12 Abstraction

SECTION 12.1

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]

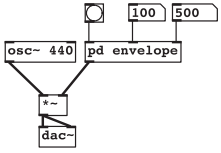


Figure 12.1
Using an envelope subpatch.

Any object that you create with a name beginning with `pd` will be a subpatch. If we create a subpatch called `pd envelope` as seen in figure 12.1 a new canvas will appear, and we can make `inlet` and `outlet` objects inside it as shown in figure 12.2. 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.

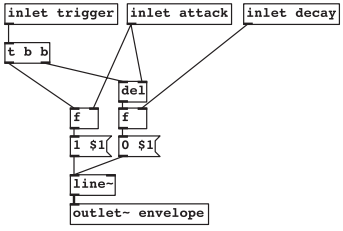


Figure 12.2
Inside the envelope subpatch.

To use `pd envelope` we supply a bang on the first inlet to trigger it, and two values for attack and decay. In figure 12.1 it modulates the output of an oscillator

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

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 one envelope generator it's a few simple steps to turn it into a MIDI mono synthesiser (shown in figure 12.3) 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.

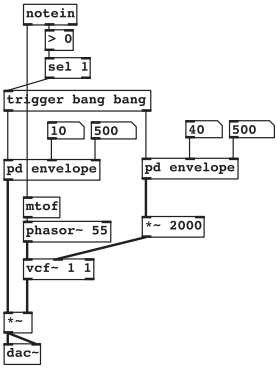


Figure 12.3
Simple mono MIDI synth made using two copies of the same envelope subpatch.

Deep Subpatches

Consider an object giving us the vector magnitude of two numbers. This is the same as the hypotenuse c of a right 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.

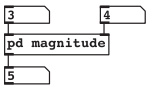


Figure 12.4
Vector magnitude.

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.

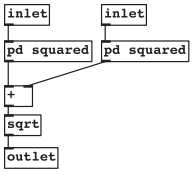


Figure 12.5
Subpatch calculates $\sqrt{a^2 + b^2}$.

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 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 figure 12.6.

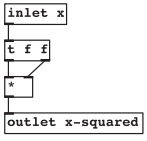


Figure 12.6
Subpatch to compute x^2.

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 figure 12.5. 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 figure 12.4 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, although 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 can have 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.

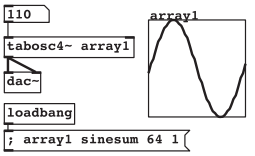


Figure 12.7
Table oscillator patch.

Consider the table oscillator patch in figure 12.7, 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 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 `tableoscillator`. 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 12.2

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 changes as shown in figure 12.8.

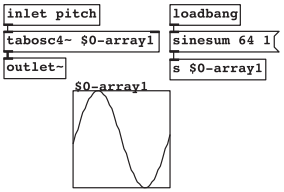


Figure 12.8
Table oscillator abstraction.

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 run time, `$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 12.3

Editing

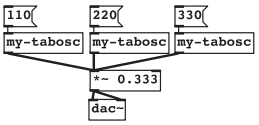


Figure 12.9
Three harmonics using the table oscillator abstraction.

Now that we have an abstracted table oscillator let's instantiate a few copies. In figure 12.9 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 12.4

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

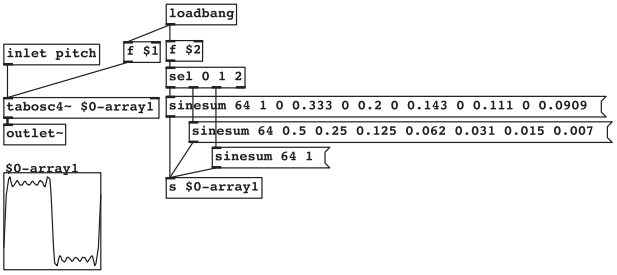


Figure 12.10
Table oscillator abstraction with initialised frequency and shape.

arguments. Let's see this in action by modifying our table oscillator to take arguments for initial frequency and waveform.

In figure 12.10 we see several interesting changes. First, 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 overridden 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.

SECTION 12.5

Defaults and States

A quick word about default parameters. Try creating some instances of the abstraction in figure 12.10 (shown as `my-tabsosc2` in figure 12.11).[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, as 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 when 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

2. The graphs with connections to them shown here, and elsewhere in the book, are abstractions that contain everything necessary 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.

(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.

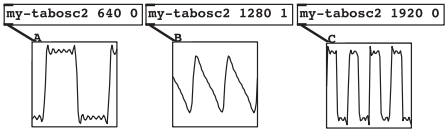


Figure 12.11
Three different waveforms and frequencies from the same table oscillator abstraction.

SECTION 12.6

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.

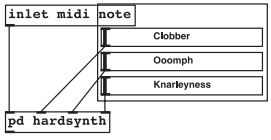


Figure 12.12
Graph on parent synth.

"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 figure 12.12 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.

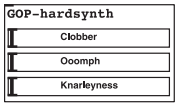


Figure 12.13
Appearance of a GOP abstraction.

Here is what the abstraction looks like when you create an instance (fig. 12.13). 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 figure 12.12 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

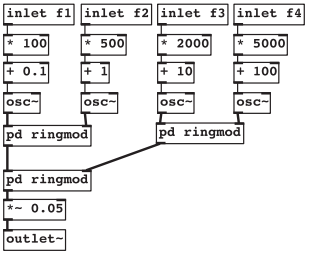


Figure 12.14
Preconditioning normalised inlets.

The patch in figure 12.14 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 preconditioners 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

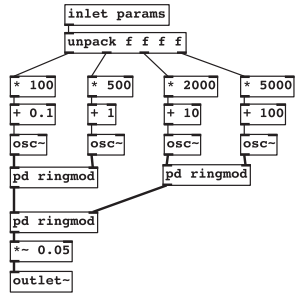
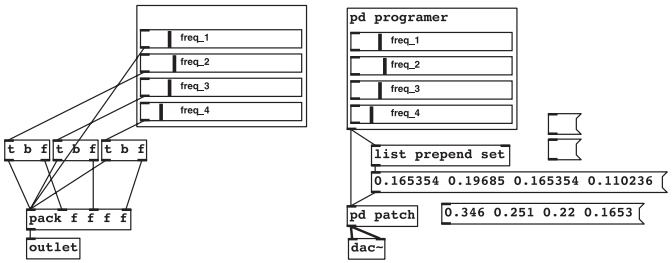


Figure 12.15
Using a list input.

What we've done here in figure 12.15 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 figure 12.16 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`

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

Figure 12.16
Packing and using parameter lists.

objects between each slider as shown in figure 12.16 (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 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 figure 12.16 (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 figure 12.17 have ranges from 0.0 to 1.0. We say the control surface is *normalised*.

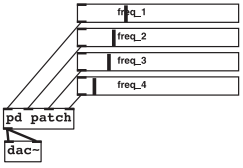


Figure 12.17
All faders are normalised 0.0 to 1.0.

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 parameters. 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. Preconditioning 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 figure 12.18) and placing a +~ object as part of the subpatch makes for easier to read patches.

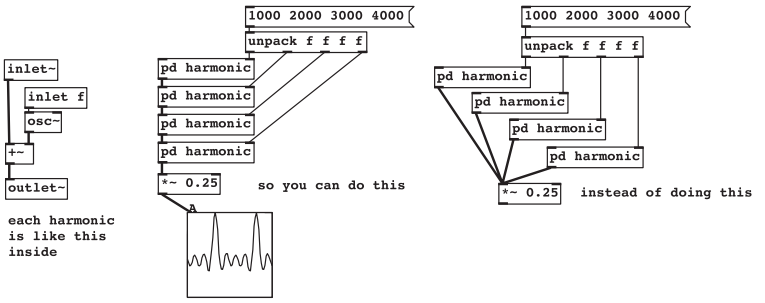


Figure 12.18
Stacking subpatches that sum with an inlet.

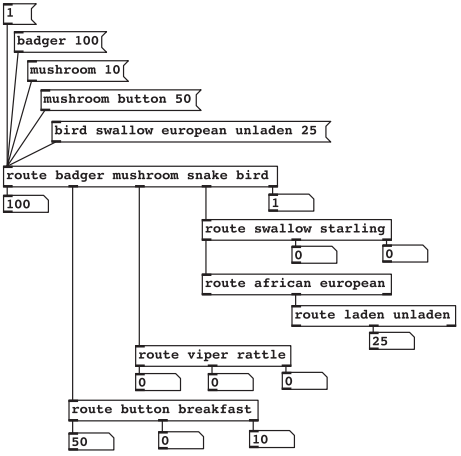


Figure 12.19
Route can channel named parameters to a destination.