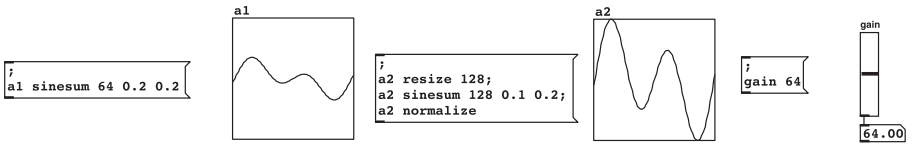

Figure 10.15
Special message broadcasts.

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.

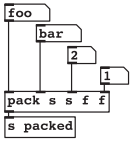

Figure 10.16
List packing.

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

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 repacking 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.

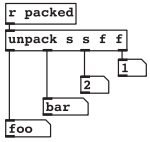

Figure 10.17
List unpacking.

Substitutions

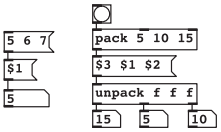

Figure 10.18
Dollar substitution.

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 figure 10.18. 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.

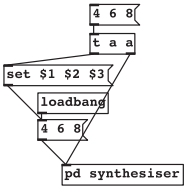

Figure 10.19
Persistence using messages.

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 will not immediately ouput it. The arrangement in figure 10.19 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.

Figure 10.20 Distribution.

The number of elements in the list must match the number of inlets and their types must be compatible. In figure 10.20 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 thing, 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 10.5

Input and Output

There are plenty of objects in Pd for reading keyboards, mice, system timers, serial ports, and USBs. 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 the 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~] (by Yves Degoyon) 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 nonnumerical 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 that 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.

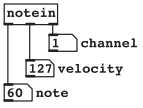

Figure 10.21
MIDI note in.

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 figure 10.22 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 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.

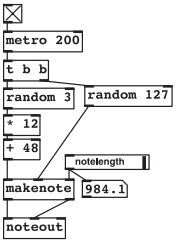

Figure 10.22
MIDI note generation.

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 figure 10.23. 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
`midiin`	Get unformatted raw MIDI	`midiout`	Send raw MIDI to device.
`sysexin`	Get system exclusive data	No output counterpart	Use `midiout` object

Figure 10.23
List of MIDI objects.