder. These samples are organised into folders which contain the samples. The name of the folder will be added as an entry to the dictionary, and the samples will be added as sub-entries.
     - For example, if you wanted to reference the second sample in the kick drum folder you would use `d["k"][1]` (`d` for the dictionary, `"k"` as kickdrums are held in directory `"k"`, and `1` as you are referencing the second sample);
6. Loads the `SynthDefs.scd` file, containing some custom SynthDefs which I use inside of patterns. Notably the necessary synthdef for playing samples `bplay`, and some instruments such as `sinfb` and `ring1`;
7. Loads the `Snippets.scd` file, which contains some snippets to be loaded into the ddwSnippets Quark, for easy access during performance, which include basic percussion patterns, some functions and some patterns that have a lot of default arguments I might not remember while performing;
8. Starts `StageLimiter` from the BatLib quark, to protect everyone's ears;
9. Posts a message to show all the above have been completed;

Once this setup file has been run, everything is set up to perform, all in one evaluation. The `.loadRelative`s in the Setup file also means if any SynthDefs or Snippets are added and saved, they will be loaded next time the setup file is loaded.

If you're following any examples/etc from this repo, and it doesn't work and I haven't said anything about the setup file, assume that you need to run it for the code to work!

PBinds and Patterns

Introduction

According to SuperCollider's Practical Guide to Patterns:

> Patterns describe calculations without explicitly stating every step. They are a higher-level representation of a computational task. While patterns are not ideally suited for every type of calculation, when they are appropriate they free the user from worrying about every detail of the process. Using patterns, one writes what is supposed to happen, rather than how to accomplish it.

A large part of my live coding performances involve using patterns, specifically Pbinds as Proxies inside of ProxySpace (see section 2.2.1) to create rhythmic elements that are synchronised to ProxySpace's TempoClock. In this case, “rhythmic elements” can mean percussion, melody, bass, pads, or generally anything that is played “in tempo”.

SynthDefs, Arguments and Pbinds

The Pbinds I perform with work in conjunction with a set of SynthDefs (these can be found in the SynthDefs.scd file of the Setup folder) which serve various musical purposes, and plays them with specifies arguments at a given duration. This isn't a particularly intuitive concept to explain, but an example can help illustrate how this works. Take the SynthDef I use the most, `bplay`:

```
SynthDef(\bplay,
    {arg out = 0, buf = 0, rate = 1, amp = 0.5, pan = 0, pos = 0, rel=15;
        var sig,env ;
        sig = Pan2.ar(PlayBuf.ar(2,buf,BufRateScale.ir(buf) * rate,
1,BufDur.kr(buf)*pos*44100,doneAction:2),pan);
        env = EnvGen.ar(Env.linen(0.0,rel,0),doneAction:2);
        sig = sig * env;
        sig = sig * amp;
        Out.ar(out,sig);
}).add;
```

`bplay` is a simple stereo-panned sample player driven by the PlayBuf class, which takes the following arguments:

- `out`: the bus to be played to (this is needed as an argument for the SynthDef to work correctly inside ProxySpace, and I don't usually touch it;
- `buf`: the buffer to be read by the synth (all of which are loaded into dictionary `d` by default;
- `rate`: the speed the sample will be played at (with no compensation for pitch);
- `amp`: how loud the sample is, with 1 being the original volume of the sample;
- `pan`: where the sound is placed in the stereo field, with `0` being centre;
- `pos`: the position from which the sample starts playing, normalised from `0` to `1`, e.g. a value of `0.5` will play the sample from the middle;
- `rel`: in this case specifies how long the server is allowed to keep the instance open before freeing it. Normally the instance will be freed when the sample is finished playing, but in the case of very long samples or samples played backwards this freeing may not occur, leading to server load building in the background due to dead running processes. This default value of 15;

Pbinds also have some arguments that need satisfying:

- `instrument`: the SynthDef that will be used to deliver this 'instance' in the pattern
- `dur`: The duration of each 'instance', if used directly in ProxySpace, a `dur` value of 1 results in an 'instance' once every clock cycle. Note: the default `dur` value of a Pbind is 1, and the default `instrument` value is SuperCollider's built in Piano synth, but specifying both anyway (especially `instrument`) is good practice.

So, if I wanted to have a kick drum playing once each beat in time with the ProxySpace timer, after I had run my setup file I would do the following:

```
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1);
~k.play;
)
```

This Pbind `~k`, spefifies that the instrument it will be using is `bplay`, the buffer `bplay` reads from will be the first index of the `k` entry in the dictionary (which contains kick drums), and that the `dur`/duration is `1`, once per cycle. If any arguments that the SynthDef takes are not specified as part of the Pbind, the SynthDef's default values will be used. Pbind arguments have to be given as key-value pairs, anything else will result in a syntax error, eg:

```
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\rate);
```

As part of these key-value pairs, Pbinds can take Pattern classes as inputs. `Pwhite` gives random values between a minimum and maximum. If I wanted to specify a random pitch of the kick drum, I could add this to the pattern:

```
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\rate,Pwhite(1,1.2));
```

Nesting pattern classes

Pattern classes can also be nested. Here are a few examples of some more complex percussive patterns. Once you start nesting pattern classes, things can get complicated quite quickly.

```
//to play with these examples, make sure the Setup File has been run

//footwork kickdrums
(
p.clock.tempo = 2.4;
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],
\dur,Pbjorklund2(Pseq([3,3,3,5],inf),8)/4,\amp,1,\rate,Pseq([1,1.2],inf));
~k.play;
)

//skittery hi-hats
(
p.clock.tempo = 1.5;
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwrand([0.25,Pseq([0.125],
2),0.5,Pseq([0.125/2],4)],[4,1,1,0.5].normalizeSum,inf),\amp,Pwhite(0.2,1));
~h.play;
)

//offset percussion patterns for techno feel behind a basic kick
(
p.clock.tempo = 135/60;
~c = Pbind(\instrument,\bplay,\buf,d["sfx"][6],
\dur,Pbjorklund2(Pexprand(2,15).round(1),16,inf,Pwhite(1,5).asStream)/4,\amp,
1,\rate,2.2);
~c2 = Pbind(\instrument,\bplay,\buf,d["sfx"][6],
\dur,Pbjorklund2(Pexprand(2,15).round(1),16,inf,Pwhite(1,5).asStream)/4,\amp,
1,\rate,1.9);
~k = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,1,\amp,5);
~c.play;
~c2.play;
~k.play;
)

//snare running forwards and back
(
p.clock.tempo = 150/60;
~sn = Pbind(\instrument,\bplay,\buf,d["s"][4],\dur,Pwhite(1,4)/2,\amp,
1,\rate,Prand([1,-1],inf),\pos,Pkey(\rate).linlin(-2,2,0.9,0));
~sn.play;
)
```

Extra arguments for melody/pitch

Pbinds also have some additional trickery for anything involving pitch.

Let's look at the `sinfb` SynthDef (the arguments are listed in the code block for simplicity)

```
//SinFB Bass
(
SynthDef(\sinfb, {
    arg freq = 440, atk = 0.01, sus = 0, rel = 1, fb = 0, amp = 0.3, out = 0,
pan=0;
    var sig, env;
    env = EnvGen.ar(Env.linen(atk,sus,rel),1,1,0,1,2);
    sig = SinOscFB.ar(freq,fb,1);
    sig = sig*env;
    Out.ar(out,Pan2.ar(sig,pan,amp));
}).add;
);
/*
freq: frequency
atk: attack
sus: sustain
rel: release
fb: phase feedback
amp: amplitude
out: output bus
pan: stereo panning
*/
```

Here, the `freq` argument is the pitch of the oscillator. Pitch can be specified manually, like so:

```
~sinfb = Pbind(\instrument,\sinfb,\dur,0.25,\freq,Pwhite(100,900));
```

However, if a variable in a SynthDef is given the name `freq`, Pbind allows the specification of the following in place of `freq` to activate a "scale mode":

- `scale`: the scale and tuning used - scales can be listed with `Scale.directory` and tunings with `Tuning.directory` (default `Scale.major(\et12)`)
- `degree`: the degree of the scale to be played (default `0`)
- `octave`: the octave of the scale to be played (default `5`)

Only one of these arguments needs to be specified to be in "scale mode", for example:

```
//run up the major scale
~sinfb = Pbind(\instrument,\sinfb,\dur,0.25,\degree,Pseq((0..7),inf));
```

But using all three gives full control over the parameters of the pitch used inside of a musical scale:

```
//run up and down chromatic scale one degree at a time
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.chromatic(\et12),
\degree,Pseq((0..12).pyramid.mirror,inf),\octave,6,\dur,0.125/2,\amp,0.3,\fb,
0.8,\rel,0.1)
```

By using the 'scale mode' of Pbinds you can easily adopt pitch structures that are organised around any scale and tuning you wish - SuperCollider has a bunch bundled in, but way more can be added with the Scala Scale library through quarks such as TuningLib and TuningTheory, and arbitrary scales can be specified.

Why I don't use Pdefs

Another approach to using patterns is to make metapatterns by placing Pbinds (and Pmonos) inside of a `Pdef`, but i've found this to be too verbose to use while performing, and i've personally had some problems getting them to sync for performances that reply on strict metric patterns.

More on Patterns

Patterns form a huge part of my live sets, so I will be referencing them frequently throughout this repo, talking about their use in both rhythmic and melodic arrangement.

3.

Rythm

Rythmic Construction for Algorave Sets

/ Introduction /

Context

In this document I'm going to talk about how I construct some basic rhythms for Algorave sets, but as my perception of rhythm is heavily influenced by the music I listen to, it's probably useful to list some of my influences to give some backdrop to the kinds of reference points I have when making repetitive electronic music designed for use in a dancefloor environment. These may or may not have manifested themselves at any point in any music I have played ever.

- Basic Channel
- DJ Rashad
- Holly Herndon
- some of Mark Fell's performances (although the “disinterested” performance aesthetic *really* doesn't do it for me)
- mobilegirl's mixes
- 'Dark DnB'
- Skepta & Grime
- Drill & Trap (for example a beat by Young Chop)

Conceptualising rhythm in live coding with SuperCollider

A problem I had with rhythm when I first started live coding was how to manage rhythm was the lack of 'cycles' or 'loops'. In DAW environments this is handled by the entire environment being organised around the time signature, and in TidalCycles it is handled by the whole musical language being structures around cycles.

Using Pbinds in SuperCollider, rhythms are specified on an individual basis using numerical representations of durations. I found this problematic as a 'natural' rhythmic progression was lacking, as well as any recognition of a 'time signature'.

As a result of this, I initially found creating rhythms faithful to dance music traditions quite difficult when live coding. Consider this track by minimal techno legend Floorplan, AKA Robert Hood. The rhythms used here are very 'locked-in' to particular parts of a 4/4 groove in order to create a set of sounds that are idiomatically in-tune with Floorplan's particular sound, situated within the tradition of dance music he is creating. I have seen a number of these performances delivered using devices such as a Electribes, which are designed to place particular notes within a grid designed around a 4/4 groove. This approach to rhythm makes sense when designing dance music rhythms as specific rhythmic onsets have to be placed with reasonable precision in order to deliver a groove that is recognisably 'dance music'.

With SuperCollider however, I had to work out a way to specify these rhythms on a per-note basis, which required a bit of thought. It's quite difficult to delivery idiomatic grooves as whole units because they depend on the alignment of a number of drum sounds in particular configurations, with some onsets often falling at parts of a bar that are difficult to specify using `dur` values within Pbinds.

While this is a setback in the instant production of very specific rhythmic units, it has allowed me to develop a more algorithmic approach to rhythm. In a live coding performance I draw upon a set of strategies that deliver rhythms reminiscent of particular aspects of dance music which I will detail in this section, and when appropriately applied these techniques yield grooves that are (to my ears and body) very dance-oriented in their construction—take for example this rehearsal excerpt. The advantage of using Pbinds for rhythm is that the aforementioned strategies can easily be modified to include extended algorithmic elements to extend or modify their functionality. If I want a kick drum that plays every beat 80% of the time, and plays a more complex rhythmic pattern 20% of the time, it is trivial to change:

```
\dur,1
```

to

```
\dur,Pwrand([1,Pbjorklund2(5,16,1)/4],[0.8,0.2],inf)
```

With these kinds of techniques I can create dance music rhythms that algorithmically manage their own repetition (or lack thereof) to create variation in individual parts, which form grooves exhibiting a compound complexity from many small variations.

I am still working on this, and I still find rhythmic complexity a difficult thing to establish in SuperCollider, especially when considering higher-level structures, or constructing rhythms in compound time signatures. This section will serve as a documentation of my continuing journey through making dance music with SuperCollider, with the intention that people will move their bodies to it in whatever way they see fit.

Drum Samples

Another fundamentally important part of my approach to rhythm live coding is the samples that I use. I (for the most part) use drum samples for percussion of any kind for the simple reason that *all of the hard work has already been done, and done well*. I could synthesise my own percussion, but I'm not the greatest as synthesis and my results would probably sound lacklustre at best. If I use samples that have already been recorded, normalised (and possibly mastered) then I can be reasonably sure that they will penetrate a mix - and in adjusting their parameters I can be reasonably sure of their operation. If I used synthesised percussion it might oddly break under certain circumstances, or not cut through a mix for instance. The other advantage of using samples is that their impact on CPU use is reasonably small.

Samples can also give a lot of context to the kinds of sounds that I'm attempting to emulate through performance. For instance, a set of 808 sounds will allow for the kind of 'rattling' hi-hat sounds common to trap and hip-hop, or distorted kicks will make it easy to draw on some gabber at some point.

In using samples I can also store a number of different 'types' of each sound, for instance sub kicks, harder kicks, softer kicks, distorted kicks and pitched kicks. I haven't quite figured out the best system both to store and categorise these samples for use in performance, but I'm getting there.

Basic Rythms

This document will be a list of some basic rhythmic techniques that are designed to deliver a simple, solid rhythmic base. Strategies for modifying these simple units will be detailed in the following document.

Preamble: How to construct rhythms

According to the Pbind docs, duration using Pbinds are determined using the following:

> ***delta:*** The time until the next event. Generally determined by: **dur.** The time until next event in a sequence of events stretch Scales event timings (i.e. stretch $= 2 \Rightarrow$ durations are twice as long)

I generally use `\dur` for basic rhythms, and when Pbinds are placed directly within ProxySpace, the `\dur` argument is in sync with the ProxySpace `TempoClock`, which is specified in `Setup.scd`. This automatic synchronisation is very handy for keeping all of your rhythms running to the same tempo.

As stated in 3.1, for the most part all percussion I use will be sample-based. For playing samples using Pbinds, I have written two simple `SynthDefs` (which can be found in the `SynthDefs.scd` file in setup)—`bplay` and `vplay`. `bplay` is a simple buffer player that takes the following arguments:

- `out`: the bus to be played to (this is needed as an argument for the SynthDef to work correctly inside ProxySpace, and I don't usually touch it;
- `buf`: the buffer to be read by the synth (all of which are loaded into dictionary `d` by default);
- `rate`: the speed the sample will be played at (with no compensation for pitch);
- `amp`: how loud the sample is, with 1 being the original volume of the sample;
- `pan`: where the sound is placed in the stereo field, with `0` being centre;

- `pos`: the position from which the sample starts playing, normalised from `0` to `1`, e.g. a value of `0.5` will play the sample from the middle;
- `rel`: in this case specifies how long the server is allowed to keep the instance open before freeing it. Normally the instance will be freed when the sample is finished playing, but in the case of very long samples or samples played backwards this freeing may not occur, leading to server load building in the background due to dead running processes. This default value of `15`;

`bplay` is a general purpose sample player, which is designed for playing back percussive sounds. It is by far my most used SynthDef, and will almost inevitably be used to build the percussion in my sets.

`vplay` also takes another argument:

- `rel1`: controls the amount of a sample played

which is for playing specific sections of a sample, or to create particular effects by cutting the playing of percussive samples short.

“The” kick

The iconic sound of a 4/4 kickdrum is probably a good starting point. A `dur` of `1` will play a kick on every beat of the clock.

Stored as snippet `kick`.

```
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
```

Alternate-beat snare

Played alongside the 4/4 kick, a snare on every other beat.

Stored as snippet `snare`.

```
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,2,\amp,1);
~sn.play;
```

Basic hi-hat pattern

Quarter-note closed hi-hats with random amplitude, good for fleshing out basic rhythms.

Stored as snippet `hat`.

```
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,0.25,\amp,Pwhite(0.25,1);
~h.play;
```

3/4 note clap

A clap every 0.75 beat. When played against the rhythms above will add a nice polyryhthmic feel.

Stored as snippet `clap`.

```
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,0.75,\amp,1);
~c.play;
```

Off-beat open hi-hat

An open hi-hat played every beat, offset every half. I use it alongside straight kicks for a kick-hat-kick-hat pattern. The sample here also sounds *really* good if it's switched out for some vocal chants. Note the `dur` uses an infinite Pbind inside of another Pbind to offset a regular pattern - this is a complexity of offsetting rhythms in SuperCollider, and is one of the only instances in which I currently do it.

Stored as snippet `oh`.

The offset `dur` is stored as snippet `offbeat`.

```
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][0],
\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,1);
~oh.play;
```

Techniques for Modifying Rhythm

In 3.2 I went over a few basic rhythmic units for some simple dance music rhythms, here I will elaborate on a few of the more simple techniques I use to get a bit of complexity in my rhythms.

Why I don't use (total) randomness

The Pwhite class is a great way to incorporate randomness into patterns, and one of the first things I tried to do when adding complexity to rhythms was to simply randomise them, however the results were often quite disappointing, especially with multiple random rhythms played at once for sounds that are played regularly (i.e. snares, hats):

```
//Random rhythm with Pwhite
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pwhite(1,5.0),\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwhite(0.25,0.75),
\amp,Pwhite(0.2,1));
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pwhite(0.75,2),\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pwhite(1,5.0),\amp,1);
~sn.play;~h.play;~c.play;~t.play;
)
//even with a regular kickdrum the other rhythms don't sound good
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)
```

With rhythms that use random floating point numbers, the durations that are used have no relationship to any central pulse, and will end up cutting across the beat a lot of the time in a way that I feel does not make sense in dance music. Instead, randomness can be incorporated within various techniques (for a great example see the way that Pwhite is used in the section on Euclidean Rhythms), or constrained to fit within a more regular pattern by using methods such as .round (which can be found in the Pattern Documentation).

Here is an example of using methods to constrain `Pwhite` into a form that is more palatable:

```
//same example but with all rhythms constrained
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pwhite(1,5.0).round(1),
\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pwhite(0.25,0.75).round(0.25),\amp,Pwhite(0.2,1));
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pwhite(0.75,2).round(0.75),
\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pwhite(1,5.0).round(0.5),
\amp,1);
~sn.play;~h.play;~c.play;~t.play;
)
//sounds more palatable with everything arranged properly
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)
```

`Pwhite` also only gives floating point results if one of the values specified is a floating point number, so for quick whole-beat durations (especially useful for occasional longer sounds) I use `Pwhite` to generate whole beats:

```
//same example again
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pwhite(1,5.0).round(1),
\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pwhite(0.25,0.75).round(0.25),\amp,Pwhite(0.2,1));
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pwhite(0.75,2).round(0.75),
\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pwhite(1,5.0).round(0.5),
\amp,1);
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~sn.play;~h.play;~c.play;~t.play;~k.play;
)
//added whole note fx, short, medium and long.
(
~fx1 = Pbind(\instrument,\bplay,\buf,d["sfx"][0],\dur,Pwhite(1,5),\amp,1);
~fx2 = Pbind(\instrument,\bplay,\buf,d["fx"][0],\dur,Pwhite(1,10),\amp,1);
~fx3 = Pbind(\instrument,\bplay,\buf,d["lfx"][0],\dur,Pwhite(10,40),\amp,1);
~fx1.play;~fx2.play;~fx3.play;
)
```

Layering

Some great advice I received from a lecturer was “if one of them is good, lots of them will be great” (paraphrased), when talking about the work of zimoun. This works really well when applied to rhythmic percussion, particularly if each layer of similar percussion serves to re-contextualise the last.

When I'm layering rhythms, there are generally a few techniques I employ to make doing so “work", or just to sound better:

- Layer at different pitches:

```
//layering at different pitches - kicks
(
p.clock.tempo = 2.3;
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1,1.2],inf));
~k.play;
)
//kicks at a different pitch. Evaluate this a few times to get different permutations
(
~k2 = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1,1.8],inf)*4);
~k2.play;
)
```

- Layer very slightly different rhythms, rhythmic units of different lengths

```
//layering of slightly different rhythms
//rhythm 1
(
p.clock.tempo = 1.7;
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pseq([1,1,1,0.5],inf),\amp,
1);
~t.play;
)
//rhythm 2, using a different tom for contrast
//also re-evaluating rhythm 1 to get them playing together
(
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pseq([1,1,1,0.5],inf),\amp,
1);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][1],\dur,Pseq([1,1,1,0.25],inf),
\amp,1);
~t2.play;
)
//rhythm 3 for more
(
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pseq([1,1,1,0.5],inf),\amp,
1);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][1],\dur,Pseq([1,1,1,0.25],inf),
\amp,1);
~t3 = Pbind(\instrument,\bplay,\buf,d["t"][2],\dur,Pseq([1,1,1,0.75],inf),
\amp,1);
~t3.play;
)
//kick underneath to illustrate
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],
\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,1,\rate,1);
~oh.play;
~k.play;
)
```

- Layer interlocking or complimentary rhythms

```
//complimentary rhythms:
//the 'polyrhythmic clap' from the Basics example
(
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,0.75,\amp,1);
~c.play;
)
//clap added at a similar rhythm (euclidean 3,8)
(
~c2 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pbjorklund2(3,8)/4,\amp,
1);
~c2.play;
)
```

- Link with StageLimiter to establish rhythms underneath other ones (more on this in the StageLimiter Abuse section)

```
//StageLimiter throttling
//a complex rhythm
(
l = Prewrite(1, // start with 1
        (    1: [0.25,2],
        0.25: [1,0.75,0.1,0.3,0.6,0.1],
        0.1: [0.5,1,2],
        2: [0.5,0.75,0.5,1]
        ), 4);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,l/2,\amp,1,\rate,2);
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l*2,\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,l,\amp,
1,\rate,Pseq([1.2,1.4,1.7],inf));
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,l*4,\amp,1,\rate,0.8);
~ding = Pbind(\instrument,\bplay,\buf,d["ding"][0],\dur,Pwhite(1,5),\amp,
1,\rate,0.2);
~h.play;~c.play;~t.play;~ding.play;~sn.play;
)
//extremely loud kick throttles everything else
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,4,\amp,100,\rate,0.5);
~k.play;
)
```

Pwrand - Weighted distribution and hassle-free controlled randomness

A technique that I started using after being inspired by Trap instrumentals (such as Ace Hood's Bugatti) was hi-hats that snapped between 1/4, 1/8, 1/6 and 1/16th note patterns. The best way I found to do this was to use Pwrand. Pwrand takes an array of items, and will select those items randomly within a weighted distribution, allowing control over the frequency of occurrence of particular elements.

The trap hi-hats looked like this:

```
//trap(ish) hi-hats
//Has a choice of four rhythmic patterns with lesser chance for each, results in a
mostly 0.25-duration hat which can potentially go quite quickly
(
p.clock.tempo = 75/60;
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwrand([0.25,Pseq([0.125],
4),Pseq([0.25]/3,3),Pseq([0.125]/2,4)],[0.6,0.3,0.09,0.01],inf),\amp,1,\rate,
2);
~h.play;
)
```

Pwrand is great to use whenever you want to control the occurrence of particular types of rhythm without explicitly specifying an order for these types of rhythm to occur. A one I've used quite a lot is to inject some variety into kick drums by switching out a straight `dur` of 1 with other values:

```
//occasional variation on 4/4 kick
(
p.clock.tempo = 2.3;
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pwrand([1,Pseq([0.75],
4),Pbjorklund2(3,8,1)/4],[0.9,0.08,0.02],inf),\amp,1);
~k.play;
)
//open hat for reference
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],
\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,1,\rate,1.4);
~oh.play;
)
```

Clipped percussion—stuttering

The SynthDef `vplay` is designed to deliver samples controlled by an envelope which by default is a square - it will abruptly start and stop sample playback according to envelope settings:

```
//cutoff percussion. This Pbind uses (0..100)/100 to split the sample into 100
sections of 0.03 and play over them
(
p.clock.tempo = 2.4;
~perc = Pbind(\instrument,\vplay,\buf,d["fx"][1],\rel,0.03,\dur,
0.25,\pos,Pseq((0..100)/100,inf));
~perc.play;
)
```

This is a useful technique for creating sputtering rhythms out of much longer sound effects or samples, which can be chopped up on-the-fly and recombined around a central rhythm with `vplay`. This approach tends to yield interesting results by each individual sample playback taking on irregular characteristics even when played inside a regular rhythm - some complexity with pretty minimal effort:

```
//sputtering rhythms based on long percussion sounds
//the Prand for \buf is a flattened array of all fx sounds. If it wasn't flat it
would play all sounds from any fx entry all at once
(
p.clock.tempo = 2.3;
~perc = Pbind(\instrument,\vplay,
\buf,Prand([d["fx"],d["sfx"],d["lfx"]].flat,inf),\rel,0.1,\dur,
0.25,\pos,Pwhite(0,0.9),\rate,Pwhite(1,3.0));
~perc.play;
)
//choose from literally every sample there is in d. Buggy because it'll also play
anything else that is in there, but good for a laugh.
(
~perc = Pbind(\instrument,\vplay,\buf,Prand(d.values,inf),\rel,0.1,\dur,
0.25,\pos,Pwhite(0.0,0.9),\rate,Pwhite(1,3.0));
~perc.play;
)
```

Back-and-forth—Pkey and linking values

Pkey is a pattern class used to embed the same value multiple times in the same pattern - for example if the release value of a SynthDef needed to be the same as the duration of the note:

```
Pbind(\instrument,\something,\dur,Pseq([2,3,4,5],inf),\rel,Pkey(\dur));
```

One way to use this in rhythm is to create sample playback that flips back and forth. Due to how the `bplay` SynthDef works, if a buffer is to be played backwards it will need to be started just before the end of the sample as the Synth will release once the sample is finished (for more information see Ugen done-actions). Using the `.linlin` linear scaling method this value can then be scaled into the rate of playback to create a back-and-forth pattern in percussion, shown here on a snare:

```
//back-and-forth snare
(
~sn = Pbind(\instrument,\vplay,\buf,d["s"][0],\dur,Pbjorklund2(Pwhite(1,6),
16)/4,\amp,1,\rate,Prand([-1,1],inf),\pos,Pkey(\rate).linlin(-1,1,0.99,0));
~sn.play;
)
```

.normalizeSum and "keeping it on 1"

Sometimes greater granularity or oddities of rhythm are needed, but still within the confines of some kind of regularity. This can be achieved with the `.normalizeSum` method, which will take an array and normalize all of its elements so that they add up to 1, for example `[10,20,30].normalizeSum` will produce the array `[ 0.16666666666667, 0.33333333333333, 0.5 ]`. This can be used to create arrays inside of `Pseq` that can easily create complex rhythmic bursts that still resolve around the central beat. Particularly useful here is to generate a sequential array and normalize it to create a rhythmic spread:

```
//.normalizeSum rhythmic spread
//spreading 1-20 over four beats
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pseq((1..20).normalizeSum,inf)*4,\amp,Pwhite(0.2,1));
~h.play;
)
//spreading 1-200 over sixteen beats (gives overtone)
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pseq((1..200).normalizeSum,inf)*16,\amp,Pwhite(0.2,1));
//spreading 1-18 over 8 beats
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pseq((1..18).normalizeSum,inf)*8,\amp,Pwhite(0.2,1));
```

\stretch

Another option for rhythmic variation is to use the `\stretch` argument built in to `Pbind`. This argument will multiply the `\dur` argument by `\stretch` to create a final duration which will be used in the pattern. I don't use this too much as it stands (April 2017), but it can be used very effectively

```
//using the \stretch argument - each time a cycle completes change the stretch
duration
//a fake argument is created here - \euclidNum is used to inform both \dur and
\stretch to ensure both work with the same number of onsets
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\euclidNum,Pwhite(1,7),
\dur,Pbjorklund2(Pkey(\euclidNum),8)/4,\amp,1,\rate,Pseq([3,4,5],inf),
\stretch,Pseq([1,0.5,0.25,2],inf).stutter(Pkey(\euclidNum).asStream));
~k.play;
```

Euclidean Rythms

Introduction

Euclidean Rhythms are described in a 2005 paper by Godfried Toussaint entitled 'The Euclidean Algorithm Generates Traditional Musical Rhythms', which describes the organisation of rhythm by placing onsets as evenly as possible within a number of possible spaces using Bjorklund's algorithm. It's not the easiest thing to verbally describe, but this online tool explains it much better, and the paper contains a bunch of illustrated examples.

As mentioned in 3.1, When I was learning how to perform Live Coding I found creating compelling, complex rhythm in SuperCollider quite hard. Euclidean Rhythms and the Bjorklund quark have ended up becoming major fixtures of my performance as a result as they handle a lot of the difficulties i have around developing rhythmic complexity in real-time as part of performance. I've always wanted to be able to make rhythms like DJ Rashad, and using Euclidean Rhythms has got me some way on that quest.

Effort-free rhythmic complexity

The problem I had with rhythm was in the fact that all rhythms for all proxies had to be specified as `dur` values, and each one had to be specified independently. Constructing TidalCycles-like 'riffs' containing multiple percussion samples is really quite hard in SuperCollider. As a result, most rhythms I ended up creating involved either using simple on-beat/off-beat patterns, or constraining a `Pwhite` or `Pexprand` into producing random rhythms in time with the ProxySpace tempo clock, and random rhythms with a uniform distribution generally sound quite boring.

The Bjorklund quark contains a few classes that help in using Euclidean Rhythms. I particularly use `Pbjorklund2`, which takes arguments for:

- `k`: Number of ‘hits’;
- `n`: Number of possible onsets;
- `length`: Number of repeats;
- `offset`: Starting onset in the pattern;

…and using this, outputs an array of durations for use as `dur` values in a pattern, for instance: `Pbjorklund2(3,8)` would produce duration arrays of `[ 3, 3, 2 ]`.

Because `Pbjorklund2` is a pattern class, it can be nested and have its arguments modulated by other pattern classes, using its inputs to generate sequences, rather than single values. In this way, 'random rhythms' create a much more interesting result, as random values will be used to create a network of onsets, which perceptually appear to be very complex interlocking rhythms.

```
//four 'randomised' rhythms, sounds okay.
(
p.clock.tempo = 2.2;
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,Pwhite(0.25,1).round(0.25),
\amp,1);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][1],
\dur,Pwhite(0.25,1).round(0.25),\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][1],
\dur,Pwhite(0.25,1).round(0.25),\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pwhite(0.25,1).round(0.25),
\amp,1);
~k.play;
~sn.play;
~h.play;
~t.play;
)

//four randomised euclidean rhythms with four different samples.
//sounds better, producing a much greater variety of rhythmic forms.
(
p.clock.tempo = 2.2;
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],
\dur,Pbjorklund2(Pwhite(1,8),Pwhite(1,16))/4,\amp,1);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][1],
\dur,Pbjorklund2(Pwhite(1,8),Pwhite(1,16))/4,\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][1],
\dur,Pbjorklund2(Pwhite(1,8),Pwhite(1,16))/4,\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],
\dur,Pbjorklund2(Pwhite(1,8),Pwhite(1,16))/4,\amp,1);
~k.play;
~sn.play;
~h.play;
~t.play;
)
```

Euclidean Rhythms vs 'the beat'

The benefit of using the Bjorklund quark like this is that it also lines up with the regular clock of ProxySpace, allowing for scattered, hypercomplex, undanceable rhythms to be established over time, and then in one movement unified under a regular rhythm, such as a straight kick drum with a `dur` of a subdivision of 1.

Here's an example that's sort-of inspired by the lasting impression that Basic Channel's Phylyps Trak made on me some time ago.

```
//Complex rhythm that obfuscates the central rhythmic centre
(
p.clock.tempo = 1.5;
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pbjorklund2(Pwhite(10,35),
41,inf,Pwhite(0,10).asStream)/8,\amp,Pexprand(0.1,1),\pan,-1);
~h2 = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pbjorklund2(Pwhite(10,35),40,inf,Pwhite(0,10).asStream)/
8,\amp,Pexprand(0.1,1),\pan,1);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],
\dur,Pbjorklund2(Pwhite(1,5),Pwhite(1,32))/4,\amp,1,\rate,Pwrand([1,-1],
[0.8,0.2],inf),\pos,Pkey(\rate).linlin(1,-1,0,0.999));
~ding = Pbind(\instrument,\bplay,\buf,d["ding"][0],
\dur,Pbjorklund2(Pwhite(1,3),25)/4,\amp,0.6,\rate,0.6,\pan,-1);
~ding2 = Pbind(\instrument,\bplay,\buf,d["ding"][0],
\dur,Pbjorklund2(Pwhite(1,3),20)/4,\amp,0.6,\rate,0.7,\pan,1);
~t1 = Pbind(\instrument,\bplay,\buf,d["mt"][0],
\dur,Pbjorklund2(Pseq([1,1,1,Pwhite(10,15,1).asStream],inf),
36,inf,Pwhite(0,2).asStream)/8,\amp,1);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][0],
\dur,Pbjorklund2(Pseq([1,1,1,Pwhite(10,15,1).asStream],inf),
40,inf,Pwhite(0,2).asStream)/8,\amp,1,\rate,2);
~t1.play;~t2.play;~h.play;~h2.play;~sn.play;~ding.play;~ding2.play;
)

//a slightly more rhythmic element, tracing the rhythm out a bit more
(
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,
\degree,Pwrand([0,4],[0.8,0.2],inf),\octave,Pwrand([2,3],[0.8,0.2],inf),\dur,
0.125,\d,0.25,\a,Pexprand(0.0001,200),\pan,0,\amp,1);
~ring1.play
)
```

```
//Add unce unce unce and simmer gently to unify flavours.
(
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,
\degree,Pwrand([0,4],[0.8,0.2],inf),\octave,Pwrand([2,3,4],
[0.6,0.2,0.2],inf),\dur,0.125,\d,0.2,\a,Pexprand(0.02,900),\pan,0,\amp,1);
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,0.5,\amp,2);
~k.play;
)

//offbeat hat because cheesy rhythms are good fun
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],
\dur,Pseq([0.5,Pseq([1],inf)],inf)/2,\amp,1)
~oh.play
)
```

Using offsets

By utilising the `offset` argument of `Pbjorklund2`, small rhythmic elements can be used multiple times with slight variation to pretty powerful effect.

The following example shows what a few basic offsets can do to liven up a very simple rhythmic pattern

```
//working with offsets - doing a lot with a little

//basic kick
(
p.clock.tempo = 2.13;
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)

//Basic 5-16 euclidean rhythm
(
~c = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16)/4,\amp,
0.7);
~c.play;
)
```

```
//add another layer at a different pitch

//NOTE: These two might not sound at the same time even though they are the same
rhythm, as the rhythmic cycle is longer than 1 beat
(
~c2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16)/4,\amp,
0.7,\rate,1.1);
~c2.play;
)

//if you want them to sound together, trigger them together
(
~c2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16)/4,\amp,
0.7,\rate,1.1);
~c = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16)/4,\amp,
0.7);
)

//offset both

//Note: I am using .asStream here, because a standard Pwhite will not work in the
offset argument of Pbjorklund2, as the values need to be embedded as a stream.

//A general rule of mine is that if pattern classes don't work properly,
use .asStream on the end of them and they likely will.
(
~c = Pbind(\instrument,\bplay,\buf,d["t"][0],
\dur,Pbjorklund2(5,16,inf,Pwhite(1,10).asStream)/4,\amp,0.7);
~c2 = Pbind(\instrument,\bplay,\buf,d["t"][0],
\dur,Pbjorklund2(5,16,inf,Pwhite(1,15).asStream)/4,\amp,0.7,\rate,1.1);
~c.play;
~c2.play;
)

//and another, slightly different sample
(
~c3 = Pbind(\instrument,\bplay,\buf,d["t"][1],
\dur,Pbjorklund2(5,16,inf,Pwhite(0,8).asStream)/4,\amp,0.7,\rate,0.9);
~c3.play
)

//now do the same to the kick
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1,1.2],inf));
)
```

```
//another kick, slightly different rhythm
(
~k2 = Pbind(\instrument,\bplay,\buf,d["k"][2],
\dur,Pbjorklund2(3,16,inf,Pwhite(1,10).asStream)/4,\amp,
1,\rate,Pseq([1.1,1.4],inf));
~k2.play;
)

//add sub kick on 1, and you have minimal techno.
(
~sk = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,1,\amp,2);
~sk.play;
)
```

Convergence & Divergence, using variables inside ProxySpace

As I mentioned in my introduction to ProxySpace, ProxySpace reserved all global variables with the format `~variableName`. I use single letter variables (besides `s` which is reserved for the server and `p` which is reserved for ProxySpace) to hold variables for use in patterns. This is handy for a technique that establishes a set of euclidean rhythms like above, and them unifies them under a central rhythm, which can be deviated from during performance.

Here are the basics of the technique. Variable `l` is used to carry a pattern, which is evaluated alongside each pattern that it contains.

```
//give a central rhythm to be used by other patterns
l = Pbjorklund2(Pseq([3,3,3,4,3,3,3,5],inf),8)/4;

//block-execute (Ctrl/Cmd+Enter) between these brackets
(
p.clock.tempo = 2.1;
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1,\rate,0.9);
~c3 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1,\rate,1.1);
~c2 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1);
~c.play;
~c2.play;
~c3.play;
)
```

```
//now individually execute (Shift+Enter) some of these lines to refresh the 'dur'.
Listen for variations in rhythm.

~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1,\rate,0.9);

~c3 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1,\rate,1.1);

~c2 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1);

//if you want to reset, execute the block again
```

Here is a fully fleshed-out example. As `Pwhite` creates random values, each pattern will create random rhythms independently of one another. Then when they are unified under a `Pseq`, they will all sound at the same time. With this technique I build up complex rhythms, then convert them to one single rhythm and texture, which I can then build structures on top of.

```
//A more fleshed-out example
//Start with a random central rhythm, to keep all of the individual parts
//also using a scale as a one-letter variable for quickness

(
p.clock.tempo = 2.32;
l = Pbjorklund2(Pwhite(3,10),16)/4;
e = Scale.chromatic(\et53);
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,
0,\degree,Pwhite(-2,2),\octave,Pwrand([3,4],[0.8,0.2],inf),\dur,l,\d,
0.4,\a,Pexprand(0.5,30),\amp,0.5,\pan,1);
~ring2 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,
0,\degree,Pwhite(-2,2),\octave,Pwrand([3,4],[0.8,0.2],inf),\dur,l,\d,
0.4,\a,Pexprand(0.5,30),\amp,0.5,\pan,-1);
~ring3 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,
0,\degree,Pwhite(-5,5),\octave,Pwrand([4,5],[0.8,0.2],inf),\dur,l,\d,
0.5,\a,Pexprand(0.5,30),\amp,0.5,\pan,0);
~ring4 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,
0,\degree,Pwhite(-5,5),\octave,Pwrand([2,3],[0.8,0.2],inf),\dur,l,\d,
0.2,\a,Pexprand(0.5,200),\amp,0.9,\pan,0);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,l,\amp,1);
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,l,\amp,Pwhite(0.2,1));
~ring1.play;~ring2.play;~ring3.play;~ring4.play;~sn.play;~c.play;~h.play;
)
```

```
//unify all of these rhythms
//sounds very different
//execute individual lines to make them diverge from this pattern
(
p.clock.tempo = 2.32;
l = Pbjorklund2(Pseq([3,8,2,5,9,10,14,3,5,5,4,9,14],inf),16)/4;
e = Scale.chromatic(\et53);
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,
0,\degree,Pwhite(-2,2),\octave,Pwrand([3,4],[0.8,0.2],inf),\dur,l,\d,
0.4,\a,Pexprand(0.5,90),\amp,0.5,\pan,1);
~ring2 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,
0,\degree,Pwhite(-2,2),\octave,Pwrand([3,4],[0.8,0.2],inf),\dur,l,\d,
0.4,\a,Pexprand(0.5,90),\amp,0.5,\pan,-1);
~ring3 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,
0,\degree,Pwhite(-5,5),\octave,Pwrand([4,5],[0.8,0.2],inf),\dur,l,\d,
0.5,\a,Pexprand(0.5,90),\amp,0.5,\pan,0);
~ring4 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,
0,\degree,Pwhite(-5,5),\octave,Pwrand([2,3],[0.8,0.2],inf),\dur,l,
\d,Pexprand(0.2,0.6),\a,Pexprand(1,200),\amp,0.9,\pan,0);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,l,\amp,1);
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,l,\amp,Pwhite(0.2,1))
)

//throw some straight rhythms in to show where the beat lies
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,1,\rate,1,\amp,3);
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],
\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,Pwhite(0.5,1),\rate,0.8);
~k.play;
~oh.play;
)
```

StageLimiter abuse and “The Guetta Effect”

Listen to the chorus of 'Titanium' by David Guetta ft Sia. That 'pumping' sound heard around the kick drums in the synth parts is (probably) a result of Sidechain Compression, an effect that's pretty common in dance music which (essentially) uses the volume of a track to duck the volume of other tracks.

I've found it very helpful to employ this technique at various points during performance to reinforce the dominant rhythmic pulse of a set. Take this rehearsal excerpt for example, where the 'bell' sounds are being 'pumped' by the bass drum, it's not too subtle. Or skip to 1.22 in this glitchy excerpt, the irregular pitched-up clap is literally cutting off the atonal chimes underneath it. There's also the first half of this set where I am attempting to riff on some tropes from Psytrance, using the kick drum to modulate the two interlocking distorted synth riffs that are being played.

With the exception of the 'Psytrance' riff, I almost always achieve this pseudo-sidechaining effect in the most brutal way possible - by abusing StageLimiter. As StageLimiter is just a Limiter.ar on the output, any sounds over an amplitude of 0dB in the mix will reduce the volume of any other sounds in the mix without distorting. As I tend to use percussion that is normalised to 0dB, any percussion that is played with an `\amp` value of greater than `1` will compress the rest of the mix in proportion to the volume that they hit above 0dB. This can range from subtle to completely ridiculous.

Here are a few examples of this.

```
//1:
//a complex polyrhythm - no need to worry about the construction of this.

(
p.clock.tempo = 2.3;
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,0.75,\amp,1);
~c2 = Pbind(\instrument,\bplay,\buf,d["c"][0],
\dur,Pbjorklund2(Pseq([3,3,3,5],inf),8)/4,\amp,1);
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],
\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,
1,\stretch,Pwhite(1,0.25).round(0.25));
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pbjorklund2(Pwhite(3,10),
16),\amp,1);
~t1 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,1/5*4,\amp,1);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,1/9*4.5,\amp,1,\rate,2);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pbjorklund2(Pwhite(10,16),
16)/8,\amp,Pwhite(0.2,1.4));
~fx1 = Pbind(\instrument,\bplay,\buf,d["sfx"][0],
\dur,Pwhite(1,4.0).round(0.5),\amp,1);
~fx2 = Pbind(\instrument,\bplay,\buf,d["sfx"][1],
\dur,Pwhite(1,8.0).round(0.25),\amp,1);
~c.play;~c2.play;~oh.play;~sn.play;~t1.play;~t2.play;~h.play;~fx1.play;~fx2.p
lay;
)

//A 0db kick which doesn't really do anything in the mix
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)

//A >0dB kick which compresses everything else and audibly 'centers' everything
around it because it is so loud.
//There's probably some psychoacoustics involved in this that i'm not qualified to
talk about.
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,4);
~k.play;
)
```

```
//a really *really* loud, very occasional percussion which silences everything else
(slowed down for exaggerated effect)
(
~hugesnare = Pbind(\instrument,\bplay,\buf,d["mt"][0],\dur,Pwhite(8,16),\amp,
4000000,\rate,1);
~hugesnare.play;
)

//2:

//some beautiful pads
//thanks Eli Fieldsteel
(
p.clock.tempo = 2;
(
~chords = Pbind(\instrument,\bpfsaw,
    \dur,Pwhite(4.5,7.0,inf),
    \midinote,Pxrand([
        [23,35,54,63,64],
        [45,52,54,59,61,64],
        [28,40,47,56,59,63],
        [42,52,57,61,63],
    ],inf),
    \detune, Pexprand(0.0001,0.1,inf),
    \cfmin,100,
    \cfmax,1500,
    \rqmin,Pexprand(0.02,0.15,inf),
    \atk,Pwhite(2.0,4.5,inf),
    \rel,Pwhite(6.5,10.0,inf),
    \ldb,6,
    \amp,Pwhite(0.8,2.0),
    \out,0)
);
~chords.play;
)

//pulse them slightly with a low-passed kick
(
~k = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,Pbjorklund2(3,8)/2,\amp,
2);
//Low Pass
~lpfSend = {[~k]};
~lpf = {RLPF.ar(Mix.ar([~lpfSend]),SinOsc.kr(0.1).range(200,100),1)};
~lpf.play;
)
```

```
//eliminate them completely with an absurdly loud low-passed kick (those with
subwoofers be careful!)
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,
9000,\rate,5);
//Low Pass
~lpfSend = {[~k]};
~lpf = {RLPF.ar(Mix.ar([~lpfSend]),SinOsc.kr(0.1).range(100,80),0.3)};
~lpf.play;
)
```

L-systems for Rhythm

L-systems are, according to Wikipedia:

> a parallel rewriting system and a type of formal grammar. An L-system consists of an alphabet of symbols that can be used to make strings, a collection of production rules that expand each symbol into some larger string of symbols, an initial "axiom" string from which to begin construction, and a mechanism for translating the generated strings into geometric structures.

For a good example to visualise what this means, this was one I found very helpful.

I was inspired to start using L-systems for rhythm after hearing one of Renick Bell's Fractal Beats tracks on SoundCloud, and in turn reading his paper about rhythmic density in live coding for the Linux Audio Conference. The approach to rhythm in this Fractal Beats track is unlike any I have heard - the rhythms are complex and don't appear to lock into common divisions of a regular beat, but do not seem to fall into the trappings of being 'random'. This stochastic approach to rhythm appears to yield something interesting and that appears to 'evolve'.

While I have no idea how to use Conductive, there are some useful implementations of L-systems in SuperCollider. Prewrite is SuperCollider's class for implementing L-systems within patterns. Prewrite takes a rule set and an initial axiom, and will expand the axiom within a Pbind.

For example:

```
//L-systems basic example
//use L-system as a duration value for a kickdrum
(
l = Prewrite(1, // start with 1
        (     1: [0.25,2],
            0.25: [3,3,2]/4,
        3/4: [0.25,1,0.125,0.125],
        ), 4);
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,l,\amp,1);
~k.play;
)
```

```
/*

With that grammar:

1 -> 0.25,2 -> 3/4,3/4,2/4 -> 0.25,1,0.125,0.125,0.25,1,0.125,0.125 -> etc.

*/

//much like with the euclidean rhythm convergence/divergence pattern, you can use
variable l for different patterns too
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,l,\amp,
1,\rate,Pseq((1..4)/2,inf));
~sn.play;
)

//and transform it
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,l,
\stretch,Pwhite(0.5,2).round(0.5),\amp,Pwhite(0.2,1));
~h.play;
)

//an off-beat open hat for reference
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],
\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,1);
~oh.play;
)
```

When I use an L-system, I often pre-write it as writing them on-the-fly (essentially just writing a list) can be time consuming.

One advantage of the rewrite system is that individual rhythms that are complex can sound alone, without 'knocking off' the rhythmic structure, keeping the emphasis on the beat, while sounding rhythms that would be hard to insert using something like a `Pwhite` or would be a little more complex to write inside of a `Pbind`.

Take this L-system for example, which generates an array of random rhythms and uses them alongside self-similar structures:

```
(
var rhythm = Array.fill(rrand(4,10),{rrand(1,10)}).normalizeSum * rrand(1,4);
l = Prewrite(1,
    (
       //equal to 2 duration units/beats
        1: #[0.25,0.5,0.5,0.25,2],
        0.25: #[1],
        2: rhythm
),15);
//play a hi-hat with that L-system as a rhythm
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,l,\amp,0.8);
~h.play;
);
```

After some experimentation with trying to integrate L-systems within sets, I ended up recording the release HSPTLFLDHS (GitHub repo for that release can be found here)in October 2017, which exclusively uses multiple variations upon one L-system to create the entire set of rhythms across the release.

```
//L-system for HSPTLFLDHS
(
l = Prewrite(0.25,
    (
        0.25: #[0.25,0.25,0.25,0.5],
        0.5: #[0.25,0.5,0.125,0.125,0.125,0.125],
        0.125: #[0.375,0.125],
        0.375: #[0.375,0.375,1],
        1: #[0.75,0.25],
        0.75: #[16]
),60)
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,l,\amp,1);
//play the L-system, listen for repetition!
~k.play;
);
```

The release uses a similar technique to the 'Convergence and Divergence' described in 3.4, but with the addition of a more organised system for multiplication and with the addition of rhythmic offsets. By using simple variations on one L-system over the course of two tracks the overall integrity of the rhythms are preserved, with recognisable self-similar patterns recurring, but with enough variation that it keeps my interest.

A simplified version of the HSPTLFLDHS setup will be included in the examples for this section. For more detail, see the HSPTLFLDHS repo.

Looping rhythms and samples with the `lplay` SynthDef

A part of rhythmic electronic music that SuperCollider isn't so great at dealing with are loops. In the Pattern library there isn't a defaulting to 'loop'-based musical structures as is the default in DAW environments such as Ableton live.

This is of course extremely powerful, but sometimes for more complex rhythmic forms, loops are a reasonable practical substitute.

I use loops particularly when there are some rhythms that I find hard to articulate by specifying duration values - an example being the classic Amen Break when I'm trying to make some fast Drum-and-bass style music.

For this, I wrote `lplay`, a variation on the `bplay` Synthdef that is ubiquitous in my Live Coding setup.

```
(
SynthDef(\lplay,
    {arg out = 0, buf = 0, amp = 0.5, pan = 0 rel=15, dur = 8;
        var sig,env ;
        sig = Mix.ar(PlayBuf.ar(2,buf,BufRateScale.ir(buf) *
((BufFrames.ir(buf)/s.sampleRate)*p.clock.tempo/dur),1,0,doneAction:2));
        env = EnvGen.ar(Env.linen(0.0,rel,0),doneAction:2);
        sig = sig * env;
        sig = sig * amp;
        out.ar(out,pan2.ar(sig,pan));
}).add;
)
```

`lplay` takes a `dur` value and plays a buffer exactly over the time period speficied by the `dur` value. For example, if you have a 8 beat drum loop, and you created this `Pbind`:

```
p.clock.tempo = 175/60
~loop = Pbind(\instrument,\lplay,\buf,d["breaks175"][0],\dur,16)
```

The loop would play over 8 cycles of the `ProxySpace` `TempoClock` (`p.clock.tempo`). This is achieved by using this equation for the `rate` control:

```
((BufFrames.ir(buf)/s.sampleRate)*p.clock.tempo/dur)
```

Note that the looping is tied to the rate of playback, so the faster the tempo, the faster the sample will be played. If you try and play a 120bpm sample at 175bpm, it will sound very high-pitched! - Be aware of this when using it during performance.

An important note is that you will have to *reload the synthdef* when the tempo is changed if you want looping to work with the updated tempo.

4.

Melody & Pitch

Pitch and Patterns

A preamble - How is pitch handled?

There are a number of different ways to arrange pitch - a brief history of pitch.

For some context, my musical background is in the western classical music tradition, but I regularly use non-'standard' pitch arrangement techniques in my music.

How Patterns handle pitch

Most times I'm specifying pitch for a synth or sound I will be specifying it as part of a Pbind. Pbinds are set up to handle pitch using the `freq` argument of a SynthDef, with various Pbind arguments designed to 'plug in' to create various kinds of pitch structures:

`freq` can be used to specify a raw frequency value, and `detune` is added to it:

```
//freq specifying a raw pitch value
(
~sinfb = Pbind(\instrument,\sinfb,\freq,Pwhite(100,800),\dur,0.1,\amp,
0.3,\fb,0.1,\rel,0.3);
~sinfb.play;
)

//frequency being detuned gradually
(
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..8),inf)*100,\dur,0.1,\amp,
0.3,\fb,0.4,\rel,1,\detune,Pseq((1..400),inf));
)
```

`scale`, `octave` and `degree` work together to easily give the ability to use a specific scale/tuning pitch arrangement inside of a Pbind, for example:

```
//using scales inside of Pbinds
//Minor scale in Just intonation, octave varying between 4 and 6, root note varying
between 0 and 4 each scale repetition.
//\detune can also be used on top of this to detune scale degrees
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.minor(\just),
\root,Pwhite(0,4).stutter(8),\octave,Pwhite(4,6).stutter(8),
\degree,Pseq((0..7),inf),\dur,0.25,\amp,0.3,\fb,1,\rel,0.2);
~sinfb.play;
)
```

Arrays can also be used to create chords:

```
//Chords used by specifying a 2-dimensional array in \degree argument.
//same can be done for the \octave argument
(
~sinfb = Pbind(\instrument,\sinfb,
    \scale,Scale.major,
    \root,0,
    \octave,Pwrand([4,[3,4],[2,3,4]],[0.9,0.08,0.02],inf),
    \degree,Prand([[0,2,4],[2,4,6],[7,2,4],[1,2,3],[0,-2,-4]],inf),
    \dur,Pwhite(5,10),
    \atk,2,\sus,1,\rel,3,\amp,0.3,\fb,0.1);
~modulation = {SinOsc.kr(0.1).range(0.01,1.41)};
~sinfb.play;
~sinfb.set(\fb,~modulation);
)
```

It's important to note that the degrees of a scale start from `0` when using patterns, with `(0..7)` being a full octave of a major or minor scale.

Types of Pitch Arrangement

Major/Minor scales

The bedrock of the western musical canon is major and minor scales. Generally it's taught in British music education that the major scale is a "happy" sound and the minor scale is a "serious" or "sad". Generally though Minor tends to be used in most music I hear on a day-to-day basis, so if I'm going to be drawing on standard musical scale I will use that. For information on what they are from a music theory perspective check this article.

A few good chords to use that will work with the Major and Minor scale very well at any point will be the following:

```
//chords I, IV and V
//in Major and Minor - re-evaluate for a different scale (using the .choose method)
(
~chords = Pbind(\instrument,\bpfsaw,
    \dur,Pwhite(4.5,7.0,inf),
    \scale,[Scale.minor,Scale.major].choose,
    \degree,Pwrand([[0,2,4],[3,5,7],[4,6,8]],[0.5,0.25,0.25],inf),
    \cfmin,100,
    \cfmax,1500,
    \rqmin,Pexprand(0.02,0.15,inf),
    \atk,Pwhite(2.0,4.5,inf),
    \rel,Pwhite(6.5,10.0,inf),
    \ldb,6,
    \lsf,1000,
    \octave,Pwrand([4,3,5],[0.6,0.3,0.1],inf),
    \amp,Pwhite(0.8,2.0),
    \out,0);
~chords.play;
);
```

The chords I, IV and V are fundamental parts of the vast majority of chord progressions in major or minor scales, with chord ii also being very common. If you randomly play these four chords over a random melody of the same (major/minor) scale, it'll sound *pretty* good:

```
//major/minor scale chords with a fairly melody which meanders around the major/minor
scale, but sounds consonant at the vast majority of points
//scale stored in a dictionary key so that it can be used in both Pbinds easily
(
d[\scale] = [Scale.major,Scale.minor].choose;
~chords = Pbind(\instrument,\bpfsaw,
    \dur,Pwhite(4.5,7.0,inf),
    \scale,d[\scale],
    \degree,Pwrand([[0,2,4],[3,5,7],[4,6,8]],[0.5,0.25,0.25],inf),
    \cfmin,100,
    \cfmax,1500,
    \rqmin,Pexprand(0.02,0.15,inf),
    \atk,Pwhite(2.0,4.5,inf),
    \rel,Pwhite(6.5,10.0,inf),
    \ldb,6,
    \lsf,1000,
    \octave,Pwrand([4,3,5],[0.6,0.3,0.1],inf),
    \amp,Pwhite(0.8,2.0),
    \out,0);
~chords.play;
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\root,0,\octave,[4,5],
\degree,Place([0,0,2,[4,5,6],[7,1,2],[6,7,8,9],[10,12,14,15],7,6,5],inf),
\dur,Pbjorklund2(Pwhite(6,8),8)/4,\amp,0.4,\fb,0.9,\rel,0.2);
~sinfb.play
);
```

The Major and Minor Pentatonic scales are also good for 'sounding good', and are very popular on Guitar for easily creating solo lines.

ChordSymbol - chord notation in SuperCollider

If you have a specific set of chords you would like to play using Patterns, the ChordSymbol addon by triss is a great way to do this, with the chords in arrays I specified in the previous section replaced by a dictionary of chord names, which are automatically translated into their note values. This is very useful if you're working with an instrumentalist and you're not too quick in translating numbers to named chords (which I am not)

```
//ChordProg - house chords with chord names in an array to make a chord sequence...
//Today is gonna be the day that they're gonna throw it back to you...
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.chromatic,\octave,
4,\degree,Pseq([\Em7,\G,\Dsus4,\A7sus4].chordProg,inf).stutter(6),\dur,
1,\atk,0.8,\amp,0.3,\fb,0.1,\rel,1);
~sinfb.play
)

//giant steps. Apparently.
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.chromatic,\octave,
4,\degree,Pseq([\Bmajor7,\D7,\Gmajor7,\Bb7,\Ebmajor7,\Am7,\D7,\Gmajor7,\Bb7,\
Ebmajor7,\Gb7,\Bmajor7,\Fm7,\Bb7,\Ebmajor7,\Am7,\D7,\Gmajor7,\Dbm7,\Gb7,\Bmaj
or7,\Fm7,\Bb7,\Ebmajor7,\Dbm7,\Gbm7].chordProg,inf),\dur,1,\atk,0.1,\amp,
0.3,\fb,0.1,\rel,1);
~sinfb.play;
)

//a musical example in context - Adapted from a set for Manchester Algorave
(
p.clock.tempo = 180/60;
~chords = Pbind(\instrument,\bpfsaw,
    \dur,Pwhite(9.5,15.0,inf),
    \scale,Scale.chromatic,
    \degree,Pxrand([\Em,\Am7,\Bm7].chordProg,inf),
    \cfmin,100,
    \cfmax,1500,
    \detune,Pexprand(0.0001,1),
    \rqmin,Pexprand(0.02,0.15,inf),
    \atk,Pwhite(2.0,4.5,inf),
    \rel,Pwhite(6.5,10.0,inf),
    \ldb,13,
    \lsf,1000,
    \octave,Pwrand([4,5,6],[0.8,0.15,0.05],inf),
    \amp,Pwhite(0.8,1.5),
    \out,0);
~chords.play;
~oh = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pbjorklund2(Pwhite(10,16),16)/4,\amp,0.4,\pan,0.2,\rate,Pwhite(1.7,2));
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(Pwhite(10,16),
16)/4,\amp,0.8,\pan,-0.2,\rate,2);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(Pwhite(10,16),
16)/4,\amp,0.8,\pan,-0.2,\rate,4);
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(Pwrand([3,6],
[0.8,0.2],inf),8)/4,\amp,1);
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,4,\amp,4);
~oh.play;~t.play;~k.play;~c.play;~t2.play;
)
```

Chromatic Scales

Microtonal/Alternative scales

SuperCollider has a bunch of built-in scales (which can be found by evaluating `Scale.directory`), all of which can be used in patterns by using them as part of the `\scale` argument.

```
//Alternative scales
//Evaluate to select a scale using the ET12 tuning and run it in ascending order,
there are a number of scales so evaluate this a bunch of times
//scales are stored in a dictionary to be referred to multiple times within the
~sinfb pbind
(
p.clock.tempo = 1;
d[\scale] = Scale.choose.postln;
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\octave,
4,\degree,Pseq((0..d[\scale].degrees.size-1),inf),\dur,0.25,\amp,0.3,\fb,
0.6,\rel,0.3);
~sinfb.play;
)
```

```
//Microtonal scales
(
p.clock.tempo = 1;
d[\scale] =
[Scale.zamzam,Scale.chromatic24,Scale.partch_o1,Scale.husseini,Scale.zanjaran
,Scale.bhairav].choose.postln;
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\octave,
4,\degree,Pseq((0..d[\scale].degrees.size-1),inf),\dur,0.25,\amp,0.3,\fb,
0.6,\rel,0.3);
~sinfb.play;
)
```

Alternative tunings

SuperCollider also has a bunch of built-in tunings (which can be found by evaluating `Tuning.directory`). These are specified as part of the `scale` argument after the scale that is used.

```
//Alternative Tunings
//Chromatic scale in a random tuning - some relatively subtle differences here
(
p.clock.tempo = 1;
d[\scale] = Scale.chromatic(Tuning.choose);
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\octave,
4,\degree,Pseq((0..d[\scale].degrees.size-1),inf),\dur,0.25,\amp,0.3,\fb,
0.6,\rel,0.3);
~sinfb.play;
)

//A musical example of alternative tunings
//one of my favourites is the et53 tuning, using it to slightly disturb a central
pitch on multiple instruments, sounds really nice in acid-type music
//by selectively deploying et53, a very narrow pitch range can become normal, making
large pitch leaps within an octave seem huge when used.
(
p.clock.tempo = 150/60;
d[\scale] = Scale.chromatic(\et53);
l = Pbjorklund2(Pwhite(1,13),16)/4;
//notice the \degree argument - ranges from -8 to +8, but this difference is nowhere
near an octave
~ring3 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,d[\scale],
\degree,Pwhite(-8,8),\octave,Pwrand([2,3],[0.8,0.2],inf),\dur,l,\d,
0.24,\a,Pexprand(10,400),\pan,0,\amp,1.5);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][1],\dur,l,\amp,0.8);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][1],\dur,l,\amp,0.8);
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,1,\amp,2);
~ring3.play;~sn.play;~h.play;~k.play;
)

//adding more acid lines which diverge even less. Also adding percussion
(
~ring2 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,d[\scale],
\degree,Pwhite(-4,4),\octave,5,\dur,l,\d,0.37,\a,Pexprand(1,40),\pan,1,\amp,
0.5);
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,d[\scale],
\degree,Pwhite(-4,4),\octave,4,\dur,l,\d,0.38,\a,Pexprand(1,40),\pan,-1,\amp,
0.5);
~ring2.play;~ring1.play;
)

//another acid line that diverges quite a bit. also open hats
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],
\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,2);
~oh.play;
~ring4 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,d[\scale],
\degree,Pwhite(-8,8),\octave,7,\dur,l,\d,0.21,\a,Pexprand(1,100),\pan,1,\amp,
0.8);
~ring4.play;
)

//repetive distorted \sinfb riff, using the whole octave
(
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\octave,[5,6],
\degree,Place([0,0,-52,[30,20,10],[52,40,25,20],[10,11,9,3,6],
[30,36,39,40]],inf),\dur,0.25,\amp,0.5,\fb,Pwhite(10.5,900.5),
\rel,Pexprand(0.1,0.5));
~sinfb.play;
)

//remove percussion
(
~k.stop;~sn.stop;~h.stop;
)
```

Harmonic (overtone) series

From Wikipedia:

> A harmonic series is the sequence of sounds where the base frequency of each sound is an integer multiple of the lowest base frequency

I generally use the Harmonic Series in SuperCollider by setting a fundamental (base) frequency as a NodeProxy and referring other NodeProxies to it. This way all of the playing elements can follow the same fundamental frequency, and the fundamental frequency can be modulated.

```
//Harmonic series
//setting up a fundamental frequency as a NodeProxy so that it can be referenced on
the fly
(
~r = {75}
)

//a straight run up the harmonic series to 10 partials. Notice how the notes converge
the higher up the harmonic series due out perception of frequency being logarithimic
//note that the \freq argument is a multiplation of a Pkr - a BenoitLib addon which
references an active NodeProxy inside of a pattern
(
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..10),inf)*Pkr(~r),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb.play;
)

//modulate the fundamental frequency to modulate the entire scale
(
~r = {SinOsc.kr(0.1).range(75,80)}
)

//raise the fundamental freqency from 75Hz to 1000Hz over two minutes
(
~r = {XLine.kr(75,1000,120)}
)
```

The 'sound' of the harmonic series is different to scales, as the further up the harmonic series is played (or the more times the fundamental frequency is multiplied), the closer the intervals 'sound' to each other:

```
//a run up the harmonic series from 1 to 50 partials - note how close together the
notes become
(
~r = {50};
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..50),inf)*Pkr(~r),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb.play;
)
```

This can be changed by changing the granularity of the multiplication of the fundamental frequency:

```
//Multiple identical harmonic frequency riffs that use a different multiplication of
the fundamental frequency
(
~r = {50};
//1x fundamental
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb.play;
)

(
//2x fundamental
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*2),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb2.play;
)

(
//4x fundamental
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*4),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb3.play;
)
```

```
(
//8x fundamental
~sinfb4 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*8),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb4.play;
)

//all together to 30:
(
~r = {50};
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*2),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*4),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb4 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*8),\dur,
0.25,\amp,0.3,\fb,0.1,\rel,0.3);
)
```

Riffs

Examples in music

A riff is a short, repeated musical phrase that is used as an anchor or a refrain in a piece of music.

I've always been drawn to guitar music with riffs, and riff-heavy electronic music is no exception. A *great*example of riff-heavy live coding is the music of Belisha Beacon's, who makes a network of shifting riffs using ixi lang.

Here are a few ways I use riffs:

The 'up-down' riff

A technique I've probably ended up using an awful lot is an 'up-down' riff, which is a way of producing a set of interlocking riffs very quickly on the spot. It can be used with any form of pitch organisation, but more common scales and the harmonic series tend to work the best.

The 'up-down' riff uses SuperCollider's `range` method to generate a sequential set of degrees of a scale playing on a SynthDef and running it alongside the same set of degrees `.reverse`-d, creating a palindrome which runs continuously. A third layer, which uses SuperCollider's `.scramble` method to create a random riff to play against the 'up-down' riff, all played in a uniform rhythm:

```
//up-down riff
//harmonic series version
//re-evaluate individual directions to create a different riff
(
//up
p.clock.tempo = 1.5;
~r = {75};
~sinfb1 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10),inf)*Pkr(~r),\dur,
0.25,\amp,0.3,\fb,Pwhite(0.1,0.4),\rel,0.3);
~sinfb1.play;
)
```

```
(
//down
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).reverse,inf)*Pkr(~r),
\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,0.4),\rel,0.3);
~sinfb2.play;
)

(
//random
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).scramble,inf)*Pkr(~r),
\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,1.0),\rel,0.3);
~sinfb3.play;
)

//up-down riff
//minor scale version
//re-evaluate individual directions to create a different riff
(
p.clock.tempo = 1.5;
//up
~sinfb1 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
5,\degree,Pseq((0..7),inf),\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,0.4),\rel,0.2);
~sinfb1.play;
)

(
//down
~sinfb2 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
5,\degree,Pseq((0..7).reverse,inf),\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,0.4),
\rel,0.2);
~sinfb2.play;
)

(
//random, an octave higher
~sinfb3 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
6,\degree,Pseq((0..7).scramble,inf),\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,1.0),
\rel,0.2);
~sinfb3.play;
)
```

An important part of this technique is that by re-evaluating individual riffs the overall structure of the riffs as a whole can be changed, giving the resulting sound a different character each time.

It can also be combined with some `Pwrand` based probabilistic rhythmic change to automatically shift the character of the riff:

```
//replacing duration of 0.25 with a Pwrand which will automatically shift the riffs
(
p.clock.tempo = 1.5;
~sinfb1 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
5,\degree,Pseq((0..7),inf),\dur,Pwrand([0.25,Pseq([0.125],2)],[0.9,0.1],inf),
\amp,0.3,\fb,Pwhite(0.1,0.4),\rel,0.2);
~sinfb2 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
5,\degree,Pseq((0..7).reverse,inf),\dur,Pwrand([0.25,Pseq([0.125],2)],
[0.9,0.1],inf),\amp,0.3,\fb,Pwhite(0.1,0.4),\rel,0.2);
~sinfb3 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
5,\degree,Pseq((0..7).scramble,inf),\dur,Pwrand([0.25,Pseq([0.125],2)],
[0.9,0.1],inf),\amp,0.3,\fb,Pwhite(0.1,1.4),\rel,0.2);
)

~sinfb1.play;
~sinfb2.play;
~sinfb3.play;
```

"Phasing"

"Phasing" was used extensively by Steve Reich in his early works, and refers to two or more similar or identical musical forms which are played at slightly differing tempi so that they shift and begin to interfere with each other (more information).

There are a few ways to emulate this during sets, both through subtle interference with playing riffs, rhythmic disturbances and omitting notes. Another example can be seen in the section on Euclidean Rhythms and Offsets.

```
//Phasing
//Using the riff from Reich's Piano Phase
//inspired by https://ccrma.stanford.edu/courses/tu/cm2008/topics/piano_phase/
index.shtml
(
p.clock.tempo = 1.8;
//riff 1 and 2 evaluated at once so that they start together.
//riff 2 will sometimes play 0.125 duration which will knock the two out of phase
~sinfb1 = Pbind(\instrument,\sinfb,\octave,4,\freq,Pseq([64, 66, 71, 73, 74,
66, 64, 73, 71, 66, 74, 73].midicps,inf),\dur,0.25,\amp,0.3,\fb,0.1,\rel,
0.3);
~sinfb2 = Pbind(\instrument,\sinfb,\octave,4,\freq,Pseq([64, 66, 71, 73, 74,
66, 64, 73, 71, 66, 74, 73].midicps,inf),\dur,Pwrand([0.25,0.125],
[0.99,0.01],inf),\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb1.play;
)
//play riff 2
~sinfb2.play;

//another version which uses a second riff which has a slightly different tempo
constantly
(
p.clock.tempo = 1.8;
//riff 1 and 2 evaluated at once so that they start together.
//riff 2 will sometimes play 0.125 duration which will knock the two out of phase
~sinfb1 = Pbind(\instrument,\sinfb,\octave,4,\freq,Pseq([64, 66, 71, 73, 74,
66, 64, 73, 71, 66, 74, 73].midicps,inf),\dur,0.25,\amp,0.3,\fb,0.8,\rel,
0.3);
~sinfb2 = Pbind(\instrument,\sinfb,\octave,4,\freq,Pseq([64, 66, 71, 73, 74,
66, 64, 73, 71, 66, 74, 73].midicps,inf),\dur,0.255,\amp,0.3,\fb,0.1,\rel,
0.3);
~sinfb1.play;
)
//play riff 2
~sinfb2.play;
```

Sample stabs

Another way to make riffs is to use pitched samples, and define the pitch of the riff using the `\rate` argument of `bplay`.

A version of this I use quite a lot is derived from '90s rave music:

```
//synth stabs - try this with both stab 0 and 1.
(
//stab 1
p.clock.tempo = 2.4;
~stab1 = Pbind(\instrument,\bplay,\buf,d["stab"][1],\euclidNum,Pwhite(3,3),
\dur,Pbjorklund2(Pkey(\euclidNum),8)/4,\amp,
2,\rate,Pseq([1,1,1,1,1,1,0.9,1.1],inf).stutter(3));
~stab1.play;
)

(
//stab 2 - double speed and greater possible number of onsets
~stab2 = Pbind(\instrument,\bplay,\buf,d["stab"][1],\euclidNum,Pwhite(3,11),
\dur,Pbjorklund2(Pkey(\euclidNum),16)/4,\amp,
1,\rate,Pseq([1,1,1,1,1,1,0.9,1.1],inf).stutter(3)*2);
~stab2.play;
)

(
//stab 3 - double speed again and greater possible number of onsets again
~stab3 = Pbind(\instrument,\bplay,\buf,d["stab"][1],\euclidNum,Pwhite(6,16),
\dur,Pbjorklund2(Pkey(\euclidNum),16)/4,\amp,
1,\rate,Pseq([1,1,1,1,1,1,0.9,1.1],inf).stutter(3)*4);
~stab3.play;
)

//drums
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1.1,1.9],inf));
~k2 = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1.1,1.9],inf)*2);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pbjorklund2(Pwhite(1,6),
16)/4,\amp,1);
~fx = Pbind(\instrument,\bplay,\buf,d["fx"][0],\dur,Pwhite(1,6),\amp,1);
~k.play;~sn.play;~fx.play;~k2.play;
)
```

Place and compound riffs

Place is "interlaced embedding of subarrays". Simply put, if you put a riff inside of another riff (or an array inside of another array), the first level of the array will be played over, and each subsequent value of the subarrays will be iterated over once every time the first level is played. This is really difficult to explain, so have a look at the first numerical example of the Place documentation for this one. Here is an example of how two riffs can be layered together using Place:

```
//Place - riffs that contain riffs
(
//first riff
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,
\degree,Place([0,7],inf),\octave,3,\dur,0.25,\d,0.6,\a,Pseq((1..40),inf),
\pan,0,\amp,0.5);
~ring1.play;
)
//stop
~ring1.stop;

(
//second riff
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,
\degree,Place([2,4,3,5,4,6,8,11],inf),\octave,3,\dur,0.25,\d,
0.6,\a,Pseq((1..40),inf),\pan,0,\amp,0.5);
~ring1.play;
)
//stop
~ring1.stop;

(
//two riffs laced together with the longer one on the inner level, playing the first
riff and then a note of the second
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,
\degree,Place([0,7,[2,4,3,5,4,6,8,11]],inf),\octave,3,\dur,0.25,\d,
0.6,\a,Pseq((1..40),inf),\pan,0,\amp,0.5);
~ring1.play
)
```

Pitch and "Static Synths"

Outside of patterns, pitch is handled primarily using the `freq` argument of UGens - for example:

```
~sin = {SinOsc.ar(440,0,0.1)};
~sin.play;
```

With `440` being the frequency.

This `freq` argument can easily be fitted to the harmonic series by using multiplication and the `.range` and `.round` methods applied to various Ugens:

```
//set a fundamental frequency
~f = {70}

//a fixed pitch sine wave, using a fundamental frequency
(
~sin = {SinOscFB.ar([~f,~f*1.01],0.7,0.3)};
~sin.play;
)

//4 saw waves that are modulated by LFNoise1 Ugens and arranged around the stereo
//field the frequency of the saw waves is a LFNoise1 that is ranged between the
//fundamental and ten times the fundamental
(
~lfn1 = {Splay.ar(Saw.ar(Array.fill(4,{LFNoise1.kr(0.3).range(~f,~f*10)}),
0.3))}
~lfn1.play;
)

//now round this LFNoise1 to the fundamental frequency to get the frequency to sweep
the harmonic frequency
(
~lfn1 = {Splay.ar(Saw.ar(Array.fill(4,
{LFNoise1.kr(0.3).range(~f,~f*10).round(~f)}),0.3))}
~lfn1.play;
)
```

```
//the frequencies are now tuned and sound GREAT (an X/Y scope also looks amazing)
s.scope

//This .range and .round method can be applied to any signal UGen, and also at any
multiplication level. Here's a silly extreme example that sounds like shrill bees
(
~lfn1 = {Splay.ar(Saw.ar(Array.fill(40,
{SinOscFB.kr(rrand(0.1,0.3),rrand(0.1,2)).range(~f,~f*100).round(~f*4)}),
0.4))}
~lfn1.play;
)

//Triggered random frequency changes, using something like TRand
(
~f = {81};
~tChange =
{Pulse.ar(TRand.kr(~f,~f*10,Dust.kr(4)).round(~f),SinOsc.kr(0.1).abs,
0.6)*SinOsc.ar([~f,~f*1.01])};
~tChange.play;
)

//specific and on-demand frequency changes using Demand.kr - Note that this is
*really* verbose for something to be used live.
//I've used an Impulse.kr that recieves the tempo clock as a trigger to show how
these synths can be synced to a central tempo clock
//Demand is a lot like having a Pattern inside of a UGen's arguments. Look at the
helpfile, it's really cool
(
~f = {66.6};
~dChange = {SawDPW.ar([~f,~f*1.02]*Demand.kr(Impulse.kr(p.clock.tempo*3),
0,Dseq([1,8,2,7,3,6,4,5],inf)),SinOsc.kr(40),0.8)};
~dChange.play;
)

//and a kick to show it's synced
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,1,\amp,1);
~k.play;
)
```

```
//and more kicks because i really liked this one
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(Pwhite(1,15),
16)/6,\amp,2,\rate,Pwrand([1,1.2,1.4,2],[0.6,0.2,0.1,0.1],inf)*1.5);
~k2 = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,1,\amp,2);
~k2.play;
)
```

Scales are a bit of a pain to use outside of patterns, but it's possible using the Scale class and its degreeToFreq method, although it is quite inflexible

```
//Scale and DegreeToFreq
//using the Demand example again
//a fifth
(
~scale = {SinOscFB.ar(Scale.minor(\just).degreeToFreq([0,4],48.midicps,1),
0.7,0.2)};
~scale.play;
)

//Note that the above does not allow scale notes to be changed once the synth is
initiated
~scale =
{SinOscFB.ar(Scale.minor(\just).degreeToFreq(TRand.kr(1,10,Impulse.kr(1)),
48.midicps,1),0.7,0.2)};
```

`.midicps` can also be used, if you know the MIDI note numbers of a scale that you want to play:

```
//using .midicps to determine pitch
~scale = {SinOscFB.ar(TRand.kr(50,80,Impulse.kr([3,3.01])).midicps,0.7,0.5)};
~scale.play
```

Between Pitch and Noise

Preamble

An important corollary when talking about pitch is to talk about unpitched sound or noise. In periods of music dominated by pitched sounds, disintegration or erosion of pitch into noise can be an important technique to drive a set forward, or just provide sonic interest. I find a lot of this kind of thing in the transformations of instruments within Holden's Music for example. Here are some techniques to achieve this.

SinOscFB

A Ugen I use a lot (read: far too much) is SinOscFB, a 'sine oscillator that has phase modulation feedback'. I've always been a big fan of bare sine waves, and SinOscFB's `feedback` argument allows a sine wave to be modulated into noise and back very easily, with extreme modulations creating a strange-sounding degraded sine wave.

```
//SinOscFB - A sine wave that can move between pitch and noise and noisy pitch
(
//polling the modulation of the 'feedback' argument, to show the way in which
SinOscFB degrades sine waves
~sinfbstatic = {SinOscFB.ar([330,440],XLine.kr(0.1,500,60).poll(10),0.6)};
~sinfbstatic.play;
)
```

A stalwart of my SynthDef arsenal is `sinfb`, a `SinOscFB` Ugen inside of an `Env.perc` which is used to control its amplitude curve. This SynthDef is very flexible - great for basses, melodies and chords, but also great for flexibly turning melodic riffs into textural noise, as well as blending the two. Notice that from values `0.0` to `20.0` there is a full spectrum from clean sine wave to noise feedback, any values above `30.0` will blend the two and are what I would consider 'extreme modulation'. In general usage during sets I tend to use the range 0.0 to 3.0, as anything above tends to be too noisy and interferes with the percussion i'm using.

```
//a pattern I use regularly with its feedback being modulated from 0 to 20. Notice
the difference in sound across the spectrum
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,[3,4,5],
\degree,Pseq([0,0,4,5],inf),\dur,Pbjorklund2(3,8)/4,\amp,0.3,\fb,0.1,\rel,
0.3);
~feedback = {SinOsc.kr(0.1,-1,1).range(0,20.0).poll(30)};
~sinfb.set(\fb,~feedback);
~sinfb.play;
)
```

Harmonic series and extreme pitch values

In 4.2 I talked about the Harmonic Series. An interesting quality of using a fundamental frequency to determine the pitch of various NodeProxies by multiplying that fundamental frequency to create a scale structure.

Some interesting techniques for distorting this harmonic series technique into the territory of noise are extreme modulation, which pushes the frequency into supersonics (and sometimes back again):

```
//Extreme modulation of fundamental frequency
//taking the up-down scale given in the 'riffs' section
( //up
p.clock.tempo = 2.4;
~r = {75};
~sinfb1 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10),inf)*Pkr(~r),\dur,
0.25,\amp,0.3,\fb,Pwhite(0.1,1.4),\rel,0.1);
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).reverse,inf)*Pkr(~r),
\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,1.4),\rel,0.1);
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).scramble,inf)*Pkr(~r),
\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,2.0),\rel,0.1);
~sinfb1.play;~sinfb2.play;~sinfb3.play;
)

//moving the frequency up and beyond sensible into supersonics - after reading around
5000Hz some interesting aliasing starts to happen
(
~r = {XLine.kr(75,8000,60).poll(10)}
)
```

```
//and even further, lower frequencies start reappearing
(
~r = {XLine.kr(8000,30000,60).poll(10)};
)

//using very extreme modulation also gives some interesing results
(
~r = {LFNoise1.kr(0.2).range(30000,90000).poll(10)};
)
```

And extreme pitch values - which appear to rise continually into supersonic frequencies and aliasing, and then looping back to the bottom of the pitch scale:

```
//extreme multiplaction of fundamental frequency
//using the previous example, a NodeProxy holding a second multiplier is added onto
the \freq argument of each Pbind
(
~r = {75};
~mult = {1};
~sinfb1 = Pbind(\instrument,\sinfb,
\freq,Pseq((1..10),inf)*(Pkr(~r)*Pkr(~mult)),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,1.4),\rel,0.1);
~sinfb2 = Pbind(\instrument,\sinfb,
\freq,Pseq((1..10).reverse,inf)*(Pkr(~r)*Pkr(~mult)),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,1.4),\rel,0.1);
~sinfb3 = Pbind(\instrument,\sinfb,
\freq,Pseq((1..10).scramble,inf)*(Pkr(~r)*Pkr(~mult)),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,2.0),\rel,0.1);
~sinfb1.play;~sinfb2.play;~sinfb3.play;
)

//increase the multiplcation over time using a .round on a Line.kr UGen. Listen to
how the scale is distorted as the multiplcation increases, eventually ending as a
series of pulses
(
~mult = {Line.kr(1,60,60).round(1).poll(5)}
)
```

Chaos UGens

SuperCollider has support for UGens that use Chaos Theory for synthesis - the Chaos UGens (note that there are also a number of additional Chaos UGens in sc3-plugins which are worth having).

While (at the time of writing) I don't know a whole lot about the particularities of chaos theory works, but the Chaos UGens are great for creating musical structures that move freely between pitched sound and noise, and these are usually handled both in the equation variables of the UGens as well as the initial conditions.

I'll use HenonN as an example of the use of chaos theory to move between melody, noise and percussion:

```
//HenonN - Chaos synths and moving between pitch and noise
(
//henon using a minor pentatonic scale at a high octave.
//The chaos Ugens will need some experimentations if you want subtle variance in
sound
//For Henon I found that an a value of 1.3 and a b value of 0.3 renders a pitch in a
pattern pretty reliably
//note that the pitches aren't quite the same as 'concert pitch'
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,
\degree,Pseq([0,2,4,6,7],inf),\octave,8,\dur,Pbjorklund2(3,8)/
4,\a,Pexprand(1.3,1.3),\b,Pexprand(0.3,0.3),\atk,0,\sus,
0,\rel,Pexprand(0.1,0.1),\amp,1);
~henon.play;
)

//increase the variation in the a and b arguments to add more noise to the mix
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,
\degree,Pseq([0,2,4,6,7],inf),\octave,8,\dur,Pbjorklund2(3,8)/
4,\a,Pexprand(1.3,1.31),\b,Pexprand(0.3,0.31),\atk,0,\sus,
0,\rel,Pexprand(0.1,0.1),\amp,1);
)
```

```
//notice that this gets very noisy VERY fast.
//adding a little more possiblity to the Pexprands in a and b turns it into pure
noise very very fast, while still retaining a little of its pitched character
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,
\degree,Pseq([0,2,4,6,7],inf),\octave,8,\dur,Pbjorklund2(3,8)/
4,\a,Pexprand(1.3,1.35),\b,Pexprand(0.3,0.35),\atk,0,\sus,
0,\rel,Pexprand(0.1,0.1),\amp,1);
)

//even more and noises become cut off and non-sounding.
//the cut off sounds would sound as DC bias, but the SynthDef \henon has a LeakDC on
its output to prevent this as it can damage sound systems and is generally quite an
unpleasant thing to deal with.
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,
\degree,Pseq([0,2,4,6,7],inf),\octave,8,\dur,Pbjorklund2(3,8)/
4,\a,Pexprand(1.3,1.45),\b,Pexprand(0.3,0.55),\atk,0,\sus,
0,\rel,Pexprand(0.1,0.1),\amp,1);
)

//at this point decreasing the \dur and \rel value turns it into rhythmic percussion
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,
\degree,Pseq([0,2,4,6,7],inf),\octave,8,\dur,0.25,\a,Pexprand(1.3,1.45),
\b,Pexprand(0.3,0.55),\atk,0,\sus,0,\rel,Pexprand(0.01,0.1),\amp,1);
)

//more extreme possible values - \dur varied, octaves doubled up, more variation in a
and b values, more octaves
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,
\degree,Pseq([0,2,4,6,7],inf),\octave,[8,12,9,10],
\dur,Pwrand([0.25,Pbjorklund2(Pwhite(3,5),8,1)/4,Pseq([0.125],4)],
[7,4,1].normalizeSum,inf),\a,Pexprand(1.2,1.55),\b,Pexprand(0.21,0.55),\atk,
0,\sus,0,\rel,Pexprand(0.01,0.6),\amp,1);
)

//against a kick drum it takes on a really strange character
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)
```

A thing to note about the Chaos synths is the type of interpolation used - taking Henon as an example; HenonC, HenonL and HenonN stand for Cubic, Linear and None respectively. The sonic effect of the type of interpolation used is in the 'smoothness' of the sound, with Cubic being the most smooth and None being the least.

```
//sound of different types of interpolation
//the default in my SynthDefs.scd file is currently to use none:
(
SynthDef(\henon,
    {arg
freq=440,mfreq=440,a=1.3,b=0.3,x0=0.30501993062401,y0=0.20938865431933,atk=0.
01,sus=1,rel=1,ts=1,out=0,pan=0,amp=0.3;
        var sig,env;
        sig = Henon2DN.ar(freq,freq+mfreq,a,b,x0,y0,amp);
        env = EnvGen.ar(Env.linen(atk,sus,rel),1,1,0,ts,2);
        sig = LeakDC.ar(sig);
        sig = sig*env;
        out.ar(out,pan2.ar(sig,pan));
}).add;
);

//the example earlier, with no interpolation (default)
(
p.clock.tempo = 2.2;
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,
\degree,Pseq([0,2,4,6,7],inf),\octave,[8,12,9,10],
\dur,Pwrand([0.25,Pbjorklund2(Pwhite(3,5),8,1)/4,Pseq([0.125],4)],
[7,4,1].normalizeSum,inf),\a,Pexprand(1.2,1.55),\b,Pexprand(0.21,0.55),\atk,
0,\sus,0,\rel,Pexprand(0.01,0.6),\amp,1);
~henon.play;
)

//now with Linear interpolation
(
SynthDef(\henon,
    {arg
freq=440,mfreq=440,a=1.3,b=0.3,x0=0.30501993062401,y0=0.20938865431933,atk=0.
01,sus=1,rel=1,ts=1,out=0,pan=0,amp=0.3;
        var sig,env;
        sig = Henon2DL.ar(freq,freq+mfreq,a,b,x0,y0,amp);
        env = EnvGen.ar(Env.linen(atk,sus,rel),1,1,0,ts,2);
        sig = LeakDC.ar(sig);
        sig = sig*env;
        out.ar(out,pan2.ar(sig,pan));
}).add;
);

//now with Cubic interpolation
(
SynthDef(\henon,
    {arg
freq=440,mfreq=440,a=1.3,b=0.3,x0=0.30501993062401,y0=0.20938865431933,atk=0.
01,sus=1,rel=1,ts=1,out=0,pan=0,amp=0.3;
        var sig,env;
        sig = Henon2DC.ar(freq,freq+mfreq,a,b,x0,y0,amp);
        env = EnvGen.ar(Env.linen(atk,sus,rel),1,1,0,ts,2);
        sig = LeakDC.ar(sig);
        sig = sig*env;
        out.ar(out,pan2.ar(sig,pan));
}).add;
);
```

Good SynthDef writing for co34pt_LiveCode

I won't cover the fundamentals of synthesis or synthdef writing, as others have done so much better than I ever will.

If you're going to be writing SynthDefs for Patterns in the format I use in these guides and in my sets, there's a few rules to ensure that things run reasonably smoothly.

It's also worth reading the SynthDef documentation and Pbind documentation

`freq` and frequency

The carrier of a main frequency of a SynthDef should have the argument name `freq` - this will allow for the use of scales, tunings and detuning within Pattern arguments, from the documentation:

> detunedFreq actual "pitch" of a synth, determined by: freq + detune; freq is determined by: (midinote + ctranspose).midicps * harmonic; midinote is determined by: (note + gtranspose + root)/stepsPerOctave * octave * 12; note is determined by: (degree + mtranspose).degreeToKey(scale, stepsPerOctave)

There are a couple of instances where you can't use `freq` as the actual frequency, so in which case, use Pkey to reroute the frequency argument like this:

```
//where x is the frequency argument
Pbind(\instrument,\foo,\x,Pkey(\freq),\scale,Scale.minor,
\degree,Pseq([4,5,6],inf))
```

`out`

Each SynthDef should have an argument `out` in its `Out.ar`. I always leave it as `0`, but it can be used to handle effects routing. I don't know why, but if it doesn't have it, it won't work inside of ProxySpace.

Envelopes

Envelopes will be automatically triggered as part of patterns, on the assumption that the trigger of any envelope is set to `1`. It's also much easier to use envelopes where it does not need a release trigger. I generally use `Env.perc` and `Env.linen`. It's also important to use a `doneAction` which will free the synth once the envelope has completed.

Sequencing MIDI using ProxySpace and Pbind

I didn't get into live coding with MIDI initially, and it's only after a couple of years of performing that I decided to get a synth to work into my sets - and while the examples I am providing here *should* work with any MIDI Synth, I've probably only tested them on mine (a Make Noise 0 Coast).

Fortunately it's relatively easy to get MIDI sequences working in conjunction with the standard ProxySpace patterns described all over this repo. I based these instructions on the ones in the Pattern Guide Cookbook.

IMPORTANT! - This is a guide for setting up MIDI using *Linux*. OSX is probably similar, but Windows I am really not too sure about.

First you need to initialise MIDI on the server with `MIDIClient.init`. This will initialise MIDI on the server and print available MIDI devices to the post window, on my system they are listed as the following:

```
MIDI Sources:
    MIDIEndPoint("System", "Timer")
    MIDIEndPoint("System", "Announce")
    MIDIEndPoint("Midi Through", "Midi Through Port-0")
    MIDIEndPoint("Scarlett 2i4 USB", "Scarlett 2i4 USB MIDI 1")
    MIDIEndPoint("SuperCollider", "out0")
    MIDIEndPoint("SuperCollider", "out1")
    MIDIEndPoint("SuperCollider", "out2")
    MIDIEndPoint("SuperCollider", "out3")
    MIDIEndPoint("SuperCollider", "out4")
    MIDIEndPoint("SuperCollider", "out5")

MIDI Destinations:
    MIDIEndPoint("Midi Through", "Midi Through Port-0")
    MIDIEndPoint("Scarlett 2i4 USB", "Scarlett 2i4 USB MIDI 1")
    MIDIEndPoint("TiMidity", "TiMidity port 0")
    MIDIEndPoint("TiMidity", "TiMidity port 1")
    MIDIEndPoint("TiMidity", "TiMidity port 2")
    MIDIEndPoint("TiMidity", "TiMidity port 3")
    MIDIEndPoint("SuperCollider", "in0")
    MIDIEndPoint("SuperCollider", "in1")
    MIDIEndPoint("SuperCollider", "in2")
    MIDIEndPoint("SuperCollider", "in3")
```

Then use the MIDIOut class to create a MIDI Output, specifying the MIDI output you would like to use as a string. I add this to the dictionary that I store samples in, like this:

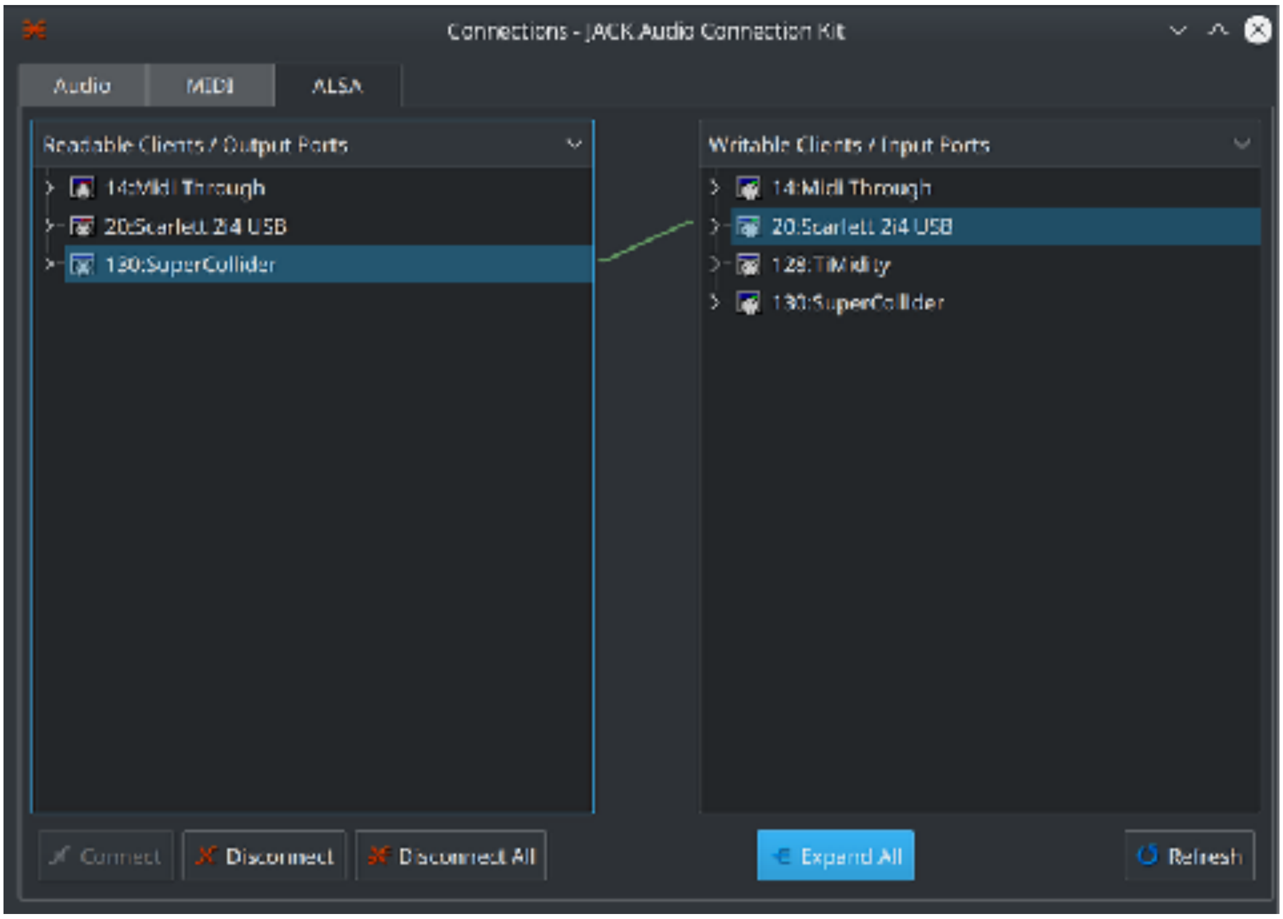
```
d[\m2] = MIDIOut.newByName("Scarlett 2i4 USB", "Scarlett 2i4 USB MIDI
1").latency = (0.2555)
```

The latency method is used to create latency in the MIDI signal, in order to sync the MIDI notes played by SuperCollider to the latency of the audio server - this will need some tweaking (see the accompanying .scd file).

MIDI sequences can then be sent from within ProxySpace as a Pbind, the same as any other pattern, with a few extra values necessary:

```
(
~midiPattern = Pbind(
    //specifies type of message sent
    \type, \midi,
    //specifies type of midi message
    \midicmd, \noteOn,
    //the MIDI Out used
    \midiout, d[\m],
  //the MIDI channel
    \chan, 0,
    //The rest of the pattern
    \scale,Scale.minor,
    \degree, Pseq([0,2,4],inf),
    \octave, 3,
    \dur, 0.5,
    \legato, 0.4
)
)
```

If this doesn't work, there's possibly a routing issue. If you're using Linux, load up `Qjackctl`, select `connect`, then go to `ALSA` and connect output `SuperCollider` to your MIDI interface:



You should now be patterning your MIDI device, Enjoy.

I don't really like MIDI as a technology because it is quite restrictive, particularly as it only takes 'note' messages rather than frequencies (messages are often limited to 0-127 ints). The result of this is that microtones of any kind are hard to specify. One way to create microtones is to use the `\bend` feature, which takes values from `0` to `16,383` (with `8,192` being the middle, or default).

```
(
~midiBend = Pbind(
\type,\midi,
\midicmd,\bend,
\midiout,d[\m],
\chan,0,
\dur,0.25,
\val,Pwhite(0,16383)
)
)
```

The amount that the pitch bend affects the pitch of the synth is set within the synth itself, in my case it is +/- 1 semitone. The code above results in a semi-microtonal scale, played out across one tone.

Note that the pitch bend *cannot* be specified at the same time as the notes, it must be specified separately, for reasons I don't quite understand.

In the setup file of this repo I have included a Setup_MIDI file, for setting up the SuperCollider server and MIDI with one execution. This will need to be edited to your MIDI device.

5.

Non-Pattern Techniques

Drones

Drones are great, both standing on their own as drone music or within other forms of music.

I've always found SuperCollider to be a really strong tool for making drones of all kinds as the types of subtle, durational modulations that can be achieved with `.kr` UGens allows for the creation of drones that vary over time very easily. The variation of these drones makes the background of sets interesting without having to maintain them directly - especially if the modulation in multiple drones are out of sync for instance. This sustained background interest can keep a set moving forward while time is spent working on preparing foreground elements without the background becoming boring too quickly (which is a problem I've come across a lot when performing live coding sets).

DFM1

sc3-plugins contains a great filter - DFM1. A “Digitally Modelled Analog Filter”, it is packed with features. It can be used a high pass and low pass, has a variable noise setting, and can self-oscillate at high resonances.

The most important feature of this for me is the self-oscillation. When overdriven, DFM1 produces a gorgeous 'warm' tone, which tends to distort softly the harder it is driven.

When this self-oscillating distortion is paired with a sine wave using the same fundamental frequency as the filter, some rich drones are created:

```
/*
A standard DFM1 drone I use an awful lot.
The filter self-oscillates at a 'res' value of >1, so here I have used a SinOsc
moving from 0.9-1.1, so the self-oscillated distortion fades in and out.
Here I am using the harmonic series to organise pitch. with the frequency of the
filter being double that of the SinOsc.
```

```
!!!!NOTE!!!!! - In my installation of SuperCollider, DFM1 is buggy and NodeProxies it
contains need to be evaluated twice slowly otherwise they will cut all sound from the
server when played. I don't know why this is (or whether it is a version/platform/OS
specific issue), but if the experience is any different for you please raise an issue
on GitHub or otherwise let me know. This only happens once per NodeProxy, once it is
initialised and playing it can be re-evaluated and changed with no effect on the
sound in the rest of the server
*/

//set the fundamental frequency
~r = {80}

//evaluate this twice with a couple of seconds of gap in between
//the stereo sine wave creates a 'beating' in stereo. For more information see
https://en.wikipedia.org/wiki/Beat_(acoustics)
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),~r*2,SinOsc.kr(0.05).range(0.9,1.1),1,0,0.0003,0.5)};

//play
~dfm1.play;

//changing the resonance changes the character of the self-oscillation, detuning it
and distorting it
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),~r*2,SinOsc.kr(0.05).range(0.9,1.6),1,0,0.0003,0.5)};

//The higher the resonance value gets, the more distortion
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),~r*2,SinOsc.kr(0.05).range(0.9,5.6),1,0,0.0003,0.5)};

//extreme resonance values get LOUD, but don't really change sonically past around
the 10 mark
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),~r*2,SinOsc.kr(1).range(10,400),1,0,0.0003,0.5)};
```

```
//DFM1 multiple drones
//Using the harmonic series technique, a number of drones at various multiplications
layered together
//Note - the modulation of the resonance is a slightly different speed for each, to
create an overall variation and non-repetition in sound

//set fundamental frequency
~r = {54};
(
//evaluate this twice with a couple of seconds of gap in between
//the argument 'mult' is used for speed - to copy and paste the entire NodeProxy and
set multiplications quickly during performance
~dfm1 = {arg mult = 1; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,SinOsc.kr(0.05).range(0.9,1.1),1,0,0.0003,0.5)};
~dfm2 = {arg mult = 2; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,SinOsc.kr(0.06).range(0.9,1.1),1,0,0.0003,0.5)};
~dfm3 = {arg mult = 3; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,SinOsc.kr(0.056).range(0.9,1.1),1,0,0.0003,0.5)};
~dfm4 = {arg mult = 4; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,SinOsc.kr(0.07).range(0.9,1.1),1,0,0.0003,0.5)};
)

//now play all
~dfm1.play;~dfm2.play;~dfm3.play;~dfm4.play;

//changing modulation from a SinOsc to an LFNoise, increasing modulation scope in
lower multiples
(
//evaluate this twice with a couple of seconds of gap in between
//the argument 'mult' is used for speed - to copy and paste the entire NodeProxy and
set multiplications quickly during performance
//this sounds like distorted guitars and is VERY rich.
~dfm1 = {arg mult = 1; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,LFNoise1.kr(0.05).range(0.9,4.5),1,0,0.0003,0.5)};
~dfm2 = {arg mult = 2; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,LFNoise1.kr(0.06).range(0.9,2.3),1,0,0.0003,0.5)};
~dfm3 = {arg mult = 3; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,LFNoise1.kr(0.056).range(0.9,1.9),1,0,0.0003,0.5)};
~dfm4 = {arg mult = 4; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,LFNoise1.kr(0.07).range(0.9,1.5),1,0,0.0003,0.5)};
)
```

Another way to use DFM1 as an oscillator is to run it up and down the harmonic series and use it as a 'melody' alongside some already running drones, and smooth it out by using the `noiselevel` argument:

```
//using DFM1 as a melody

//set harmonic frequency
~r = {60};

//start the first drone from the first example in this document
//evate this twice with a couple of seconds in between
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),~r*2,SinOsc.kr(0.05).range(0.9,1.1),1,0,0.0003,0.5)};

//play
~dfm1.play

//another drone, but one that contains a LFNoise1 used to give sweeps around the
harmonic series
//evaluate this twice with a couple of seconds in between
~dfmharm = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),LFNoise1.kr(0.1).range(100,1000).round(~r),SinOsc.kr(0.05).range(0.9,
1.1),1,0,0.0003,0.5)};

//play
~dfmharm.play;

//up the resonance
~dfmharm = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),LFNoise1.kr(0.1).range(100,1000).round(~r),SinOsc.kr(0.05).range(0.9,
1.4),1,0,0.0003,0.5)};

//up the speed of pitch change
~dfmharm = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),LFNoise1.kr(1.4).range(100,1000).round(~r),SinOsc.kr(0.05).range(0.9,
1.4),1,0,0.0003,0.5)};

//up the noise
~dfmharm = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),LFNoise1.kr(1.4).range(100,1000).round(~r),SinOsc.kr(0.05).range(0.9,
1.4),1,0,0.1,0.5)};
```

SuperCollider as a Modular Synth

A performance technique I don't generally employ a whole lot during Algorave-type sets using SuperCollider as a modular synth - and ProxySpace is *very* strong in this regard too. Each NodeProxy can be seen as an individual module, and each module can be plugged into others to create a complex network of interconnected musical and control elements. This is achieved by setting up audio (`.ar`) and control (`*.kr`) proxies - for more info on audio vs control rate see this and this

I can't really talk about this in any great depth, so here is an in-depth example of how SuperCollider can be used as a live-codeable modular synth. An important thing to note though is that if you want a lot of freedom in this approach, a lot of familiarity with types of UGens available (as well as some of the stranger quirks of SuperCollider syntax) will be very helpful.

```
//load setup
("../../Setup/Setup.scd").loadRelative

//run this to smooth out transitions
p.fadeTime=5

//Using SuperCollider as a Modular synth
//snippets help with building these sets a LOT, as standard elements such as
modulation signals can be called upon very quickly
//NOTE: this will get !!! L O U D !!! - there's protection from StageLimiter of
course, but be aware.
//NOTE II: There may also be some DC bias. Be prepared for this. more information
here - http://en.wikiaudio.org/DC_offset

//a sine wave
~sin = {SinOsc.ar([80,82],0,0.5)}

//a pulse wave
~pulse = {Pulse.ar([20,21],SinOsc.kr(0.1).range(0.01,1),0.5)}

//a new proxy multiplying sine and pulse waves
~sinpulse = {~sin.ar * ~pulse.ar}
~sinpulse.play
```

```
//feed this into a delay with its delay line modulated slightly
~delay = {CombC.ar(~sinpulse.ar,1,LFNoise1.kr(0.1).range(0.1,0.3),4)}
~delay.play

//increase the pulse speed and decrease the width, play it alongside the original
~pulse2 = {Pulse.ar([40,41],SinOsc.kr(0.1).range(0.001,0.1),0.5)}
~pulse2.play;

//actually no that would sound much better just in the delay, so ~pulse2 from playing
and add it into ~delay by using Mix.ar
(
~pulse2.stop;
~delay = {CombC.ar(Mix.ar([~sinpulse.ar,~pulse2.ar]),
1,LFNoise1.kr(0.1).range(0.1,0.3),4)};
)

//now we have some drones, some heavily gated and filtered noise would be good.
(
~noise =
{RLPF.ar(WhiteNoise.ar(1),LFNoise1.kr(0.1).range(100,2000),SinOsc.kr(0.1).ran
ge(0.1,0.4),1)};
~noiseEnv = {EnvGen.ar(Env.perc(0.0001,0.1),Dust.kr(4))};
~totalNoise = {~noise.ar*~noiseEnv.ar};
~totalNoise.play;
)

//oh no. it is mono. i'm going to pan it over 2.
//In order to make a mono proxy stereo, I will have to .clear it and then evaluate a
stereo version, as the number of channels is set at initialisation time.
//luckily with Pan2 I will only have to re-evaluate the ~totalNoise proxy
~totalNoise.clear;
(
~totalNoise = {Pan2.ar(~noise.ar*~noiseEnv.ar,SinOsc.kr(0.1))};
~totalNoise.play;
)
```

```
//the filtering on the noise isn't extreme enough, change it!
~noise = {RLPF.ar(WhiteNoise.ar(1),LFNoise1.kr(0.6).range(100,2000),
SinOsc.kr(0.04).range(0.00001,0.2),1)};
//the noise could also do with some delay, which would sound nice if it was fed back
through a pitch shifter:
//set up the delay, and play it
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar]),1,0.4,7,1)}
~noiseDelay.play;

//establish the pitch shifter
~pitchShift = {PitchShift.ar(~noiseDelay,0.2,TRand.kr(0.1,2,Dust.kr(0.5)))}
//play the pitch shifter, it will slow the delay speed by half
~pitchShift.play

//if we then put the results of ~pitchShift back into ~noiseDelay, then things get
interesting.
//NB - this is bad practice and gets very loud before ending up in being DC bias, but
i'm doing it here to prove a point.
//If you have super high end audio equipment or just don't want any DC bias then skip
this step
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar,~pitchShift.ar]),1,0.4,7,1)}
//in order to avoid this getting totally out of control, reduce the volume of
~pitchShift inside of ~noiseDelay
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar,(~pitchShift.ar*0.11)]),
1,0.4,7,1)}
//or modulate it to get varying amounts of feedback
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),1,0.4,7,1)}
//modulating the delay time too will make things get a bit wild
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),
1,LFNoise1.kr(0.1).range(0.01,0.6),7,1)}
//~noiseDelay seems to be glitching a bit and throwing DC bias - add a LeakDC around
it
~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),
1,LFNoise1.kr(0.1).range(0.01,0.6),7,1))}
//let's cut the original pulse/sine waves over a few seconds
~delay.stop(20)
~sinpulse.stop(20)
//then put them inside of a DFM1 that can self-oscillate
//make sure you evaluate ~noiseDelayAdd twice before you .play it
~noiseDelayAdd = {DFM1.ar(Mix.ar([~delay.ar,~sinpulse.ar]),
500,SinOsc.kr(0.1).range(0.5,2),1,0,0.03)}
//if you've evaluated the above line twice, play it
~noiseDelayAdd.play
```

```
//a lot of these sounds are quite degraded, some harsh sounds would be nice, let's
have some chaos
//go to the help file for Henon2DC and copy-paste the second example but don't
evaluate it (you'll need sc3-plugins for this)
/*
(
{ Henon2DN.ar(
    2200, 8800,
    LFNoise2.kr(1, 0.2, 1.2),
    LFNoise2.kr(1, 0.15, 0.15)
) * 0.2 }.play(s);
)
*/

//turn it into a node proxy and remove the .play(s) from the end
(
~henon = { Henon2DN.ar(
    2200, 8800,
    LFNoise2.kr(1, 0.2, 1.2),
    LFNoise2.kr(1, 0.15, 0.15)
) * 0.2 };
)

//make an envelope that has a long sweeping modulation on the amount of envelopes
triggered
~chaosEnv =
{EnvGen.ar(Env.perc(0,0.02),Dust.kr(SinOsc.kr(0.01).range(1,10)))}
//and combine in stereo
~chaos = {Pan2.ar(~henon*~chaosEnv)}
~chaos.play

//it is SUPER quiet, up the volume on ~henon
(
~henon = { Henon2DN.ar(
    2200, 8800,
    LFNoise2.kr(1, 0.2, 1.2),
    LFNoise2.kr(1, 0.15, 0.15)
) * 3.5 };
)
```

```
//add some reverb which will work in parallel
//if you want to change the parameters of any effect without re-evaluating it - set
up that value as another NodeProxy
~room = {30};
~time = {3};
~verb = {GVerb.ar(~chaosEnv,~room,~time)}
~verb.play
//increase the reverb time
~time = {40};

//this needs some melody - add two melodies in stereo, slightly out of phase:
~saws =
{LFSaw.ar([LFSaw.kr(0.1).range(100,1000).round(50),LFSaw.kr(0.11).range(100,
1000).round(50)],0,0.3)}
~saws.play
//too harsh, needs filtering
~saws = {RLPFD.ar(LFSaw.ar([LFSaw.kr(0.1).range(100,1000).round(50),
LFSaw.kr(0.101).range(100,1000).round(50)],0,0.8),1000,0.8,0.6,10)};

//another delay would be nice
~sawDelay = {CombC.ar(~saws.ar,1,0.5,10)};
~sawDelay.play;
//some heavy decimation on the delay
~sawDelay = {Decimator.ar(CombC.ar(~saws.ar,1,0.5,10),2200,10)};
//further bit reduction
~sawDelay = {Decimator.ar(CombC.ar(~saws.ar,1,0.5,10),2200,5)};
//even further
~sawDelay = {Decimator.ar(CombC.ar(~saws.ar,1,0.5,10),2020,3)};
//plugging the ~sawDelay into the original for more noise

~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),
1,LFNoise1.kr(0.1).range(0.01,0.6),7,1))}
//plugging ChaosEnv into ~noiseDelay too
~noiseDelay =
{LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),
1,LFNoise1.kr(0.1).range(0.01,0.6),7,1))};
//then plugging it also into a more intense ~noiseDelayAdd for more mad effects
~noiseDelayAdd = {DFM1.ar(Mix.ar([~delay.ar,~sinpulse.ar,~noiseDelay]),
LFNoise1.kr(100).range(100,10000),SinOsc.kr(0.1).range(0.5,100),1,0,0.03)}
~noiseDelayAdd.play
```

```
//it doesn't appear to be playing, probably because ~noiseDelay is SO loud. Multiply
it by half
~noiseDelay =
{LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),
1,LFNoise1.kr(0.1).range(0.01,0.6),7,1)) * 0.3};
//then plug ~noiseDelayAdd into ~noiseDelay and roll off the multiplication for
maximum damage
~noiseDelay =
{LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs),~noiseDelayAdd.ar]),
1,LFNoise1.kr(0.1).range(0.01,0.6),7,1))};
//increase the ridiculousness of the modulation of the delaytime
~noiseDelay =
{LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs),~noiseDelayAdd.ar]),
1,LFNoise1.kr(1).range(0.001,4),7,1))};

//put another delay on top of that?
~delay2 = {CombC.ar(~noiseDelay.ar,1,0.4,30)}
~delay2.play

//then plug that back into ~noiseDelay (which by now contains most things that are
playing.
~noiseDelay =
{LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs),~noiseDelayAdd.ar,~delay2.ar]),
1,LFNoise1.kr(1).range(0.001,4),7,1))};
//also modulate ~delay2, really slowly
~delay2 = {LeakDC.ar(CombC.ar(~noiseDelay.ar,
1,SinOsc.kr(0.01).range(0.0001,0.2),80))}

//things broke up for me here and I have no idea why, there's multiple things feeding
back through each other here.
//and you have noise music!
```

6.

Visuals and Data

FreqScope and Visuals

Note: This guide is in general terms because it is platform-specific. I'd recommend some research on how this can be realised on your particular platform

I really like having visuals as part of my sets, I think it adds a lot of energy to sets, regardless of how 'audio-responsive' they are. In addition to adding a bunch of colour to my projection, it gives some relief from just looking at code, and can serve as a low-budget light show in absence of actual lighting.

I have written my own programs to make visuals for sets before in openFrameworks, which was a lot of effort. While this in itself was not an issue, I find it *extremely* difficult to live code visuals and sound at the same time, as it involves a lot of parallel thinking, which disturbs my flow when live coding music. What I've found is that SuperCollider's FreqScope is a great way of instantly adding visuals to sets with very little actual effort.

Inside ProxySpace, a `FreqScope` can be started to monitor all sound by evaluating `s.scope` (which is contained within this repo's `Setup.scd`. This will give an oscilloscope-type visualisation of the sound currently taking place, and can be shown as independent channels, an overlay, or an x/y chart of the sound on a stereo spectrum. My technique is to fullscreen the `FreqScope` window, and drop it behind my SuperCollider IDE window, and make the IDE window semi transparent with a black background (the black background is especially important as it will not tint the scope), showing the scope behind the code I am writing (as can be seen here). This is an effort-free way to get some responsive visuals which work alongside my code which do not need attention themselves. I won't post any guides here on how to make your IDE transparent, as this depends entirely on your platform. I found it quite hard to do on Mac OSX, and quite easy on Linux (but a little harder in Ubuntu Unity than my current KDE). I usually do this in `Tracks` mode, although `Overlay` works too.

X/Y is where things get more interesting. This mode plots stereo sound on a two-dimensional plane by frequency and amplitude to form a geometric shape rather than a wave. The best example of how this works can be seen in this Techmoan video (or anything that can be found by googling Oscilloscope Music). X/Y mode can be a great way to create music that directly results in interesting visual forms by using complimentary frequencies across the stereo field. The specifics of this revolve around the harmonic series and different types of intonation which is explained in this video. The shapes made can be changed by the type of waves used, as well as the volume and frequency, and performing according to this is an interesting way of shaking up one's performance strategies, as normal performance techniques will not yield interesting shapes, here is X/Y used as visuals on a project entirely sounding entirely sine waves and here are a couple of more clear code examples:

```
//Example 1 - Static Frequencies
(
//two low sine waves at the same frequency showing a diagonal line
~sin1 = {SinOsc.ar([80,80],0,0.3)};
~sin1.play;
)

//two low sine waves at slightly different frequencies turning the line into a slowly
turning disc
~sin1 = {SinOsc.ar([80,80.1],0,0.3)};
(
//two sine waves at double the frequency - notice the change in shape - turning the
line a number of times on itself
~sin2 = {SinOsc.ar([80*2,80.01*2],0,0.3)};
~sin2.play;
)

(
//two sine waves at 10x the frequency - notice the change in shape - turning the line
a whole bunch more times on itself
~sin3 = {SinOsc.ar([80*10,80.01*10],0,0.3)};
~sin3.play;
)

//stop everything
~sin1.stop;~sin2.stop;~sin3.stop;
```

```
(
//changing the frequency difference in the lower sine waves, changing how the
original circle moves
~sin1 = {SinOsc.ar([80,80+LFNoise1.kr(0.1,4)],0,0.3)};
~sin1.play;
)

//replay the other sine waves and see how the entire shape moves faster
~sin2.play;~sin3.play;

//stop the highest sines
~sin3.stop;
(
//re-align the two low sine waves
~sin1 = {SinOsc.ar([80,80.01],0,0.3)};
~sin1.play;
)

(
//play a sine that doesn't align with the harmonic series, notice that the shape gets
much less clear
~sin4 = {SinOsc.ar([94.234,99.1315],0,0.3)};
~sin4.play;
)

//stop the non-aligning sines
~sin4.stop;
//stop the second sine
~sin2.stop;

//play some quiet width-modulated pulse waves at 2x the frequency of the low sine
waves
//notice that the shape changes according to the width of the pulse and that the
'notches' interact with each other across the stereo field
(
~pulse1 = {Pulse.ar([80*4,80.1*4],SinOsc.kr(0.05).abs,0.08)};
~pulse1.play;
)

//change the pulse to a saw wave at the same frequency
(
~pulse1.stop;
~saw1 = {Saw.ar([80*4,80.1*4],0.08)};
~saw1.play;
)
```

```
//note that the higher the volume, the greater the effect a sound has on the overall
shape
~saw1 = {Saw.ar([80*4,80.1*4],0.08)};
//also the higher the frequency, the lesser the effect on the 'overall' shape and the
greater the effect on the 'detail' of the shape
~saw1 = {Saw.ar([80*100,80.1*100],0.1)};
//stop everything
~sin1.stop;~saw1.stop;

//Example 2 - Moving frequencies and non-standard waveforms
//make a (really) low sine wave/spinning disc again
(
~sin1 = {SinOsc.ar([50,50.01],0,0.4)};
~sin1.play;
)

//make a stereo sine wave that sweeps the harmonic series
(
~sin2 = {SinOsc.ar(Saw.kr(0.1).range(10,1000).round(50),0,0.4)!2};
~sin2.play;
)

//make those two sine waves sweep the harmonic series at phasing (sightly different)
rates
(
~sin2 = {SinOsc.ar(Saw.kr([0.1,0.11]).range(10,1000).round(50),0,0.4)};
~sin2.play;
)

//turn off the original sine wave
~sin1.stop
//speed the sweeping and make it a sine wave
~sin2 = {SinOsc.ar(SinOsc.kr([0.5,0.56]).range(10,1000).round(50),0,0.4)};

//make two meandering SinOscFB Ugens around the lower end of the harmonic series and
see how they interact
(
~sinfb1 =
{SinOscFB.ar([LFNoise1.kr(0.1).range(50,100).round(25),LFNoise1.kr(0.1).range
(50,100).round(25)],SinOsc.kr(0.1).range(0.01,1),0.8)};
~sinfb1.play;
)

//stop the second sine waves
~sin2.stop
```

```
//make a big sub kick drum - notice the effect on the shape
(
~k = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,4,\amp,1);
~k.play
)

//make a panned hi-hat
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,
0.25,\amp,Pexprand(0.05,1),\pan,Pwhite(-1,1.0));
~h.play;
)

//make the feedback in the sinfb much more pronounced
~sinfb1 =
{SinOscFB.ar([LFNoise1.kr(0.1).range(50,100).round(25),LFNoise1.kr(0.1).range
(50,100).round(25)],SinOsc.kr(0.1).range(0.01,3),0.8)};
```

OSC Communication and Data Streams

I have done a set of performances that revolve around live coding and using data streams in the past, particularly performances where I have a continuous stream of data to interpret during the performance. These performances involve negotiating my relationship with sensor data (e.g. movement data, temperature, light levels), live-coding my interpretation of this data to deliver a performance.

For this type of performance I have used a particular set of technologies mutliple times, and have developed a reasonably quick way to work, which I will share here. Depending on the kind of data you want to use, some of the items described in this part of the guide may not apply to you directly, but using OSC to handle messages in SuperCollider is a really useful skill that can apply to many types of situation, so it's worth knowing if you want to work outside of SuperCollider at all.

Open Sound Control (hereafter OSC) is an *absurdly* useful protocol for communicating between programs, and across networks.

Before I learned how to use it I heard people refer to it a lot as 'modern MIDI', which I think is a bit of a misnomer and actually confused me quite a lot while learning it as someone used to MIDI a la DAWs and plugging cables into synthesizers. While MIDI is a set of commonly understood messages between programs (plug a MIDI cable into an interface and you can expect your DAW to react in a certain way), OSC is more of a “common language” that enables programs to communicate effectively. OSC is very useful for getting multiple programs and machines to “talk" to each other, and I have found it very useful for performances involving multiple programs and machines running together.

For example:

- sampler-sampler uses OSC to communicate information about emulated stitching between two machines and multiple programs:

```
MACHINE 1: Processing → MACHINE 2: SuperCollider → Processing
```

- tome. uses OSC to parse sensor data and manage lighting.

```
Sensor array (serial data) → Python Serial Parser[1] → SuperCollider → QLCPlus
→ OpenDMXUSB
```

While the above setups might seem complex or convoluted, using OSC makes these connections very easy, and using OSC is very similar across platforms.

It's first worth understanding a bit about how OSC sends its messages:

OSC Messages are sent over a network, and that network can be internally within a machine (to sent messages between programs), or across machines in a network of any kind (commonly a local network). Messages are sent to a particular port of a particular network address (for example, `127.0.0.1`, port `51720`), with an address (for example `/hello`), and parameters that can be of various types (for example `1`, `32.32`, `'message'`).

[1] https://github.com/theseanco/python-SerialToOSC

Sending this message from SCLang to be received by SCLang internally would look like this (adapted from the OSC Communication tutorial):

```
//monitor all incoming OSC Messages
OSCFunc.trace;
//set the relevant IP and port - both arbitrary, but these will be sent to
//SuperCollider internally (assuming that NetAddr.langPort == 57120)
b = NetAddr.new("127.0.0.1", 57120);
//send the above message, and it should be shown in the post window
b.sendMsg("/hello", 1, 32.32, 'message')
// If this doesn't work, evaluate:
NetAddr.langPort
// Then change the port of NetAddr.new accordingly
```

This is the basic way to send OSC Messages using SuperCollider. These messages can be sent to any IP and port, and the message will be sent regardless whether or not it is received. In order for the message to mean anything, a receiver will have to be built to interpret the message.

Taking the above example, here is a simple setup that will make a sound every time a message is sent to address `/ding`, it uses a class called `OSCdef` which triggers a particular function when an OSC message is received:

```
// set address
b = NetAddr.new("127.0.0.1",NetAddr.langPort);
// create OSCdef (very similar syntax to SynthDef)
(
OSCdef(\dinger,
    {
    // a simple function that triggers an envelope
        {Pulse.ar(1000,rrand(0.01,0.5),0.3)!2 *
EnvGen.ar(Env.perc,doneAction:2)}.play
}, '/ding')
)
// Send a message with no parameters. It'll trigger the function within the OSCdef.
b.sendMsg("/ding")
```

There are a few tools for diagnosing issues with OSC use in SuperCollider, and we have touched on both of them here. To check if messages from another application are being received correctly, evaluate `OSCFunc.trace(true)`, which will print all incoming OSC messages to the post window (incluing any internal communications within sclang). If you are expecting to recieve messages to SuperCollider and they're not coming through (the default port of 57120 is where I usually direct all my messages), evaluate `NetAddr.langPort` to check the internal server port, as it can be re-assigned through multiple instances of SClang.

Messages sent over OSC can also be interpreted and passed into these functions, here is an elaboration on the above example, using a message to set the pitch of the sound:

```
//set address (if you've already done this no need to do it again)
b = NetAddr.new("127.0.0.1",NetAddr.langPort);
//msg will receive the OSC message as an array, with index 0 being the address and
//index 1 onwards being the message.
//setting msg[1] as the frequency will give the first parameter of the
//OSCmessage as an argument
//setting msg[2] as the pulse width would allow you to send the second
//message parameter as the pulse width, and so on...
(
OSCdef(\dinger,
    {
        |msg|
        {Pulse.ar(msg[1],rrand(0.01,0.5),0.3)!2 *
EnvGen.ar(Env.perc,doneAction:2)}.play
}, '/ding')
)
//make a 900Hz ding
b.sendMsg("/ding",900);
//make a ding at a random pitch
b.sendMsg("/ding",rrand(100,2000))
```

In terms of using live data, and live coding your response to the data, the OSCdef can be changed and re-evaluated on the fly, changing data mappings and using OSCdefs to send messages to various items running on the server, and this fits into ProxySpace very nicely. In order to use live data however, you need a live data source, which is not readily available from within SuperCollider - check the examples folder for a Python script which simulates a live data input to be used in a live coding context, covering inter-program communication and live-mapping of data.

If you are wanting to use data from an Arduino to get data into SuperCollider, I wrote this tool, which generates Python scripts based on a specification you provide that parses Serial data and sends it as a high-speed OSC stream, for which you can build custom OSCdefs in SuperCollider.

Using Datasets

As well as using live data, using static datasets is another technique for using external inputs.

I've used datasets during live coding sets in the past, including for the first half of my Chemical Algorave set. I've also used SuperCollider to create some works using sonification of static data sets, including this.

There are a number of ways to interpret datasets as sound, part of a technique commonly referred to as Sonification.

There's a great resource on Sonification here, but i'll cover the techniques that I have used to leverage Data in SuperCollider here, for which you should refer to the example in this repo.

7.

Source Code

2.2 - ProxySpace Basics.scd

```
//SuperCollider ProxySpace tutorial. See ProxySpace.md for explanation.

//Live coding can unexpectedly get loud. Always use protection
StageLimiter.activate

//1: Proxies and patching
//In ProxySpace you live code SuperCollider much like you would live patch a modular synth

//A pair of sine waves:
{SinOsc.ar([400,500],0,0.1)}.play;

//stop the server (Ctrl/Cmd+.)
Server.hardFreeAll

//Start ProxySpace
p = ProxySpace.push(s);

//make the basic sine wave again, but with a named proxy
/*
A few notes:

Proxies work on a couple of naming conventions, one is:

~foo123 (has to start with a lowercase letter)

or:

p[\name]

single letter variables can't hold a proxy, I generally use them to hold other things such as
lists if I need them during performances.

I don't know why single letter variables don't work. They just don't.

You also can ONLY store proxies in those two naming conventions. If you need to store lots of
variables elsewhere, i'd suggest making a dictionary on a single letter variable to refer to
later.

*/
~sine1 = {SinOsc.ar([400,500],0,0.1)};

//play those sine waves
~sine1.play;

//change the frequency of those sine waves instantly
~sine1 = {SinOsc.ar([500,600],0,0.1)};

//add a fade to ProxySpace
p.fadeTime = 3;
//you can also assign individual fade times to proxies
~sine1.fadeTime = 3;

//change the frequencies again and hear a fade
~sine1 = {SinOsc.ar([200,300],0,0.1)};

//make a second sine wave and play it alongside the first
~sine2 = {SinOsc.ar([350,450],0,0.1)};
~sine2.play;

//add modulation to the second sine wave
~sine2 = {SinOsc.ar([350+Saw.kr(1,100),450+Saw.kr(0.99,100)],0,0.1)};
```

```
//stop the two sine waves
~sine1.stop;
//stops can also be faded
~sine2.stop(5);

//make two modulated saw waves
~saw = {Saw.ar([LFNoise1.kr(0.1).range(8,12),LFNoise1.kr(0.1).range(8,12)],2)}

//Amplitude modulate the original second sine with the saw
~sine2 = {SinOsc.ar([350,450],0,0.1*~saw)};
~sine2.play

//frequency modulate the sine wave with the saw
~sine2 = {SinOsc.ar([350,450]*~saw,0,0.1)};

~sine2.stop;

//you can also combine proxies in a new proxy for modulation
~modSine = {~sine2 * ~saw}
~modSine.play

//mix in other proxies
~modSine = {Mix.ar([~sine1 * ~saw,~sine2]) * ~saw}

//Create an effects proxy, to send other proxies to
//(note the multichannel expansion in ~delay)
~delay = {CombN.ar(~modSine,0.2,[0.2,0.21],2,1)}
~delay.play

//Create another effects proxy, just for fun
~decimator = {Decimator.ar(~delay,2205,10)}
~decimator.play

//Note that when you create a new proxy, the old one will always keep going, so chaning like
this will keep sounding more and more layers unless you stop the existing ones
~modSine.stop
~delay.stop

/*

Note, if you .stop a proxy, it will keep running in proxyspace, but it won't be sounding (but
if it is affecting any other proxies it will still do so).

This means that if you have any particularly CPU intensive proxies running but not sounding,
or affecting any other proxies, and you don't intend to use that proxy again, you should use
the .free method to kill them completely:

i.e. (don't evaluate these now)
~modSine.free
~delay.free

*/


//You can plug any part of this proxy chain into any other part (but not a proxy into itself)
//This can get very loud and unruly very quickly.

//plugging the end of the effects chain back into the frequency of the first sine wave, this
will get some nice modulation
~sine1 = {SinOsc.ar([200,300]* ~decimator,~delay * ~saw,0.4 * ~decimator)};

//plugging the results of the delay back into the delay again. This will get unruly and REALLY
loud.
```

```
//Turn down your volume. You have been warned.
~delay = {CombN.ar(~modSine+~decimator,0.2,[0.2,0.21],2,1)}

//note, you can't plug a proxy into itself.
//see also: dividing by zero
~delay = {CombN.ar(~modSine,0.2,[0.2,0.21]*~delay,2,1)}

//free all proxies in ProxySpace.
//wasn't that fun?
p.clear

//For using patterns in ProxySpace, see ProxySpace ii
```

2.2 - ProxySpace Patterns.scd

```
/*

Using Patterns in ProxySpace

You don't have to be super well versed in patterns to follow this. This will be explained
further on in the repo

If you haven't looked at the 'recommended addons' section, please do so now, as you will need
some of the Quarks listed to play these examples.

NOTE: If you have come here from ProxySpace i, please quit the server and recompile

*/

//To start, either execute the setup file or run this
("../../Setup/Setup.scd").loadRelative

//Patterns can also be written directly into ProxySpace. They will be synced to ProxySpace's
TempoClock
//The tempoclock is initialised at setup by p.makeTempoClock
//The speed of the clock is controlled by modifying this value
p.clock.tempo = 1

//start a basic kick drum pattern
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;

//start a basic snare pattern
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,2,\amp,1);
~sn.play;

//start a basic hi-hat pattern
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,0.25,\amp,Pwhite(0.2,1));
~h.play;

//double the clock speed.
p.clock.tempo = 2

//with p.clock.tempo at 2, one full cyle in the 'dur' argument of patterns happens twice a
second, making the BPM 120
//You can get the BPM value of p.clock.tempo by multiplying it by 60
//You can also do this to set the tempo clock by BPM
p.clock.tempo = 135/60

//the proxyspace clock can be pushed very hard, with super low clock speeds resulting in
silence as patterns run too slowly (if only patterns are running)
p.clock.tempo = 0.00001

//Extremely high clock speeds will result in extratone-like drums
p.clock.tempo = 20

//Absurd clock speeds will result in hideous crashing and you having to recompile
p.clock.tempo = 999999
//(go on, I dare you)

//anyway...
p.clock.tempo = 135/60

//Patterns can also have fades applied to them, much like the function proxies in the first
tutorial
p.fadeTime = 4;
```

```
//hear what fades sound like on patterns, either run these one at a time (shift+return) or all
at once (ctrl+return)
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][1],\dur,Pwrand([0.25,Pseq([0.125],2)],
[0.8,0.2],inf),\amp,Pwhite(0.2,1));
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pbjorklund2(5,16)/4,\amp,1);
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,1);
)

//Note how all the proxies have stayed in time with one-another no problem. This 'just works'.
//I've experimented with using Pdefs in the past, and never managed to get them to quite sync
up, or i've had issues syncing Pdefs together.

//let's make things a little less intense
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwrand([0.25,Pseq([0.125],2)],
[0.8,0.2],inf),\amp,Pwhite(0.2,1));
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,2,\amp,1)
)

//You can also use Pbinds to specify melodies
(
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\root,0,\degree,0,\octave,
5,\dur,1,\d,0.3,\a,0.6,\pan,0,\amp,1);
~ring1.play;
)

//The beauty of using patterns inside of ProxySpace is that you can build them up element by
element, and evaluate as often as you want, building complexity during performance in a way
that you and an audience can hear.

//Evaluate these one by one, waiting a little while between each

//i'm going to change one value at a time to really illustrate how these things can build. In
performance i'd usually do more than that before re-evaluating a proxy

//1
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\root,0,\degree,0,\octave,
5,\dur,0.25,\d,0.3,\a,0.6,\pan,0,\amp,1)
//2
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\root,0,\degree,0,\octave,
5,\dur,0.25,\d,0.3,\a,Pexprand(0.6,10),\pan,0,\amp,1)
//3
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\root,0,\degree,
0,\octave,Pwrand([5,4,3],[0.6,0.2,0.2],inf),\dur,0.25,\d,0.3,\a,Pexprand(0.6,10),\pan,0,\amp,
1)
//4
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\root,0,\degree,
0,\octave,Pwrand([5,4,3],[0.6,0.2,0.2],inf),\dur,Pbjorklund2(Pwhite(10,15),16)/4,\d,
0.3,\a,Pexprand(0.6,10),\pan,0,\amp,1)
//5
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\root,0,\degree,
0,\octave,Pwrand([5,4,3],[0.6,0.2,0.2],inf),\dur,Pbjorklund2(Pwhite(10,15),16)/4,\d,
0.3,\a,Pexprand(0.6,80),\pan,0,\amp,1)
//6
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\root,
0,\degree,Prand([0,2,4,6,7],inf),\octave,Pwrand([5,4,3],[0.6,0.2,0.2],inf),
\dur,Pbjorklund2(Pwhite(10,15),16)/4,\d,0.3,\a,Pexprand(0.6,80),\pan,0,\amp,1)

//As you can probably imagine, things can get pretty complicated, but let's keep going.

//So far we're running all patterns, but the beauty of ProxySpace means we can run function
proxies alongside pattern proxies
~sines = {SinOscFB.ar([36.midicps,48.midicps+1],SinOsc.kr(0.1).range(0.1,1),0.8)}
~sines.play
```

```
//You can also use function proxies to effect patterns. I've been told that this shouldn't
work, but i've never had much of a problem with it. I'll create another pattern
~sinfb = Pbind(\instrument,\sinfb,\degree,0,\octave,6,\dur,0.25,\amp,0.7,\fb,0.1)
~sinfb.play

//then create a Control Rate proxy to control that pattern
~sinfbControl = {SinOsc.kr(0.1).range(0.1,1.5)}

//then set an argument on the pattern against the control rate proxy
~sinfb.set(\fb,~sinfbControl)

//And you can chain pattern proxies too

//let's add reverb to ~ring
~verb = {FreeVerb.ar(~ring1,1)}
~verb.play

//and make the reverb ridiculous
~verb = {FreeVerb.ar(~ring1,1,1,0.1)}

//and cut the percussion for some ambience
(
~k.stop;
~sn.stop;
~h.stop;
)

//and make another sinfb pattern a fifth above the old one for more ambience
~sinfb2 = Pbind(\instrument,\sinfb,\degree,4,\octave,6,\dur,0.25,\amp,0.7,\fb,0.1)
~sinfb2.play

//make sure it has the control proxy assigned to it too! Or, make another control proxy to get
two alternating washes of feedback
~sinfbControl2 = {SinOsc.kr(0.11).range(0.1,1.5)}
~sinfb2.set(\fb,~sinfbControl2)

//slow the whole thing down a little
p.clock.tempo = 120/60

//super loud kick for the head-nodders out there...
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,1,\amp,10);
~k.play;
)

//percussion crossrhythm
(
~p = Pbind(\instrument,\bplay,\buf,d["ding"][0],\dur,0.75,\amp,1);
~p.play;
)

//alternate the percussion crossrhythm across itself using a task for maximum enjoyment. Also
vary the pitch for even maximumer enjoyment.
(
~p.fadeTime = 4;
(
Tdef(\task,{
    loop{
        ~p = Pbind(\instrument,\bplay,\buf,d["ding"][0],\dur,0.75,\amp,
             1.5,\rate,rrand(1,1.2));
        rrand(1,5).wait;
}});
);
Tdef(\task).play;
)

//really slow euclidean snare hitting just away from the beat
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pbjorklund2(5,32)/4,\amp,4);
~sn.play;
)

//raise the tempo because people had time to go to the bar during the ambient section and now
people want to dance
p.clock.tempo = 135/60

//and so on, and so on...

//Oh, you want to do some mixing?
//cool, tweak volumes here
//I don't do this much myself during sets but it's worth knowing about
ProxyMixer.new(p)

//note that you can also do this for individual proxies by specifying this argument. It
defaults to 1
~sn.vol = 0.1;

~k.vol = 3;

~sines.vol = 0.5;

//I got a bit carried away, but I think i've gone some way to demonstrating the power of
ProxySpace when combined with SuperCollider's native functionality.

//start, stop and modify proxies to your heart's content, change the volumes on the ProxyMixer
as you wish, and don't forget to free your proxies when done with them!

//when you're finished
(
~k.clear;
~sn.clear;
~h.clear;
~sines.clear;
~ring1.clear;
~p.clear;
~sinfb.clear;
~sinfb2.clear;
Tdef(\task).stop;
)

//or
(
p.clear;
Server.hardFreeAll;
)
```

2.4 - Pbinds and Patterns - Examples.scd

```
/*
These examples are extracted from section 2.4 – Pbinds and Patterns – The Basics

I'd advise hard-stopping (Ctrl+.) between examples
*/

//Load setup file to get examples working
("../../Setup/Setup.scd").loadRelative

//So, if I wanted to have a kick drum playing once each beat in time with the ProxySpace
timer, after I had run my setup file I would do the following:
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1);
~k.play;
)



//This will return a syntax error
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\rate);

//As part of these key-value pairs, Pbinds can take Pattern classes as inputs. [`Pwhite`]
(http://doc.sccode.org/Classes/Pwhite.html) gives random values between a minimum and maximum.
If I wanted to specify a random pitch of the kick drum, I could add this to the pattern:
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\rate,Pwhite(1,1.2));
~k.play;
)

//footwork kickdrums
(
p.clock.tempo = 2.4;
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(Pseq([3,3,3,5],inf),8)/4,\amp,
1,\rate,Pseq([1,1.2],inf));
~k.play;
)

//skittery hi-hats
(
p.clock.tempo = 1.5;
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwrand([0.25,Pseq([0.125],2),
0.5,Pseq([0.125/2],4)],[4,1,1,0.5].normalizeSum,inf),\amp,Pwhite(0.2,1));
~h.play;
)

//offset percussion patterns for techno feel behind a basic kick
(
p.clock.tempo = 135/60;
~c = Pbind(\instrument,\bplay,\buf,d["sfx"][6],\dur,Pbjorklund2(Pexprand(2,15).round(1),
16,inf,Pwhite(1,5).asStream)/4,\amp,1,\rate,2.2);
~c2 = Pbind(\instrument,\bplay,\buf,d["sfx"][6],\dur,Pbjorklund2(Pexprand(2,15).round(1),
16,inf,Pwhite(1,5).asStream)/4,\amp,1,\rate,1.9);
~k = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,1,\amp,5);
~c.play;
~c2.play;
~k.play;
)
```

```
//snare running forwards and back
(
p.clock.tempo = 150/60;
~sn = Pbind(\instrument,\bplay,\buf,d["s"][4],\dur,Pwhite(1,4)/2,\amp,
1,\rate,Prand([1,-1],inf),\pos,Pkey(\rate).linlin(-2,2,0.9,0));
~sn.play;
)

//Here, the `freq` argument is the pitch of the oscillator. Pitch can be specified manually,
like so:
(
~sinfb = Pbind(\instrument,\sinfb,\dur,0.25,\freq,Pwhite(100,900));
~sinfb.play;
)

//run up and down chromatic scale one degree at a time
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.chromatic(\et12),
\degree,Pseq((0..12).pyramid.mirror,inf),\octave,6,\dur,0.125/2,\amp,0.3,\fb,0.8,\rel,0.1);
~sinfb.play;
)
```

3.2 - Basic Rhythms.scd

```
//3.2 - Basic Rhythms - Examples

//Load setup file to get examples working
("../../Setup/Setup.scd").loadRelative

//basic kick
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)

//alternate-beat snare
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,2,\amp,1);
~sn.play;
)

//basic hi-hat pattern
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,0.25,\amp,Pwhite(0.25,1));
~h.play
)

//3/4 note clap
(
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,0.75,\amp,1);
~c.play;
)

//off-beat open hi-hat
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][0],\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,1);
~oh.play;
)
```

3.3 - Techniques for Modifying Rhythm - Examples.scd

```
//load setup file
("../../Setup/Setup.scd").loadRelative

//Random rhythm with Pwhite
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pwhite(1,5.0),\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwhite(0.25,0.75),\amp,Pwhite(0.2,1));
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pwhite(0.75,2),\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pwhite(1,5.0),\amp,1);
~sn.play;~h.play;~c.play;~t.play;
)
//even with a regular kickdrum the other rhythms don't sound good
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play
)

//same example but with all rhythms constrained
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pwhite(1,5.0).round(1),\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwhite(0.25,0.75).round(0.25),
\amp,Pwhite(0.2,1));
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pwhite(0.75,2).round(0.75),\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pwhite(1,5.0).round(0.5),\amp,1);
~sn.play;~h.play;~c.play;~t.play;
)
//sounds more palatable with everything arranged properly
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play
)

//same example again
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pwhite(1,5.0).round(1),\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwhite(0.25,0.75).round(0.25),
\amp,Pwhite(0.2,1));
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pwhite(0.75,2).round(0.75),\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pwhite(1,5.0).round(0.5),\amp,1);
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~sn.play;~h.play;~c.play;~t.play;~k.play;
)
//added whole note fx, short, medium and long.
(
~fx1 = Pbind(\instrument,\bplay,\buf,d["sfx"][0],\dur,Pwhite(1,5),\amp,1);
~fx2 = Pbind(\instrument,\bplay,\buf,d["fx"][0],\dur,Pwhite(1,10),\amp,1);
~fx3 = Pbind(\instrument,\bplay,\buf,d["lfx"][0],\dur,Pwhite(10,40),\amp,1);
~fx1.play;~fx2.play;~fx3.play;
)

//layering at different pitches – kicks
(
p.clock.tempo = 2.3;
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1,1.2],inf));
~k.play;
)
```

```
//kicks at a different pitch. Evaluate this a few times to get different permutations
(
~k2 = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1,1.8],inf)*4);
~k2.play;
)

//layering of slightly different rhythms
//rhythm 1
(
p.clock.tempo = 1.7;
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pseq([1,1,1,0.5],inf),\amp,1);
~t.play;
)
//rhythm 2, using a different tom for contrast
//also re-evaluating rhythm 1 to get them playing together
(
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pseq([1,1,1,0.5],inf),\amp,1);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][1],\dur,Pseq([1,1,1,0.25],inf),\amp,1);
~t2.play;
)
//rhythm 3 for more
(
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pseq([1,1,1,0.5],inf),\amp,1);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][1],\dur,Pseq([1,1,1,0.25],inf),\amp,1);
~t3 = Pbind(\instrument,\bplay,\buf,d["t"][2],\dur,Pseq([1,1,1,0.75],inf),\amp,1);
~t3.play;
)
//kick underneath to illustrate
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,
1,\rate,1);
~oh.play;
~k.play;
)

//complimentary rhythms:
//the 'polyrhythmic clap' from the Basics example
(
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,0.75,\amp,1);
~c.play;
)
//clap added at a similar rhythm (euclidean 3,8)
(
~c2 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pbjorklund2(3,8)/4,\amp,1);
~c2.play;
)
```

```
//StageLimiter throttling
//a complex rhythm
(
l = Prewrite(1, // start with 1
(    1: [0.25,2],
0.25: [1,0.75,0.1,0.3,0.6,0.1],
0.1: [0.5,1,2],
2: [0.5,0.75,0.5,1]
), 4);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,l/2,\amp,1,\rate,2);
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l*2,\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,l,\amp,1,\rate,Pseq([1.2,1.4,1.7],inf));
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,l*4,\amp,1,\rate,0.8);
~ding = Pbind(\instrument,\bplay,\buf,d["ding"][0],\dur,Pwhite(1,5),\amp,1,\rate,0.2);
~h.play;~c.play;~t.play;~ding.play;~sn.play;
)
//extremely loud kick throttles everything elese
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,4,\amp,100,\rate,0.5);
~k.play;
)

//trap(ish) hi-hats
//Has a choice of four rhythmic patterns with lesser chance for each, results in a mostly
0.25-duration hat which can potentially go quite quickly
(
p.clock.tempo = 75/60;
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pwrand([0.25,Pseq([0.125],4),Pseq([0.25]/
3,3),Pseq([0.125]/2,4)],[0.6,0.3,0.09,0.01],inf),\amp,1,\rate,2);
~h.play;
)

//occasional variation on 4/4 kick
(
p.clock.tempo = 2.3;
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pwrand([1,Pseq([0.75],4),Pbjorklund2(3,8,1)/
4],[0.9,0.08,0.02],inf),\amp,1);
~k.play
)
//open hat for reference
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,
1,\rate,1.4);
~oh.play;
)

//cutoff percussion. This Pbind uses (0..100)/100 to split the sample into 100 sections of
0.03 and play over them
(
p.clock.tempo = 2.4;
~perc = Pbind(\instrument,\vplay,\buf,d["fx"][1],\rel,0.03,\dur,0.25,\pos,Pseq((0..100)/
100,inf));
~perc.play;
)

//sputtering rhythms based on long percussion sounds
//the Prand for \buf is a flattened array of all fx sounds. If it wasn't flat it would play
all sounds from any fx entry all at once
(
p.clock.tempo = 2.3;
~perc = Pbind(\instrument,\vplay,\buf,Prand([d["fx"],d["sfx"],d["lfx"]].flat,inf),\rel,
0.1,\dur,0.25,\pos,Pwhite(0,0.9),\rate,Pwhite(1,3.0));
~perc.play;
)
```

```
//choose from literally every sample there is in d. Buggy because it'll also play anything
else that is in there, but good for a laugh.
(
~perc = Pbind(\instrument,\vplay,\buf,Prand(d.values,inf),\rel,0.1,\dur,
0.25,\pos,Pwhite(0.0,0.9),\rate,Pwhite(1,3.0));
~perc.play;
)

//back-and-forth snare
(
~sn = Pbind(\instrument,\vplay,\buf,d["s"][0],\dur,Pbjorklund2(Pwhite(1,6),16)/4,\amp,
1,\rate,Prand([-1,1],inf),\pos,Pkey(\rate).linlin(-1,1,0.99,0));
~sn.play;
)


//.normalizeSum rhythmic spread
//spreading 1-20 over four beats
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pseq((1..20).normalizeSum,inf)*4,\amp,Pwhite(0.2,1));
~h.play;
)
//spreading 1-200 over sixteen beats (gives overtone)
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pseq((1..200).normalizeSum,inf)*16,\amp,Pwhite(0.2,1));
//spreading 1-18 over 8 beats
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],
\dur,Pseq((1..18).normalizeSum,inf)*8,\amp,Pwhite(0.2,1));


//using the \stretch argument - each time a cycle completes change the stretch duration
//a non-synthdef argument is created here - \euclidNum is used to inform both \dur and
\stretch to ensure both work with the same number of onsets
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\euclidNum,Pwhite(1,7),
\dur,Pbjorklund2(Pkey(\euclidNum),8)/4,\amp,1,\rate,Pseq([3,4,5],inf),
\stretch,Pseq([1,0.5,0.25,2],inf).stutter(Pkey(\euclidNum).asStream));
~k.play;
)
```

3.4- Euclidean Rhythms - Examples.scd

```
/*

These examples are extracted from section 3.x — Euclidean Rhythms

Individual examples are separated by a blank line

I'd advise hard-stopping (Ctrl+.) between examples

*/

//Load setup file to get examples working
("../../Setup/Setup.scd").loadRelative

//four 'randomised' rhythms, sounds okay.
(
p.clock.tempo = 2.2;
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,Pwhite(0.25,1).round(0.25),\amp,1);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][1],\dur,Pwhite(0.25,1).round(0.25),\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][1],\dur,Pwhite(0.25,1).round(0.25),\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pwhite(0.25,1).round(0.25),\amp,1);
~k.play;
~sn.play;
~h.play;
~t.play;
)

//four randomised euclidean rhythms with four different samples.
//sounds better, producing a much greater variety of rhythmic forms.
(
p.clock.tempo = 2.2;
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,Pbjorklund2(Pwhite(1,8),Pwhite(1,16))/4,\amp,
1);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][1],\dur,Pbjorklund2(Pwhite(1,8),Pwhite(1,16))/
4,\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][1],\dur,Pbjorklund2(Pwhite(1,8),Pwhite(1,16))/
4,\amp,1);
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(Pwhite(1,8),Pwhite(1,16))/4,\amp,
1);
~k.play;
~sn.play;
~h.play;
~t.play;
)

//Complex rhythm that obfuscates the central rhythmic centre
(
p.clock.tempo = 1.45;
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pbjorklund2(Pwhite(10,35),
41,inf,Pwhite(0,10).asStream)/8,\amp,Pexprand(0.1,1),\pan,-1);
~h2 = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pbjorklund2(Pwhite(10,35),
40,inf,Pwhite(0,10).asStream)/8,\amp,Pexprand(0.1,1),\pan,1);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pbjorklund2(Pwhite(1,5),Pwhite(1,32))/
4,\amp,1,\rate,Pwrand([1,-1],[0.8,0.2],inf),\pos,Pkey(\rate).linlin(1,-1,0,0.999));
~ding = Pbind(\instrument,\bplay,\buf,d["ding"][0],\dur,Pbjorklund2(Pwhite(1,3),25)/4,\amp,
0.6,\rate,0.6,\pan,-1);
~ding2 = Pbind(\instrument,\bplay,\buf,d["ding"][0],\dur,Pbjorklund2(Pwhite(1,3),20)/4,\amp,
0.6,\rate,0.7,\pan,1);
~t1 = Pbind(\instrument,\bplay,\buf,d["mt"][0],
\dur,Pbjorklund2(Pseq([1,1,1,Pwhite(10,15,1).asStream],inf),36,inf,Pwhite(0,2).asStream)/
8,\amp,1);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][0],
\dur,Pbjorklund2(Pseq([1,1,1,Pwhite(10,15,1).asStream],inf),40,inf,Pwhite(0,2).asStream)/
8,\amp,1,\rate,2);
```

```
~t1.play;~t2.play;~h.play;~h2.play;~sn.play;~ding.play;~ding2.play;
)
//a slightly more rhythmic element, tracing the rhythm out a bit more
(
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\degree,Pwrand([0,4],
[0.8,0.2],inf),\octave,Pwrand([2,3],[0.8,0.2],inf),\dur,0.125,\d,0.25,\a,Pexprand(0.0001,200),
\pan,0,\amp,1);
~ring1.play
)
//Add unce unce unce and simmer gently to unify flavours.
(
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\degree,Pwrand([0,4],
[0.8,0.2],inf),\octave,Pwrand([2,3,4],[0.6,0.2,0.2],inf),\dur,0.125,\d,
0.2,\a,Pexprand(0.02,900),\pan,0,\amp,1);
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,0.5,\amp,2);
~k.play;
)
//offbeat hat because cheesy rhythms are good fun
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,Pseq([0.5,Pseq([1],inf)],inf)/2,\amp,1);
~oh.play
)

//working with offsets – doing a lot with a little
//basic kick
(
p.clock.tempo = 2.13;
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)
//Basic 5-16 euclidean rhythm
(
~c = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16)/4,\amp,0.7);
~c.play;
)
//add another layer at a different pitch
//NOTE: These two might not sound at the same time even though they are the same rhythm, as the
rhythmic cycle is longer than 1 beat
(
~c2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16)/4,\amp,0.7,\rate,1.1);
~c2.play;
)
//if you want them to sound together, trigger them together
(
~c2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16)/4,\amp,0.7,\rate,1.1);
~c = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16)/4,\amp,0.7);
)
//offset both
(
~c = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16,inf,Pwhite(1,10).asStream)/
4,\amp,0.7);
~c2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(5,16,inf,Pwhite(1,15).asStream)/
4,\amp,0.7,\rate,1.1);
~c.play;
~c2.play;
)
//and another, slightly different sample
(
~c3 = Pbind(\instrument,\bplay,\buf,d["t"][1],\dur,Pbjorklund2(5,16,inf,Pwhite(0,8).asStream)/
4,\amp,0.7,\rate,0.9);
~c3.play
)
//now do the same to the kick
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1,1.2],inf));
)
//another kick, slightly different rhythm
(
~k2 = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(3,16,inf,Pwhite(1,10).asStream)/
4,\amp,1,\rate,Pseq([1.1,1.4],inf));
~k2.play;
)
//add sub kick on 1, and you have minimal techno.
(
~sk = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,1,\amp,2);
~sk.play;
)


//give a central rhythm to be used by other patterns
l = Pbjorklund2(Pseq([3,3,3,4,3,3,3,5],inf),8)/4;
//block-execute (Ctrl/Cmd+Enter) between these brackets
(
p.clock.tempo = 2.1;
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1,\rate,0.9);
~c3 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1,\rate,1.1);
~c2 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1);
~c.play;
~c2.play;
~c3.play;
)
//now individually evaluate (Shift+Enter) some of these lines to refresh the 'dur'. Listen for
variations in rhythm.
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1,\rate,0.9);
~c3 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1,\rate,1.1);
~c2 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1);
//if you want to reset, execute the block again


//A more fleshed-out example
//Start with a random central rhythm, to keep all of the individual parts seperate
//also using a scale as a one-letter variable for quickness
(
p.clock.tempo = 2.32;
l = Pbjorklund2(Pwhite(3,10),16)/4;
e = Scale.chromatic(\et53);
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,0,\degree,Pwhite(-2,2),
\octave,Pwrand([3,4],[0.8,0.2],inf),\dur,l,\d,0.4,\a,Pexprand(0.5,30),\amp,0.5,\pan,1);
~ring2 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,0,\degree,Pwhite(-2,2),
\octave,Pwrand([3,4],[0.8,0.2],inf),\dur,l,\d,0.4,\a,Pexprand(0.5,30),\amp,0.5,\pan,-1);
~ring3 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,0,\degree,Pwhite(-5,5),
\octave,Pwrand([4,5],[0.8,0.2],inf),\dur,l,\d,0.5,\a,Pexprand(0.5,30),\amp,0.5,\pan,0);
~ring4 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,0,\degree,Pwhite(-5,5),
\octave,Pwrand([2,3],[0.8,0.2],inf),\dur,l,\d,0.2,\a,Pexprand(0.5,200),\amp,0.9,\pan,0);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,l,\amp,1);
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,l,\amp,Pwhite(0.2,1));
~ring1.play;~ring2.play;~ring3.play;~ring4.play;~sn.play;~c.play;~h.play;
)
//unify all of these rhythms
//sounds very different, with all elements sounding at the same time.
//execute individual lines to make them diverge from this pattern
(
p.clock.tempo = 2.32;
l = Pbjorklund2(Pseq([3,8,2,5,9,10,14,3,5,5,4,9,14],inf),16)/4;
e = Scale.chromatic(\et53);
```

```
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,0,\degree,Pwhite(-2,2),
\octave,Pwrand([3,4],[0.8,0.2],inf),\dur,l,\d,0.4,\a,Pexprand(0.5,90),\amp,0.5,\pan,1);
~ring2 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,0,\degree,Pwhite(-2,2),
\octave,Pwrand([3,4],[0.8,0.2],inf),\dur,l,\d,0.4,\a,Pexprand(0.5,90),\amp,0.5,\pan,-1);
~ring3 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,0,\degree,Pwhite(-5,5),
\octave,Pwrand([4,5],[0.8,0.2],inf),\dur,l,\d,0.5,\a,Pexprand(0.5,90),\amp,0.5,\pan,0);
~ring4 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,e,\root,0,\degree,Pwhite(-5,5),
\octave,Pwrand([2,3],[0.8,0.2],inf),\dur,l,\d,Pexprand(0.2,0.6),\a,Pexprand(1,900),\amp,
0.9,\pan,0);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,l,\amp,1);
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,l,\amp,1);
~h = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,l,\amp,Pwhite(0.2,1))
)
//throw some straight rhythms in to show where the beat lies - this one i could genuinely
listen to for a while...
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,1,\rate,1,\amp,3);
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,Pseq([0.5,Pseq([1],inf)],inf),
\amp,Pwhite(0.5,1),\rate,0.8);
~k.play;
~oh.play;
)
```

3.5- StageLimiter Abuse - Examples.scd

```
//load setup file
("../../Setup/Setup.scd").loadRelative

//a complex polyrhythm
(
p.clock.tempo = 2.3;
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,0.75,\amp,1);
~c2 = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,Pbjorklund2(Pseq([3,3,3,5],inf),8)/4,\amp,
1);
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,
1,\stretch,Pwhite(1,0.25).round(0.25));
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pbjorklund2(Pwhite(3,10),16),\amp,1);
~t1 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,1/5*4,\amp,1);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,1/9*4.5,\amp,1,\rate,2);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pbjorklund2(Pwhite(10,16),16)/
8,\amp,Pwhite(0.2,1.4));
~fx1 = Pbind(\instrument,\bplay,\buf,d["sfx"][0],\dur,Pwhite(1,4.0).round(0.5),\amp,1);
~fx2 = Pbind(\instrument,\bplay,\buf,d["sfx"][1],\dur,Pwhite(1,8.0).round(0.25),\amp,1);
~c.play;~c2.play;~oh.play;~sn.play;~t1.play;~t2.play;~h.play;~fx1.play;~fx2.play;
)
//A 0db kick which doesn't really do anything in the mix
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)
//A >0dB kick which compresses everything else and audibly 'centers' everything around it
because it is so loud.
//There's probably some psychoacoustics involved in this that i'm not qualified to talk about.
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,4);
~k.play;
)
//a really *really* loud, very occasional percussion which silences everything else (slowed
down for exaggerated effect)
(
~hugesnare = Pbind(\instrument,\bplay,\buf,d["mt"][0],\dur,Pwhite(8,16),\amp,4000000,\rate,1);
~hugesnare.play;
)


//some beautiful pads
//thanks Eli Fieldsteel
(
p.clock.tempo = 2;
(
~chords = Pbind(\instrument,\bpfsaw,
\dur,Pwhite(4.5,7.0,inf),
\midinote,Pxrand([
[23,35,54,63,64],
[45,52,54,59,61,64],
[28,40,47,56,59,63],
[42,52,57,61,63],
],inf),
\detune, Pexprand(0.0001,0.1,inf),
\cfmin,100,
\cfmax,1500,
\rqmin,Pexprand(0.02,0.15,inf),
\atk,Pwhite(2.0,4.5,inf),
\rel,Pwhite(6.5,10.0,inf),
\ldb,6,
\amp,Pwhite(0.8,2.0),
\out,0)
);
~chords.play;
)
//pulse them slightly with a low-passed kick
(
~k = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,Pbjorklund2(3,8)/2,\amp,2);
//Low Pass
~lpfSend = {[~k]};
~lpf = {RLPF.ar(Mix.ar([~lpfSend]),SinOsc.kr(0.1).range(200,100),1)};
~lpf.play;
)
//eliminate them completely with an absurdly loud low-passed kick (those with subwoofers be
careful!)
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,9000,\rate,5);
//Low Pass
~lpfSend = {[~k]};
~lpf = {RLPF.ar(Mix.ar([~lpfSend]),SinOsc.kr(0.1).range(100,80),0.3)};
~lpf.play;
)
```

3.6- L-Systems For Rhythm - Examples.scd

```
//load setup file
("../../Setup/Setup.scd").loadRelative

//use L-system as a duration value for a kickdrum
(
l = Prewrite(1, // start with 1
(    1: [0.25,2],
0.25: [3,3,2]/4,
3/4: [0.25,1,0.125,0.125],
), 4);
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,l,\amp,1);
~k.play;
)
/*

With that grammar:

1 -> 0.25,2 -> 3/4,3/4,2/4 -> 0.25,1,0.125,0.125,0.25,1,0.125,0.125 -> etc.

*/
//much like with the euclidean rhythm convergence/divergence pattern, you can use variable l
for different patterns too
(
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,l,\amp,1,\rate,Pseq((1..4)/2,inf));
~sn.play;
)
//and transform it
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,l,\stretch,Pwhite(0.5,2).round(0.5),
\amp,Pwhite(0.2,1));
~h.play;
)
//an off-beat open hat for reference
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,1);
~oh.play;
)

//Building non-grid rhythms into L-systems, and adding complexity beyond self-similar patterns
//Super basic L-system
(
l = Prewrite(1,
(
//equal to 2 duration units/beats
1: #[0.25,0.5,0.5,0.25,2],
0.25: #[1],
),15);
//play a hi-hat with that L-system as a rhythm
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,l,\amp,0.8);
~h.play;
);
//Make the L-system more complex. '2' expands into a rhythm that will still fall with the
emphasis of the beat.
// This is done by creating an array of random numbers, and using normalizeSum to constrain the
array to equalling 1 overall, and multiplying it to spread those random numbers across multiple
beats.
//To get an idea of how this works, evaluate this a few times and look at the output in the
post window
Array.fill(rrand(4,10),{rrand(1,10)}).normalizeSum * rrand(1,4);
//use this is a variable within an evaluation (when this block is executed, rhythm will remain
as a local variable within that evaluation)
```

```
//when '2' is expanded, it will expand into a random rhythm
(
var rhythm = Array.fill(rrand(4,10),{rrand(1,10)}).normalizeSum * rrand(1,4);
l = Prewrite(1,
(
//equal to 2 duration units/beats
1: #[0.25,0.5,0.5,0.25,2],
0.25: #[1],
2: rhythm
),15);
//play a hi-hat with that L-system as a rhythm
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,l,\amp,0.8);
~h.play;
);
//although h contains some really off-kilter rhythms it will resolve back to the beat, and will
also still contain the self-similar patterns laid out in the rest of the L-system, combining
self-simliar patterns and 'random' rhythms.
//check this with an on-beat kick
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play
//more extreme values can be used of course
(
var rhythm = Array.fill(rrand(30,40),{rrand(1,30)}).normalizeSum * rrand(1,4);
l = Prewrite(1,
(
//equal to 2 duration units/beats
1: #[0.25,0.5,0.5,0.25,2],
0.25: #[1],
2: rhythm
),15);
//play a hi-hat with that L-system as a rhythm
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,l,\amp,0.8);
~h.play;
);
//Note every time '2' is expanded into a randomly generated rhythm it will expand differently.

//Reduced version of HSPTLFLDHS
//set the tempo
p.clock.tempo = 2;

//Evaluate the entire block with Cmd/Ctrl+Enter, if the post window shows 'FAILURE IN SERVER /
s_new too many nodes', wait a couple of seconds and evaluate again.
(
//The L-System used to generate rhythm. For more information see the Prewrite class
(
l = Prewrite(0.25,
(
0.25: #[0.25,0.25,0.25,0.5],
0.5: #[0.25,0.5,0.125,0.125,0.125,0.125],
0.125: #[0.375,0.125],
0.375: #[0.375,0.375,1],
1: #[0.75,0.25],
0.75: #[16]
),60)
);
//Offsets for each L-system, used to create 'repeating' effects.
//These will sound when both the offset AND the sample instruments are evaluated, the offset
will _not_ be inserted on the fly.
//For example, change d[\offstab2] to 0.125 and d[\offstab3] to 0.25, and re-evaluate the
entire
d[\offk] = Pseq([0],1);
d[\offh] = Pseq([0],1);
d[\offsn] = Pseq([0],1);
//Multipliers for durations - A multiplier of 2 will double the duration, 0.5 will half the
duration, etc.
//Multipliters generally work best if kept to even multiplications & divisions of 1
//These will sound when they are evaluated and can be changed on the fly
//Once the multipliers have been changed it will be very difficult to make individual sampes
sound in unison
//The EP starts with all of these in unison and gradually diverges
//NOTE: Very low multipliers or a zero multiplier will crash the server. Exercise caution.
~multk = {3};
~multh = {1};
~multsn = {4};
//kicks
~k = Pbind(\instrument,\bplaym,\buf,d["k"][0],\dur,Pseq([d[\offk],(l * Pkr(~multk))],inf),\amp,
1);
~h = Pbind(\instrument,\bplaym,\buf,d["ch"][0],\dur,Pseq([d[\offh],(l * Pkr(~multh))],inf),
\amp,0.8,\rate,0.8,\pan,Pwhite(-0.8,0.8).stutter(Pwhite(40,100).asStream));
~sn = Pbind(\instrument,\bplaym,\buf,d["s"][1],\dur,Pseq([d[\offsn],(l * Pkr(~multsn))],inf),
\amp,0.8,\rel,1,\pan,Pwhite(-0.8,0.8).stutter(Pwhite(40,100).asStream),\rate,1);
)
//play these to bring in the various samples, then change the multiplier, or change the offset
and re-evaluate either the block or re-evaluate the offset and then the Proxy that uses it.
~k.play;
~sn.play;
~h.play;
```

3.7- Looping.scd

```
//load setup file
("../../Setup/Setup.scd").loadRelative

//set tempo to level of breaks
p.clock.tempo = 175/60

//make background for loops
~stab = Pbind(\instrument,\bplay,\buf,d["stab"][0],\dur,Pbjorklund2(3,8)/
4,\rate,Pseq([1,1,0.9,1.1],inf).stutter(3),\amp,0.6)


//reload SynthDefs.scd for the updated tempo
("../../Setup/SynthDefs.scd").loadRelative

//make breaks using lplay
~loop = Pbind(\instrument,\lplay,\buf,d["breaks175"][4],\dur,16,\amp,1)
~loop.play

//if they are out of sync, trigger together
(
~loop = Pbind(\instrument,\lplay,\buf,d["breaks175"][4],\dur,16,\amp,1);
~stab = Pbind(\instrument,\bplay,\buf,d["stab"][0],\dur,Pbjorklund2(3,8)/
4,\rate,Pseq([1,1,0.9,1.1],inf).stutter(3),\amp,0.6)
)

//if you're going to be reloading the tempo a lot it might be nice to specify this as a
function
a = {("../../Setup/SynthDefs.scd").loadRelative}

p.fadeTime = 0

(
//set random tempo
p.clock.tempo = rrand(2.0,3.0);
//reload synthdefs
a.();
~loop = Pbind(\instrument,\lplay,\buf,d["breaks175"][4],\dur,16,\amp,1);
~stab = Pbind(\instrument,\bplay,\buf,d["stab"][0],\dur,Pbjorklund2(3,8)/
4,\rate,Pseq([1,1,0.9,1.1],inf).stutter(3),\amp,0.6)
)

//more stabs for fun
~stab2 = Pbind(\instrument,\bplay,\buf,d["stab"][0],\dur,Pbjorklund2(Pwhite(4,11),16)/
4,\rate,Pseq([2,2,2,2,2,2,1.8,2.2],inf).stutter(3),\amp,0.7)
~stab2.play

//in its current state, it is a little bit janky, but it does work to set loops to tempo
```

4.1 - Pitch And Patterns - Examples.scd

```
//load the setup file
("../../Setup/Setup.scd").loadRelative

//freq specifying a raw pitch value
(
~sinfb = Pbind(\instrument,\sinfb,\freq,Pwhite(100,800),\dur,0.1,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb.play;
)
//frequency being detuned gradually using a gradual increasing of \detine argument
(
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..8),inf)*100,\dur,0.1,\amp,0.3,\fb,0.4,\rel,
1,\detune,Pseq((1..400),inf));
)

//using scales inside of Pbinds
//Minor scale in Just intonation, octave varying between 4 and 6, root note varying between 0
and 4 each scale repetition.
//\detune can also be used on top of this to detune scale degrees
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.minor(\just),\root,Pwhite(0,4).stutter(8),
\octave,Pwhite(4,6).stutter(8),\degree,Pseq((0..7),inf),\dur,0.25,\amp,0.3,\fb,1,\rel,0.2);
~sinfb.play;
)

//Chords used by specifying a 2-dimensional array in \degree argument.
//same can be done for the \octave argument
(
~sinfb = Pbind(\instrument,\sinfb,
\scale,Scale.major,
\root,0,
\octave,Pwrand([4,[3,4],[2,3,4]],[0.9,0.08,0.02],inf),
\degree,Prand([[0,2,4],[2,4,6],[7,2,4],[1,2,3],[0,-2,-4]],inf),
\dur,Pwhite(5,10),
\atk,2,\sus,1,\rel,3,\amp,0.3,\fb,0.1);
~modulation = {SinOsc.kr(0.1).range(0.01,1.41)};
~sinfb.play;
~sinfb.set(\fb,~modulation);
)
```

4.2 - Types of Pitch Arrangement - Examples.scd

```
//load the setup file
("../../Setup/Setup.scd").loadRelative

//chords I, IV and V
//in Major and Minor - re-evaluate for a different scale (using the .choose method)
(
~chords = Pbind(\instrument,\bpfsaw,
        \dur,Pwhite(4.5,7.0,inf),
        \scale,[Scale.minor,Scale.major].choose,
        \degree,Pwrand([[0,2,4],[3,5,7],[4,6,8]],[0.5,0.25,0.25],inf),
        \cfmin,100,
        \cfmax,1500,
        \rqmin,Pexprand(0.02,0.15,inf),
        \atk,Pwhite(2.0,4.5,inf),
        \rel,Pwhite(6.5,10.0,inf),
        \ldb,6,
        \lsf,1000,
        \octave,Pwrand([4,3,5],[0.6,0.3,0.1],inf),
        \amp,Pwhite(0.8,2.0),
        \out,0);
~chords.play;
);
//major/minor scale chords with a fairly melody which meanders around the major/minor scale,
but sounds consonant at the vast majority of points
//scale stored in a dictionary key so that it can be used in both Pbinds easily
(
d[\scale] = [Scale.major,Scale.minor].choose;
~chords = Pbind(\instrument,\bpfsaw,
        \dur,Pwhite(4.5,7.0,inf),
        \scale,d[\scale],
        \degree,Pwrand([[0,2,4],[3,5,7],[4,6,8]],[0.5,0.25,0.25],inf),
        \cfmin,100,
        \cfmax,1500,
        \rqmin,Pexprand(0.02,0.15,inf),
        \atk,Pwhite(2.0,4.5,inf),
        \rel,Pwhite(6.5,10.0,inf),
        \ldb,6,
        \lsf,1000,
        \octave,Pwrand([4,3,5],[0.6,0.3,0.1],inf),
        \amp,Pwhite(0.8,2.0),
        \out,0);
~chords.play;
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\root,0,\octave,[4,5],\degree,Place([0,0,2,
[4,5,6],[7,1,2],[6,7,8,9],[10,12,14,15],7,6,5],inf),\dur,Pbjorklund2(Pwhite(6,8),8)/4,\amp,
0.4,\fb,0.9,\rel,0.2);
~sinfb.play
);

//ChordProg - house chords with chord names in an array to make a chord sequence...
//Today is gonna be the day that they're gonna throw it back to you...
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.chromatic,\octave,4,\degree,Pseq([\Em7,\G,
\Dsus4,\A7sus4].chordProg,inf).stutter(6),\dur,1,\atk,0.8,\amp,0.3,\fb,0.1,\rel,1);
~sinfb.play
)

//giant steps. Apparently.
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.chromatic,\octave,
4,\degree,Pseq([\Bmajor7,\D7,\Gmajor7,\Bb7,\Ebmajor7,\Am7,\D7,\Gmajor7,\Bb7,\Ebmajor7,\Gb7,\Bma
jor7,\Fm7,\Bb7,\Ebmajor7,\Am7,\D7,\Gmajor7,\Dbm7,\Gb7,\Bmajor7,\Fm7,\Bb7,\Ebmajor7,\Dbm7,\Gbm7]
.chordProg,inf),\dur,1,\atk,0.1,\amp,0.3,\fb,0.1,\rel,1);
```

```
~sinfb.play;
)

//a musical example in context
(
p.clock.tempo = 180/60;
~chords = Pbind(\instrument,\bpfsaw,
        \dur,Pwhite(9.5,15.0,inf),
        \scale,Scale.chromatic,
        \degree,Pxrand([\Em,\Am7,\Bm7].chordProg,inf),
        \cfmin,100,
        \cfmax,1500,
        \detune,Pexprand(0.0001,1),
        \rqmin,Pexprand(0.02,0.15,inf),
        \atk,Pwhite(2.0,4.5,inf),
        \rel,Pwhite(6.5,10.0,inf),
        \ldb,13,
        \lsf,1000,
        \octave,Pwrand([4,5,6],[0.8,0.15,0.05],inf),
        \amp,Pwhite(0.8,1.5),
        \out,0);
~chords.play;
~oh = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,Pbjorklund2(Pwhite(10,16),16)/4,\amp,
0.4,\pan,0.2,\rate,Pwhite(1.7,2));
~t = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(Pwhite(10,16),16)/4,\amp,
0.8,\pan,-0.2,\rate,2);
~t2 = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(Pwhite(10,16),16)/4,\amp,
0.8,\pan,-0.2,\rate,4);
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(Pwrand([3,6],[0.8,0.2],inf),8)/
4,\amp,1);
~c = Pbind(\instrument,\bplay,\buf,d["c"][0],\dur,4,\amp,4);
~oh.play;~t.play;~k.play;~c.play;~t2.play;
)

//Alternative scales
//Evaluate to select a scale using the ET12 tuning and run it in ascending order, there are a
number of scales so evaluate this a bunch of times
//scales are stored in a dictionary to be referred to multiple times within the ~sinfb pbind
(
p.clock.tempo = 1;
d[\scale] = Scale.choose.postln;
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\octave,
4,\degree,Pseq((0..d[\scale].degrees.size-1),inf),\dur,0.25,\amp,0.3,\fb,0.6,\rel,0.3);
~sinfb.play;
)

//Microtonal scales
(
p.clock.tempo = 1;
d[\scale] =
[Scale.zamzam,Scale.chromatic24,Scale.partch_o1,Scale.husseini,Scale.zanjaran,Scale.bhairav].ch
oose.postln;
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\octave,
4,\degree,Pseq((0..d[\scale].degrees.size-1),inf),\dur,0.25,\amp,0.3,\fb,0.6,\rel,0.3);
~sinfb.play;
)


//Alternative Tunings
//Chromatic scale in a random tuning - some relatively subtle differences here
(
p.clock.tempo = 1;
d[\scale] = Scale.chromatic(Tuning.choose);
```

```
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\octave,
4,\degree,Pseq((0..d[\scale].degrees.size-1),inf),\dur,0.25,\amp,0.3,\fb,0.6,\rel,0.3);
~sinfb.play;
)

//A musical example of alternative tunings
//one of my favourites is the et53 tuning, using it to slightly disturb a central pitch on
multiple instruments, sounds really nice in acid-type music
//by selectively deploying et53, a very narrow pitch range can become normal, making large
pitch leaps within an octave seem huge when used.
(
p.clock.tempo = 150/60;
d[\scale] = Scale.chromatic(\et53);
l = Pbjorklund2(Pwhite(1,13),16)/4;
//notice the \degree argument - ranges from -8 to +8, but this difference is nowhere near an
octave
~ring3 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,d[\scale],\degree,Pwhite(-8,8),
\octave,Pwrand([2,3],[0.8,0.2],inf),\dur,l,\d,0.24,\a,Pexprand(10,400),\pan,0,\amp,1.5);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][1],\dur,l,\amp,0.8);
~h = Pbind(\instrument,\bplay,\buf,d["ch"][1],\dur,l,\amp,0.8);
~k = Pbind(\instrument,\bplay,\buf,d["k"][1],\dur,1,\amp,2);
~ring3.play;~sn.play;~h.play;~k.play;
)
//adding more acid lines which diverge even less. Also adding percussion
(
~ring2 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,d[\scale],\degree,Pwhite(-4,4),\octave,
5,\dur,l,\d,0.37,\a,Pexprand(1,40),\pan,1,\amp,0.5);
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,d[\scale],\degree,Pwhite(-4,4),\octave,
4,\dur,l,\d,0.38,\a,Pexprand(1,40),\pan,-1,\amp,0.5);
~ring2.play;~ring1.play;
)
//another acid line that diverges quite a bit. also open hats
(
~oh = Pbind(\instrument,\bplay,\buf,d["oh"][1],\dur,Pseq([0.5,Pseq([1],inf)],inf),\amp,2);
~oh.play;
~ring4 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,d[\scale],\degree,Pwhite(-8,8),\octave,
7,\dur,l,\d,0.21,\a,Pexprand(1,100),\pan,1,\amp,0.8);
~ring4.play;
)
//repetive distorted \sinfb riff, using the whole octave
(
~sinfb = Pbind(\instrument,\sinfb,\scale,d[\scale],\octave,[5,6],\degree,Place([0,0,-52,
[30,20,10],[52,40,25,20],[10,11,9,3,6],[30,36,39,40]],inf),\dur,0.25,\amp,
0.5,\fb,Pwhite(10.5,900.5),\rel,Pexprand(0.1,0.5));
~sinfb.play;
)
//remove percussion
(
~k.stop;~sn.stop;~h.stop;
)



//Harmonic series
//setting up a fundamental frequency as a NodeProxy so that it can be referenced on the fly
(
~r = {75}
)
//a straight run up the harmonic series to 10 partials. Notice how the notes converge the
higher up the harmonic series due out perception of frequency being logarithimic
//note that the \freq argument is a multiplation of a Pkr - a BenoitLib addon which references
an active NodeProxy inside of a pattern
(
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..10),inf)*Pkr(~r),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb.play;
```

```
)
//modulate the fundamental frequency to modulate the playing of entire scale
(
~r = {SinOsc.kr(0.1).range(75,80)}
)
//raise the fundamental freqency from 75Hz to 1000Hz over two minutes using XLine (note that
XLine is used as frequency is logarithmic)
(
~r = {XLine.kr(75,1000,120)}
)


//a run up the harmonic series from 1 to 50 partials – note how close together the notes become

(
~r = {50};
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..50),inf)*Pkr(~r),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb.play;
)

//Multiple identical harmonic frequency riffs that use a different multiplication of the
fundamental frequency
(
~r = {50};
//1x fundamental
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb.play;
)
(
//2x fundamental
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*2),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb2.play;
)
(
//4x fundamental
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*4),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb3.play;
)
(
//8x fundamental
~sinfb4 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*8),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb4.play;
)
//all together to 30:
(
~r = {50};
~sinfb = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*2),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*4),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
~sinfb4 = Pbind(\instrument,\sinfb,\freq,Pseq((1..20),inf)*(Pkr(~r)*8),\dur,0.25,\amp,0.3,\fb,
0.1,\rel,0.3);
)
```

4.3 - Riffs - Examples.scd

```
//load setup file
("../../Setup/Setup.scd").loadRelative

//up-down riff
//harmonic series version
//re-evaluate individual directions to create a different riff
(
//up
p.clock.tempo = 1.5;
~r = {75};
~sinfb1 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10),inf)*Pkr(~r),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,0.4),\rel,0.3);
~sinfb1.play;
)
(
//down
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).reverse,inf)*Pkr(~r),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,0.4),\rel,0.3);
~sinfb2.play;
)
(
//random
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).scramble,inf)*Pkr(~r),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,1.0),\rel,0.3);
~sinfb3.play;
)

//up-down riff
//minor scale version
//re-evaluate individual directions to create a different riff
(
p.clock.tempo = 1.5;
//up
~sinfb1 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,5,\degree,Pseq((0..7),inf),\dur,
0.25,\amp,0.3,\fb,Pwhite(0.1,0.4),\rel,0.2);
~sinfb1.play;
)
(
//down
~sinfb2 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
5,\degree,Pseq((0..7).reverse,inf),\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,0.4),\rel,0.2);
~sinfb2.play;
)
(
//random, an octave higher
~sinfb3 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
6,\degree,Pseq((0..7).scramble,inf),\dur,0.25,\amp,0.3,\fb,Pwhite(0.1,1.0),\rel,0.2);
~sinfb3.play;
)

//replacing duration of 0.25 with a Pwrand which will automatically shift the riffs
(
p.clock.tempo = 1.5;
~sinfb1 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,5,\degree,Pseq((0..7),inf),
\dur,Pwrand([0.25,Pseq([0.125],2)],[0.9,0.1],inf),\amp,0.3,\fb,Pwhite(0.1,0.4),\rel,0.2);
~sinfb2 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
5,\degree,Pseq((0..7).reverse,inf),\dur,Pwrand([0.25,Pseq([0.125],2)],[0.9,0.1],inf),\amp,
0.3,\fb,Pwhite(0.1,0.4),\rel,0.2);
~sinfb3 = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,
5,\degree,Pseq((0..7).scramble,inf),\dur,Pwrand([0.25,Pseq([0.125],2)],[0.9,0.1],inf),\amp,
0.3,\fb,Pwhite(0.1,1.4),\rel,0.2);
)
```

```
~sinfb1.play;
~sinfb2.play;
~sinfb3.play;


//Phasing
//Using the riff from Reich's Piano Phase
//inspired by https://ccrma.stanford.edu/courses/tu/cm2008/topics/piano_phase/index.shtml
(
p.clock.tempo = 1.8;
//riff 1 and 2 evaluated at once so that they start together.
//riff 2 will sometimes play 0.125 duration which will knock the two out of phase
~sinfb1 = Pbind(\instrument,\sinfb,\octave,4,\freq,Pseq([64, 66, 71, 73, 74, 66, 64, 73, 71,
66, 74, 73].midicps,inf),\dur,0.25,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb2 = Pbind(\instrument,\sinfb,\octave,4,\freq,Pseq([64, 66, 71, 73, 74, 66, 64, 73, 71,
66, 74, 73].midicps,inf),\dur,Pwrand([0.25,0.125],[0.99,0.01],inf),\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb1.play;
)
//play riff 2
~sinfb2.play;

//another version which uses a second riff which has a slightly different tempo constantly
(
p.clock.tempo = 1.8;
//riff 1 and 2 evaluated at once so that they start together.
//riff 2 will sometimes play 0.125 duration which will knock the two out of phase
~sinfb1 = Pbind(\instrument,\sinfb,\octave,4,\freq,Pseq([64, 66, 71, 73, 74, 66, 64, 73, 71,
66, 74, 73].midicps,inf),\dur,0.25,\amp,0.3,\fb,0.8,\rel,0.3);
~sinfb2 = Pbind(\instrument,\sinfb,\octave,4,\freq,Pseq([64, 66, 71, 73, 74, 66, 64, 73, 71,
66, 74, 73].midicps,inf),\dur,0.255,\amp,0.3,\fb,0.1,\rel,0.3);
~sinfb1.play;
)
//play riff 2
~sinfb2.play;

//synth stabs - try this with both stab 0 and 1.
(
//stab 1
p.clock.tempo = 2.4;
~stab1 = Pbind(\instrument,\bplay,\buf,d["stab"][1],\euclidNum,Pwhite(3,3),
\dur,Pbjorklund2(Pkey(\euclidNum),8)/4,\amp,
2,\rate,Pseq([1,1,1,1,1,1,0.9,1.1],inf).stutter(3));
~stab1.play;
)
(
//stab 2 - double speed and greater possible number of onsets
~stab2 = Pbind(\instrument,\bplay,\buf,d["stab"][1],\euclidNum,Pwhite(3,11),
\dur,Pbjorklund2(Pkey(\euclidNum),16)/4,\amp,
1,\rate,Pseq([1,1,1,1,1,1,0.9,1.1],inf).stutter(3)*2);
~stab2.play;
)
(
//stab 3 - double speed again and greater possible number of onsets again
~stab3 = Pbind(\instrument,\bplay,\buf,d["stab"][1],\euclidNum,Pwhite(6,16),
\dur,Pbjorklund2(Pkey(\euclidNum),16)/4,\amp,
1,\rate,Pseq([1,1,1,1,1,1,0.9,1.1],inf).stutter(3)*4);
~stab3.play;
)
//drums
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1.1,1.9],inf));
~k2 = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(3,8)/4,\amp,
1,\rate,Pseq([1.1,1.9],inf)*2);
~sn = Pbind(\instrument,\bplay,\buf,d["s"][0],\dur,Pbjorklund2(Pwhite(1,6),16)/4,\amp,1);
~fx = Pbind(\instrument,\bplay,\buf,d["fx"][0],\dur,Pwhite(1,6),\amp,1);
~k.play;~sn.play;~fx.play;~k2.play;
)

//Place - riffs that contain riffs
(
//first riff
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\degree,Place([0,7],inf),
\octave,3,\dur,0.25,\d,0.6,\a,Pseq((1..40),inf),\pan,0,\amp,0.5);
~ring1.play;
)
//stop
~ring1.stop;
(
//second riff
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,
\degree,Place([2,4,3,5,4,6,8,11],inf),\octave,3,\dur,0.25,\d,0.6,\a,Pseq((1..40),inf),\pan,
0,\amp,0.5);
~ring1.play;
)
//stop
~ring1.stop;
(
//two riffs laced together with the longer one on the inner level, playing the first riff and
then a note of the second
~ring1 = Pbind(\instrument,\ring1,\f,Pkey(\freq),\scale,Scale.minor,\degree,Place([0,7,
[2,4,3,5,4,6,8,11]],inf),\octave,3,\dur,0.25,\d,0.6,\a,Pseq((1..40),inf),\pan,0,\amp,0.5);
~ring1.play
)
```

4.4 - Pitch and Static Synths - Examples.scd

```
//load the setup file
("../../Setup/Setup.scd").loadRelative

//set a fundamental frequency
~f = {70}
//a fixed pitch sine wave, using a fundamental frequency
(
~sin = {SinOscFB.ar([~f,~f*1.01],0.7,0.3)};
~sin.play;
)
//4 saw waves that are modulated by LFNoise1 Ugens and arranged around the stereo field
//the frequency of the saw waves is a LFNoise1 that is ranged between the fundamental and ten
times the fundamental
(
~lfn1 = {Splay.ar(Saw.ar(Array.fill(4,{LFNoise1.kr(0.3).range(~f,~f*10)}),0.3))}
~lfn1.play;
)
//now round this LFNoise1 to the fundamental frequency to get the frequency to sweep the
harmonic frequency
(
~lfn1 = {Splay.ar(Saw.ar(Array.fill(4,{LFNoise1.kr(0.3).range(~f,~f*10).round(~f)}),0.3))}
~lfn1.play;
)
//the frequencies are now tuned and sound GREAT (an X/Y scope also looks amazing)
s.scope
//This .range and .round method can be applied to any signal UGen, and also at any
multiplication level. Here's a silly extreme example that sounds like shrill bees
(
~lfn1 = {Splay.ar(Saw.ar(Array.fill(40,
{SinOscFB.kr(rrand(0.1,0.3),rrand(0.1,2)).range(~f,~f*100).round(~f*4)}),0.4))}
~lfn1.play;
)

//Triggered random frequency changes, using something like TRand
(
~f = {81};
~tChange = {Pulse.ar(TRand.kr(~f,~f*10,Dust.kr(4)).round(~f),SinOsc.kr(0.1).abs,
0.6)*SinOsc.ar([~f,~f*1.01])};
~tChange.play;
)

//specific and on-demand frequency changes using Demand.kr - Note that this is *really* verbose
for something to be used live.
//I've used an Impulse.kr that recieves the tempo clock as a trigger to show how these synths
can be synced to a central tempo clock
//Demand is a lot like having a Pattern inside of a UGen's arguments. Look at the helpfile,
it's really cool
(
~f = {66.6};
~dChange = {SawDPW.ar([~f,~f*1.02]*Demand.kr(Impulse.kr(p.clock.tempo*3),
0,Dseq([1,8,2,7,3,6,4,5],inf)),SinOsc.kr(40),0.8)};
~dChange.play;
)
//and a kick to show it's synced
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,1,\amp,1);
~k.play;
)
//and more kicks because i really liked this one
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pbjorklund2(Pwhite(1,15),16)/6,\amp,
2,\rate,Pwrand([1,1.2,1.4,2,4],[0.55,0.2,0.1,1,0.05],inf)*1.5);
~k2 = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,1,\amp,2);
~k2.play;
)

//Scale and DegreeToFreq
//using the Demand example again
//a fifth
(
~scale = {SinOscFB.ar(Scale.minor(\just).degreeToFreq([0,4],48.midicps,1),0.7,0.2)};
~scale.play;
)
//Note that the above does not allow scale notes to be changed once the synth is initiated
~scale = {SinOscFB.ar(Scale.minor(\just).degreeToFreq(TRand.kr(1,10,Impulse.kr(1)),48.midicps,
1),0.7,0.2)};

//using .midicps to determine pitch
~scale = {SinOscFB.ar(TRand.kr(50,80,Impulse.kr([3,3.01])).midicps,0.7,0.5)};
~scale.play
```

4.5 - Between Pitch And Noise - Examples.scd

```
//load the setup file
("../../Setup/Setup.scd").loadRelative

//SinOscFB – A sine wave that can move between pitch and noise and noisy pitch
(
//polling the modulation of the 'feedback' argument, to show the way in which SinOscFB degrades
sine waves
~sinfbstatic = {SinOscFB.ar([330,440],XLine.kr(0.1,500,60).poll(10),0.6)};
~sinfbstatic.play;
)

//a pattern I use regularly with its feedback being modulated from 0 to 20. Notice the
difference in sound across the spectrum
(
~sinfb = Pbind(\instrument,\sinfb,\scale,Scale.minor,\octave,[3,4,5],
\degree,Pseq([0,0,4,5],inf),\dur,Pbjorklund2(3,8)/4,\amp,0.3,\fb,0.1,\rel,0.3);
~feedback = {SinOsc.kr(0.1,-1,1).range(0,20.0).poll(30)};
~sinfb.set(\fb,~feedback);
~sinfb.play;
)

//Extreme modulation of fundamental frequency
//taking the up-down scale given in the 'riffs' section
(
p.clock.tempo = 2.4;
~r = {75};
~sinfb1 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10),inf)*Pkr(~r),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,1.4),\rel,0.1);
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).reverse,inf)*Pkr(~r),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,1.4),\rel,0.1);
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).scramble,inf)*Pkr(~r),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,2.0),\rel,0.1);
~sinfb1.play;~sinfb2.play;~sinfb3.play;
)
//moving the frequency up and beyond sensible into supersonics – after reading around 5000Hz
some interesting aliasing starts to happen
(
~r = {XLine.kr(75,8000,60).poll(10)}
)
//and even further, lower frequencies start reappearing
(
~r = {XLine.kr(8000,30000,60).poll(10)};
)
//using very extreme modulation also gives some interesing results
(
~r = {LFNoise1.kr(0.2).range(30000,90000).poll(10)};
)

//extreme multiplaction of fundamental frequency
//using the previous example, a NodeProxy holding a second multiplier is added onto the \freq
argument of each Pbind
(
~r = {75};
~mult = {1};
~sinfb1 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10),inf)*(Pkr(~r)*Pkr(~mult)),\dur,0.25,\amp,
0.3,\fb,Pwhite(0.1,1.4),\rel,0.1);
~sinfb2 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).reverse,inf)*(Pkr(~r)*Pkr(~mult)),\dur,
0.25,\amp,0.3,\fb,Pwhite(0.1,1.4),\rel,0.1);
~sinfb3 = Pbind(\instrument,\sinfb,\freq,Pseq((1..10).scramble,inf)*(Pkr(~r)*Pkr(~mult)),\dur,
0.25,\amp,0.3,\fb,Pwhite(0.1,2.0),\rel,0.1);
~sinfb1.play;~sinfb2.play;~sinfb3.play;
)
```

```
//increase the multiplcation over time using a .round on a Line.kr UGen. Listen to how the
scale is distorted as the multiplcation increases, eventually ending as a series of pulses
(
~mult = {Line.kr(1,60,60).round(1).poll(5)}
)


//Henon2DN - Chaos synths and moving between pitch and noise
(
//henon using a minor pentatonic scale at a high octave.
//The chaos Ugens will need some experimentations if you want subtle variance in sound
//For Henon I found that an a value of 1.3 and a b value of 0.3 renders a pitch in a pattern
pretty reliably
//note that the pitches aren't quite the same as 'concert pitch'
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,\degree,Pseq([0,2,4,6,7],inf),
\octave,8,\dur,Pbjorklund2(3,8)/4,\a,Pexprand(1.3,1.3),\b,Pexprand(0.3,0.3),\atk,0,\sus,
0,\rel,Pexprand(0.1,0.1),\amp,1);
~henon.play;
)
//increase the variation in the a and b arguments to add more noise to the mix
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,\degree,Pseq([0,2,4,6,7],inf),
\octave,8,\dur,Pbjorklund2(3,8)/4,\a,Pexprand(1.3,1.31),\b,Pexprand(0.3,0.31),\atk,0,\sus,
0,\rel,Pexprand(0.1,0.1),\amp,1);
)
//notice that this gets very noisy VERY fast.
//adding a little more possiblity to the Pexprands in a and b turns it into pure noise very
very fast, while still retaining a little of its pitched character
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,\degree,Pseq([0,2,4,6,7],inf),
\octave,8,\dur,Pbjorklund2(3,8)/4,\a,Pexprand(1.3,1.35),\b,Pexprand(0.3,0.35),\atk,0,\sus,
0,\rel,Pexprand(0.1,0.1),\amp,1);
)
//even more and noises become cut off and non-sounding.
//the cut off sounds would sound as DC bias, but the SynthDef \henon has a LeakDC on its output
to prevent this as it can damage sound systems and is generally quite an unpleasant thing to
deal with.
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,\degree,Pseq([0,2,4,6,7],inf),
\octave,8,\dur,Pbjorklund2(3,8)/4,\a,Pexprand(1.3,1.45),\b,Pexprand(0.3,0.55),\atk,0,\sus,
0,\rel,Pexprand(0.1,0.1),\amp,1);
)
//at this point decreasing the \dur and \rel value turns it into rhythmic percussion
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,\degree,Pseq([0,2,4,6,7],inf),
\octave,8,\dur,0.25,\a,Pexprand(1.3,1.45),\b,Pexprand(0.3,0.55),\atk,0,\sus,
0,\rel,Pexprand(0.01,0.1),\amp,1);
)
//more extreme possible values - \dur varied, octaves doubled up, more variation in a and b
values, more octaves
(
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,\degree,Pseq([0,2,4,6,7],inf),
\octave,[8,12,9,10],\dur,Pwrand([0.25,Pbjorklund2(Pwhite(3,5),8,1)/4,Pseq([0.125],4)],
[7,4,1].normalizeSum,inf),\a,Pexprand(1.2,1.55),\b,Pexprand(0.21,0.55),\atk,0,\sus,
0,\rel,Pexprand(0.01,0.6),\amp,1);
)
//against a kick drum it takes on a really strange character
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play;
)


//sound of different types of interpolation
//the default in my SynthDefs.scd file is currently to use none:
(
SynthDef(\henon,
{arg
freq=440,mfreq=440,a=1.3,b=0.3,x0=0.30501993062401,y0=0.20938865431933,atk=0.01,sus=1,rel=1,ts=
1,out=0,pan=0,amp=0.3;
var sig,env;
sig = Henon2DN.ar(freq,freq+mfreq,a,b,x0,y0,amp);
env = EnvGen.ar(Env.linen(atk,sus,rel),1,1,0,ts,2);
sig = LeakDC.ar(sig);
sig = sig*env;
Out.ar(out,Pan2.ar(sig,pan));
}).add;
);
//the example earlier, with no interpolation (default)
(
p.clock.tempo = 2.2;
~henon = Pbind(\instrument,\henon,\scale,Scale.minorPentatonic,\degree,Pseq([0,2,4,6,7],inf),
\octave,[8,12,9,10],\dur,Pwrand([0.25,Pbjorklund2(Pwhite(3,5),8,1)/4,Pseq([0.125],4)],
[7,4,1].normalizeSum,inf),\a,Pexprand(1.2,1.55),\b,Pexprand(0.21,0.55),\atk,0,\sus,
0,\rel,Pexprand(0.01,0.6),\amp,1);
~henon.play;
)
//now with Linear interpolation
(
SynthDef(\henon,
{arg
freq=440,mfreq=440,a=1.3,b=0.3,x0=0.30501993062401,y0=0.20938865431933,atk=0.01,sus=1,rel=1,ts=
1,out=0,pan=0,amp=0.3;
var sig,env;
sig = Henon2DL.ar(freq,freq+mfreq,a,b,x0,y0,amp);
env = EnvGen.ar(Env.linen(atk,sus,rel),1,1,0,ts,2);
sig = LeakDC.ar(sig);
sig = sig*env;
Out.ar(out,Pan2.ar(sig,pan));
}).add;
);
//now with Cubic interpolation
(
SynthDef(\henon,
{arg
freq=440,mfreq=440,a=1.3,b=0.3,x0=0.30501993062401,y0=0.20938865431933,atk=0.01,sus=1,rel=1,ts=
1,out=0,pan=0,amp=0.3;
var sig,env;
sig = Henon2DC.ar(freq,freq+mfreq,a,b,x0,y0,amp);
env = EnvGen.ar(Env.linen(atk,sus,rel),1,1,0,ts,2);
sig = LeakDC.ar(sig);
sig = sig*env;
Out.ar(out,Pan2.ar(sig,pan));
}).add;
);
```

4.7 - MIDI.scd

```
//Find your MIDI device by running this and checking the available devices in the post window
MIDIClient.init;

//edit the MIDI-enabled setup file to include your own MIDI device

//Once you have done this, load the MIDI-enabled setup file
("../../Setup/Setup_MIDI.scd").loadRelative;

//create a scale for the MIDI pattern to use (note: Scales used with MIDI must conform to 12-
tone chromatic format)
d[\scale] = Scale.minor

//Create a very basic MIDI/Kick Drum setup to check if the server latency is correct:
//If the MIDI note and the kick drum are playing at exactly the same time, the .latency method
on the MIDI set is correct, if not, it will need tweaking. Note that this will need tweaking
every time you set a different server latency or a different latency on your sound card.
(
(
~midiPattern = Pbind(
    \type, \midi,
    \midicmd, \noteOn,
        \midiout, d[\m],
    \chan, 0,
        \scale,d[\scale],
        \degree, 0,
        \octave, 3,
        \dur, 1,
        \legato, 0.01
)
);
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1,\rate,3);
~k.play;
)

//Example 1 - MIDI Basics:
//create a basic MIDI pattern
(
~midiPattern = Pbind(
        \type, \midi,
        \midicmd, \noteOn,
        \midiout, d[\m],
        \chan, 0,
        \scale,d[\scale],
        \degree, 0,
        \octave, 3,
        \dur, 1,
        \legato, 0.1
)
)
//elaborate on the pattern a little, changing the degree, octave, duration and legato
(
~midiPattern = Pbind(
        \type, \midi,
        \midicmd, \noteOn,
        \midiout, d[\m],
        \chan, 0,
        \scale,d[\scale],
        \degree, Pseq([0,0,3,4],inf),
        \octave, Pwrand([3,2],[10,1].normalizeSum,inf),
        \dur, Pbjorklund2(3,8)/4,
        \legato, Pexprand(0.1,0.99)
```

```
)
)
//add some percussion
(
~tom = Pbind(\instrument,\bplay,\buf,d["t"][0],\dur,Pbjorklund2(Pwhite(3,12),16)/4,\amp,
1,\rate,Pexprand(1.5,1.6),\pan,Pwhite(-0.8,0.8));
~tom.play;
)
(
~k = Pbind(\instrument,\bplay,\buf,d["k"][2],\dur,Pwrand([1,Pbjorklund2(5,8,1)/4],
[0.8,0.2],inf),\amp,1);
~k.play
)
//expand the MIDI pattern further:
(
~midiPattern = Pbind(
        \type, \midi,
        \midicmd, \noteOn,
        \midiout, d[\m],
        \chan, 0,
        \scale,d[\scale],
        \degree, Pwhite(-3,5),
        \octave, Pwrand([3,2],[10,1].normalizeSum,inf),
        \dur, Pbjorklund2(Pwhite(10,15),16)/4,
        \legato, Pexprand(0.1,0.99)
)
)

//and so on...

//Example 2 - Microtones:
//create a MIDI pattern with a long legato that creates one held note
(
~midiPattern = Pbind(
        \type, \midi,
        \midicmd, \noteOn,
        \midiout, d[\m],
        \chan, 0,
        \scale,d[\scale],
        \degree, 0,
        \octave, 3,
        \dur, 4,
        \legato, 1
))
//create a bend Pbind (within snippet bend)
(
~midiBend = Pbind(
        \type,\midi,
        \midicmd,\bend,
        \midiout,d[\m],
        \chan,0,
        \dur,1,
        \val,Pwhite(8192,8192)
)
)
//change the bend \val key to get a pitch bend
(
~midiBend = Pbind(
        \type,\midi,
        \midicmd,\bend,
        \midiout,d[\m],
        \chan,0,
        \dur,1,
        \val,Pwhite(0,16384)
)
)
//make the \dur of the bend pattern work independently of the ~midiPattern to get a microtonal
pattern within the existing MIDI pattern
(
~midiBend = Pbind(
        \type,\midi,
        \midicmd,\bend,
        \midiout,d[\m],
        \chan,0,
        \dur,Pwhite(0.25,0.75).round(0.25),
        \val,Pwhite(0,16384)
)
)
//change the available notes on the original MIDI pattern to get a greater variation of
available tones
(
~midiPattern = Pbind(
        \type, \midi,
        \midicmd, \noteOn,
        \midiout, d[\m],
        \chan, 0,
        \scale,Scale.chromatic,
        \degree, Pwhite(0,2),
        \octave, 3,
        \dur, 4,
        \legato, 1
))
//and so on...
```

5.1 - Drones - Examples.scd

```
//load the setup file
("../../Setup/Setup.scd").loadRelative

//DFM1

/*
A standard DFM1 drone I use an awful lot.
The filter self-oscillates at a 'res' value of >1, so here I have used a SinOsc moving from
0.9-1.1, so the self-oscillated distortion fades in and out.
Here I am using the harmonic series to organise pitch. with the frequency of the filter being
double that of the SinOsc.
!!!!NOTE!!!!! - In my installation of SuperCollider, DFM1 is buggy and NodeProxies it
contains need to be evaluated twice slowly otherwise they will cut all sound from the server
when played. I don't know why this is (or whether it is a version/platform/OS specific
issue), but if the experience is any different for you please raise an issue on GitHub or
otherwise let me know. This only happens once per NodeProxy, once it is initialised and
playing it can be re-evaluated and changed with no effect on the sound in the rest of the
server
*/
//set the fundamental frequency
~r = {80}
//evaluate this twice with a couple of seconds of gap in between
//the stereo sine wave creates a 'beating' in stereo. For more information see https://
en.wikipedia.org/wiki/Beat_(acoustics)
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],0,0.1),~r*2,SinOsc.kr(0.05).range(0.9,1.1),
1,0,0.0003,0.5)};
//play
~dfm1.play;
//changing the resonance changes the character of the self-oscillation, detuning it and
distorting it
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],0,0.1),~r*2,SinOsc.kr(0.05).range(0.9,1.6),
1,0,0.0003,0.5)};
//The higher the resonance value gets, the more distortion
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],0,0.1),~r*2,SinOsc.kr(0.05).range(0.9,5.6),
1,0,0.0003,0.5)};
//extreme resonance values get LOUD, but don't really change sonically past around the 10
mark
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],0,0.1),~r*2,SinOsc.kr(1).range(10,400),
1,0,0.0003,0.5)};

//DFM1 multiple drones
//Using the harmonic series technique, a number of drones at various multiplications layered
together
//Note - the modulation of the resonance is a slightly different speed for each, to create an
overall variation and non-repetition in sound
//set fundamental frequency
~r = {54};
(
//evaluate this twice with a couple of seconds of gap in between
//the argument 'mult' is used for speed - to copy and paste the entire NodeProxy and set
multiplications quickly during performance
~dfm1 = {arg mult = 1; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,SinOsc.kr(0.05).range(0.9,1.1),1,0,0.0003,0.5)};
~dfm2 = {arg mult = 2; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,SinOsc.kr(0.06).range(0.9,1.1),1,0,0.0003,0.5)};
~dfm3 = {arg mult = 3; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,SinOsc.kr(0.056).range(0.9,1.1),1,0,0.0003,0.5)};
~dfm4 = {arg mult = 4; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,SinOsc.kr(0.07).range(0.9,1.1),1,0,0.0003,0.5)};
)
//now play all
~dfm1.play;~dfm2.play;~dfm3.play;~dfm4.play;
//changing modulation from a SinOsc to an LFNoise, increasing modulation scope in lower
multiples
(
//evaluate this twice with a couple of seconds of gap in between
//the argument 'mult' is used for speed - to copy and paste the entire NodeProxy and set
multiplications quickly during performance
//this sounds like distorted guitars and is VERY rich.
~dfm1 = {arg mult = 1; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,LFNoise1.kr(0.05).range(0.9,4.5),1,0,0.0003,0.5)};
~dfm2 = {arg mult = 2; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,LFNoise1.kr(0.06).range(0.9,2.3),1,0,0.0003,0.5)};
~dfm3 = {arg mult = 3; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,LFNoise1.kr(0.056).range(0.9,1.9),1,0,0.0003,0.5)};
~dfm4 = {arg mult = 4; DFM1.ar(SinOsc.ar([~r,~r*1.01]*mult,0,0.1),
(~r*2)*mult,LFNoise1.kr(0.07).range(0.9,1.5),1,0,0.0003,0.5)};
)

//using DFM1 as a melody
//set harmonic frequency
~r = {60};
//start the first drone from the first example in this document
//evate this twice with a couple of seconds in between
~dfm1 = {DFM1.ar(SinOsc.ar([~r,~r*1.01],0,0.1),~r*2,SinOsc.kr(0.05).range(0.9,1.1),
1,0,0.0003,0.5)};
//play
~dfm1.play
//another drone, but one that contains a LFNoise1 used to give sweeps around the harmonic
series
//evaluate this twice with a couple of seconds in between
~dfmharm = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),LFNoise1.kr(0.1).range(100,1000).round(~r),SinOsc.kr(0.05).range(0.9,1.1),
1,0,0.0003,0.5)};
//play
~dfmharm.play;
//up the resonance
~dfmharm = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),LFNoise1.kr(0.1).range(100,1000).round(~r),SinOsc.kr(0.05).range(0.9,1.4),
1,0,0.0003,0.5)};
//up the speed of pitch change
~dfmharm = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),LFNoise1.kr(1.4).range(100,1000).round(~r),SinOsc.kr(0.05).range(0.9,1.4),
1,0,0.0003,0.5)};
//up the noise
~dfmharm = {DFM1.ar(SinOsc.ar([~r,~r*1.01],
0,0.1),LFNoise1.kr(1.4).range(100,1000).round(~r),SinOsc.kr(0.05).range(0.9,1.4),
1,0,0.1,0.5)};
```

5.2 - SuperCollider as a Modular Synth.scd

```
//load setup
("../../Setup/Setup.scd").loadRelative

//run this to smooth out transitions
p.fadeTime=5

//Using SuperCollider as a Modular synth
//snippets help with building these sets a LOT, as standard elements such as modulation signals
can be called upon very quickly
//NOTE: this will get !!! L O U D !!! - there's protection from StageLimiter of course, but be
aware.

//a sine wave
~sin = {SinOsc.ar([80,82],0,0.5)}
//a pulse wave
~pulse = {Pulse.ar([20,21],SinOsc.kr(0.1).range(0.01,1),0.5)}
//a new proxy multiplying sine and pulse waves
~sinpulse = {~sin.ar * ~pulse.ar}
~sinpulse.play
//feed this into a delay with its delay line modulated slightly
~delay = {CombC.ar(~sinpulse.ar,1,LFNoise1.kr(0.1).range(0.1,0.3),4)}
~delay.play
//increase the pulse speed and decrease the width, play it alongside the original
~pulse2 = {Pulse.ar([40,41],SinOsc.kr(0.1).range(0.001,0.1),0.5)}
~pulse2.play;
//actually no that would sound much better just in the delay, so ~pulse2 from playing and add
it into ~delay by using Mix.ar
(
~pulse2.stop;
~delay = {CombC.ar(Mix.ar([~sinpulse.ar,~pulse2.ar]),1,LFNoise1.kr(0.1).range(0.1,0.3),4)};
)
//now we have some drones, some heavily gated and filtered noise would be good.
(
~noise =
{RLPF.ar(WhiteNoise.ar(1),LFNoise1.kr(0.1).range(100,2000),SinOsc.kr(0.1).range(0.1,0.4),1)};
~noiseEnv = {EnvGen.ar(Env.perc(0.0001,0.1),Dust.kr(4))};
~totalNoise = {~noise.ar*~noiseEnv.ar};
~totalNoise.play;
)
//oh no. it is mono. i'm going to pan it over 2.
//In order to make a mono proxy stereo, I will have to .clear it and then evaluate a stereo
version, as the number of channels is set at initialisation time.
//luckily with Pan2 I will only have to re-evaluate the ~totalNoise proxy
~totalNoise.clear;
(
~totalNoise = {Pan2.ar(~noise.ar*~noiseEnv.ar,SinOsc.kr(0.1))};
~totalNoise.play;
)
//the filtering on the noise isn't extreme enough, change it!
~noise =
{RLPF.ar(WhiteNoise.ar(1),LFNoise1.kr(0.6).range(100,2000),SinOsc.kr(0.04).range(0.00001,0.2),
1)};
//the noise could also do with some delay, which would sound nice if it was fed back through a
pitch shifter:
//set up the delay, and play it
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar]),1,0.4,7,1)}
~noiseDelay.play;
//establish the pitch shifter
~pitchShift = {PitchShift.ar(~noiseDelay,0.2,TRand.kr(0.1,2,Dust.kr(0.5)))}
//play the pitch shifter, it will slow the delay speed by half
~pitchShift.play
//if we then put the results of ~pitchShift back into ~noiseDelay, then things get interesting.
```

```
//NB - this is bad practice and gets very loud before ending up in being DC bias, but i'm doing
it here to prove a point.
//If you have super high end audio equipment or just don't want any DC bias then skip this step
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar,~pitchShift.ar]),1,0.4,7,1)}
//in order to avoid this getting totally out of control, reduce the volume of ~pitchShift
inside of ~noiseDelay
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar,(~pitchShift.ar*0.11)]),1,0.4,7,1)}
//or modulate it to get varying amounts of feedback
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar,(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),
1,0.4,7,1)}
//modulating the delay time too will make things get a bit wild
~noiseDelay = {CombC.ar(Mix.ar([~totalNoise.ar,(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),
1,LFNoise1.kr(0.1).range(0.01,0.6),7,1)}
//~noiseDelay seems to be glitching a bit and throwing DC bias - add a LeakDC around it
~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),1,LFNoise1.kr(0.1).range(0.01,0.6),7,1))}
//let's cut the original pulse/sine waves over a few seconds
~delay.stop(20)
~sinpulse.stop(20)
//then put them inside of a DFM1 that can self-oscillate
//make sure you evaluate ~noiseDelayAdd twice before you .play it
~noiseDelayAdd = {DFM1.ar(Mix.ar([~delay.ar,~sinpulse.ar]),500,SinOsc.kr(0.1).range(0.5,2),
1,0,0.03)}
//if you've evaluated the above line twice, play it
~noiseDelayAdd.play
//a lot of these sounds are quite degraded, some harsh sounds would be nice, let's have some
chaos
//go to the help file for Henon2DC and copy-paste the second example but don't evaluate it
(you'll need sc3-plugins for this)
/*
(
{ Henon2DN.ar(
    2200, 8800,
    LFNoise2.kr(1, 0.2, 1.2),
    LFNoise2.kr(1, 0.15, 0.15)
) * 0.2 }.play(s);
)
*/
//turn it into a node proxy and remove the .play(s) from the end
(
~henon = { Henon2DN.ar(
    2200, 8800,
    LFNoise2.kr(1, 0.2, 1.2),
    LFNoise2.kr(1, 0.15, 0.15)
) * 0.2 };
)
//make an envelope that has a long sweeping modulation on the amount of envelopes triggered
~chaosEnv = {EnvGen.ar(Env.perc(0,0.02),Dust.kr(SinOsc.kr(0.01).range(1,10)))}
//and combine in stereo
~chaos = {Pan2.ar(~henon*~chaosEnv)}
~chaos.play
//it is SUPER quiet, up the volume on ~henon
(
~henon = { Henon2DN.ar(
    2200, 8800,
    LFNoise2.kr(1, 0.2, 1.2),
    LFNoise2.kr(1, 0.15, 0.15)
) * 3.5 };
)
//add some reverb which will work in parallel
//if you want to change the parameters of any effect without re-evaluating it - set up that
value as another NodeProxy
~room = {30};
~time = {3};
```

```
~verb = {GVerb.ar(~chaosEnv,~room,~time)}
~verb.play
//increase the reverb time
~time = {40};
//this needs some melody - add two melodies in stereo, slightly out of phase:
~saws =
{LFSaw.ar([LFSaw.kr(0.1).range(100,1000).round(50),LFSaw.kr(0.11).range(100,1000).round(50)],
0,0.3)}
~saws.play
//too harsh, needs filtering
~saws =
{RLPFD.ar(LFSaw.ar([LFSaw.kr(0.1).range(100,1000).round(50),LFSaw.kr(0.101).range(100,1000).rou
nd(50)],0,0.8),1000,0.8,0.6,10)};
//another delay would be nice
~sawDelay = {CombC.ar(~saws.ar,1,0.5,10)};
~sawDelay.play;
//some heavy decimation on the delay
~sawDelay = {Decimator.ar(CombC.ar(~saws.ar,1,0.5,10),2200,10)};
//further bit reduction
~sawDelay = {Decimator.ar(CombC.ar(~saws.ar,1,0.5,10),2200,5)};
//even further
~sawDelay = {Decimator.ar(CombC.ar(~saws.ar,1,0.5,10),2020,3)};
//plugging the ~sawDelay into the original for more noise
~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),1,LFNoise1.kr(0.1).range(0.01,0.6),7,1))}
//plugging ChaosEnv into ~noiseDelay too
~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),1,LFNoise1.kr(0.1).range(0.01,0.6),7,1))};
//then plugging it also into a more intense ~noiseDelayAdd for more mad effects
~noiseDelayAdd =
{DFM1.ar(Mix.ar([~delay.ar,~sinpulse.ar,~noiseDelay]),LFNoise1.kr(100).range(100,10000),SinOsc.
kr(0.1).range(0.5,100),1,0,0.03)}
~noiseDelayAdd.play
//it doesn't appear to be playing, probablt because ~noiseDelay is SO loud. Multiply it by half
~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs)]),1,LFNoise1.kr(0.1).range(0.01,0.6),7,1)) * 0.3};
//then plug ~noiseDelayAdd into ~noiseDelay and roll off the multiplication for maximum damage
~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs),~noiseDelayAdd.ar]),
1,LFNoise1.kr(0.1).range(0.01,0.6),7,1))};
//increase the ridiculousness of the modulation of the delaytime
~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs),~noiseDelayAdd.ar]),1,LFNoise1.kr(1).range(0.001,4),
7,1))};
//put another delay on top of that?
~delay2 = {CombC.ar(~noiseDelay.ar,1,0.4,30)}
~delay2.play
//then plug that back into ~noiseDelay (which by now contains most things that are playing.
~noiseDelay = {LeakDC.ar(CombC.ar(Mix.ar([~chaosEnv.ar,~sawDelay.ar,~totalNoise.ar,
(~pitchShift.ar*LFNoise1.kr(0.01,0.2).abs),~noiseDelayAdd.ar,~delay2.ar]),
1,LFNoise1.kr(1).range(0.001,4),7,1))};
//also modulate ~delay2, really slowly
~delay2 = {LeakDC.ar(CombC.ar(~noiseDelay.ar,1,SinOsc.kr(0.01).range(0.0001,0.2),80))}
//things broke up for me here and I have no idea why, there's multiple things feeding back
through each other here.
//and you have noise music!
```

6.1 - FreqScope and Visuals - Example.scd

```
//load setup file
("../../Setup/Setup.scd").loadRelative;

//Example 1 - Static Frequencies
(
//two low sine waves at the same frequency showing a diagonal line
~sin1 = {SinOsc.ar([80,80],0,0.3)};
~sin1.play;
)
//two low sine waves at slightly different frequencies turning the line into a slowly turning
disc
~sin1 = {SinOsc.ar([80,80.1],0,0.3)};
(
//two sine waves at double the frequency - notice the change in shape - turning the line a
number of times on itself
~sin2 = {SinOsc.ar([80*2,80.01*2],0,0.3)};
~sin2.play;
)
(
//two sine waves at 10x the frequency - notice the change in shape - turning the line a whole
bunch more times on itself
~sin3 = {SinOsc.ar([80*10,80.01*10],0,0.3)};
~sin3.play;
)
//stop everything
~sin1.stop;~sin2.stop;~sin3.stop;
(
//changing the frequency difference in the lower sine waves, changing how the original circle
moves
~sin1 = {SinOsc.ar([80,80+LFNoise1.kr(0.1,4)],0,0.3)};
~sin1.play;
)
//replay the other sine waves and see how the entire shape moves faster
~sin2.play;~sin3.play;
//stop the highest sines
~sin3.stop;
(
//re-align the two low sine waves
~sin1 = {SinOsc.ar([80,80.01],0,0.3)};
~sin1.play;
)
(
//play a sine that doesn't align with the harmonic series, notice that the shape gets much less
clear
~sin4 = {SinOsc.ar([94.234,99.1315],0,0.3)};
~sin4.play;
)
//stop the non-aligning sines
~sin4.stop;
//stop the second sine
~sin2.stop;
//play some quiet width-modulated pulse waves at 2x the frequency of the low sine waves
//notice that the shape changes according to the width of the pulse and that the 'notches'
interact with each other across the stereo field
(
~pulse1 = {Pulse.ar([80*4,80.1*4],SinOsc.kr(0.05).abs,0.08)};
~pulse1.play;
)
//change the pulse to a saw wave at the same frequency
(
~pulse1.stop;
~saw1 = {Saw.ar([80*4,80.1*4],0.08)};
~saw1.play;
)
//note that the higher the volume, the greater the effect a sound has on the overall shape
~saw1 = {Saw.ar([80*4,80.1*4],0.08)};
//also the higher the frequency, the lesser the effect on the 'overall' shape and the greater
the effect on the 'detail' of the shape
~saw1 = {Saw.ar([80*100,80.1*100],0.1)};
//stop everything
~sin1.stop;~saw1.stop;

//Example 2 - Moving frequencies and non-standard waveforms
//make a (really) low sine wave/spinning disc again
(
~sin1 = {SinOsc.ar([50,50.01],0,0.4)};
~sin1.play;
)
//make a stereo sine wave that sweeps the harmonic series
(
~sin2 = {SinOsc.ar(Saw.kr(0.1).range(10,1000).round(50),0,0.4)!2};
~sin2.play;
)
//make those two sine waves sweep the harmonic series at phasing (sightly different) rates
(
~sin2 = {SinOsc.ar(Saw.kr([0.1,0.11]).range(10,1000).round(50),0,0.4)};
~sin2.play;
)
//turn off the original sine wave
~sin1.stop
//speed the sweeping and make it a sine wave
~sin2 = {SinOsc.ar(SinOsc.kr([0.5,0.56]).range(10,1000).round(50),0,0.4)};
//make two meandering SinOscFB Ugens around the lower end of the harmonic series and see how
they interact
(
~sinfb1 =
{SinOscFB.ar([LFNoise1.kr(0.1).range(50,100).round(25),LFNoise1.kr(0.1).range(50,100).round(25)
],SinOsc.kr(0.1).range(0.01,1),0.8)};
~sinfb1.play;
)
//stop the second sine waves
~sin2.stop
//make a big sub kick drum - notice the effect on the shape
(
~k = Pbind(\instrument,\bplay,\buf,d["sk"][0],\dur,4,\amp,1);
~k.play
)
//make a panned hi-hat
(
~h = Pbind(\instrument,\bplay,\buf,d["ch"][0],\dur,0.25,\amp,Pexprand(0.05,1),
\pan,Pwhite(-1,1.0));
~h.play;
)
//make the feedback in the sinfb much more pronounced
~sinfb1 =
{SinOscFB.ar([LFNoise1.kr(0.1).range(50,100).round(25),LFNoise1.kr(0.1).range(50,100).round(25)
],SinOsc.kr(0.1).range(0.01,3),0.8)};
```

6.2 – python_randomNumber.py

```
# import python modules

import random

import time

import OSC


# Connect to SuperCollider's internal port

c = OSC.OSCClient()

c.connect(('127.0.0.1', 57120))


# Repeatedly send random messages which will be turned into a Warp1 Ugen pointer in ProxySpace

while True:

oscmsg = OSC.OSCMessage()

oscmsg.setAddress("/warpPointer")

oscmsg.append(random.random())

c.send(oscmsg)

time.sleep(random.uniform(0.1, 1))

oscmsg = OSC.OSCMessage()

oscmsg.setAddress("/warpRate")

oscmsg.append(random.uniform(0.1,3))

c.send(oscmsg)

time.sleep(random.uniform(0.1,1))

oscmsg = OSC.OSCMessage()

oscmsg.setAddress("/warpWindow")

oscmsg.append(random.uniform(0.01,0.9))

c.send(oscmsg)

time.sleep(random.uniform(0.1,1))
```

6.2- OSC and Data Streams - Example.scd

```
//TODO: make an example using the /netInfo that controls a warp1 Ugen. Also make a Python OSC
responder so that you can change the speed of the random number generation on Python side.

//Live data stream examples

//An example using SuperCollider's internal messages, outside of ProxySpace
//set address (if you've already done this no need to do it again)
b = NetAddr.new("127.0.0.1",NetAddr.langPort);
//msg will receive the OSC message as an array, with index 0 being the address and index 1
onwards being the message.
//setting msg[1] as the frequency will give the first parameter of the OSC message as an
argument
//setting msg[2] as the pulse width would allow you to send the second message parameter as the
pulse width, and so on...
(
OSCdef(\dinger,
{
|msg|
{Pulse.ar(msg[1],rrand(0.01,0.5),0.3)!2 * EnvGen.ar(Env.perc,doneAction:2)}.play
}, '/ding')
)
//make a 900Hz ding
b.sendMsg("/ding",900);
//make a ding at a random pitch
b.sendMsg("/ding",rrand(100,2000))

//An example of using live data
//In this folder there is a Python script: python_randomNumber.py
//To run this you will need Python 2.x and pyOSC (https://pypi.python.org/pypi/pyOSC)
//Once you have got the sketch running, this example should work.
//check that messages are being sent to '/warpPointer', '/warpWindow' and '/warpRate'
OSCFunc.trace
//load setup if you haven't already
("../../../Setup/Setup.scd").loadRelative;
//check this value
NetAddr.langPort
//if it is 57120, continue, if not, close all instances of SuperCollider and start again.
b = NetAddr.new("127.0.0.1", 57120);
//create warp1
~warp1 = {arg pos = 0, winsize = 0.1, rate = 1; Warp1.ar(2,d["lfx"]
[1],pos,rate,winsize,-1,16,SinOsc.kr(0.01).range(0.0001,0.1),4,0.6)}
//use this OSCdef to use messages coming from Python to change the pointer position of ~warp1
(
OSCdef(\pointerResponder,
{
|msg|
~warp1.set(\pos,msg[1]);
},'/warpPointer')
)
//and the rate
(
OSCdef(\rateResponder,
{
|msg|
~warp1.set(\rate,msg[1]);
},'/warpRate')
)
//and the window size
(
OSCdef(\winSizeResponder,
{
|msg|
~warp1.set(\winsize,msg[1]);
},'/warpWindow')
)
```

6.3 - Using Datasets - Example.scd

```
//Using Datasets - example

//load the setup file if you haven't already
("../../Setup/Setup.scd").loadRelative

//This is an example using some environmental data that I logged in 2015 in a part of my
university building. It was logged to a CSV file which is included in this repo
//load the CSV file as a multi-dimensional array - Storing this within the dictonary I usualy
store sample in
//note the 'flop' method, which takes columns as rows and vice versa.
//Without the flop method, each line of the CSV file would be an array entry
//startRow is worth setting at 1, so that the header line is not parsed. I'm going to start a
few hundred lines in because the start of this dataset is skewed by the sensors taking a while
to 'warm up'
d[\data] = CSVFileReader.readInterpret(("Datasets/
ArmstrongFoyer_Data.csv").resolveRelative,true,true,startRow:1000).flop
//In the current data configuration each type of data is an index of the arrady d[\data]
d[\data][0] // An array of the date (rendered out as garbage because of the / character in each
field
d[\data][1] // An array of the time (rendered out as garbage because of the - character in each
field
d[\data][2] // An array of the temperature
d[\data][3] // An array of the %rh
d[\data][4] // An array of the light level
//if you want to get an idea of the form of a dataset, calling the plot method will draw a
graph using the GUI
d[\data][4].plot
//If you're going to use the data in a sonification it's worth scaling it to useful values.
//Use this function to convert data to frequencies.
//the linlin, linexp, explin and expexp scaling methods can be used depending on the dataset
available. As i'm going to be converting environmental data to frequency, i'll be using the
linexp method
(
d[\scaleData] = {
        arg data = d[\data][3], minimum = 100, maximum = 1000;
        data.linexp(data.minItem,data.maxItem,minimum,maximum);
}
)
//scale all of the data
(
d[\temp] = d[\scaleData].(d[\data][2],100,2000);
d[\humidity] = d[\scaleData].(d[\data][3],100,2000);
d[\lux] = d[\scaleData].(d[\data][4],100,2000);
)
//Put the data inside some Pbinds and set them away. They should all finish at the same time.
There is a LOT of data here, so it'll run for quite some time.
(
~temp = Pbind(\instrument,\sinfb,\freq,Pseq(d[\temp],inf),\dur,0.25,\amp,Pwhite(0.05,0.3),\fb,
0.1,\rel,0.3,\atk,1,\rel,3);
~sin = {SinOsc.kr(0.1).range(0.01,1.41)};
~temp.set(\fb,~sin);
~humidity = Pbind(\instrument,\ring1,\f,Pseq(d[\humidity],1),\dur,0.25,\d,0.6,\a,1,\pan,0,\amp,
0.5);
~lux = Pbind(\instrument,\ring1,\f,Pseq(d[\lux],1),\dur,0.25,\d,5,\a,Pexprand(1,30),\pan,
0,\amp,0.1);
~temp.play;
~humidity.play;
~lux.play;
)
//feel free to add some percussion and use ProxySpace as normal
~k = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,1,\amp,1);
~k.play
~t = Pbind(\instrument,\bplay,\buf,d["k"][0],\dur,Pbjorklund2(Pwhite(3,24),32)/4,\amp,
1,\rate,Pwhite(4,6.0));
```

```
~t.play


//another example, using the normalize method to scale data from 0 to 1, and use it to set a
Warp1 Ugen
//the host Ugen. We will manipulate it using the "pos" argument
~warp1 = {arg pos = 0, winsize = 0.1, rate = 1; Warp1.ar(2,d["lfx"][0],pos,
1,winsize,-1,16,0,4,1)};
(
//a thing I just found out recently after using SuperCollider for years is that local variable
scope is determined on a per-execution basis!
//This makes iterating during a Tdef way easier than I thought it was.
var i = 0;
Tdef(\dataIterator,
        {
                var data = d[\data][4].normalize;
                //a simple loop to iterate over the
                loop{
                        ~warp1.set(\pos,data[i]);
                        i = i + 1;
                        i = i%(data.size-1);
                        0.003.wait;
                        data[i].postln;
                }
        }
)
)
//play the Warp1
~warp1.play;
//Use the normalised data to control the position of the Warp1 Ugen based on the light levels.
Tdef(\dataIterator).play
```

The End.