10

Using Pure Data

SECTION 10.1

Basic Objects and Principles of Operation

Now that we are familiar with the basics of Pd, let's look at some essential objects and rules for connecting them together. There are about 20 message objects you should try to learn by heart because almost everything else is built from them.

Hot and Cold Inlets

Most objects operating on messages have a "hot" inlet and (optionally) one or more "cold" inlets. Messages received at the hot inlet, usually the leftmost one, will cause computation to happen and output to be generated. Messages on a cold inlet will update the internal value of an object but not cause it to output the result yet. This seems strange at first, like a bug. The reason for it is so that we can order evaluation. This means waiting for subparts of a program to finish in the right order before proceeding to the next step. From maths you know that brackets describe the order of a calculation. The result of $4 \times 10 - 3$ is not the same as $4 \times (10 - 3)$, we need to calculate the parenthesised parts first. A Pd program works the same way: you need to wait for the results from certain parts before moving on.

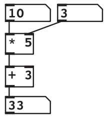


Figure 10.1
Hot and cold inlets.

In figure 10.1 a new number box is added to right inlet of `*`. This new value represents a constant multiplier k so we can compute $y = kx + 3$. It overrides the 5 given as an initial parameter when changed. In figure 10.1 it's set to 3 so we have $y = 3x + 3$. Experiment setting it to another value and then changing the left number box. Notice that changes to the right number box don't immediately affect the output, because it connects to the cold inlet of `*`, but changes to the left number box cause the output to change, because it is connected to the hot inlet of `*`.

Bad Evaluation Order

A problem arises when messages fan out from a single outlet into other operations. Look at the two patches in figure 10.2. Can you tell the difference? It is impossible to tell just by looking that one is a working patch and the other contains a nasty error. Each is an attempt to double the value of a number by connecting it to both sides of a `+`. When connections are made this way the

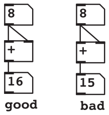


Figure 10.2
Bad ordering.

behaviour is undefined, but usually happens in the order the connections were made. The first one works because the right (cold) inlet was connected before the left (hot) one. In the second patch the arriving number is added to the *last* number received because the hot inlet is addressed first. Try making these patches by connecting the inlets to `+` in a different order. If you accidentally create errors this way they are hard to debug.

Trigger Objects

A trigger is an object that splits a message up into parts and sends them over several outlets in order. It solves the evaluation order problem by making the order explicit.

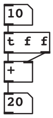


Figure 10.3
Ordering with trigger.

The order of output is right to left, so a `trigger bang float` object outputs a float on the right outlet first, then a bang on the left one. This can be abbreviated as `t b f`. Proper use of triggers ensures correct operation of units further down the connection graph. The arguments to a trigger may be **s** for symbol, **f** for float, **b** for bang, **p** for pointers, and **a** for any. The "any" type will pass lists and pointers too. The patch in figure 10.3 always works correctly, whatever order you connect to the `+` inlets. The float from the right outlet of `t f f` is always sent to the cold inlet of `+` first, and the left one to the hot inlet afterwards.

Making Cold Inlets Hot

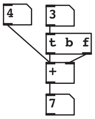


Figure 10.4
Warming an inlet.

An immediate use for our new knowledge of triggers is to make an arithmetic operator like `+` respond to either of its inlets immediately. Make the patch shown in figure 10.4 and try changing the number boxes. When the left one is changed it sends a float number message to the left (hot) inlet which updates the output as usual. But now, when you change the right number box it is split by `t b f` into two messages, a float which is sent to the cold (right) inlet of `+`, and a bang, which is sent to the hot inlet immediately afterwards. When it receives a bang on its hot inlet, `+` computes the sum of the two numbers last seen on its inlets, which gives the right result.

Float Objects

The object `f` is very common. A shorthand for `float`, which you can also use if you like to make things clearer, it holds the value of a single floating point number. You might like to think of it as a variable, a temporary place to store a number. There are two inlets on `f`; the rightmost one will set the value of the object, and the leftmost one will both set the value and/or output it depending on what message it receives. If it receives a bang message it will just output whatever value is currently stored, but if the message is a float it will override the currently stored value with a new float and immediately output that. This gives us a way to both set and query the object contents.

Int Objects

Although we have noted that integers don't really exist in Pd, not in a way that a programmer would understand, whole numbers certainly do. `int` stores a float as if it were an integer in that it provides a rounding (truncation) function of any extra decimal places. Thus 1.6789 becomes 1.0000, equal to 1, when passed to `int`.

Symbol and List Objects

As for numbers, there are likewise object boxes to store lists and symbols in a temporary location. Both work just like their numerical counterparts. A list can be given to the right inlet of `list` and recalled by banging the left inlet. Similarly `symbol` can store a single symbol until it is needed.

Merging Message Connections

When several message connections are all connected to the same inlet that's fine. The object will process each of them as they arrive, though it's up to you to ensure that they arrive in the right order to do what you expect. Be aware of race hazards when the sequence is important.

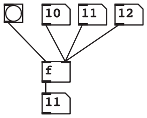


Figure 10.5
Messages to same inlet.

Messages arriving from different sources at the same hot inlet have no effect on each another; they remain separate and are simply interleaved in the order they arrive, each producing output. But be mindful that where several connections are made to a cold inlet only the last one to arrive will be relevant. Each of the number boxes in figure 10.5 connects to the same cold inlet of the float box `f` and a bang button to the hot inlet. Whenever the bang button is pressed the output will be whatever is currently stored in `f`, which will be the last number box changed. Which number box was updated last in figure 10.5? It was the middle one with a value of 11.

SECTION 10.2

Working with Time and Events

With our simple knowledge of objects we can now begin making patches that work on functions of time, the basis of all sound and music.

Metronome

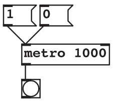


Figure 10.6
Metronome.

Perhaps the most important primitive operation is to get a beat or timebase. To get a regular series of bang events `metro` provides a clock. Tempo is given as a period in milliseconds rather than beats per minute (as is usual with most music programs).

The left inlet toggles the metronome on and off when it receives a 1 or 0, while the right one allows you to set the period. Periods that are fractions of a millisecond are allowed. The `metro` emits a bang as soon as it is switched on and the following bang occurs after the time period. In figure 10.6 the time period is 1000ms (equal to 1 second). The bang button here is used as an indicator. As soon as you click the message box to send 1 to `metro` it begins sending out bangs which make the bang button flash once per second, until you send a 0 message to turn it off.

A Counter Timebase

We could use the metronome to trigger a sound repeatedly, like a steady drum beat, but on their own a series of bang events aren't much use. Although they are separated in time we cannot keep track of time this way because bang messages contain no information.

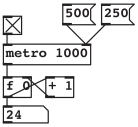


Figure 10.7
Counter.

In figure 10.7 we see the metronome again. This time the messages to start and stop it have been conveniently replaced by a toggle switch. I have also added two new messages which can change the period and thus make the metronome faster or slower. The interesting part is just below the metronome. A float box receives bang messages on its hot inlet. Its initial value is 0, so upon receiving the first bang message it outputs a float number 0 which the number box then displays. Were it not for the `+ 1` object the patch would continue outputting 0 once per beat forever. However, look closely at the wiring of these two objects: `f` and `+ 1` are connected to form an *incrementor* or *counter*. Each time `f` recieves a bang it ouputs the number currently stored to `+ 1` which adds 1 to it. This is fed back into the cold inlet of `f` which updates its value, now 1. The next time a bang arrives, 1 is output, which goes round again through `+ 1` and becomes 2. This repeats as long as bang messages arrive: each time the output increases by 1. If you start the metronome in figure 10.7 you will see the number box slowly counting up, once per second. Clicking the message boxes to change the period will make it count up faster with a 500ms delay between beats (twice per second), or still faster at 4 times per second (250ms period).

Time Objects

Three related objects help us manipulate time in the message domain. `timer` accurately measures the interval between receiving two bang messages, the first on its left inlet and the second on its right inlet. It is shown on the left in figure 10.8.

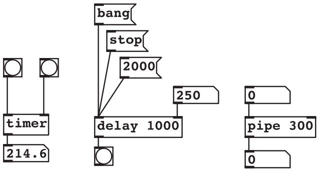


Figure 10.8
Time objects.

Clicking the first bang button will reset and start `timer` and then hitting the second one will output the time elapsed (in ms). Notice that `timer` is unusual; it's one of the few objects where the right inlet behaves as the hot control. `delay` shown in the middle of figure 10.8 will output a single bang message a certain time period after receiving a bang on its left inlet. This interval is set by its first argument or right inlet, or by the value of a float arriving at its left inlet, so there are three ways of setting the time delay. If a new bang arrives, any pending one is cancelled and a new delay is initiated. If a `stop` message arrives, then `delay` is reset and all pending events are cancelled. Sometimes we want to delay a stream of number messages by a fixed amount, which is where `pipe` comes in. This allocates a memory buffer that moves messages from its inlet to its outlet, taking a time set by its first argument or second inlet. If you change the top number box of the right patch in figure 10.8 you will see the lower number box follow it, but lagging behind by 300ms.

Select

This object outputs a bang on one of its outlets matching something in its argument list. For example, `select 2 4 6` will output a bang on its second outlet if it receives a number 4, or on its third outlet when a number 6 arrives. Messages that do not match any argument are passed through to the rightmost outlet.

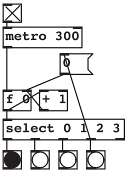


Figure 10.9
Simple sequencer.

This makes it rather easy to begin making simple sequences. The patch in figure 10.9 cycles around four steps, blinking each bang button in turn. It is a metronome running with a 300ms period and a counter. On the first step the counter holds 0, and when this is output to `select` it sends a bang to its first outlet which matches 0. As the counter increments, successive outlets of `select` produce a bang, until the fourth one is reached. When this happens a message containing 0 is triggered which feeds into the cold inlet of `f` resetting the counter to 0.

SECTION 10.3

Data Flow Control

In this section are a few common objects used to control the flow of data around patches. As you have just seen, `select` can send bang messages along a choice of connections, so it gives us a kind of selective flow.

Route

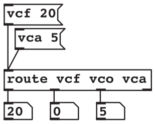


Figure 10.10
Routing values.

Route behaves in a similar fashion to select, only it operates on lists. If the first element of a list matches an argument, the remainder of the list is passed to the corresponding outlet.

So, `route badger mushroom snake` will send 20.0 to its third outlet when it receives the message {*snake 20*}. Nonmatching lists are passed unchanged to the rightmost outlet. Arguments can be numbers or symbols, but we tend to use symbols because a combination of `route` with lists is a great way to give parameters names so we don't forget what they are for.

We have a few named values in figure 10.10 for synthesiser controls. Each message box contains a two-element list, a name-value pair. When `route` encounters one that matches one of its arguments it sends it to the correct number box.

Moses

This is a "stream splitter" which sends numbers below a threshold to its left outlet, and numbers greater than or equal to the threshold to the right outlet. The threshold is set by the first argument or a value appearing on the right inlet. `moses 20` splits any incoming numbers at 20.0.

Spigot

This is a switch that can control any stream of messages including lists and symbols. A zero on the right inlet of `spigot` stops any messages on the left inlet passing to the outlet. Any non-zero number turns the spigot on.

Swap

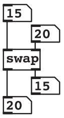


Figure 10.11
Swapping values.

It might look like a very trivial thing to do, and you may ask—why not just cross two wires? In fact `swap` is a really useful object. It just exchanges the two values on its inlets and passes them to its outlets, but it can take an argument, so it always exchanges a number with a constant. It's useful when this constant is 1 as shown later for calculating complement $1-x$ and inverse $1/x$ of a number, or where it is 100 for calculating values as a percent.

Change

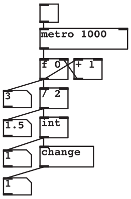


Figure 10.12
Pass values that change.

This is useful if we have a stream of numbers, perhaps from a physical controller like a joystick that is polled at regular intervals, but we only want to know values when they change. It is frequently seen preceded by `int` to denoise a jittery signal or when dividing timebases. In figure 10.12 we see a counter that has been stopped after reaching 3. The components below it are designed to divide the timebase in half. That is to say, for a sequence {*1, 2, 3, 4, 5, 6 ...*} we will get {*1, 2, 3 ...*}. There should be half as many numbers in the output during the same time interval. In other words, the output changes half as often as the input. Since the counter has just passed 3 the output of `/` is 1.5 and `int` truncates this to 1. But this is the second time we have seen 1 appear, since the same number was sent when the input was 2. Without using `change` we would get {*1, 1, 2, 2, 3, 3 ...*} as output.

Send and Receive Objects

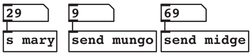


Figure 10.13
Sends.

These are very useful when patches get too visually dense, or when you are working with patches spread across many canvases. `send` and `receive` objects, abbreviated as `s` and `r`, work as named pairs. Anything that goes into the send unit is transmitted by an invisible wire and appears immediately on the receiver, so whatever goes into `send bob` reappears at `receive bob`.

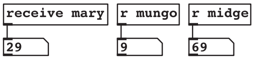


Figure 10.14
Receives.

Matching sends and receives have global names by default and can exist in different canvases loaded at the same time. So if the `receive` objects in figure 10.14 are in a different patch they will still pick up the send values from figure 10.13. The relationship is one to many, so only one send can have a particular name but can be picked up by multiple `receive` objects with the same name. In the latest versions of Pd the destination is dynamic and can be changed by a message on the right inlet.

Broadcast Messages

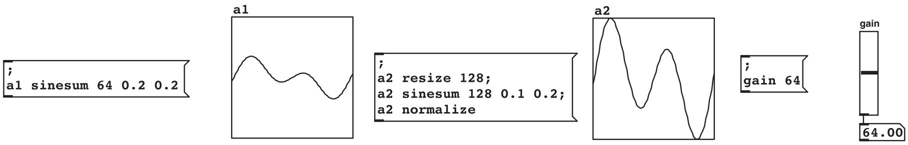
As we have just seen, there is an "invisible" environment through which messages may travel as well as through wires. A message box containing a message that begins with a semicolon is *broadcast*, and Pd will route it to any destination that matches the first symbol. This way, activating the message box `; foo 20` is the same as sending a float message with a value of 20 to the object `s foo`.

Special Message Destinations

This method can be used to address arrays with special commands, to talk to GUI elements that have a defined *receive symbol* or as an alternative way to talk to `receive` objects. If you want to change the size of arrays dynamically they recognise a special *resize* message. There is also a special destination (which always exists) called `pd` which is the audio engine. It can act on broadcast messages like `; pd dsp 1` to turn on the audio computation from a patch. Some examples are shown in figure 10.15.

Message Sequences

Several messages can be stored in the same message box as a sequence if separated by commas, so `2, 3, 4, 5` is a message box that will send four values one after another when clicked or banged. This happens instantly (in *logical time*). This is often confusing to beginners when comparing sequences to lists. When you send the contents of a message box containing a sequence all the elements are sent in one go, but as separate messages in a stream. Lists, on the other hand, which are not separated by commas, also send all the elements at the



```
;
a1 sinesum 64 0.2 0.2
```

```
;
a2 resize 128;
a2 sinesum 128 0.1 0.2;
a2 normalize
```

```
;
gain 64
```

Figure 10.15
Special message broadcasts.

same time, but as a single list message. Lists and sequences can be mixed, so a message box might contain a sequence of lists.

SECTION 10.4

List Objects and Operations

Lists can be quite an advanced topic and we could devote an entire chapter to this subject. Pd has all the capabilities of a full programming language like LISP, using only list operations, but like that language all the more complex functions are defined in terms of just a few intrinsic operations and abstractions. The *list-abs* collection by Frank Barknecht and others is available in *pd-extended*. It contains scores of advanced operations like sorting, reversing, inserting, searching, and performing conditional operations on every element of a list. Here we will look at a handful of very simple objects and leave it as an exercise to the reader to research the more advanced capabilities of lists for building sequencers and data analysis tools.

Packing and Unpacking Lists

The usual way to create and disassemble lists is to use `pack` and `unpack`. Arguments are given to each which are type identifiers, so `pack f f f f` is an object that will wrap up four floats given on its inlets into a single list. They should be presented in right-to-left order so that the hot inlet is filled last. You can also give float values directly as arguments of a `pack` object where you want them to be fixed; so `pack 1 f f 4` is legal, the first and last list elements will be 1 and 4 unless overridden by the inlets, and the two middle ones will be variable.

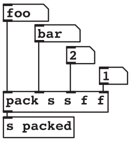


Figure 10.16
List packing.

Start by changing the right number in figure 10.16, then the one to its left, then click on the symbol boxes and type a short string before hitting RETURN. When you enter the last symbol connected to the hot inlet of `pack`, you will see the data received by figure 10.17 appear in the display boxes after it is unpacked.

The `unpack s s f f` will expect two symbols and two floats and send them to its four outlets. Items are packed and unpacked in the sequence given in the list, but in right-to-left order. That means the floats from `unpack s s f f` will appear first, starting