1 Synopsis

AAA302

SECTION 1.1

Introduction

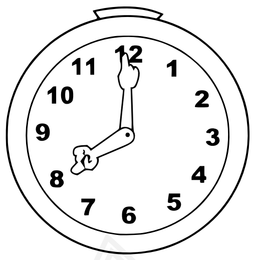
In this exercise we will use dataflow methods to create a piece of musical accompaniment. In doing so we construct all the synthesis and sequencing code required for the task. An exploration of real-time parameterisation will be undertaken. All SAE students on course 302 "advanced audio applications" should attempt this task since it exercises and reinforces many of the procedural audio techniques we have studied on the course.

The musical for is late 1980's style "Acid" since this offers a simple form requiring only basic instrumentation and sequencing, yet it demonstrates most of the essential features of an embedded musical construct that may be adapted to other styles. The project consists of the following items:

Instrument	Description
Timebase	Source of a global clock to drive music
Bar sequencer	Simple list based loop for constructing phrases
Mixer	Summation of audio signals to a stereo mix
Kick drum	A short low frequency burst
Hi Hat	A short inharmonic spectrum with noisy qualities
TB303	Sawtooth synthesiser with highly resonant filter
FM Bass	Rich, deep bass sound with complex spectrum
Pad sound	Dense chordal effects
Stereo delay	A simple effect to add space to the mix
Reverb	A basic Schroeder reverb to add ambiance

You should construct them in the order given for maximum experimental potential. Customise your own composition and modify the synthesis elements to create a unique piece of procedural music.

2 Timebase



A very simple timebase starts us off as seen in 2.1. The metro object is switched on when it receives a 1 at its left inlet (and off when a 0 is applied), and after that it sends out a steam of bang messages separated by the period given on its right inlet.

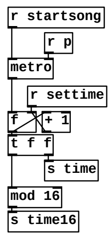


fig 2.1: Timebase

A counter is clocked by the metronome and it sends its value, which represents the current time through `s time`. Another copy, constrained modulo 16 is sent on `s time16`. We have arranged for some broadcast messages to control the subpatch. The destination `r startsong` can switch the timebase on and off, a period can be received via `r p` to change the tempo, and `r settime` allows us to reset the counter or jump to any new time in the song. The subpatch that contains this should be named `timebase` and used as follows.
In use, we might start by sending suitable messages to get the timebase running. 2.2 shows a broadcast message contained in a message box which does the following:

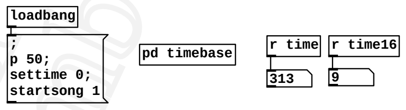


fig 2.2: Testing out the timebase

Destination	Value	Description
p	50	Set the metronome period to 50 milliseconds
settime	0	Make sure the global time is reset to zero
startsong	1	Begin the timebase by starting the metronome

You can see the subpatch containing the code from 2.1 in the middle, and two receivers connected to number boxes which demonstrate that everything is working. A loadbang object is used to fire this message when the patch is loaded, but you can also click the message by hand any time to reset the timebase.

SECTION 2.1 Exercises and extension

Notice we specified the song tempo in milliseconds. That is something commonly done in computer music, but it isn't the classical way to do things. Most people like to express tempo in BPM (beats per minute). What is the tempo in BPM for a 50ms period? Design a subpatch to convert from BPM to milliseconds to you can input new tempos in a more friendly fashion.

Solution

There are 1000 milliseconds in one second. And there are 60 seconds in a minute. Therefore a minute contains $60 \times 1000 = 60000$ms. If T_{bpm} is the tempo in BPM and T_{ms} is the tempo in milliseconds then $60000/T_{bpm} = T_{ms}$, and going the other way, milliseconds to BPM, $60000/T_{ms} = T_{bpm}$. So, tempo in BPM is $60000/50\text{ms} = 1200\text{BPM}$.

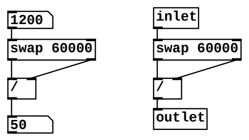


fig 2.3: Left: BPM to milliseconds. Right: As a subpatch.

The easiest message domain method of dividing a constant by a variable is to do as in 2.3 and use swap (which exchanges its argument with the value on its inlet and outputs them on separate outlets), preceding a / object. A way to incorporate it into the main patch is shown in 2.4.

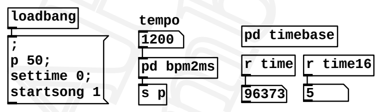


fig 2.4: Adding a human readable tempo control.

3 Kick drum

Now the fun begins. Essentially a kick drum is just a low frequency sine wave enveloped with a decay curve so its amplitude diminishes to zero. Let's develop a really nice way of doing this that is both efficient and accurate and see in the process some pitfalls of doing it wrong.

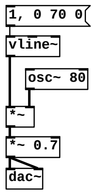


fig 3.1: A simple but inadequate kick drum.

3.1 shows a sine oscillator at 80Hz modulated by a short envelope. The envelope, produced by vline~ and its message, rises to 1.0 immediately then decays to 0.0 in 70ms.

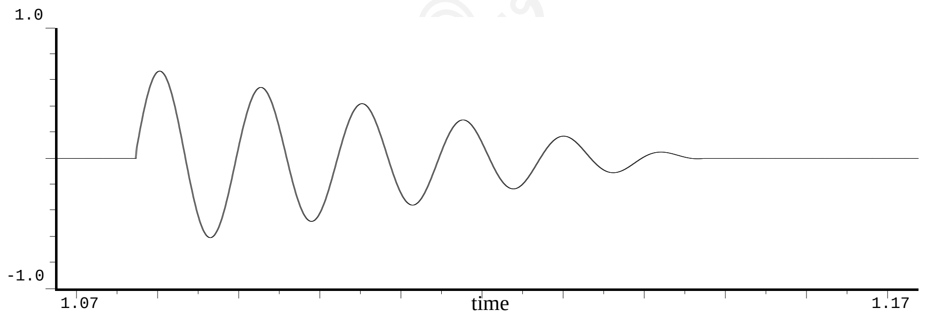


fig 3.2: Waveform of a kickdrum using a free running oscillator.

3.2 and 3.3 show the waves of two different kick drums taken from this same patch at different times. You can listen to a sequence of three of them next.

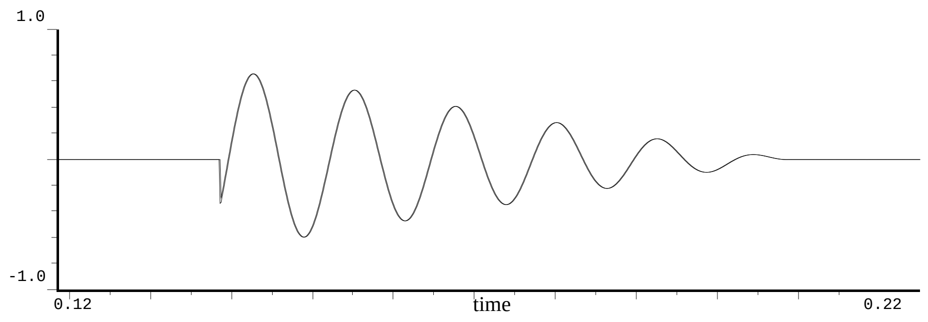


fig 3.3: Waveform of another kickdrum from the same patch using a free running oscillator.

Listen to three kickdrums created from a free running oscillator.
The problem is subtle and you may need to listen carefully. When using a free running oscillator we never know what phase it will be on when the envelope is triggered. That means that while the waveform starts nicely in 3.2 it shows a truncated start in 3.3. This causes the sound to change on different beats. Sometimes the sound has a hard click to it, sometimes it is soft and quiet.
To fix this we need a way of making sure the oscillator is synced, so it always starts in the same way. There is a clever trick to do this without even needing an oscillator. We can get a sinusoidal waveform $f(t) = A\cos(\omega t + \phi)$ just by taking the cosine of the line segment used for the amplitude envelope. See how this is done in 3.4.

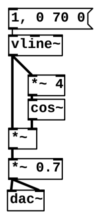


fig 3.4: A phase synchronised kick drum.

It doesn't matter that the line is decreasing in value because the cosine wave is time symmetrical (it sounds the same played backwards as forwards). Since $\phi = 1.0$ and remembering that Pure Data uses rotation normalised form so really $\phi = 2\pi$, we know the amplitude will start at 1.0 since $\cos(2\pi) = 1$. All we need to specify is ω to get the right frequency. This is actually more obvious than it may seem. The line segment covers a range of 1.0 in 70ms. If we multiply it by a number n there will be n cosine cycles in 70ms, so the frequency will be $n/0.07$ cycles per second. For $n = 5$ we get about 70Hz.

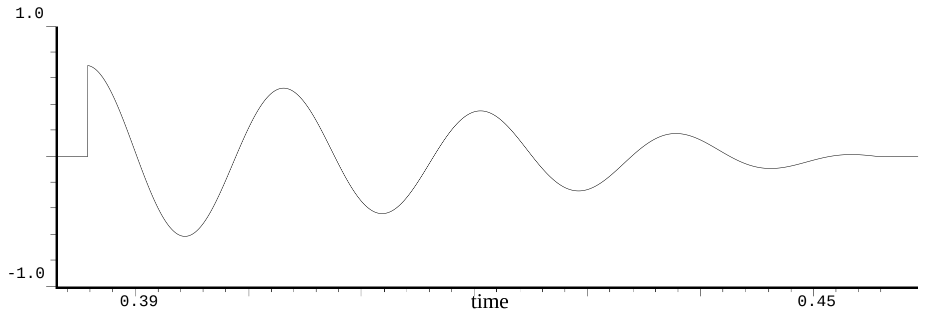


fig 3.5: Waveform of a phase synced kickdrum.

3.5 shows the waveform. They all look the same because every kick starts on exactly the same phase, with amplitude as 1.0. This gives a solid, meaty click to the sound, it always has a hard attack, perfect for techno.

Listen to some kickdrums created with a phase synced oscillator.
One more improvement can be made at very little cost. Giving the waveform a frequency sweep will fatten it up and turn the rather clean kick into something more of a thud. By sweeping the frequency we are performing FM, so adding sidebands to the signal. 3.6 shows how we can take a new function of the line, which is $f(x) = 2.4x^2$, giving a sweep from about $(2.4 \times 2.4)/0.07 = 5.76/0.07 =$ 82Hz down to zero in a square law curve.

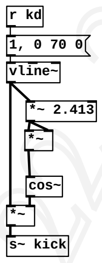


fig 3.6: A frequency swept phase synchronised kick drum.

The results can be seen in 3.7. Note there's no reason to assume we can pick an upper frequency that sounds good that also happens to be an integer, so the choice is an aesthetic one that gives a nice click and a good range. Of course we can also change the frequency by changing the duration, because this is a codependent parameter, so the perfect kick drum comes from experimenting with duration and sweep factor.
Listen to some kickdrums created with a swept phase synced oscillator. We have obtained a fairly punchy kick drum using only a cosine, a line, two multiplies

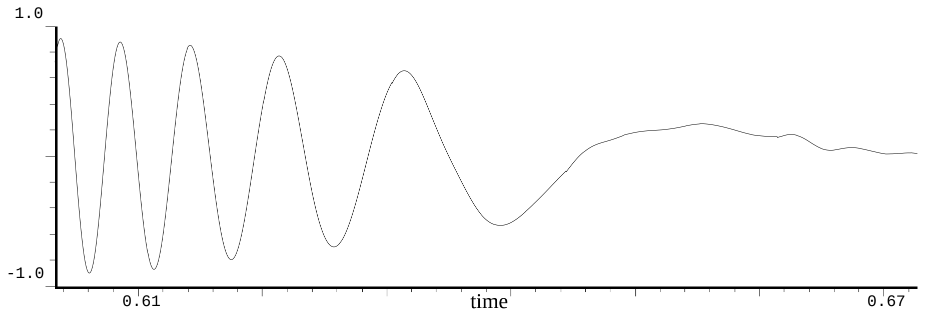


fig 3.7: Waveform of a frequency swept phase synced kickdrum.

and a constant. Not bad going!

SECTION 3.1

Exercises and extension

The kick shown in 3.7 is wired with two receivers, the first `r kd` just picks up a bang message to trigger it, while the second `s~ kick` is an audio send that we can route to a mixer later. Wrap it all into a subpatch `pd kickdrum` and hook up the kick to the timebase to produce a "four on the floor" pulse at 150BPM.

Solution

Refer to 3.8. We keep the timebase at the same tempo of 1200BPM, so that we have many more beats to a bar than we really need. This will come in handy later. To place the kick on every 4 beat at 150BPM we divide the timebase using `mod` and use `select` to get a bang on every 4 beat. A factor is added to the control so the tempo we read is correct for the desired timebase period.

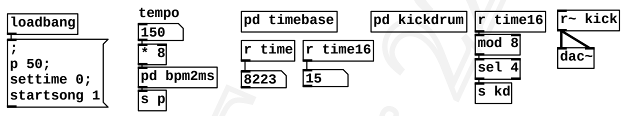
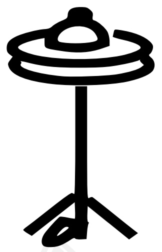


fig 3.8: A beat at 150BPM using timebase division.

4 Hihat



The hihat consists of two small metal cymbals in close proximity that can rattle when hit. Many descriptions for synthesising a hihat rely on noise. In fact using pure white noise isn't a very efficient or good way to make a hihat since the spectrum is sparser than one might think. A better approach is to use FM to form a dense spectrum.

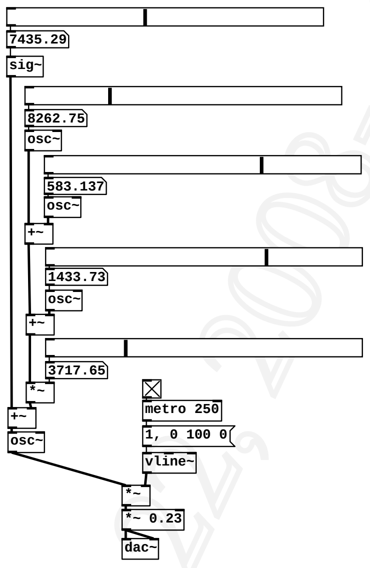


fig 4.1: Making a hihit spectrum with complex FM.

4.1 shows a complex FM patch with three oscillators in the modulator. They are summed and added to a base carrier frequency and then this signal is used to modulate the primary oscillator. The patch follows

$$f(t) = 0.23(1 - \frac{t}{10})(\cos(7435t + (\cos(8263t) + \cos(583t) + \cos(1434t))))$$

Those numbers didn't come out of nowhere. It's crazy to pick values by calculation, so to get the sound right a bunch of faders are added so that each

parameter can be tweaked. Number boxes are used so that we can see the final values and then plug them in as fixed arguments when finished. During tweaking a temporary metronome is added to keep retriggering the hihat and apply an overall amplitude envelope with a 100ms decay.
Listen to the sound change as I tweak the parameters, starting with just the carrier base, then adjusting the modulation index, then fixing the three components of the modulator.

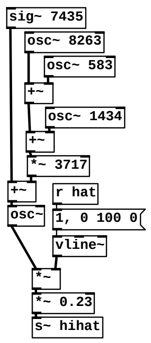


fig 4.2: Fixing the values for the hihat subpatch.

In 4.2 you can see how the values discovered while tweaking are set as arguments to the oscillators and FM index factor. The whole patch can be collapsed and encapsulated in a subpatch, `pd hihat`, with a receiver `r hat` and an audio send `s~ hihat`. A wave view and spectrum are shown in 4.3.

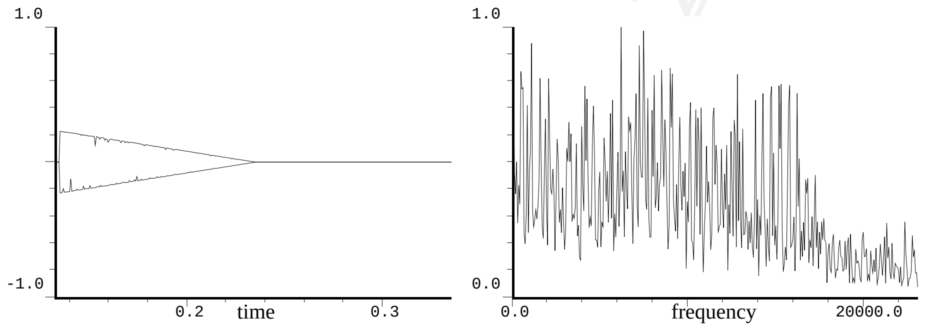


fig 4.3: Waveform and spectrum of complex FM hihat.

Next we need to patch the hihat in so it plays on the off-beat forming a classic techno alternation. In 4.4 a `select` is used directly on the `time16` time stream to pick out beats 0 and 8 (that's 1 and 9 if you're counting in the normal musical way).

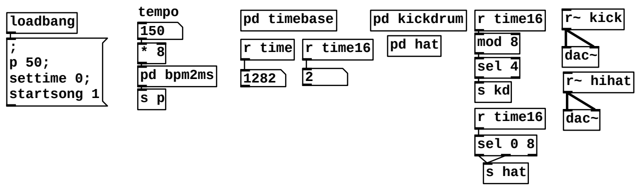


fig 4.4: Patching the hihat in to make a beat.

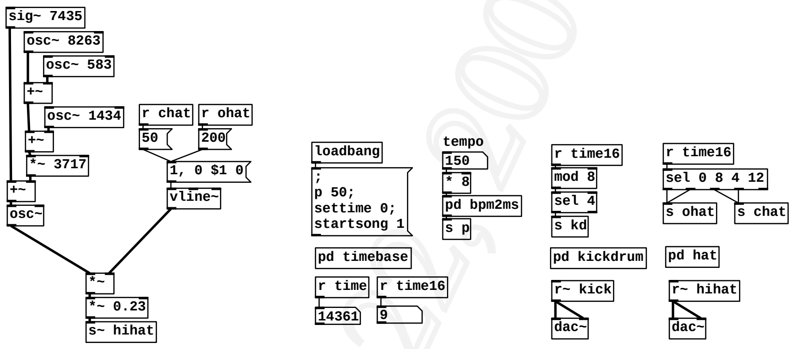
SECTION 4.1

Exercises and extension

The hihat pedal adjusts the separation of the two cymbals. Further apart they ring for longer and rattle less. A real drummer can vary the separation with nuance, but usually a drum machine hihats have two settings, open and closed. Implement a change in the envelope to provide longer a "open" sound. Test this by patching in a trigger pattern.

Solution

You could substitute the decay time into the line generator message as seen in 4.5(left). For the short, closed hat use 50ms and for the open version try 200ms



(a) Substitute decay (b) Open or closed on alternate beats

fig 4.5: A hihat with open and closed options.

A slight rearrangement of the main patch is seen in 4.5 (right). Now there's four bangs taken from the `select` to trigger both closed and open hihats.

5
Mixer

Σ

Now that we have more than one instrument playing it's time to construct a mixer. A simple device with faders, pan position controls, and aux sends for effects should suffice. Take a look at 5.1 which shows a single channel strip we can duplicate. This architecture should be familiar enough. An audio input signal arrives through the first inlet. The next inlet takes a `mute` control, either 1 to mute the channel, or 0 to unmute it. To preserve the correct sense of the mute function the control must be inverted to become 1 when the audio is on and 0 when it is off. Mute is then combined with an audio level control using a "latching multiply". It feeds the hot inlet of a `*` while the current level is held on the cold inlet. The level is also scaled using a decibel to RMS curve, which gives the fader a nicer range. Because the level is scaled by 100dB the actual fader is normalised 0.0 to 1.0. Converting the level control to a signal and then using `lop~` to low pass it makes the fades and mute operations smooth and free from clicks.

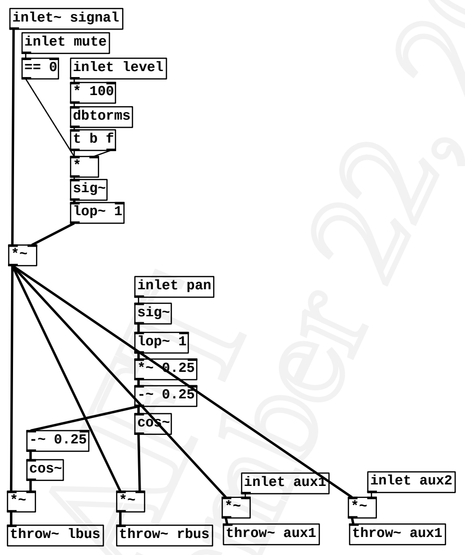


fig 5.1: A basic channel strip with mute, level, pan and aux.

After the input audio has been scaled to level it passes to a cosine panner and auxiliary bus sends (so they are post-fade sends). Two main bus destinations are fed from multipliers modulated by cosine functions 90° out of phase, so we get an approximately equal power pan operation.

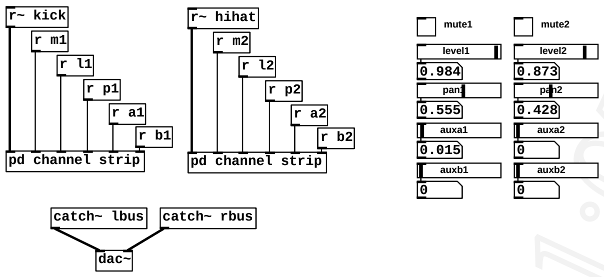


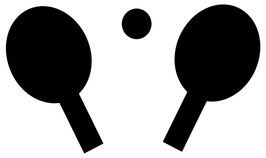
fig 5.2: Using the mixer with some controls.

5.2 shows how we use the channel strips. Two are used, by copying the first and adding suitable receiver boxes and controls, one for the kick drum and one for the hihat. Later we will add more channels as we need them. All the controls for level, pan, and aux sends are normalised 0.0 to 1.0 The two main bus destinations `lbus` and `rbus` are now connected directly to the dac~. So far we haven't used the axillary sends, but we will add a master bus section when we do that next.

SECTION 5.1 Exercise and extension

Add four more channels to the mixer to give it 8 channels. Set the canvas area and make it a "graph on parent" subpatch so the controls can be set from the main patch without having to open it each time.

6 Ping-pong delay



At this point some FX would be good to test out the aux sends of the mixer. 6.1 shows a simple way to make a tempo controllable ping pong delay. The feedback between the delay elements crosses over, so the left delay feeds back to the right delay input and the right delay feeds back to the left delay inlet. In other words the delay lines are circular with two taps diametrically opposite. Two delays are defined `a` and `b` with 10 seconds of time allocate, which is plenty! Although the delay read objects have default arguments their actual times are set by multiples of the timebase period. So when the tempo changes the delays change with it. I have fixed the feedback to 0.2 and the delay pattern to a triplet on 6 and 12 beats.

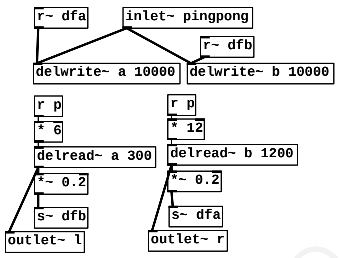


fig 6.1: A tempo control ping-pong delay.

Let's see how the pingpong delay effect fits into the bigger picture by wiring it in to an auxiliary send. As you can see in 6.2, now `aux1` controls the amount of audio signal sent to the pingpong.
Try adding a little bit of delay to the hihat to make the beat roll along. Listen to a snippet of the beat we have so far with a little ping pong effect.

SECTION 6.1 Exercise and extension

Provide controls for the delay pattern and feedback amounts. Limit the feedback so it is never equal to or greater than 1.

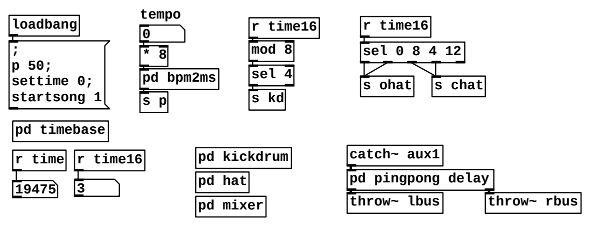


fig 6.2: Main patch showing the mixer subpatch and the pingpong delay effect.

Solution

A solution is shown in 6.3. This isn't the definitive way to do it, there are several solutions, but this one is flexible. We define three new controls. A normalised fader sends to `s ppfb` which is limited to stay below 0.9 by the `min` object. Two other numbers are sent to `ppat1` and `ppat2` so you can set the number of beats separately for the left and right sides.

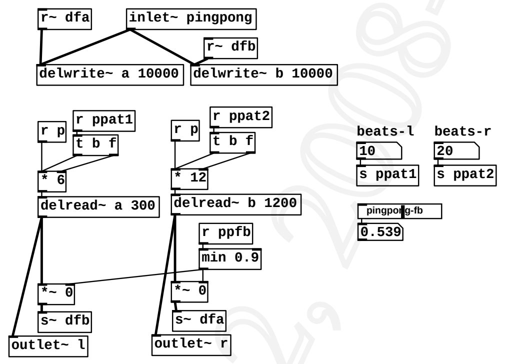
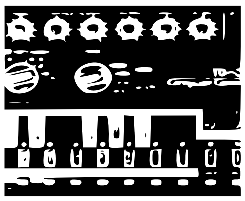


fig 6.3: Adding feedback and beat pattern controls to the pingpong delay effect.

7
TB303



Nothing very musical is happening yet, so it's time to build out first synthesiser. The TB303 is a unique sound, quite unlike any other synth. The way that we will construct this example involves an interesting "cheat" that greatly accentuates the sound we are after. In essence it is just a sawtooth or square wave synth with a resonant filter. But the filter is non linear and able to self oscillate, so the kind of sounds the 303 became famous for were not within its intended range as a bass accompaniment, but as a highly overdriven and resonant "acid" sound. The 303 sound is the basis for an entire genre of music spanning the between the late 1980's and the mid 1990's.
First create an audio receiver in the mixer for `tb1`. This will be where we send the synth sound we are working on. Set the beat running and create the following subpatch as shown in 7.1.

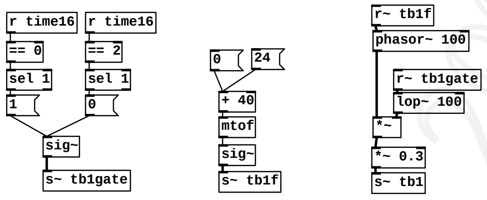


fig 7.1: The start of our TB303 synth, just a phasor and gate.

As a test pattern we create a gate signal that switches on when `time16` is 0 and off again when it's 2. This makes a single note at the start of the bar. To set the pitch of the phasor we use `mtof` with a base MIDI note of 40 and messages to manually shift it up and down by two octaves. When the gate is 1 the phasor is passed though, attenuated to a third, and sent to `tb1`.
Two problems arise from using the phasor directly as a sawtooth. Firstly it lies above zero, so we get a DC offset in the signal. Secondly, it isn't band limited, so very high notes will generate aliasing and sound muddy. Looking to 7.2 we see a solution.

Instead of using the phasor directly we use it to lookup a band limited sawtooth wave stored in a small table `saw1` of 2051 points (because it must be a power of two plus three). Multiplying the phasor by the table size (three

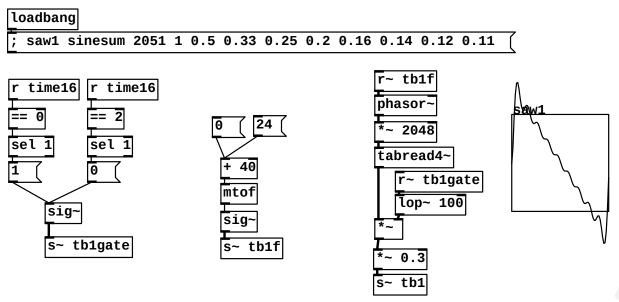


fig 7.2: Replacing the oscillator with a band limited table lookup.

less than the full table for correct interpolation by `tabread4~`) gives us a much smoother and more mellow saw sound. 7.3 shows the wave and spectrum of this implementation.

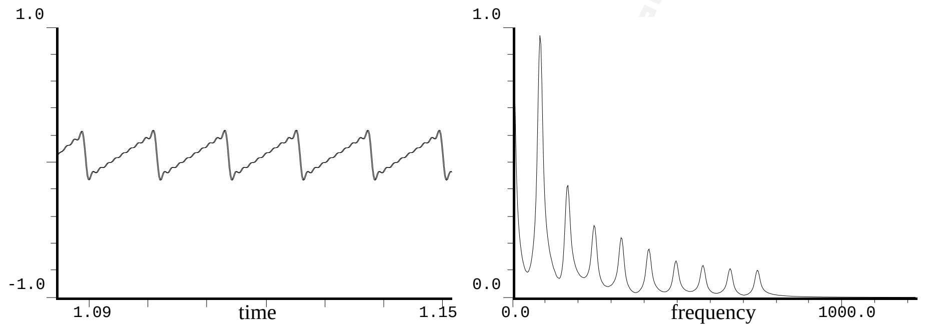


fig 7.3: Wave and spectrum of band limited 303 saw.

As it stands the sound lacks the biting resonance of the 303. Normally, we would now add a resonant filter like the `vcf~` or `moog~`, but I would like to demonstrate a really nice trick here. Since the filter will be driven into self oscillation, why bother with a filter? Why not just add a new sine wave at the frequency of the filter cutoff? This trick of letting a new sine ride on top of the saw actually produces an amazing 303 filter substitute with much less cost than a real filter. Let's see how to do this now.

In 7.4 the output from the phasor is split into three streams. The output of the band limited saw passes down on the right and is added to the other two flows at the bottom. The other two combine to make a sine wave that rides on top of the saw. In the centre you can see a `cos~` preceded by a `wrap~`. When the phasor is multiplied by the resonant frequency control `tbrf1` then wrapped it

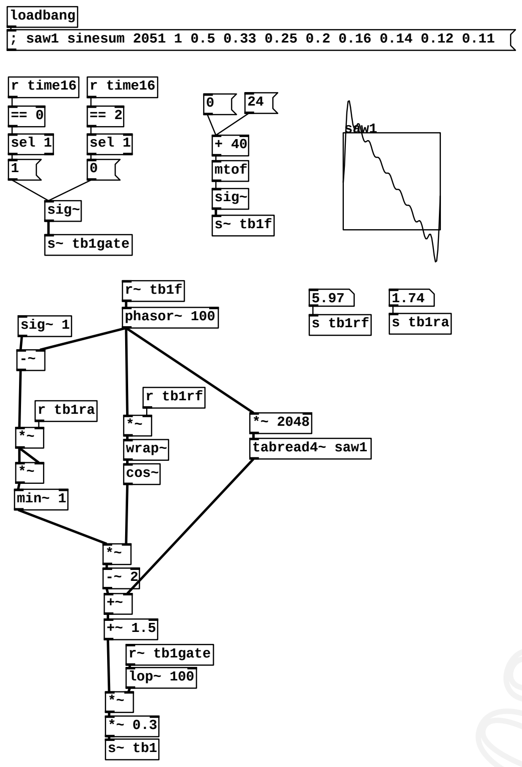


fig 7.4: A 303 with simulated non-linear filter system.

produces a new phasor at a higher frequency. Once this is turned to a sinusoidal wave it is enveloped on each wave cycle by a square law curve in phase with the main phasor, the height of which is determined by `tb1ra`, the resonance amount control. Typically the frequency control will range from 0 to 10 and the amount control from 0 to 1. I have added a min~ object that makes an interesting modification so that you can extend the resonance amount above 1. Values as high as 3 or 4 will produce a "saturated" filter effect. 7.3 shows the wave and spectrum. Notice the accentuated harmonics around $10f$, just like a real resonant filter.
Listen to the band limited saw wave with and without the fake filter applied.

Before the 303 is complete we must add some more essential ingredients The first is the ability to slide between notes, called *glissando*. In fact this is part of the 303's unique sound, not because it isn't found on other instruments but because of the way it does it.
A secret known only to experienced dance music producers is that it's very difficult to correctly create a 303 glissando using MIDI because it uses a pre-

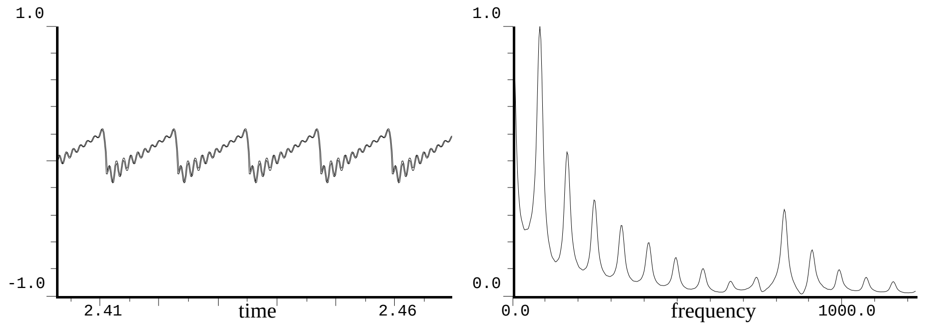


fig 7.5: Wave and spectrum of fake 303 filter.

emptive form in which the slide starts before the note, thus it arrives at the target pitch exactly at the desired start position. We won't attempt to create that here since it involves some extra logic that complicates the sequencer. Instead we use a `line~` object as shown in 7.6 to interpolate the pitch signal between the last and next note.

The control logic to the left of 7.6 consolidates the operation into a list driven note method. Instead of having separate messages to turn the gate on and off we add a new `delay` object to create a gate of given duration, which is multiple of the timebase period. Notes are given in the form `MIDI note`, `duration`, `slide`. Slide is either 1 0r 0, to switch the slide on or off. This particular ordering is necessary so that the slide duration can be set up properly before the gate is triggered.

Listen to the instrument with notes at two pitches with and without slide.

Finally a proper 303 sound requires a ripping overdrive effect. The function of $\tan^{-1}$, often written `tanh(x)`, and seen in 7.7 is useful here.

When the signal is small in amplitude, below about a third (-0.33 to $+0.33$), then the function behaves like an almost perfectly linear transfer. Approaching 1.0 the signal becomes greatly distorted. Since this is a form of waveshaping extra harmonics are added that make the sound much richer.

Adding an amplitude control to the gate and placing the resonance amount and frequency controls into a single list we arrive at 7.8. An important addition is `moses` which strips out any notes that have a zero MIDI pitch. This is because we will use this value in our sequencer to mean don't play anything. If zero notes were allowed through they would create a very low note that sounds as a click. Notice also how the gate has been reconfigured to trigger from the duration parameter which similarly avoids tiny clicks occurring for zero notes. The graph for the saw wave array has also been hidden by unticking its graph-on-parent box. The test sequence has been removed so that the whole instrument can be encapsulated neatly into a subpatch. Distortion using $\tan^{-1}$ is provided by an expression. The multiplier that precedes it is used to set the drive level. Note

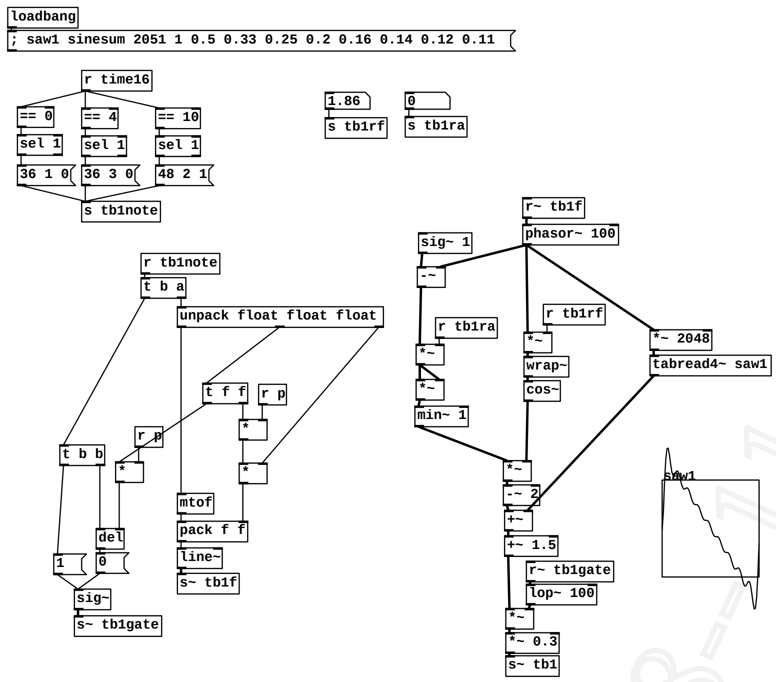


fig 7.6: 303 with note slide.

that the distortion level is set prior to the gate level so we can have independent control of clipping amount and overall level. The full parameter sequence is now;

Position	Parameter
1	MIDI note number
2	Duration in timebase periods
3	Slide, on or off
4	Filter frequency, in harmonics
5	Filter amount, 0 to 1
6	Overall amplitude level
7	Distortion amount

Finally, refer to 7.9 to see the top level patch with a test sequence.

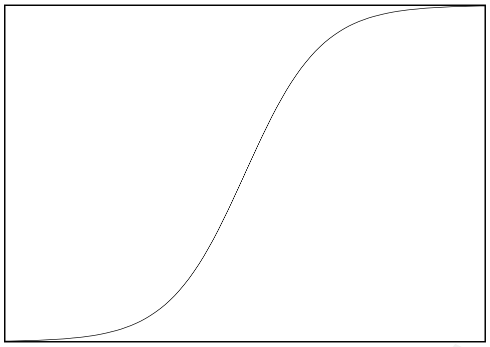


fig 7.7: The function $\tan^{-1}$ can be used as a nice distortion effect to waveshape the signal and add new harmonics.

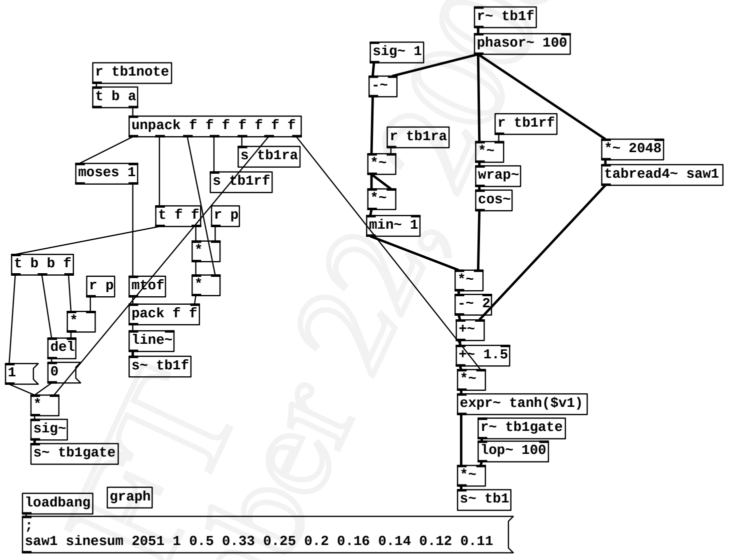


fig 7.8: Consolidating the TB303 controls into a per-note list.

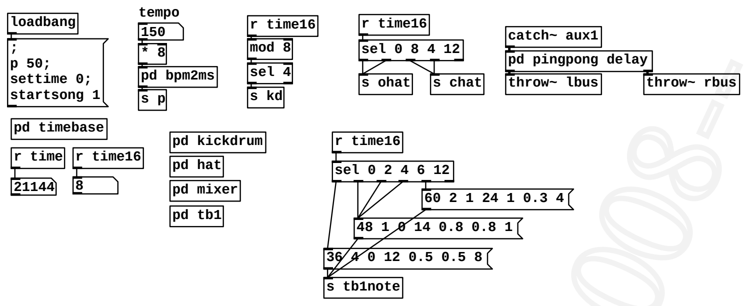


fig 7.9: The top level patch now shows the added TB303 with a little sequence to test all of its functions.

8
Bar sequencer

All the things needed to make interesting music are coming together, except for one important tool, a sequencer. At the moment we are just using hard-wired select objects to hang events off. Of course this has just been to test our instruments, but now it's time to get serious about constructing compositions so we need something a bit more flexible.

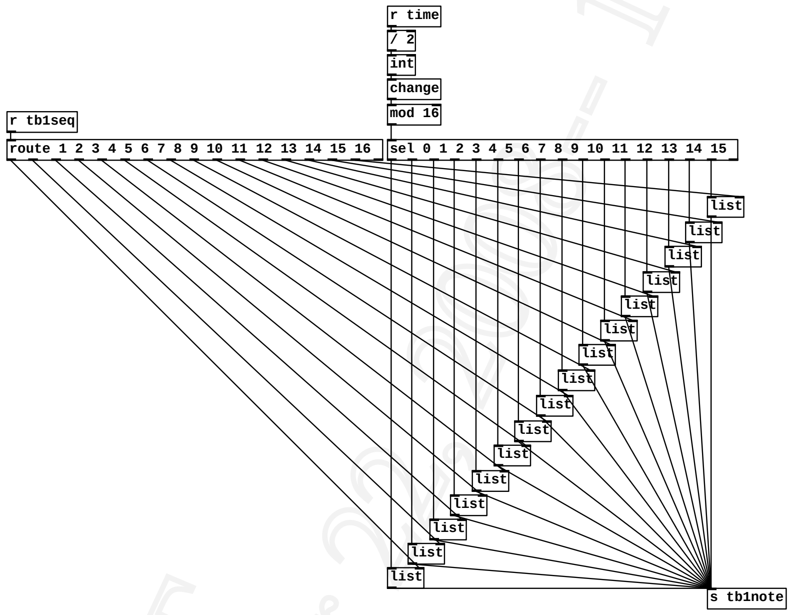

r time
/ 2
int
change
mod 16
r tb1seq
route 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
sel 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
list
s tb1note

fig 8.1: A synchronous list sequencer.

There are many, many ways of constructing sequencers. We have looked at list sequencers in earlier lectures. There are also ways of using textfiles, OSC, MIDI files and other data formats. The patch in 8.1 is something we haven't seen before though. It's a neat compromise between storing a lot of redundant data and having a compact bar/loop based approach. The timebase is divided to the right rate of beats per bar using /, int, mod and change. The resulting timebase loops over a 16 step select statement that fires the contents of a list box into the TB303 instrument. Each list box can actually contain a full parameter list, of arbitrary length for any synthesiser. But, I should point out here that it isn't necessary to hold a full parameter list for each step. Because of the way we constructed the synthesiser the importance of parameters decreases left to right. So, if we omit one or more parameters the next note will keep the last values unpacked. In fact we can send nothing more than a new MIDI note value and duration.

This makes using the synchronous list sequencer really easy. By prepending a step position to the head of a parameter list `route` will place it in the correct list box for us and it will be fired each time the bar cycles. We can drop in or erase new data on the fly. Sending the two element list `n 0` will erase the note at position `n`. At any time a comma delimited list sequence can be dropped into the bar sequencer in one go. The bar sequencer of 8.1 is encapsulated in a subpatch `pd tb1 bar seq`, because we will use this one to sequence the TB303. It receives instructions on `tb1seq` and outputs its live data on `tb1note`

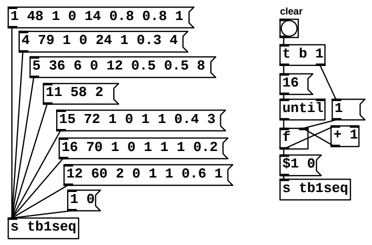


fig 8.2: Using the bar list sequencer.

8.2 shows how to use it. For human readability the sequence starts on beat 1 and ends on beat 16 (rather than numbering from 0) Each list adds a new entry to the sequencer at the position given by the first number, so the top example adds MIDI note 48 with a duration of 1 timebase period at beat 1. This note also specifies, no slide, a peak at the 10th harmonic with a resonance of 0.8, overall amplitude 0.8 and a drive factor of 1.0. The lists can be given in any order, and you can see that the one for beat 11 only gives the note and duration, so its parameters will be the same as the prior note, which is on beat 5. On the right of 8.2 is a demonstration of how to clear the whole bar in one operation.

Have a listen to some old skool type acid sequences made this way.

9 FM Bass

A great bass sound for techno and dance is obtained with simple FM. We have already studied the principles of FM and the evolution of sidebands according to modulator frequency, carrier frequency, FM index and Bessel functions, but let's recap with a very simple summary of the rules for those harmonic speactra useful in bass synthesis. For a small modulation index $i < 2$,

Carrier:Modulator	Waveform
$f_m = f_c$	Sawtooth wave
$f_m = 2f_c$	Square wave
$f_m = 3f_c$	Pulse wave

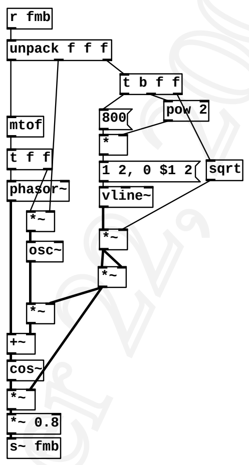


fig 9.1: A simple FM bass sound.

9.1 shows the patch for an FM bass sound with very few parameters. A note is received as a list of three parameters on `r fmb` and is unpacked into separate floats. The last, rightmost value represents the intensity of the note and affects both the decay time and modulation index. At unity (1) the decay is 800ms and

the FM index is 1.0. The amplitude and index are both square law functions of the line envelope, but while the decay increases as the square of the note intensity the index is modified as the square root. This produces a nice, natural change that sounds like damping over the range of about 0.5 to 1.25. The second (middle) parameter sets the FM modulator ratio and should be either 1 for a natural electric bass sound, 2 for synth type "hollow" bass sounds, or 3 for a buzz type bass. On each note the FM index is modulated from its peak value to zero by the envelope, giving a plucked effect.
As with the last instrument we will use a list style bar sequencer. 9.2 shows a sequence that exercises some of the possibilities of this patch.

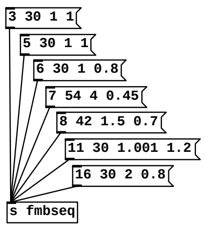
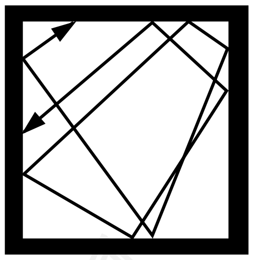


fig 9.2: Sequencing the FM bass sound.

Create a bar sequencer like the one used for the TB303 but give it an input destination `fmbseq` and an output send to `fmb`. If you haven't already, create a mixer channel for the FM bass. A little pingpong delay helps the sound.

Listen to the FM bass playing a short sequence.

10 Reverb



A reverb increses the density of a sound and spreads it out in time.

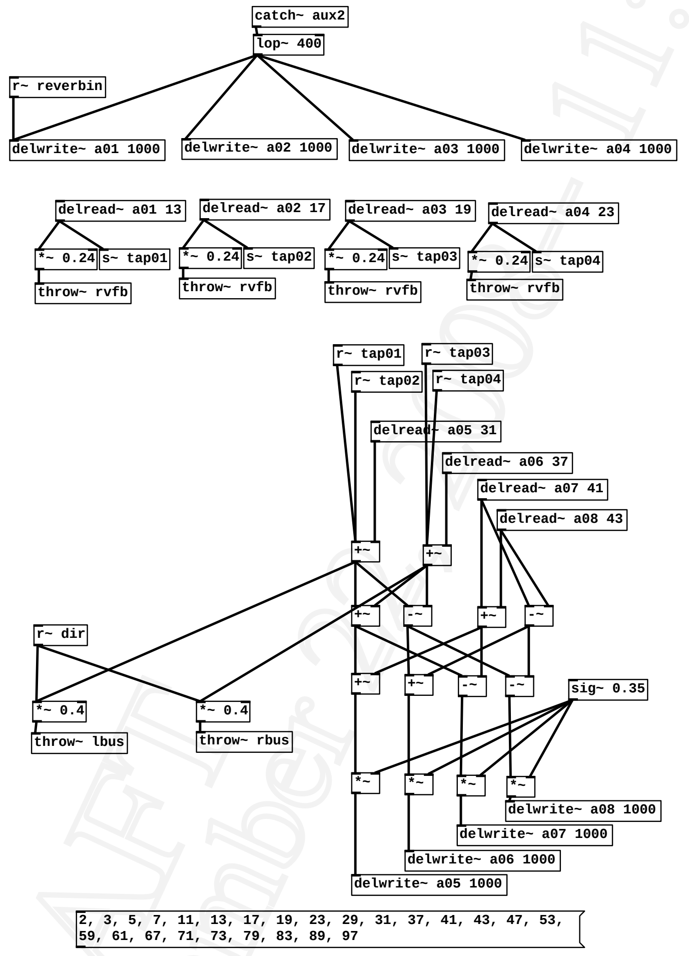
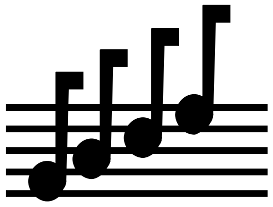


fig 10.1: A two stage simple reverb.

Natural reverb requires many delay objects or convolution. We have seen the Schroeder reverb and some other designs made from allpass networks. Here, in 10.1 is a greatly simplified two stage reverb, the first being to add early reflections and thicken the sound, the second a recirculator to add body over time.
Notice the use of prime numbers as delay parameters and a lop~ in the early reflection part to dampen the initial response. There are no controls for high or low frequency decay in the recirculator so the colour of this simple reverb is set by the choice of delay times. Feedback settings of 0.25 in the first stage and 0.33 in the second stage give a reverb time of about half a second which produces a fairly tight room type effect.
This effect is wired in to `aux2` on the mixer and returns its audio signal to the main bus.
Listen to the effect of reverb on the kick drum sound.

11
Pad



A pad synthesiser capable of string or brass like sounds is used to fill out the composition. We would like a dense spectrum, but unlike the hihat a perfectly harmonic one is needed. 11.1 shows a great trick for doing this efficiently.

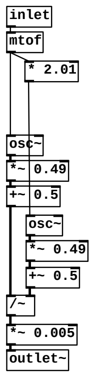


fig 11.1: Rich pulse oscillator.

Two oscillators are used with a /~ to obtain a "combing pulse train". This is a very fat sound. As you can see from 11.2 it consists of two sets of narrow pulses that slowly exchange with each other. They differ by one harmonic interval plus a small amount so a beating effect occurs. One set produces a spectrum starting at f with every other harmonic, and the other produces a spectrum starting at $2f$ with the same spacing. The result is a constantly shifting spectrum odd, even, odd, even . . .

Combining three such oscillators in subpatches `pd fatpulse voice` we obtain a three note pad synth. Because a DC offset is produced by this method some harsh highpass filtering is used to remove it. The particular way this pad is made produces a stuttering, gated effect which works well with a techno style of music. A gate having a duration set in timebase periods is given as the fourth patch parameter after the three MIDI note values. The gate is then multiplied by the timebase shifted back one position to place it on the right beat.

Listen to the gated pad sound.

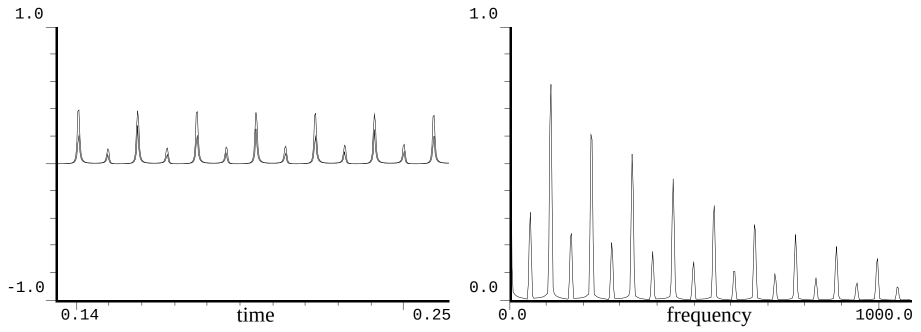


fig 11.2: Time graph and spectrum pulse oscillator.

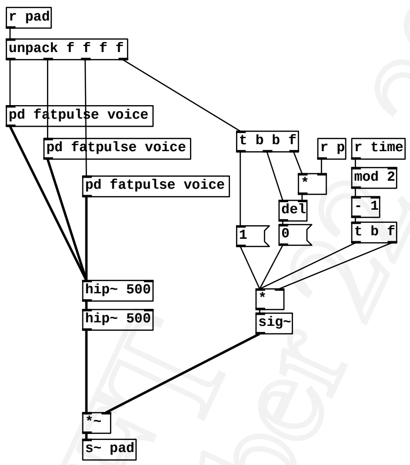


fig 11.3: Gated pulse pad synth.

12
Integration

$\int$

Pulling all the parts together is the last step. As you go, you will want to build up a more complex piece of music using the bar list sequences, chaining them together or providing interactive rules for when each should begin. To create a more coherent overview I've made the mixer and effects sections graph-on-parent as seen in the top level view of 12.1

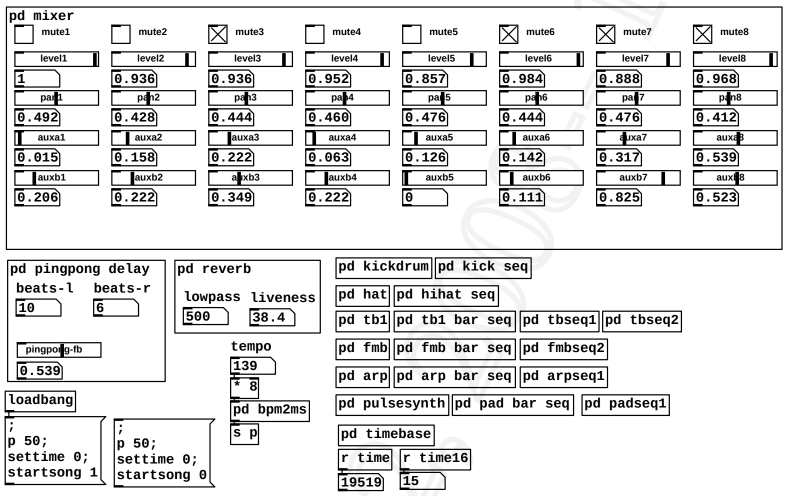


fig 12.1: Top level view of piece still in development.

With the ability to automate every part via parameter destinations, including the mixer settings you should be able to build up quite long and elaborate scores. Listen to a short exerpt of a sequence.