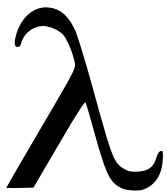
CHAPTER 5
Abstraction



SECTION 5.1

Subpatches

Any patch canvas can contain *subpatches* which have their own canvas but reside within the same file as the main patch, called the *parent*. They have inlets and outlets, which you define, so they behave very much like regular objects. When you save a canvas all subpatches that belong to it are automatically saved. A subpatch is just a neat way to hide code, it does not automatically offer the benefit of local scope[1].

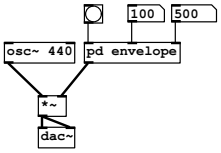


fig 5.1: Using an envelope subpatch

Any object that you create with a name beginning `pd` will be a subpatch. If we create a subpatch called `pd envelope` as seen in Fig. 5.1 a new canvas will appear and we can make `inlet` and `outlet` objects inside it as shown in Fig. 5.2. These appear as connections on the outside of the subpatch box in the same order they appear left to right inside the subpatch. I've given extra (optional) name parameters to the subpatch inlets and outlets. These are unnecessary, but when you have a subpatch with several inlets or outlets it's good to give them names to keep track of things and remind yourself of their function.

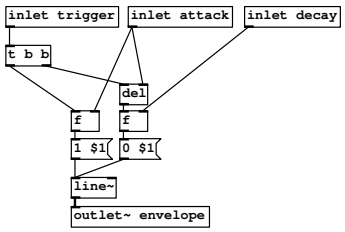


fig 5.2: Inside the envelope subpatch

To use `pd envelope` we supply a bang on the first inlet to trigger it, and two values for attack and decay. In Fig. 5.1 it modulates the output of an oscillator running at 440Hz before the signal is sent to `dac~`. The envelope has a trigger inlet for a message to bang two floats stored from the remaining inlets, one for the attack time in milliseconds and one for the decay time in milliseconds. The attack time also sets the period of a delay so that the decay portion of the envelope is not triggered until the attack part has finished. These values are substituted into the time parameter of a 2 element list for `line~`.

Copying subpatches

So long as we haven't used any objects requiring unique names any subpatch can be copied. Select `pd envelope` and hit `CTRL+D` to duplicate it. Having made

[1] As an advanced topic subpatches can be used as target name for dynamic patching commands or to hold datastructures.

one envelope generator it's a few simple steps to turn it into a MIDI mono synthesiser (shown in Fig. 5.3) based on an earlier example by replacing the osc~ with a phasor~ and adding a filter controlled by the second envelope in the range 0 to 2000Hz. Try duplicating the envelope again to add a pitch sweep to the synthesiser.

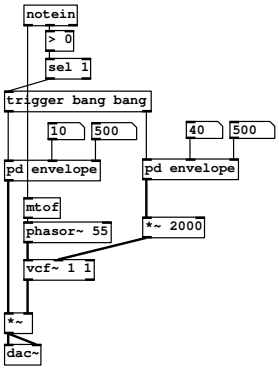


fig 5.3: Simple mono MIDI synth made using two copies of the same envelope subpatch

Deep subpatches

Consider an object giving us the vector magnitude of two numbers. This is the same as the hypotenuse c of a right angled triangle with opposite and adjacent sides a and b and has the formula $c = \sqrt{a^2 + b^2}$. There is no intrinsic object to compute this, so let's make our own subpatch to do the job as an exercise.

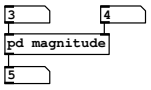


fig 5.4: Vector magnitude

We begin by creating a new object box and typing `pd magnitude` into it. A new blank canvas will immediately open for us to define the internals. Inside this new canvas create two new object boxes at the top by typing the word `inlet` into each. Create one more object box at the bottom as an `outlet`. Two input numbers a and b will come in through these inlets and the result c will go to the outlet.

When turning a formula into a dataflow patch it sometimes helps to think in reverse, from the bottom up towards the top. In words, c is the square root of the sum of two other terms, the square of a and the square of b. Begin by creating a sqrt object and connecting it to the outlet. Now create and connect a + object to the inlet of the sqrt. All we need to complete the example is an object that gives us the square of a number. We will define our own as a way to show that subpatches can contain other subpatches. And in fact this can go as deep as you like. It is one of the *principles of abstraction*, that we can define new

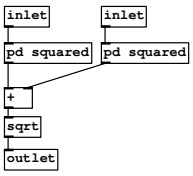


fig 5.5: Subpatch calculates $\sqrt{a^2 + b^2}$

objects, build bigger objects from those and still bigger objects in turn. Make a new object `pd squared` and when the canvas opens add the parts shown in Fig. 5.6.

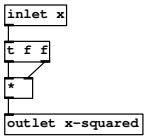


fig 5.6: Subpatch to compute x^2

To square a number you multiply it by itself. Remember why we use a trigger to split the input before sending it to each inlet of the multiply? We must respect evaluation order, so the trigger here distributes both copies of its input from right to left, the "cold" right inlet of * is filled first, then the "hot" left inlet. Close this canvas and connect up your new pd squared subpatch. Notice it now has an inlet and outlet on its box. Since we need two of them duplicate it by selecting then hitting `CTRL+D` on the keyboard. Your complete subpatch to calculate magnitude should look like Fig. 5.5. Close this canvas to return to the original topmost level and see pd magnitude now defined with two inlets and one outlet. Connect some number boxes to these as in Fig. 5.4 and test it out.

Abstractions

An abstraction is something that distances an idea from an object, it captures the essence and generalises it. It makes it useful in other contexts. Superficially an abstraction is a subpatch that exists in a separate file, but there is more to it. Subpatches add modularity and make patches easier to understand, which is one good reason to use them. However, while a subpatch seems like a separate object it is still part of a larger thing. *Abstractions* are reusable components written in plain Pd, but with two important properties. They can be loaded many times by many patches and although the same code defines all instances each instance has a separate internal namespace. They can also take creation arguments, so you can create multiple instances each with a different behaviour by typing different creation arguments in the object box. Basically, they behave like regular programming functions that can be called by many other parts of the program in different ways.

Scope and $0

Some objects like arrays and send objects must have a unique identifier, otherwise the interpreter cannot be sure which one we are referring to. In programming we have the idea of *scope* which is like a frame of reference. If I am talking to Simon in the same room as Kate I don't need to use Kate's surname every time I speak. Simon assumes, from context, that the Kate I am referring to is the most immediate one. We say that Kate has local scope. If we create an array within a patch and call it `array1` then that's fine so long as only one copy of it exists.

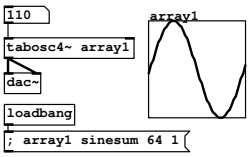


fig 5.7: Table oscillator patch

Consider the table oscillator patch in Fig. 5.7 which uses an array to hold a sine wave. There are three significant parts, a tabosc4~ running at 110Hz, a table to hold one cycle of the waveform and an initialisation message to fill the table with a waveform. What if we want to make a multi-oscillator synthesiser using this method, but with a square wave in a one table and a triangle wave in another? We could make a subpatch of this arrangement and copy it, or just copy everything shown here within the main canvas. But if we do that without changing the array name, Pd will say;

```
warning: array1: multiply defined
warning: array1: multiply defined
```

The warning message is given twice because while checking the first array it notices another one with the same name, then later, while checking the duplicate array, it notices the first one has the same name. This is a serious warning and if we ignore it erratic, ill defined behaviour will result. We could rename each array we create as `array1`, `array2`, `array3` etc, but that becomes tedious. What we can to do is make the table oscillator an abstraction and give the array a special name that will give it local scope. To do this, select everything with `CTRL+E, CTRL+A` and make a new file from the file menu (Or you can use `CTRL+N` as a shortcut to make a new canvas). Paste the objects into the new canvas with `CTRL+V` and save it as `my-tabosc.pd` in a directory called `tableocillator`. The name of the directory isn't important, but it is important that we know where this abstraction lives so that other patches that will use it can find it. Now create another new blank file and save it as `wavetablesynth` in the *same* directory as the abstraction. This is a patch that will use the abstraction. By default a patch can find any abstraction that lives in the same directory as itself.

SECTION 5.2

Instantiation

Create a new object in the empty patch and type `my-tabosc` in the object box. Now you have an instance of the abstraction. Open it just as you would edit a normal subpatch and make the following changes as shown in Fig. 5.8;

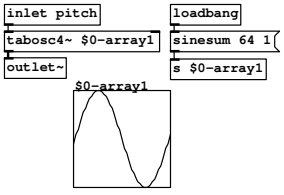


fig 5.8: Table oscillator abstraction

First we have replaced the number box with an inlet so that pitch data can come from outside the abstraction. Instead of a dac~ the audio signal appears on an outlet we've provided. The most important change is the name of the array. Changing it to `$0-array1` gives it a special property. Adding the `$0-` prefix makes it local to the abstraction because at runtime, `$0-` is replaced by a unique per instance number. Of course we have renamed the array referenced by tabosc4~ too. Notice another slight change in the table initialisation

code, the message to create a sine wave is sent explicitly through a send because $0- inside a message box is treated in a different way.

SECTION 5.3

Editing

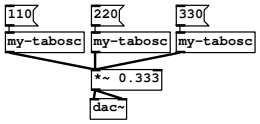


fig 5.9: Three harmonics using the table oscillator abstraction

Now that we have an abstracted table oscillator let's instantiate a few copies. In Fig. 5.9 there are three copies. Notice that no error messages appear at the console, as far as Pd is concerned each table is now unique. There is something important to note here though. If you open one of the abstraction instances and begin to edit it the changes you make will immediately take effect as with a subpatch, but they will only affect that instance. Not until you save an edited abstraction do the changes take place in *all* instances of the abstraction. Unlike subpatches, abstractions will not automatically be saved along with their parent patch and must be saved explicitly. Always be extra careful when editing abstractions to consider what the effects will be on all patches that use them. As you begin to build a library of reusable abstractions you may sometimes make a change for the benefit of one project that breaks another. How do you get around this problem? The answer is to develop a disciplined use of namespaces, prefixing each abstraction with something unique until you are sure you have a finished, general version that can used in all patches and will not change any more. It is also good practice to write help files for your abstractions. A file in the same directory as an abstraction, with the same name but ending `-help.pd` will be displayed when using the object help facility.

SECTION 5.4

Parameters

Making local data and variables is only one of the benefits of abstraction. A far more powerful property is that an abstraction passes any parameters given as creation arguments through local variables `$1, $2, $3`... In traditional programming terms this behaviour is more like a function than a code block. Each instance of an abstraction can be created with completely different initial arguments. Let's see this in action by modifying our table oscillator to take arguments for initial frequency and waveform. In Fig. 5.10 we see several interesting changes. Firstly, there are two float boxes that have `$n` parameters. You can use as many of these as you like and each of them will contain the nth creation parameter. They are all banged when the abstraction is loaded by the loadbang. The first sets the initial pitch of the oscillator, though of course this can still be over-ridden by later messages at the pitch inlet. The second activates one of three messages via select which contain harmonic series of square, sawtooth and sine waves respectively.

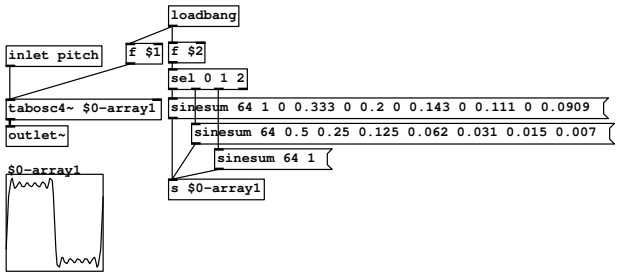


fig 5.10: Table oscillator abstraction with initialised frequency and shape.

SECTION 5.5

Defaults and states

A quick word about default parameters. Try creating some instances of the abstraction in Fig. 5.10 (shown as `my-tabsosc2` in Fig. 5.11)[2]. Give one a first parameter of 100Hz but no second parameter. What happens is useful, the missing parameter is taken to be zero. That's because `float` defaults to zero for an undefined argument. That's fine most of the time, because you can arrange for a zero to produce the behaviour you want. But, what happens if you create the object with no parameters at all? The frequency is set to 0Hz of course, which is probably useful behaviour, but let's say we wanted to have the oscillator start at 440Hz when the pitch is unspecified. You can do this with `sel 0` so that zero value floats trigger a message with the desired default. Be careful choosing default behaviours for abstractions, they are one of the most common causes of problems later when the defaults that seemed good in one case are wrong in another. Another important point pertains to initial parameters of GUI components, which will be clearer in just a moment as we consider abstractions with built in interfaces. Any object that persistently maintains state (keeps its value between saves and loads) will be the same for *all* instances of the abstraction loaded. It can only have one set of values (those saved in the abstraction file). In other words it is the abstraction *class* that holds state, not the object instances. This is annoying when you have several instances of the same abstraction in a patch and want them to individually maintain persistent state. To do this you need a state saving wrapper like `memento` or `sssad`, but that is a bit beyond the scope of this textbook.

[2]The graphs with connections to them shown here, and elsewhere in the book, are abstractions that contain everything necsessary to display a small time or spectrum graph from signals received at an inlet. This is done to save space by not showing this in every diagram.

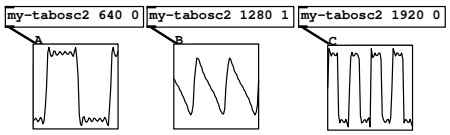


fig 5.11: Three different waveforms and frequencies from the same table oscillator abstraction

SECTION 5.6

Common abstraction techniques

Here are a few tricks regularly used with abstractions and subpatches. With these you can create neat and tidy patches and manage large projects made of reusable general components.

Graph On Parent

It's easy to build nice looking interfaces in Pd using GUI components like sliders and buttons. As a rule it is best to collect all interface components for an application together in one place and send the values to where they are needed deeper within subpatches. At some point it's necessary to expose the interface to the user, so that when an object is created it appears with a selection of GUI components laid out in a neat way.

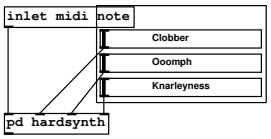


fig 5.12: Graph on parent synth

"Graph on Parent" (or GOP) is a property of the canvas which lets you see inside from outside the object box. Normal objects like oscillators are not visible, but GUI components, including graphs are. GOP abstractions can be nested, so that controls exposed in one abstraction are visible in a higher abstraction if it is also set to be GOP. In Fig. 5.12 we see a subpatch which is a MIDI synthesiser with three controls. We have added three sliders and connected them to the synth. Now we want to make this abstraction, called `GOP-hardsynth`, into a GOP abstraction that reveals the controls. Click anywhere on a blank part of the canvas, choose `properties` and activate the GOP toggle button. A frame will appear in the middle of the canvas. In the canvas properties box, set the size to $width = 140$ and $height = 80$, which will nicely frame three standard size sliders with a little border. Move the sliders into the frame, save the abstraction and exit.

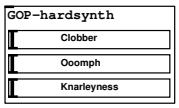


fig 5.13: Appearance of a GOP abstraction

Here is what the abstraction looks like when you create an instance (Fig. 5.13). Notice that the name of the abstraction appears at the top, which is why we left a little top margin to give this space. Although the inlet box partly enters the frame in Fig. 5.12 it cannot be seen in the abstraction instance because only GUI elements are displayed. Coloured *canvases*[3] also appear in GOP abstractions so if you want decorations they can be used to make things prettier. Any canvases appear above the name in the drawing order so if you want to hide the name make a canvas that fills up the whole GOP window. The abstraction name can be turned off altogether from the `properties` menu by activating `hide object name and arguments`.

Using list inputs

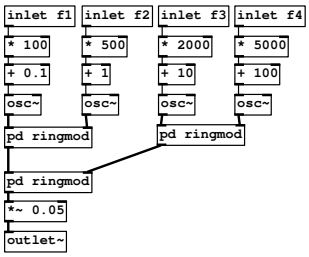


fig 5.14: Preconditioning normalised inlets

The patch in Fig. 5.14 is a fairly arbitrary example (a 4 source cross ring modulator). It's the kind of thing you might develop while working on a sound or composition. This is the way you might construct a patch during initial experiments, with a separate inlet for each parameter you want to modify. There are four inlets in this case, one for each different frequency that goes into the modulator stages. The first trick to take note of is the control pre-conditioners all lined up nicely at the top. These set the range and offset of each parameter so we can use uniform controls as explained below.

Packing and unpacking

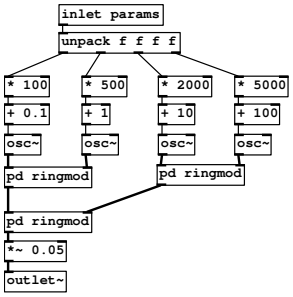
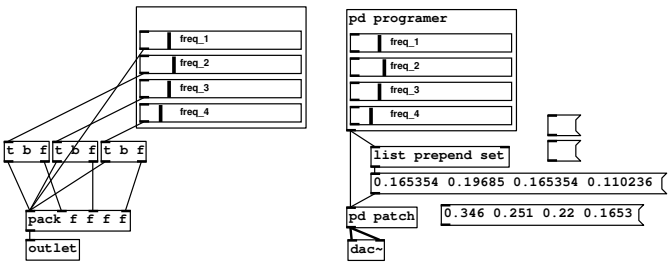


fig 5.15: Using a list input

What we've done here in Fig. 5.15 is simply replace the inlets with a single inlet that carries a list. The list is then unpacked into its individual members which are distributed to each internal parameter. Remember that lists are unpacked right to left, so if there was any computational order that needed taking care of you should start from the rightmost value and move left. This modification to the patch means we can use the flexible arrangement shown in Fig. 5.16 called a "programmer". It's just a collection of normalised sliders connected to a `pack` object so that a new list is transmitted each time a fader is moved. In order to do this it is necessary to insert `trigger bang float` objects between each slider as shown in Fig. 5.16 (left). These go on all but the far left inlet. Doing so ensures that the float value is loaded into `pack` before all the values are sent again. By prepending

[3]Here the word "canvas" is just used to mean a decorative background, different from the regular meaning of patch window.



(a) Packing a list (b) Making a programmer

fig 5.16: Packing and using parameter lists

the keyword `set` to a list, a message box that receives it will store those values. Now we have a way of creating patch presets, because the message box always contains a snapshot of the current fader values. You can see in Fig. 5.16 (right) some empty messages ready to be filled and one that's been copied ready to use later as a preset.

Control normalisation

Most patches require different parameter sets with some control ranges between 0.0 and 1.0, maybe some between 0.0 and 20000, maybe some bipolar ones −100.0 to +100.0 and so on. But all the sliders in the interface of Fig. 5.17 have ranges from 0.0 to 1.0. We say the control surface is *normalised*.

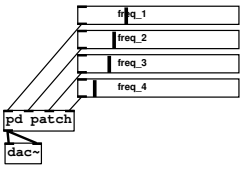


fig 5.17: All faders are normalised 0.0 to 1.0

If you build an interface where the input parameters have mixed ranges it can get confusing. It means you generally need a customised set of sliders for each patch. A better alternative is to normalise the controls, making each input range 0.0 to 1.0 and then adapting the control ranges as required inside the patch. Pre-conditioning means adapting the input parameters to best fit the synthesis pamaters. Normalisation is just one of the tasks carried out at this stage. Occasionally you will see a `log` or `sqrt` used to adjust the parameter curves. Pre-conditioning operations belong together as close to where the control signals are to be used as possible, They nearly always follow the same pattern, multiplier, then offset, then curve adjustment.

Summation chains

Sometimes when you have a lot of subpatches that will be summed to produce an output it's nicer to be able to stack them vertically instead of having many connections going to one place. Giving each an inlet (as in Fig. 5.18) and placing a `+~` object as part of the subpatch makes for easier to read patches.

Routed inputs

A powerful way to assign parameters to destinations while making them human readable is to use `route`. Look at Fig. 5.19 to see how you can construct arbitrary

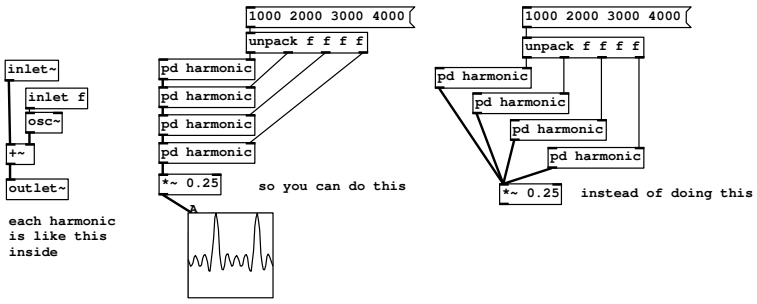


fig 5.18: Stacking subpatches that sum with an inlet

paths like URLs to break subpatches into individually addressable areas.

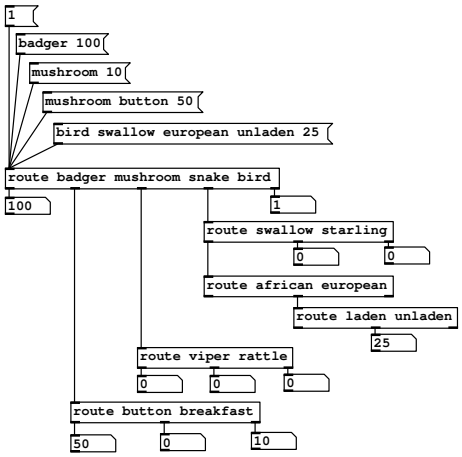
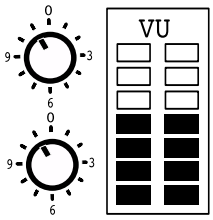


fig 5.19: Route can channel named parameters to a destination

CHAPTER 7
Pure Data essentials



This chapter will present some commonly used configurations for mixing, reading and writing files, communication and sequencing. You may want to build up a library of abstractions for things you do again and again, or to find existing ones from the pd-extended distribution. All the same, it helps to understand how these are built from primitive objects since you may wish to customise them to your own needs.

SECTION 7.1

Channel strip

For most work you will use Pd with multiple audio outlets and an external mixing desk. But you might find you want to develop software which implements its own mixing. All mixing desks consist of a few basic elements like gain controls, buses, panners and mute or channel select buttons. Here we introduce some basic concepts that can be plugged together to make complex mixers.

Signal switch

All we have to do to control the level of a signal is multiply it by a number between 0.0 and 1.0. The simplest form of this is a signal switch where we connect a toggle to one side of a *~ and an audio signal to the other (Fig. 7.1). The toggle outputs either 1 or 0, so the signal is either on or off. You will use this frequently to temporarily block a signal. Because the toggle changes value abruptly it usually produces a click, so don't use this simple signal switch when recording audio, for that you must apply some smoothing as in the mute button below.

fig 7.1: Signal switch

Simple level control

To create a level fader start with a vertical slider and set its `properties` to a lower value of 0.0 and upper value of 1.0. In Fig. 7.2 the slider is connected to one inlet of *~ and the signal to the other, just like the signal switch above except the slider gives a continuous change between 0.0 and 1.0. A number box displays the current fader value, 0.5 for a halfway position here. A sine oscillator at 40Hz provides a test signal. It is okay to mix messages and audio signals on opposite sides of *~ like this, but because the slider generates messages any updates will only happen on each

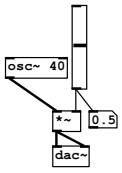


fig 7.2: Direct level control

block, normally every 64 samples. Move it up and down quickly and listen to the result. Fading is not perfectly smooth. You will hear a clicking sound when you move the slider. This *zipper noise* is caused by the level suddenly jumping to a new value on a block boundary.

Using a log law fader

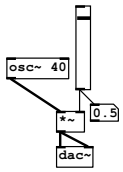


fig 7.3: log level control

The behaviour of slider objects can be changed. If you set its properties to log instead of linear then smaller values are spread out over a wider range and larger values are squashed into the upper part of the movement. This gives you a finer degree of control over level and is how most real mixing desks work. The smallest value the slider will output is 0.01. With its top value as 1.0 it will also output 1.0 when fully moved. Between these values it follows a logarithmic curve. When set to halfway it outputs a value of about 0.1 and at three quarters of full movement its output is a little over 0.3. It doesn't reach an output of 0.5 until nearly nine tenths of its full movement (shown in Fig. 7.3). This means half the output range is squashed into the final ten percent of the movement range, so be careful when you have this log law fader connected to a loud amplifier. Often log law faders are limited to constrain their range, which can be done with a `clip` unit.

MIDI fader

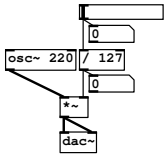


fig 7.4: Scaling a level

You won't always want to control a mix from Pd GUI sliders, sometimes you might wish to use a MIDI fader board or other external control surface. These generally provide a linear control signal in the range 0 to 127 in integer steps, which is also the default range of GUI sliders. To convert a MIDI controller message into the range 0.0 to 1.0 it is divided by 127 (the same as multiplying by 0.0078745) as shown in Fig. 7.4. The normalised output can be further scaled to a log curve, or multiplied by 100 to obtain a decibel scale and converted via the `dbtorms` object.

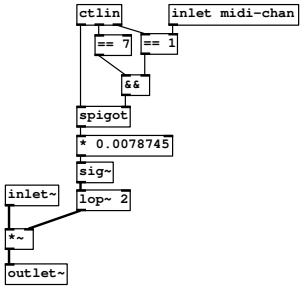


fig 7.5: MIDI level

To connect the fader to an external MIDI device you need to add a `ctlin` object. The first outlet gives the current fader value, the second indicates the continuous controller number and the third provides the current MIDI channel. Volume messages are sent on controller number 7. We combine the outlets using `==` and `spigot` so that only volume control messages on a particular channel are passed to the fader. The patch shown in Fig. 7.5 has an audio inlet and outlet. It has an inlet to set the MIDI channel. It can be subpatched or abstracted to form one of several components in a complete MIDI controlled fader board.

Mute button and smooth fades

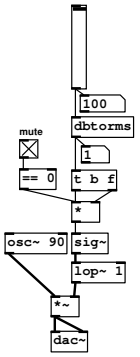


fig 7.6: mute switch

After carefully adjusting a level you may want to temporarily silence a channel without moving the slider. A *mute button* solves this problem. The fader value is stored at the cold inlet of a * while the left inlet receives a Boolean value from a toggle switch. The usual sense of a mute button is that the channel is silent when the mute is active, so first the toggle output is inverted. Some solutions to zipper noise use line or line~ objects to interpolate the slider values. Using line is efficient but somewhat unsatisfactory since we are still interfacing a message to a signal and will hear clicks on each block boundary even though the jumps are smaller. Better is to use line~, but this can introduce corners into the control signal if the slider moves several times during a fade. A good way to obtain a smooth fade is to convert messages to a signal with sig~ and then low pass filter it with lop~. A cutoff value of 1Hz will make a fade that smoothly adjusts over 1 second.

Panning

Pan is short for panorama, meaning a view in all directions. The purpose of a pan control is to place a *mono* or *point* source within a listening panorama. It should be distinguished from *balance* which positions a sound already containing stereo information. The field of an audio panorama is called the *image* and with plain old stereo we are limited to a theoretical *image width* of 180°. In practice a narrower width of 120° is used. Some software applications specify the pan position in degrees, but this is fairly meaningless unless you know precisely how the loudspeakers are arranged or whether the listener is using headphones. Mixing a stereo image for anything other than movie theatres is always a compromise to account for the unknown final listening arrangement. In movie sound however, the specifications of theatre PA systems are reliable enough to accurately predict the listeners experience.

Simple linear panner

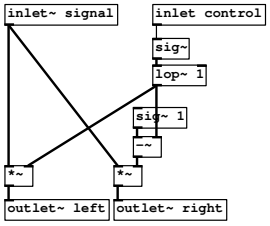


fig 7.7: simple panner

In the simplest case a pan control provides for two speakers, left and right. It requires that an increase on one side has a corresponding decrease on the other. In the center position the sound is distributed equally to both loudspeakers. The pan patch in Fig. 7.7 shows a signal inlet and control message inlet at the top and two signal outlets at the bottom, one for the left channel and one for the right. Each outlet is preceded by a multiplier to set the level for that channel, so the patch is essentially two level controls in one. As with our level control, zipper noise is removed by converting control messages to a signal and then smoothing them with a filter. The resulting control signal, which is in the range 0.0 to

1.0, is fed to the left channel multiplier, while its complement (obtained by subtracting it from 1.0) governs the right side. With a control signal of 0.5 both sides are multiplied by 0.5. If the control signal moves to 0.75 then the opposing side will be 0.25. When the control signal reaches 1.0 the complement will be 0.0, so one side of the stereo image will be completely silent.

Square root panner

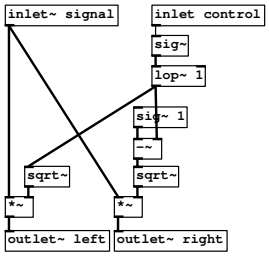


fig 7.8: root law panner

The problem with simple linear panning is that when a signal of amplitude 1.0 is divided in half and sent to two loudspeakers, so each receives an amplitude of 0.5, the result is quieter than sending an amplitude of 1.0 to only one speaker. This doesn't seem intuitive to begin with, but remember loudness is a consequence of sound power level, which is the square of amplitude. Let's say our amplitude of 1.0 represents a current of 10A. In one loudspeaker we get a power of $10^2 = 100$W. Now we send it to equally amongst two speakers, each receiving a current of 5A. The power from each speaker is therefore $5^2 = 25$W and the sum of them both is only 50W. The real loudness has halved! To remedy this we can modify the curve used to multiply each channel, giving it a new *taper*. Taking the square root of the control signal for one channel and the square root of the complement of the control signal for the other, gives panning that follows an *equal power law*. This has a 3dB amplitude increase in the center position.

Cosine panner

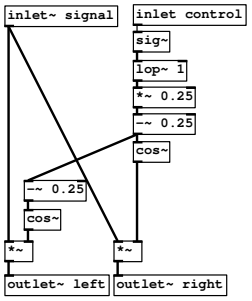


fig 7.9: cos-sin law panner

While the square root law panner gives a correct amplitude reduction for centre position it has a problem of its own. The curve of $\sqrt{A}$ is perpendicular to the x axis as it approaches it, so when adjusting the panning close to one side the image suddenly disappears completely from the other. An alternative taper follows the *sine-cosine law*. This also gives a smaller amplitude reduction in the centre position, but it approaches the edges of the image smoothly, at 45 degrees. The cosine panner is not only better in this regard but slightly cheaper in CPU cycles since it's easier to compute a cosine than a square root. It also mimics the placement of the source on a circle around the listener and is nice for classical music as an orchestra is generally arranged in a semicircle, however some engineers and producers prefer the root law panner because it has a nicer response around the center position and signals are rarely panned hard left or right.

Fig. 7.10 shows the taper of each panning law. You can see that the linear method is 3dB lower than the others in the centre position and that the root and cosine laws have different approaches at the edge of the image.

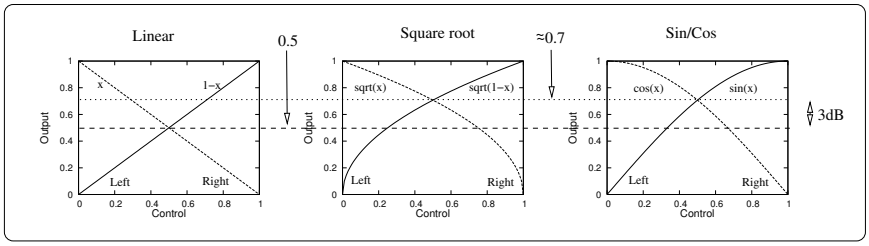


fig 7.10: Linear, root and sin/cos panning laws

Combining the cosine panner patch with a ctlin we now have a MIDI controlled pan unit to add to the MIDI controlled fader. Pan information is sent on controller number 10, with 64 representing the centre position. Once again an inlet is provided to select the MIDI channel the patch responds to. You may like to expand this idea into a complete MIDI fader board by adding a mute, bus outlet and auxiliary send/return loop. It might be a good solution to combine the level control, panning, mute and routing into a single abstraction that takes the desired MIDI channel and output bus as creation arguments. Remember to use dollar notation to create local variables if you intend to override MIDI control with duplicate controls from GUI objects.

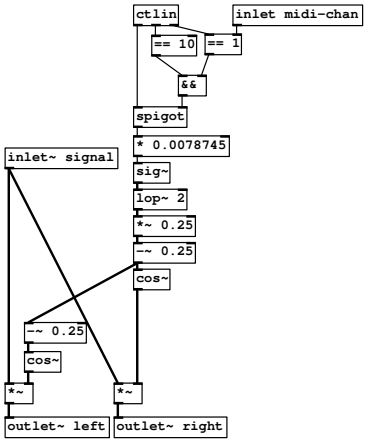


fig 7.11: MIDI panner

Crossfader

The opposite of a pan control, a reverse panner if you like, is a crossfader. When you want to smoothly transfer between two sound sources by mixing them to a common signal path, the patch shown in Fig. 7.12 can be used. There are three signal inlets, two of them are signals to be mixed and one is a control signal to set the ratio (of course a message domain version would work equally well with appropriate anti-zipper smoothing). It can be used in the final stage of a reverb effects unit to set the wet/dry proportion, or in a DJ console to crossfade between two tunes. Just like the simple panner, the control signal is split into two parts, a direct version and the complement with each modulating an input signal. The output is the sum of both multipliers. This type is a linear

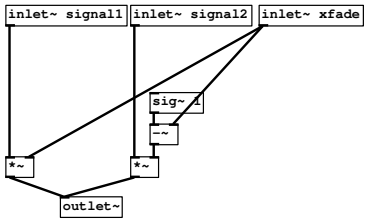


fig 7.12: crossfader

crossfader, but in some situations crossfades may be better with constant power fading done using sine or square root transfer functions.

Demultiplexer

A demultiplexer or signal source selector is a multi-way switch that can choose between a number of signal sources. Fig. 7.13 is useful in synthesiser construction where you want to select from a few different waveforms. In this design the choice is exclusive so only one input channel can be sent to the output at any time. A number at the control inlet causes `select` to choose one of four possible messages to send to `unpack`. The first turns off all channels, the second switches on only channel one and so forth. The Boolean values appearing in the `unpack` output are converted to signals and then lowpassed at 80Hz to give a fast but click free transition.

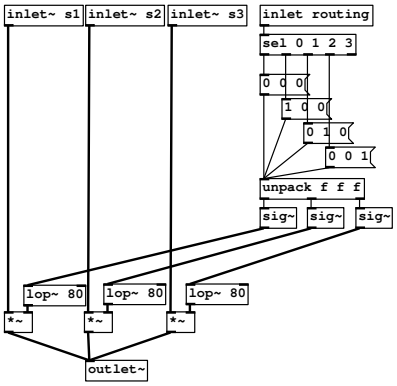


fig 7.13: demultiplex

SECTION 7.2

Audio file tools

Monophonic sampler

A useful object to have around is a simple sampler that can grab a few seconds of audio input and play it back. Audio arrives at the first inlet and is scaled by a gain stage before being fed to `tabwrite~`. It's nice to have a gain control so that you don't have to mess with the level of the signal you are recording elsewhere. In Fig. 7.14 a table of 88200 samples is created called `$0-a1`, so we have a couple of seconds recording time. Obviously this can be changed in the code or a control created to use the `resize` method. When it receives a bang `tabwrite~` starts recording from the beginning of the table. To play back the recording we issue a bang to `tabplay~` which connects directly to the outlet. The use of dollar arguments means this patch can be abstracted and multiple instances created, it's not unusual to want a bunch of samplers when working on a sound.

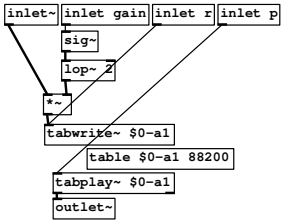


fig 7.14: simple sampler

Using the sampler is very easy. Create an instance and connect it to a signal source via the first inlet. In Fig. 7.15 the left audio input is taken from adc~. A slider with a range 0.0 to 1.0 connects to the gain inlet and two bang buttons are used to start recording or playback. Sound files of up to 3min can be stored happily in memory. Beyond this limit you need to use other objects for 32 bit machines because the sound quality will suffer due to pointer inaccuracies. If you have files longer than 3 minutes then you may want to think about using disk based storage and playback.

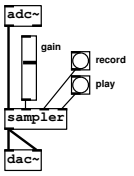


fig 7.15: using a sampler

File recorder

When creating sounds for use in other applications, like multitracks or samplers you could choose to record the output of Pd directly from the dac~ using your favourite wave file editor or software like *Timemachine*. This could mean editing long recordings later, so sometimes you want to just write fixed length files directly from Pd.

In Fig. 7.16 we see a file writer in use, which I will show you how to make in a moment. It catches audio, perhaps from other patches, on a bus called `audio`. It was created with two arguments, the length of each file to record (in this case 1s) and the name of an existing folder beneath the current working directory in which to put them. Each time you hit the `start` button a new file is written to disk and then the `done` indicator tells you when it's finished. A numerical suffix is appended to each file, which you can see on the second outlet, in order to keep track of how many files you've created. The internals of the file writer are shown

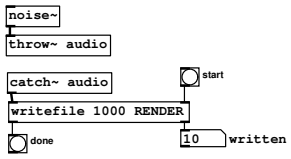


fig 7.16: Using a file writer

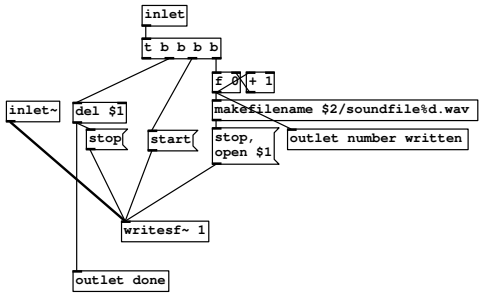


fig 7.17: Making a file writer

in Fig. 7.17. Audio comes into the first inlet and to the writesf~ object which has an argument of 1, so writes a single channel (mono) file. There are three commands that writesf~ needs, the name of a file to open for writing, a start command, and a stop command. Each bang on the second inlet increments

a counter and the value of this is appended to the current file name using makefilename which can substitute numerical values into a string like the C printf statement does. This string is then substituted after the `open` keyword in the following message. As soon as this is done a `start` message is sent to writesf~ and a bang to the delay which waits for a period given by the first argument before stopping writesf~.

Loop player

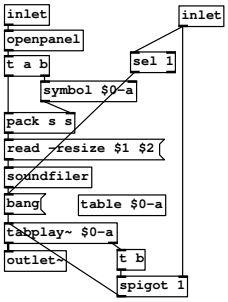


fig 7.18: sample loop player

A looping sample player is useful in many situations, to create a texture from looped background samples, or to provide a beat from a drum loop, especially if you need a continuous sound to test some process with. In Fig. 7.18 we see a patch that should be created as an abstraction so that many can be instantiated if required. It's operation is unsophisticated, just playing a loop of a sound file forever. When the abstraction receives a bang openpanel is activated and provides a nice file dialogue for you to choose a sound file. You should pick a Microsoft .wav or Mac .aiff type, either stereo or mono will do but this player patch will only give mono output. The name and path of this file is passed through the trigger "any" outlet and packed as the first part of a list along with a second part which is a symbol `$0-a`. The second symbol is the name of our storage table, the place in memory where the contents of the soundfile will be put once read. It has the prefix `$-` to give it local scope so we can have many sample loop players in a patch. Now the elements of the list will be substituted in $1 and $2 of the message `read -resize $1 $2`, which forms a complete command to soundfiler telling it to read in a sound file and put it in an array resizing the array as required. Once this operation is complete soundfiler returns the number of bytes read, which in this case we ignore and simply trigger a new bang message to start tabplay~. Notice the argument is the name of the array living in the table just above it. tabplay~ will now play once through the file at its original sample rate, so there is no need to tune it. When it has finished, the right outlet emits a bang. We take this bang, buffering it through another trigger and apply it back to the tabplay~ inlet, which means it plays the sound forever in a loop. A zero arriving at the second inlet allows you to stop the loop playing.

SECTION 7.3

Events and sequencing

Now let's look at a few concepts used for creating time, sequences and event triggers.

Timebase

At the heart of many audio scenes or musical constructions is a timebase to drive events. We've already seen how to construct a simple timebase from a

metronome and counter. A more useful timebase is given in Fig. 7.19 that allows you to specify the tempo as beats per minute (BPM) and to add a "swing"[1] to the beat. Notice first that start and stop control via the first inlet also resets

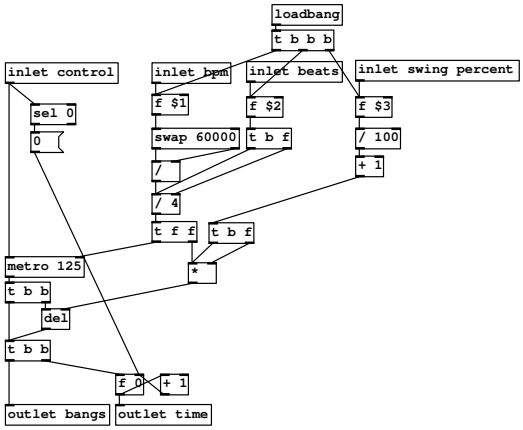


fig 7.19: A more useful musical timebase abstraction with BPM and swing

the counter when the timebase is stopped. Bangs from `metro` are duplicated with a `delay` object so we can position every other beat relative to the main rhythm. To convert beats per minute to a period in milliseconds it is divided by 60000 and multiplied by the number of beats per bar. The last parameter provides swing as a percentage which is added to the delay prior to incrementing the counter.

Select sequencer

The simplest way to obtain regular patterns for repetitive sounds is by using `mod` to wrap the incoming time to a small range, say 8 beats, and then use `select` to trigger events within this range. You do not have to fill out all the select values, so for example, to produce a single trigger at $time = 1024$ you can connect one `select` matching this number. A good practice is to broadcast a global time message so that other patches can pick up a common reference. In Fig. 7.20 the output from the timebase abstraction goes to a `send`. To create a sequencer where you can manually set the time at which an event is triggered, use a com-

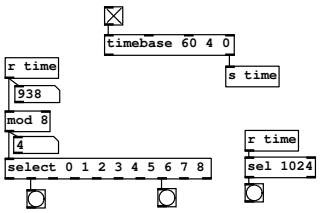


fig 7.20: Select based triggering

[1]Swing is where every other beat is moved slightly in time giving a different feel to the rhythm.

bination of == and select with a number box attached to the cold inlet of == and the current time going to the left inlet.

Partitioning time

For long musical compositions, interactive installations or generating event structures for a long game or animation you may want to offset timing sequences by a large number but keep the relative timings within a section. This is how bars and measures work in a traditional sequencer. In Fig. 7.21 moses is used to split the global time into smaller frames of reference. A chain of moses objects splits off numbers that fall within a range. You can see that the last value present on the left outlet of the first split was 127. Numbers of 128 or more are passing through the right outlet and into the second moses which partitions values between 128 and 255. We subtract the base value of 128 from this stream to reposition it, as if it were a sequence starting at zero. This can be further processed, such as wrapping it into the range 0 to 64 to create 2 bars of 64 beats in the range 128 to 256. In Fig. 7.21 you see the timebase at 208, which is in the second bar of the partitioned timeframe.

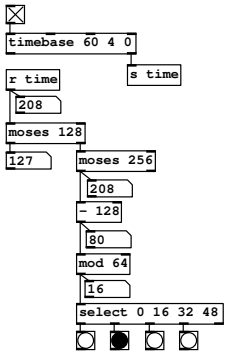


fig 7.21: Bar offset by partitioning time

Dividing time

With time expressed as a number you can perform arithmetic on it to obtain different rates. Be aware that although the value of numerical time changes with a different scale it still updates at the rate set by the timebase. Since for musical purposes you want to express time in whole beats and bars a problem is presented. Dividing time by two and rounding it to an integer means two messages will now be sent with the same value. To get around this problem change is used so that redundant messages are eliminated. Using int means values are rounded to the time floor, so if rhythms constructed this way seem one beat out of alignment you can try using a "closest integer" rounding explained earlier. Sometimes rounding time is not what you want as shown in the next example.

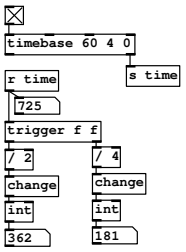


fig 7.22: Dividing time into different rates

Event synchronised LFO

An application and pitfall of timebase division is shown in Fig. 7.23 where low frequency control signals are derived from the timebase. Notice how the sine wave is not interpolated, so you get two or four consecutive equal values when using a divided timebase. This makes the LFO jumpy so to avoid it we scale the raw time values before the trig operation using a higher mod and /. It

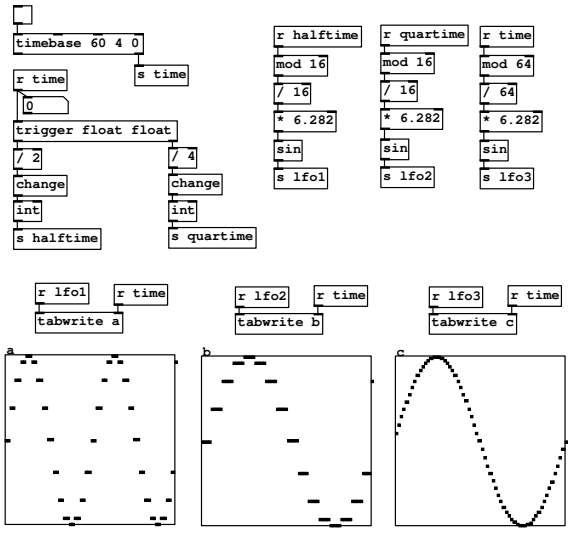


fig 7.23: Synchronous message LFOs

illustrates why you should often use a timebase that is a large multiple (say 64 times) of the real event rate you want. You might use this to create interesting polyrhythms or elaborate slow moving control signals for wind, rain or spinning objects.

List sequencer

An alternative to an absolute timebase is using lists and delays to make a relative time sequencer. Events are stored in a list, which we define to have a particular meaning to a sequencer that will interpret it. In this case the list is read in pairs, an event type and a time offset from the last event. So, a list like {*1 0 2 200 1 400*} describes three events and two event types. Event 1 occurs at $time = 0$ and then at $time = 200$ event 2 occurs, followed by event 1 again at $time = 200 + 400 = 600$. Times are in milliseconds and event types usually correspond to an object name or a MIDI note number. The patch in Fig. 7.24 is hard to follow, so I will describe it in detail. The sequence list arrives at the first inlet of `list split 2` where it is chopped at the second element. The first two elements pass to the `unpack` where they are separated and processed, while the remainder of the list passes out of the second outlet of `list split 2` and into the right inlet of `list append`. Returning to `unpack`, our first half of the current pair which identifies a float event type is sent to the cold inlet of a `float` where it waits, while the second part which represents a time delay is passed to `delay`. After a delay corresponding to this second value `delay` emits a bang message which flushes out the value stored in `float` for output. Finally, `list append` is

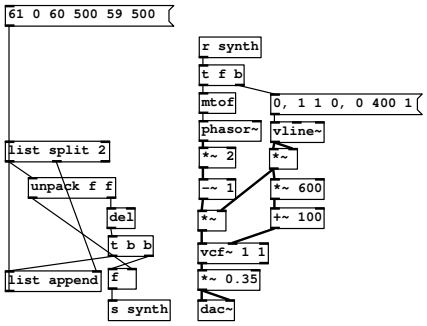


fig 7.24: An asynchronous list sequencer

banged so the remainder of the list is passed back to `list split 2` and the whole process repeats, chomping 2 elements off each time until the list is empty. To the right of Fig. 7.24 is a simple monophonic music synthesiser used to test the sequencer. It converts MIDI note numbers to Hertz with `mtof` and provides a filtered sawtooth wave with a 400ms curved decay envelope. To scale the sequence delay times, and thus change the tempo without rewriting the entire list, you can make each time offset be a scaling factor for the delay which is then multiplied by some other fraction. List sequencers of this type behave asynchronously, so don't need a timebase.

Textfile control

Eventually lists stored in message boxes become unwieldy for large data sets and it's time to move to secondary storage with textfiles. The `textfile` object provides an easy way to write and read plain text files. These can have any format you like, but a general method is to use a comma or linebreak delimited structure to store events or program data. It is somewhat beyond this textbook to describe the many ways you can use this object, so I will present only one example of how to implement a text file based MIDI sequencer. A combination of `textfile` and `route` can provide complex score control for music or games. If you need even larger data sets with rapid access a SQL object is available in `pd-extended` which can interface to a database.

Starting at the top left corner of Fig. 7.25 you can see a monophonic synthesiser used to test the patch. Replace this with a MIDI note out function if you like. The remainder of the patch consists of two sections, one to store and write the sequence and one to read and play it back. Recording commences when the `start-record` button is pressed. This causes a `clear` message to be sent to `textfile`, the list accumulator is cleared and the `timer` object reset. When a note is received by `notein` and then reduced to just its note-on value by `stripnote` it passes to the trigger unit below which dispatches two bangs to `timer`. The result of this is for `timer` to output the time since the last bang it received, then

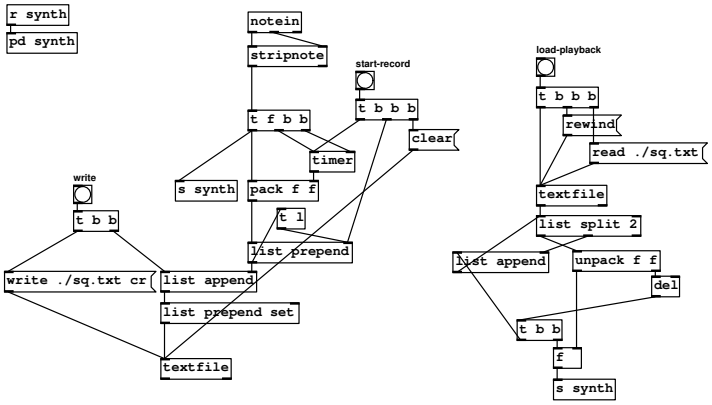


fig 7.25: A MIDI sequencer that uses textfiles to store data

restart from zero. This time value, along with the current MIDI note number, is packed by `pack` into a pair and appended to the list accumulator. When you are done playing, hit the `write` button to flush the list into `textfile` and write it to a file called `sq.txt` in the current working directory. Moving to the load and replay side of things, banging the `load-replay` button reads in the textfile and issues a `rewind` message setting `textfile` to the start of the sequence. It then receives a bang which squirts the whole list into a list sequencer like the one we just looked at.

SECTION 7.4

Effects

For the last part of this chapter I am going to introduce simple effects. Chorus and reverb are used to add depth and space to a sound. They are particularly useful in music making, but also have utility in game sound effects to thicken up weaker sources. Always use them sparingly and be aware that it is probably better to make use of effects available in your external mixer, as plugins, or as part of the game audio engine.

Stereo chorus/flanger effect

The effect of chorus is to produce a multitude of sources by doubling up many copies of the same sound. To do this we use a several delays and position them slightly apart. The aim is to deliberately cause beating and swirling as the copies move in and out of phase with one another. In Fig. 7.26 an input signal at the first inlet is split three ways. An attenuated copy is fed directly to the right stereo outlet while two other copies are fed to separate delay lines. In the centre you see two variable delay taps, `vd~`, which are summed.

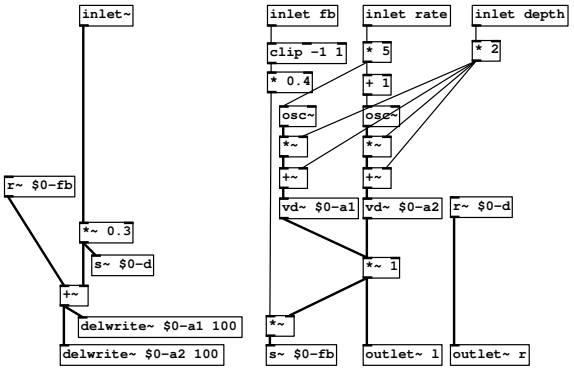


fig 7.26: A chorus type effect

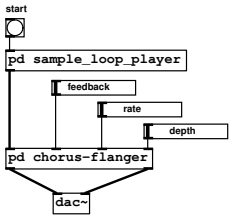


fig 7.27: Testing the chorus

A small part, scaled by the feedback value on the second inlet, is sent back to be mixed in with the input signal, while another copy is sent to the left stereo outlet. So there is a dry copy of the signal on one side of the stereo image and two time shifted copies on the other. By slowly varying the delay times with a couple of signal rate LFOs a swirling chorus effect is achieved. The low frequency oscillators are always 1Hz apart and vary between 1Hz and 5Hz. It is necessary to limit the feedback control to be sure the effect cannot become unstable. Notice that feedback can be applied in positive or negative phase to create a notching effect (phaser/flanger) and a reinforcing effect (chorus). Testing out the effect is best with a sample loop player. Try loading a few drum loops or music loop clips.

Simple reverberation

A reverb simulates dense reflections as a sound bounces around inside some space. There are several ways of achieving this effect, such as convolving a sound with the impulse response of a room or using allpass filters to do a similar thing. In Fig. 7.28 you can see a design for a recirculating reverb type that uses only delay lines. There are four delays which mutually feed back into one another, so once a signal is introduced into the patch it will circulate through a complex path. So that reinforcement doesn't make the signal level keep growing some feedback paths are negative. The recirculating design is known as a Schroeder reverb (this example by Claude Heiland-Allen) and mimics four walls of a room. As you can see the number of feedback paths gets hard to patch if we move to 6 walls (with floor and ceiling) or to more complex room shapes. Reverb design is a fine art. Choosing the exact feedback and delay values is not easy. If they are wrong then a feedback path may exist for certain frequencies producing an unstable effect. This can be hard to detect in practice and complex to predict in

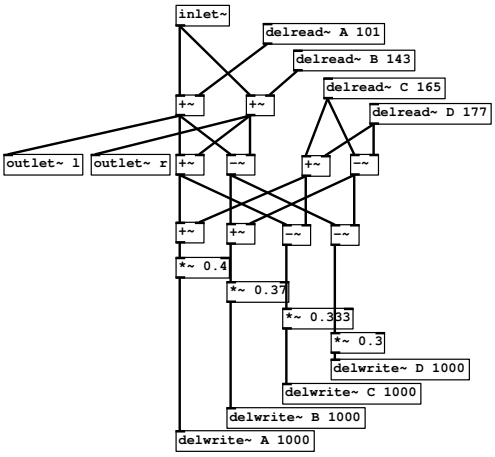


fig 7.28: A recirculating Schroeder reverb effect

theory. An apparently well designed reverb can mysteriously explode after many seconds or even minutes so a common design safety measure is to attenuate the feedback paths as the reverb decays away. What defines the reverb time is the point at which the reverb is has fallen to −60dB of the first reflection intensity. A good design should not be too coloured, which means feedback paths must not be too short leading to a pitched effect. The minimum delay time should be at least a quarter of the reverberation time and the lengths of delays should be prime, or collectively co-prime [2]. The density of the reverb is important too. Too little and you will hear individual echos, too much and the effect will become muddy and noisy. Schroeder suggests 1000 echoes per second for a reasonable reverb effect. If you look in the `extra` directory that comes with Pd there are three nice reverb abstractions `rev1~`, `rev2~` and `rev3~` by Miller Puckette

Exercises

Exercise 1

Create any one of the following effects.

- Guitar tremolo effect
- Multi-stage phaser
- Multi-tap tempo-sync delay
- A high quality vocal reverb

Exercise 2

Create a sequencer that provides any two of,

[2] A set of integers with no common factors are said to be collectively co-prime.

- Hierarchical structure
- Microtonal tuning scales
- Polyrhythmic capabilities
- A way to load and save your compositions

Exercise 3

Design and implement a mixing desk with at least three of,

- MIDI or OSC parameter automation
- Switchable fader and pan laws
- Surround sound panning (eg 5.1, quadraphonic)
- Effect send and return bus
- Accurate signal level monitoring
- Group buses and mute groups
- Scene store and recall

Exercise 4

Essay: Research datastructures in Pd. How can graphical representations help composition? What are the limitations of graphics in Pd? Generally, what are the challenges for expressing music and sound signals visually?

References

Zoelzer, U. (2008) "Digital Audio Signal Processing" (Wiley) ISBN-13: 978-0470997857

Penttinen, H., Tikander, M. (2001) Spank the reverb: In "Reverb Algorithms, Course report for Audio Signal Processing S-89.128"

Gardner, W. G. (1998) "Reverberation Algorithms" in M. Kahrs and K. Brandenburg (eds.), Applications of Digital Signal Processing to Audio and Acoustics. Kluwer, pp. 85-131.

Schroeder, M. R. (1962) "Natural Sounding Artificial Reverberation" J. Audio Eng. Soc., vol. 10, no. 3, pp. 219-224.

Case, A. (2007) "Sound FX: Unlocking the Creative Potential of Recording Studio Effects" (Focal) ISBN-13: 978-0240520322

Izhaki, R. (2007) "Mixing Audio: Concepts, Practices and Tools" (Focal) ISBN-13: 978-0240520681

Online resources

http://puredata.info/ is the site of the main Pure Data portal.

http://crca.ucsd.edu/ is the current home of official Pure Data documentation by Miller Puckette.

Beau Sievers "The Amateur Gentleman's Introduction to Music Synthesis" An introductory online resource geared toward synth building in Pure Data. http://beausievers.com/synth/synthbasics/

http://www.musicdsp.org/ is the home of the music DSP list archive, with categorised source code and comments.

http://www.dafx.de/ is home of the DAFx (Digital Audio Effects) project containing many resources.

Acknowledgements

I would like to thank Frank Barknecht, Steffen Juul, Marius Schebella, Joan Hiscock, Philippe-Aubert Gauthier, Charles Henry, Cyrille Henry and Thomas Grill for their valuable help in preparing this part.