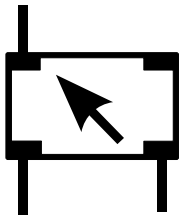
CHAPTER 3
Using Pure Data



SECTION 3.1
Basic objects and principles of operation

Now 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 is so that we can order evaluation. This means waiting for sub-parts 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.

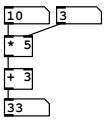


fig 3.1: Hot and cold inlets

In Fig. 3.1 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 Fig. 3.1 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 effect 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

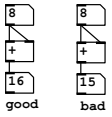


fig 3.2: Bad ordering

A problem arises when messages fan out from a single outlet into other operations. Look at the two patches in Fig. 3.2. 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 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.

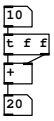


fig 3.3: Ordering with trigger

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 Fig. 3.3 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

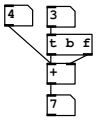


fig 3.4: Warming an inlet

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 Fig. 3.4 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.

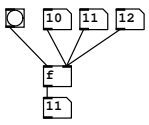


fig 3.5: Messages to same inlet

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 Fig. 3.5 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 Fig. 3.5? It was the middle one with a value of 11.

SECTION 3.2

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

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).

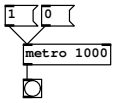


fig 3.6: Metronome

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 Fig. 3.6 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.

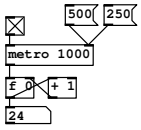


fig 3.7: Counter

In Fig. 3.7 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 Fig. 3.7 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 of Fig. 3.8.

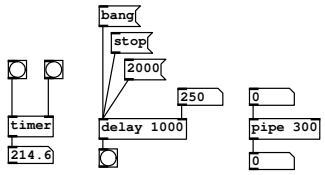


fig 3.8: Time objects

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 Fig. 3.8 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 Fig. 3.8 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.

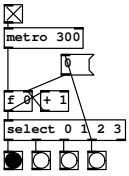


fig 3.9: Simple sequencer

This makes it rather easy to begin making simple sequences. The patch in Fig. 3.9 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 3.3

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

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.

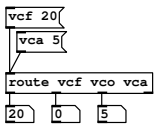


fig 3.10: Routing values

So, `route badger mushroom snake` will send 20.0 to its third outlet when it receives the message {*snake 20*}. Non matching 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 Fig. 3.10 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

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

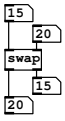


fig 3.11: Swapping values

It might look like a very trivial thing to do, and you may ask - why not just cross two wires? In fact `swap` is 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

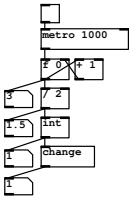


fig 3.12: Pass values that change

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 Fig. 3.12 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 `f` 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

fig 3.13: Sends

Very useful for 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.

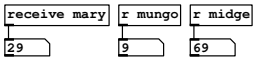


fig 3.14: Receives

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 Fig. 3.14 are in a different patch they will still pick up the send values from Fig. 3.13. 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 Fig. 3.15

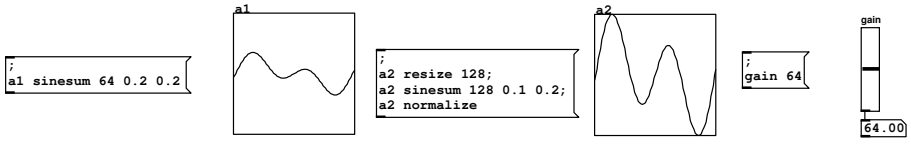


fig 3.15: Special message broadcasts

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 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 3.4

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 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 over-ridden by the inlets, and the two middle ones will be variable.

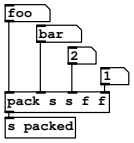


fig 3.16: List packing

Start by changing the right number in Fig. 3.16, 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 Fig. 3.17 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 with the rightmost one, then the two symbols ending on the leftmost one. Of course this happens so quickly you cannot see the ordering, but it makes sense to happen this way so that if you are unpacking data, changing it and re-packing into a list everything occurs in the right order. Note that the types of data in the list must match the arguments of each object. Unless you use the **a** (any) type Pd will complain if you try to pack or unpack a mismatched type.

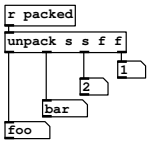


fig 3.17: List unpacking

Substitutions

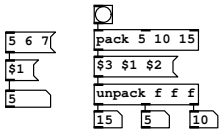


fig 3.18: Dollar substitution.

A message box can also act as a template. When an item in a message box is written $1 it behaves as an empty slot that assumes the value of the first element of a given list. Each of the dollar arguments $1, $2 and so on, are replaced by the corresponding item in the input list. The message box then sends the new message with any slots filled in. List elements can be substituted in multiple positions as seen in Fig. 3.18. The list {*5 10 15*} becomes {*15 5 10*} when put through the substitution $3 $1 $2.

Persistence

You will often want to set up a patch so it's in a certain state when loaded. It's possible to tell most GUI objects to output the last value they had when the patch was saved. You can do this by setting the `init` checkbox in the `properties` panel. But what if the data you want to keep comes from another source, like an external MIDI fader board? A useful object is loadbang which generates a bang message as soon as the patch loads.

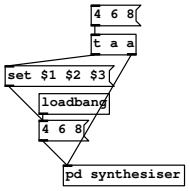


fig 3.19: Persistence using messages

You can use this in combination with a message box to initialise some values. The contents of message boxes are saved and loaded with the patch. When you need to stop working on a project but have it load the last state next time around then list data can be saved in the patch with a message box by using the special `set` prefix. If a message box receives a list prefixed by `set` it will be filled with the list, but not immediately ouput it. The arrangement in Fig. 3.19 is used to keep a 3 element list for `pd synthesiser` in a message box that will be saved with the patch, then generate it to initialise the synthesiser again when the patch is reloaded.

List distribution

An object with 2 or more message inlets will distribute a list of parameters to all inlets using only the first inlet.

fig 3.20: Distribution

The number of elements in the list must match the number of inlets and their types must be compatible. In Fig. 3.20 a message box contains a list of two numbers, 9 and 7. When a pair of values like this are sent to - with its right inlet unconnected they are spread over the two inlets, in the order they appear, thus $9 - 7 = 2$.

More advanced list operations

To concatenate two lists together we use list append. It takes two lists and creates a new one with the second list attached to the end of the first. If given an argument it will append this to every list it receives. It may be worth knowing that list is an alias for list append. You can choose to type in either in order to make it clearer what you are doing. Very similar is list prepend which does almost the same, but returns a new list with the argument or list at the second inlet concatenated to the beginning. For disassembling lists we can use list split. This takes a list on its left inlet and a number on the right inlet (or as an argument) which indicates the position to split the list. It produces two new lists, one containing elements below the split point appears on the left outlet, and the remainder of the list appears on the right. If the supplied list is shorter than the split number then the entire list is passed unchanged to the right outlet. The list trim object strips off any selector at the start leaving the raw elements.

SECTION 3.5

Input and output

There are plenty of objects in Pd for reading keyboards, mice, system timers, serial ports and USB. There's not enough room in this book to do much more than summarise them, so please refer to the Pd online documentation for your platform. Many of these are available only as external objects, but several are built into Pd core. Some depend on the platform used, for example comport and key are only available on Linux and MacOS. One of the most useful externals available is hid which is the "human interface device". With this you can connect joysticks, game controllers, dance mats, steering wheels, graphics tablets and all kinds of fun things. File IO is available using textfile and qlist objects, objects are available to make database transactions to MySQL, and of course audio file IO is simple using a range of objects like writesf~ and readsf~. MIDI files can be imported and written with similar objects. Network access is available through netsend and netreceive which offer UDP or TCP services. Open Sound Control is available using the external OSC library by Martin Peach or dumpOSC and sendOSC objects. You can even generate or open compressed audio streams using mp3cast~ and similar externals, and you can run code from other languages like python and lua. A popular hardware peripheral for use in combination with

Pd is the Arduino board which gives a number of buffered analog and digital lines, serial and parallel, for robotics and control applications. Nearly all of this is quite beyond the scope of this book. The way you set up your DAW and build your sound design studio is an individual matter, but Pd should not disappoint you when it comes to I/O connectivity. We will now look at a few common input and output channels.

The print object

Where would we be without a print object? Not much use for making sound, but vital for debugging patches. Message domain data is dumped to the console so you can see what is going on. You can give it a non-numerical argument which will prefix any output and make it easier to find in a long printout.

MIDI

When working with musical keyboards there are objects to help integrate these devices so you can build patches with traditional synthesiser and sampler behaviours. For sound design this is great for attaching MIDI fader boards to control parameters, and of course musical interface devices like breath controllers and MIDI guitars can be used. Hook up any MIDI source to Pd by activating a MIDI device from the `Media->MIDI` menu (you can check this is working from `Media->Test Audio and MIDI`).

Notes in

You can create single events to trigger from individual keys, or have layers and velocity fades by adding extra logic.

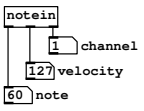


fig 3.21: MIDI note in

The notein object produces note number, velocity and channel values on its left, middle and right outlets. You may assign an object to listen to only one channel by giving it an argument from 1 to 15. Remember that note-off messages are equivalent to a note-on with zero velocity in many MIDI implementations and Pd follows this method. You therefore need to add extra logic before connecting an oscillator or sample player to notein so that zero valued MIDI notes are not played.

Notes out

Another object noteout sends MIDI to external devices. The first, second and third inlets set note number, velocity and channel respectively. The channel is 1 by default. Make sure you have something connected that can play back MIDI and set the patch shown in Fig. 3.22 running with its toggle switch. Every 200ms it produces a `C` on a random octave with a random velocity value between 0 and 127. Without further ado these could be sent to noteout, but it would cause each MIDI note to "hang", since we never send a note-off message. To properly construct MIDI notes you need makenote

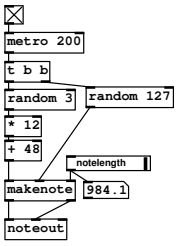


fig 3.22: MIDI note generation

which takes a note-number and velocity, and a duration (in milliseconds) as its third argument. After the duration has expired it automatically adds a note-off. If more than one physical MIDI port is enabled then `noteout` sends channels 1 to 16 to port 1 and channels 17 to 32 to port 2 etc.

Continuous controllers

Two MIDI input/output objects are provided to receive and send continuous controllers, `ctlin` and `ctlout`. Their three connections provide, or let you set, the controller value, controller number and MIDI channel. They can be instantiated with arguments, so `ctlin 10 1` picks up controller 10 (pan position) on MIDI channel 1.

MIDI to Frequency

Two numerical conversion utilities are provided to convert between MIDI note numbers and Hz. To get from MIDI to Hz use `mtof`. To convert a frequency in Hz to a MIDI note number use `ftom`.

Other MIDI objects

For pitchbend, program changes, system exclusive, aftertouch and other MIDI functions you may use any of the objects summarised in Tbl. 3.23. System exclusive messages may be sent by hand crafting raw MIDI bytes and outputting via the `midiout` object. Most follow the inlet and outlet template of `notein` and `noteout` having a channel as the last argument, except for `midiin` and `sysexin` which receive omni (all channels) data.

MIDI in object		MIDI out object	
Object	Function	Object	Function
`notein`	Get note data	`noteout`	Send note data.
`bendin`	Get pitchbend data −63 to +64	`bendout`	Send pitchbend data −64 to +64.
`pgmin`	Get program changes.	`pgmout`	Send program changes.
`ctlin`	Get continuous controller messages.	`ctlout`	Send continuous controller messages.
`touchin`	Get channel aftertouch data.	`touchout`	Send channel aftertouch data.
`polytouchin`	Polyphonic touch data in	`polytouchout`	Polyphonic touch output
`polytouchin`	Send polyphonic aftertouch.	`polytouchin`	Get polyphonic aftertouch.
`midiin`	Get unformatted raw MIDI	`midiout`	Send raw MIDI to device.
`sysexin`	Get system exclusive data	No output counterpart	Use `midiout` object

fig 3.23: List of MIDI objects

SECTION 3.6

Working with numbers

Arithmetic objects

Objects that operate on ordinary numbers to provide basic maths functions are summarised in Tbl. 3.24 All have hot left and cold right inlets and all take one argument that initialises the value otherwise received on the right inlet. Note the difference between arithmetic division with / and the div object. The modulo operator gives the remainder of dividing the left number by the right.

Object	Function
+	Add two floating point numbers
-	Subtract number on right inlet from number on left inlet
/	Divide lefthand number by number on right inlet
*	Multiply two floating point numbers
div	Integer divide, how many times the number on the right inlet divides exactly into the number on the left inlet
mod	Modulo, the smallest remainder of dividing the left number into any integer multiple of the right number

fig 3.24: Table of message arithmetic operators

Trigonometric maths objects

A summary of higher maths functions is given in Tbl. 3.25.

Random numbers

A useful ability is to make random numbers. The random object gives integers over the range given by its argument including zero, so random 10 gives 10 possible values from 0 to 9.

Arithmetic example

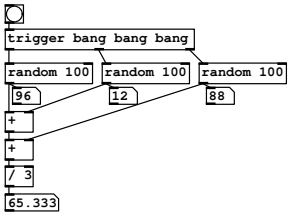


fig 3.26: Mean of three random floats

An example is given in Fig. 3.26 to show correct ordering in a patch to calculate the mean of three random numbers. We don't have to make every inlet hot, just ensure that everything arrives in the correct sequence by triggering the random objects properly. The first random (on the right) supplies the cold inlet of the lower +, the middle one to the cold inlet of the upper +. When the final (left) random is generated it passes to the hot inlet of the first +, which computes the sum and passes it to the second + hot inlet. Finally we divide by 3 to get the mean value.

Object	Function
cos	The cosine of a number given in radians. Domain: $-\pi/2$ to $+\pi/2$. Range: -1.0 to $+1.0$.
sin	The sine of a number in radians, domain $-\pi/2$ to $+\pi/2$, range -1.0 to $+1.0$
tan	Tangent of number given in radians. Range: 0.0 to ∞ at $\pm\pi/2$
atan	Arctangent of any number in domain $\pm\infty$ Range: $\pm\pi/2$
atan2	Arctangent of the quotient of two numbers in Cartesian plane. Domain: any floats representing X, Y pair. Range: angle in radians $\pm\pi$
exp	Exponential function e^x for any number. Range 0.0 to ∞
log	Natural log (base e) of any number. Domain: 0.0 to ∞. Range: $\pm\infty$ ($-\infty$ is -1000.0)
abs	Absolute value of any number. Domain $\pm\infty$. Range 0.0 to ∞
sqrt	The square root of any positive number. Domain 0.0 to ∞
pow	Exponentiate the left inlet to the power of the right inlet. Domain: positive left values only.

fig 3.25: Table of message trigonometric and higher math operators

Comparative objects

In Tbl. 3.27 you can see a summary of comparative objects. Output is either 1 or 0 depending on whether the comparison is true or false. All have hot left inlets and cold right inlets and can take an argument to initialise the righthand value.

Object	Function
>	True if the number at the left inlet is greater than the right inlet.
<	True if the number at the left inlet is less than the right inlet.
>=	True if the number at the left inlet is greater than or equal to the right inlet.
<=	True if the number at the left inlet is less than or equal to the right inlet.
==	True if the number at the left inlet is equal to the right inlet.
!=	True if the number at the left inlet is not equal to the right inlet

fig 3.27: List of comparative operators

Boolean logical objects

There are a whole bunch of logical objects in Pd including bitwise operations that work exactly like C code. Most of them aren't of much interest to us in this book, but we will mention the two important ones || and &&. The output of ||, logical OR, is true if either of its inputs are true. The output of &&, logical AND, is true only when both its inputs are true. In Pd any non-zero number is "true", so the logical inverter or "not" function is unnecessary because there are many ways of achieving this using other objects. For example, you can make a logical inverter by using != with 1 as its argument.

SECTION 3.7

Common idioms

There are design patterns that crop up frequently in all types of programming. Later we will look at abstraction and how to encapsulate code into new objects so you don't find yourself writing the same thing again and again. Here I will introduce a few very common patterns.

Constrained counting

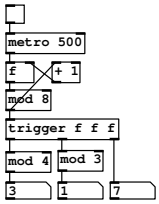


fig 3.28: Constrained counter.

We have already seen how to make a counter by repeatedly incrementing the value stored in a float box. To turn an increasing or decreasing counter into a cycle for repeated sequences there is an easier way than resetting the counter when it matches an upper limit, we wrap the numbers using mod. By inserting mod into the feedback path before the increment we can ensure the counter stays bounded. Further mod units can be added to the number stream to generate polyrhythmic sequences. You will frequently see variations on the idiom shown in Fig. 3.28. This is the way we produce multi-rate timebases for musical sequencers, rolling objects or machine sounds that have complex repetitive patterns.

Accumulator

fig 3.29: Accumulator.

A similar construct to a counter is the accumulator or integrator. This reverses the positions of f and + to create an integrator that stores the sum of all previous number messages sent to it. Such an arrangement is useful for turning "up and down" messages from an input controller into a position. Whether to use a counter or accumulator is a subtle choice. Although you can change the increment step of the counter by placing a new value on the right inlet of + it will not take effect until the previous value in f has been used. An accumulator on the other hand can be made to jump different intervals immediately by the value sent to it. Note the important difference, an accumulator takes floats as an input while a counter takes bang messages.

Rounding

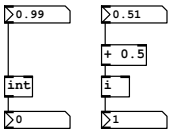


fig 3.30: Rounding

An integer function, int, also abbreviated i gives the whole part of a floating point number. This is a *truncation*, which just throws away any decimal digits. For positive numbers it gives the *floor* function, written $\lfloor x \rfloor$ which is the integer *less than or equal to* the input value. But take note of what happens for *negative* values, applying int to -3.4 will give 3.0, an integer *greater than or equal to* the input. Truncation is shown on the left of Fig. 3.30. To get a regular rounding for positive numbers, to pick the *closest* integer, use the method shown on the right side of Fig. 3.30. This will return 1 for an input of 0.5 or more and 0 for an input of 0.49999999 or less.

Scaling

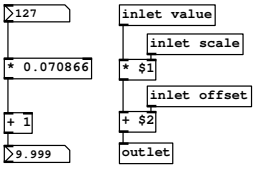


fig 3.31: Scaling

This is such a common idiom you will see it almost everywhere. Given a range of values such as 0 to 127 we may wish to map this onto another set of values, the domain, such as 1 to 10. This is the same as changing the slope and zero intersect of a line following $y = mx + c$. To work out the values you first obtain the bottom value or *offset*, in this case +1. Then a *multiplier* value is needed to scale for the upper value, which given an input of 127 would satisfy $10 = 1 + 127x$, so moving the offset we get $9 = 127x$, and dividing by 127 we get $x = 9/127$ or $x = 0.070866$. You can make a subpatch or an abstraction for this as shown in Fig. 6.1, but since only two objects are used it's more sensible to do scaling and offset as you need it.

Looping with until

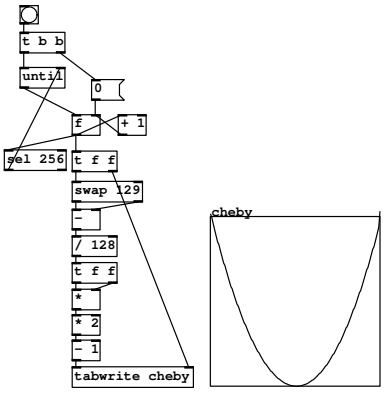


fig 3.32: Using until

Unfortunately, because it must be designed this way, until has the potential to cause a complete system lock-up. Be very careful to understand what you are doing with this. A bang message on the left inlet of until will set it producing bang messages as fast as the system can handle! These do not stop *until* a bang message is received on the right inlet. Its purpose is to behave as a fast loop construct performing message domain computation quickly. This way you can fill an entire wavetable or calculate a complex formula in the time it takes to process a single audio block. Always make sure the right inlet is connected to a valid terminating condition. In Fig. 3.32 you can see an example that computes the second Chebyshev polynomial according

to $y = 2x^2 - 1$ for the range -1.0 to $+1.0$ and fills a 256 step table with the result. As soon as the bang button is pressed a counter is reset to zero and then until begins sending out bangs. These cause the counter to rapidly increment until select matches 256 whereupon a bang is sent to the right inlet of until stopping the process. All this will happen in a fraction of a millisecond. Meanwhile we use the counter output to calculate a Chebyshev curve and put it into the table.

fig 3.33: for 256

A safer way to use until is shown in Fig. 3.33. If you know in advance that you want to perform a fixed number of operations then use it like a **for loop**. In this case you pass a non-zero float to the left inlet. There is no terminating condition, it stops when the specified number of bangs has been sent, 256 bangs in the example shown.

Message complement and inverse

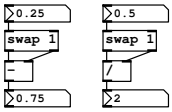


fig 3.34: Message reciprocal and inverse

Here is how we obtain the number that is $1 - x$ for any x. The *complement* of x is useful when you want to balance two numbers so they add up to a constant value, such as in panning. The swap object exchanges its inlet values, or any left inlet value with its first argument. Therefore, what happens with the left example of Fig. 3.34 is the - calculates $1 - x$, which for an input of 0.25 gives 0.75. Similarly the inverse of a float message $1/x$ can be calculated by replacing the - with a /.

Random selection

To choose one of several events at random a combination of random and select will generate a bang message on the select outlet corresponding to one of its arguments. With an initial argument of 4 random produces a *range* of 4 random integer numbers starting at 0, so we use select 0 1 2 3 to select amongst them. Each has an equal probability, so every outlet will be triggered 25% of the time on average.

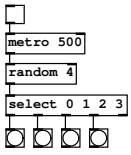


fig 3.35: Random select.

Weighted random selection

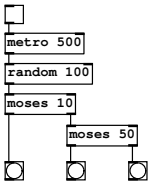


fig 3.36: Weighted random select.

A simple way to get a bunch of events with a certain probability distribution is to generate uniformly distributed numbers and stream them with moses. For example moses 10 sends integers greater than 9.0 to its right outlet. A cascade of moses objects will distribute them in a ratio over the combined outlets when the sum of all ratios equals the range of random numbers. The outlets of moses 10 distribute the numbers in the ratio 1 : 9. When the right outlet is further split by moses 50 as in Fig. 3.36 numbers in the range 0.0 to 100.0 are split in the ratio 10 : 40 : 50, and

since the distribution of input numbers is uniform they are sent to one of three outlets with 10%, 40% and 50% probability.

Delay cascade

Sometimes we want a quick succession of bangs in a certain fixed timing pattern. An easy way to do this is to cascade `delay` objects. Each `delay 100` in Fig. 3.37 adds a delay of 100 milliseconds. Notice the abbrieved form of the object name is used.

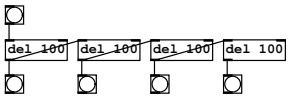


fig 3.37: Delay cascade.

Last float and averages

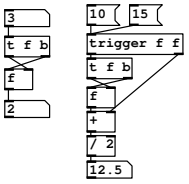


fig 3.38: Last value and averaging

If you have a stream of float values and want to keep the previous value to compare to the current one then the idiom shown on the left of Fig. 3.38 will do the job. Notice how a trigger is employed to first bang the *last* value stored in the float box and then update it with the current value via the right inlet. This can be turned into a simple "lowpass" or averaging filter for float messages as shown on the right of Fig. 3.38. If you add the previous value to the current one and divide by two you obtain the average. In the example shown the values were 10 followed by 15, resulting in $(10 + 15)/2 = 12.5$.

Running maximum (or minimum)

Giving `max` a very small argument and connecting whatever passes through it back to its right inlet gives us a way to keep track of the largest value. In Fig. 3.39 the greatest past value in the stream has been 35. Giving a very large argument to `min` provides the opposite behaviour for tracking a lowest value. If you need to reset the maximum or minimum tracker just send a very large or small float value to the cold inlet to start again.

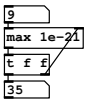


fig 3.39: Biggest so far

Float lowpass

Using only `*` and `+` as shown in Fig. 3.40 we can low pass filter a stream of float values. This is useful to smooth data from an external controller where values are occasionally anomalous. It follows the filter equation $y_n = Ax_n + Bx_{n-1}$. The strength of the filter is set by the ratio $A : B$. Both A and B should be between 0.0 and 1.0 and add up to 1.0. Note that this method will not converge on the exact input value, so you might like to follow it with `int` if you need numbers rounded to integer values.

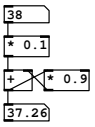
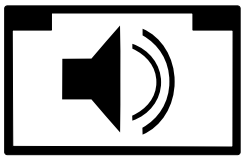


fig 3.40: Low pass for floats

CHAPTER 4
Pure Data Audio



SECTION 4.1

Audio objects

We have looked at Pd in enough detail now to move on to the next level. You have a basic grasp of dataflow programming and know how to make patches that process numbers and symbols. But why has no mention been made of audio yet? Surely it is the main purpose of our study? The reason for this is that audio signal processing is a little more complex in Pd than the numbers and symbols we have so far considered, so I wanted to leave this until now.

Audio connections

I already mentioned that there two kinds of objects and data for messages and signals. Corresponding to these there are two kinds of connections, audio connections and message connections. There is no need to do anything special to make the right kind of connection. When you connect two objects together Pd will work out what type of outlet you are attempting to connect to what kind of inlet and create the appropriate connection. If you try to connect an audio signal to a message inlet, then Pd will not let you, or it will complain if there is allowable but ambiguous connection. Audio objects always have a name ending with a tilde (∼) and the connections between them look fatter than ordinary message connections.

Blocks

The signal data travelling down audio cords is made of *samples*, single floating point values in a sequence that forms an audio signal. Samples are grouped together in *blocks*.

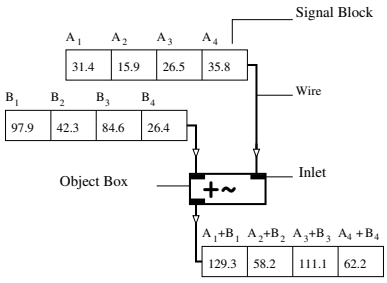


fig 4.1: Object processing data.

A block, sometimes also called a *vector*, typically has 64 samples inside it, but you can change this in certain circumstances. Objects operating on signal blocks behave like ordinary message objects, they can add, subtract, delay or store blocks of data, but do so by processing one whole block at a time. In Fig. 4.1 streams of blocks are fed to the two inlets. Blocks appearing at the outlet have values which are the sum of the corresponding values in the two input blocks. Because they process signals made of blocks, audio objects do a lot more work than objects that process messages.

Audio object CPU use

All the message objects we looked at in the last chapters only use CPU when event driven dataflow occurs, so most of the time they sit idle and consume no resources. Many of the boxes we put on our sound design canvases will be audio objects, so it's worth noting that they use up some CPU power just being idle. Whenever `compute audio` is switched on they are processing a constant stream of signal blocks, even if the blocks only contain zeros. Unlike messages which are processed in logical time, signals are processed synchronously with the soundcard sample rate. This *real-time* constraint means glitches will occur unless every signal object in the patch can be computed before the next block is sent out. Pd will not simply give up when this happens, it will struggle along trying to maintain real-time processing, so you need to listen carefully, as you hit the CPU limit of the computer you may hear crackles or pops. It is also worth knowing how audio computation relates to messages computation. Messages operations are executed at the beginning of each pass of audio block processing, so a patch where audio depends on message operations which don't complete in time will also fail to produce correct output.

SECTION 4.2

Audio objects and principles

There are a few ways that audio objects differ from message objects so let's look at those rules now before starting to create sounds.

Fanout and merging

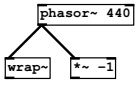


fig 4.2: Signal fanout is Okay.

You can connect the same signal outlet to as many other audio signal inlets as you like, and blocks are sent in an order which corresponds to the creation of the connections, much like message connections. But unlike messages, most of the time this will have no effect whatsoever, so you can treat audio signals that fan out as if they were perfect simultaneous copies. Very seldom you may meet rare and interesting problems, especially with delays and feedback, that can be fixed by reordering audio signals (see Chapter 7 of Puckette, "Theory and technique" regarding time shifts and block delays).

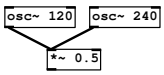


fig 4.3: Merging signals is Okay.

When several signal connections all come into the same signal inlet that's also fine. In this case they are implicitly summed, so you may need to scale your signal to reduce its range again at the output of the object. You can connect as many signals to the same inlet as you like, but sometimes it makes a patch easier to understand if you explicitly sum them with a +~ unit.

Time and resolution

Time is measured in seconds, milliseconds (one thousandth of a second, written 1ms) or samples. Most Pd times are in ms. Where time is measured in

samples this depends on the sampling rate of the program or the sound card of the computer system on which it runs. The current sample rate is returned by the samplerate~ object. Typically a sample is 1/44100th of a second and is the smallest unit of time that can be measured as a signal. But the time resolution also depends on the object doing the computation. For example metro and vline~ are able to deal in fractions of a millisecond, even less than one sample. Timing irregularities can occur where some objects are only accurate to one block boundary and some are not.

Audio signal block to messages

To see the contents of a signal block we can take a snapshot or an average. The env~ object provides the RMS value of one block of audio data scaled 0 to 100 in dB, while snapshot~ gives the instantaneous value of the last sample in the previous block. To view an entire block for debugging print~ can be used. It accepts an audio signal and a bang message on the same inlet and prints the current audio block contents when banged.

Sending and receiving audio signals

Audio equivalents of send and receive are written send~ and receive~, with shortened forms s~ and r~ . Unlike message sends only one audio send can exist with a given name. If you want to create a signal bus with many to one connectivity use throw~ and catch~ instead. Within subpatches and abstractions we use the signal objects inlet~ and outlet~ to create inlets and outlets.

Audio generators

Only a few objects are signal sources. The most important and simple one is the phasor~. This outputs an asymmetrical periodic ramp wave and is used at the heart of many other digital oscillators we are going to make. Its left inlet specifies the frequency in Hz, and its right inlet sets the phase, between 0.0 and 1.0. The first and only argument is for frequency, so a typical instance of a phasor looks like phasor~ 110. For sinusoidal waveforms we can use osc~ . Again, frequency and phase are set by the left and right inlets, or frequency is set by the creation parameter. A sinusoidal oscillator at concert A pitch is defined by osc~ 440. White noise is another commonly used source in sound design. The noise generator in Pd is simply noise~ and has no creation arguments. Its output is in the range −1.0 to 1.0. Looped waveforms stored in an array can be used to implement *wavetable* synthesis using the tabosc4~ object. This is a 4 point interpolating table ocillator and requires an array that is a power of 2, plus 3 (eg. 0 to 258) in order to work properly. It can be instantiated like phasor~ or osc~ with a frequency argument. A table oscillator running at 3kHz is shown in Fig. 4.4. It takes the waveform stored in array `A` and loops around this at the frequency given by its argument or left inlet value. To make sound samplers we need to read and write audio data from an array. The index to tabread~ and its interpolating friend tabread4~ is a sample number, so you need to supply a signal with the correct slope and magnitude to get the proper playback rate. You can use the special `set` message to reassign tabread4~ to read from another

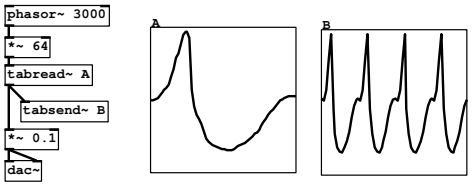


fig 4.4: Table oscillator

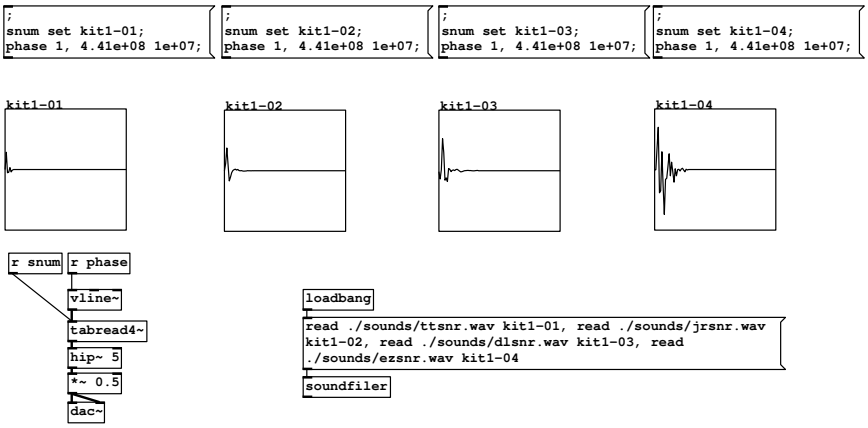


fig 4.5: Sample replay from arrays

array. The message boxes in Fig. 4.5 allow a single object to play back from more than one sample table. First the target array is given via a message to `snum`, and then a message is sent to `phase` which sets `vline~` moving up at 44100 samples per second. The arrays are initially loaded, using a multi-part message, from a `sounds` folder in the current patch directory.

Audio line objects

For signal rate control data the `line~` object is useful. It is generally programmed with a sequence of lists. Each list consists of a pair of numbers, the first being a level to move to and the second number is the time in milliseconds to take getting there. The range is usually between 1.0 and 0.0 when used as an audio control signal, but it can be any value such as when using `line~` to index a table. A more versatile line object is called `vline~`, which we will meet in much more detail later. Amongst its advantages are very accurate sub-millisecond timing and the ability to read multi-segment lists in one go and to delay stages of movement. Both these objects are essential for constructing envelope generators and other control signals.

Audio input and output

Audio IO is achieved with the adc~ and dac~ objects. By default these offer two inlets or outlets for stereo operation, but you can request as many additional sound channels as your sound system will handle by giving them numerical arguments.

Example: A simple MIDI monosynth

Using the objects we've just discussed let's create a little MIDI keyboard controlled music synthesiser as shown in Fig. 4.6. Numbers appearing at the left outlet of notein control the frequency of an oscillator. MIDI numbers are converted to a Hertz frequency by mtof. The MIDI standard, or rather general adherence to it, is a bit woolly by allowing note-off to also be a note-on with a velocity of zero. Pd follows this definition, so when a key is released it produces a note with a zero velocity. For this simple example we remove it with stripnote, which only passes note-on messages when their velocity is greater than zero. The velocity value, ranging between 1 and 127 is scaled to between 0 and 1 in order to provide a rudimentary amplitude control.

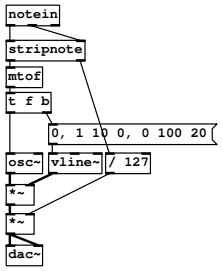


fig 4.6: MIDI note control

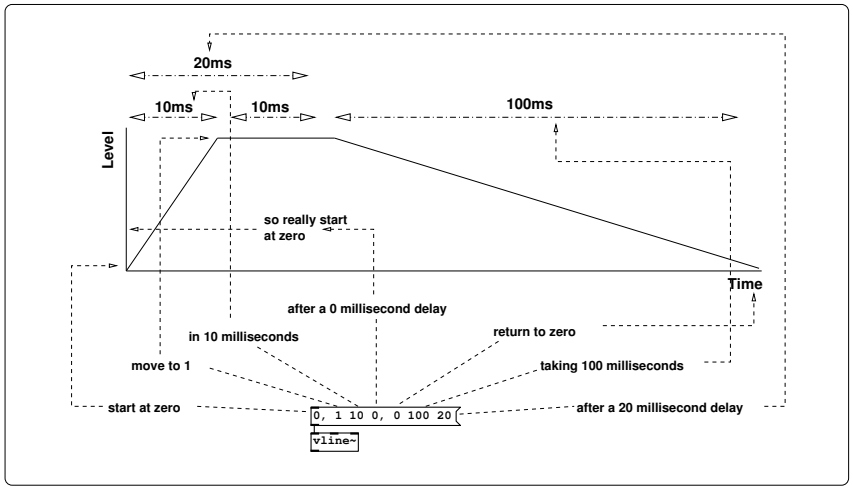


fig 4.7: Anatomy of vline message

So, here's a great place to elaborate on the anatomy of the message used to control vline~ as shown in Fig. 4.7. The syntax makes perfect sense, but sometimes it's hard to visualise without practice. The general form has three numbers per list. It says: *"go to some value"*, given by the first number, then

"take a certain time to get there", which is the second number in each list. The last number in the list is a time to wait before executing the command, so it adds an extra *wait for a time before doing it"*. What makes vline~ cool is you can send a sequence of list messages in any order, and so long as they make temporal sense then vline~ will execute them all. This means you can make very complex control envelopes. Any missing arguments in a list are dropped in right to left order, so a valid exception is seen in the first element of Fig. 4.7 where a single 0 means *"jump immediately to zero"* (don't bother to wait or take any time getting there).

Audio filter objects

Six or seven filters are used in this book. We will not look at them in much detail until we need to because there is a lot to say about their usage in each case. Simple one pole and one zero real filters are given by rpole~ and rzero~. Complex one pole and one zero filters are cpole~ and czero~. A static biquad filter biquad~ also comes with a selection of helper objects to calculate coefficients for common configurations and lop~, hip~, and bp~ 1 provide the standard low, high and bandpass responses. These are easy to use and allow message rate control of their cutoff frequencies and, in the case of bandpass, resonance. The first and only argument of the low and high pass filters is frequency, so typical instances may look like lop~ 500 and hip~ 500. Bandpass takes a second parameter for resonance like this bp~ 100 3. Fast signal rate control of cutoff is possible using the versatile vcf~ "voltage controlled filter". Its first argument is cutoff frequency and its second argument is resonance, so you might use it like vcf~ 100 2. With high resonances this provides a sharp filter that can give narrow bands. An even more colourful filter for use in music synthesiser designs is available as an external called moog~, which provides a classic design that can self oscillate.

Audio arithmetic objects

Audio signal objects for simple arithmetic are summarised in Tbl. 4.8.

Object	Function
+~	Add two signals (either input will also accept a message)
-~	Subtract righthand signal from lefthand signal
/~	Divide lefthand signal by right signal
*~	Signal multiplication
wrap~	Signal wrap, constrain any signal between 0.0 and 1.0

fig 4.8: List of arithmetic operators

Trigonometric and math objects

A summary of higher maths functions is given in Tbl. 4.9. Some signal units are abstractions defined in terms of more elementary intrinsic objects and those marked * are only available through external libraries in some Pd versions.

Object	Function
cos~	Signal version of cosine function. Domain: −1.0 to + 1.0. Note the input domain is "rotation normalised"
sin~	Not intrinsic but defined in terms of signal cosine by subtracting 0.25 from the input.
atan~ *	Signal version of arctangent with normalised range.
log~	Signal version of natural log.
abs~ *	Signal version of abs.
sqrt~	A square root for signals.
q8_sqrt~	A fast square root with less accuracy.
pow~	Signal version of power function.

fig 4.9: List of trig and higher math operators

Audio delay objects

Delaying an audio signal requires us to create a memory buffer using delwrite~. Two arguments must be supplied at creation time, a unique name for the memory buffer and a maximum size in milliseconds. For example, delwrite~ mydelay 500 creates a named delay buffer "mydelay" of size 500ms. This object can now be used to write audio data to the delay buffer through its left inlet. Getting delayed signals back from a buffer needs delread~. The only argument needed is the name of a buffer to read from, so delread~ mydelay will listen to the contents of `mydelay`. The delay time is set by a second argument, or by the left inlet. It can range from zero to the maximum buffer size. Setting a delay time larger than the buffer results in a delay of the maximum size. It is not possible to alter the maximum size of a delwrite~ buffer once created. But it is possible to change the delay time of delread~ for chorus and other effects. This often results in clicks and pops [1] so we have a vd~ variable-delay object. Instead of moving the read point vd~ changes the rate it reads the buffer, so we get tape echo and Doppler shift type effects. Using vd~ is as easy as before, create an object that reads from a named buffer like vd~ mydelay. The left inlet (or argument following the name) sets the delay time.

[1] Hearing clicks when moving a delay read point is normal, not a bug. There is no reason to assume that wavforms will align nicely once we jump to a new location in the buffer. An advanced solution crossfades between more than one buffer.