9
Starting with Pure Data

Pure Data

Pure Data is a visual signal programming language which makes it easy to construct programs to operate on signals. We are going to use it extensively in this textbook as a tool for sound design. The program is in active development and improving all the time. It is a free alternative to Max/MSP that many see as an improvement.

The primary application of Pure Data is processing sound, which is what it was designed for. However, it has grown into a general-purpose signal-processing environment with many other uses. Collections of video-processing externals exist called Gem, PDP, and Gridflow, which can be used to create 3D scenes and manipulate 2D images. It has a great collection of interfacing objects, so you can easily attach joysticks, sensors, and motors to prototype robotics or make interactive media installations. It is also a wonderful teaching tool for audio signal processing. Its economy of visual expression is a blessing: it doesn't look too fancy, which makes looking at complex programs much easier on the eye. There is a very powerful idea behind "The diagram is the program." Each patch contains its complete state visually so you can reproduce any example just from the diagram. That makes it a visual description of sound.

The question is often asked, "Is Pure Data a programming language?" The answer is yes; in fact, it is a Turing complete language capable of doing anything that can be expressed algorithmically, but there are tasks such as building text applications or websites that Pure Data is ill suited to. It is a specialised programming language that does the job it was designed for very well, namely processing signals. It is like many other GUI frameworks or DSP environments which operate inside a "canned loop"[1] and are not truly open programming languages. There is a limited concept of iteration, programmatic branching, and conditional behaviour. At heart dataflow programming is very simple. If you understand object oriented programming, think of the objects as having methods which are called by data, and can only return data. Behind the scenes Pure Data is quite sophisticated. To make signal programming simple it hides

---

1. A *canned loop* is used to refer to languages in which the real low-level programmatic flow is handled by an interpreter that the user is unaware of.

away behaviour like deallocation of deleted objects and manages the execution
graph of a multirate DSP object interpreter and scheduler.

Installing and Running Pure Data

Grab the latest version for your computer platform by searching the Internet
for it. There are versions available for Mac, Windows and Linux systems. On
Debian-based Linux systems you can easily install it by typing:

```
$ apt-get install puredata
```

Ubuntu and RedHat users will find the appropriate installer in their package
management systems, and MacOSX or Windows users will find an installer
program online. Try to use the most up-to-date version with libraries. The
pd-extended build includes extra libraries so you don't need to install them
separately. When you run it you should see a console window that looks some-
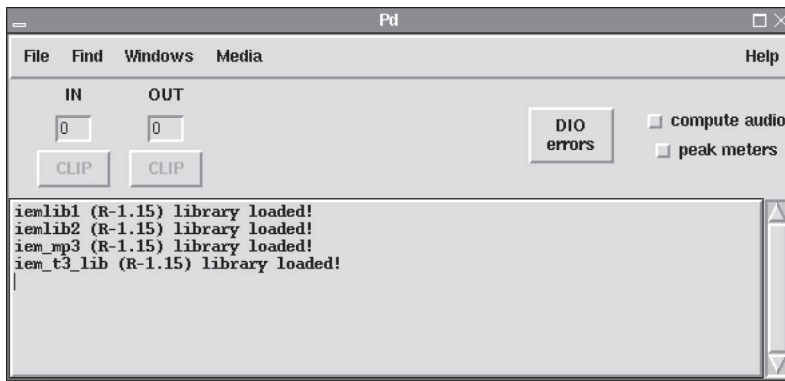thing like figure 9.1.

Figure 9.1
Pure Data console.

Testing Pure Data

The first thing to do is turn on the audio and test it. Start by entering the
`Media` menu on the top bar and select `Audio ON` (or either check the `compute`
`audio` box in the console window, or press `CTRL+/` on the keyboard.) From the
`Media→Test-Audio-and-MIDI` menu, turn on the test signal (fig. 9.2). You
should hear a clear tone through your speakers, quiet when set to –40.0dB and
much louder when set to –20.0dB. When you are satisfied that Pure Data is
making sound, close the test window and continue reading. If you don't hear
a sound you may need to choose the correct audio settings for your machine.
The audio settings summary will look like that shown in figure 9.3. Choices
available might be Jack, ASIO, OSS, ALSA, or the name of a specific device
you have installed as a sound card. Most times the default settings will work. If
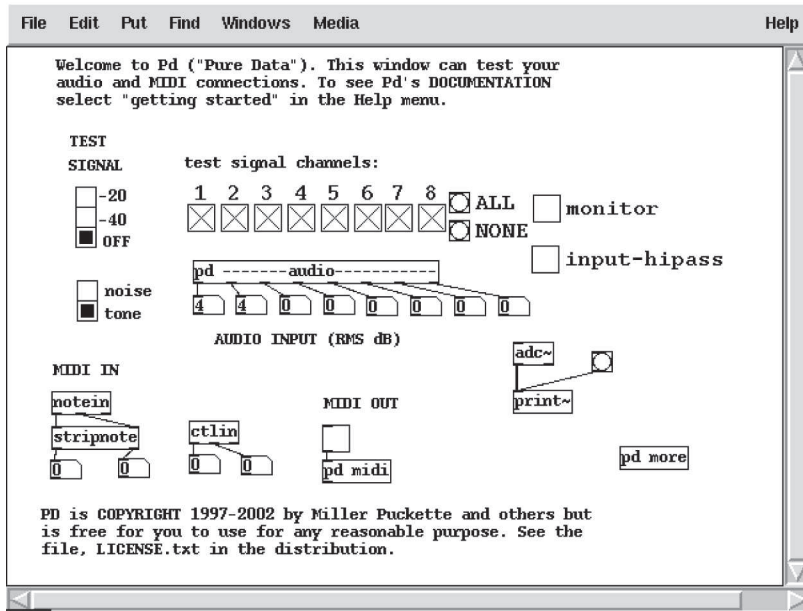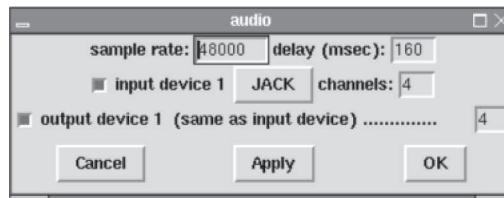
Figure 9.2
Test signal.

Figure 9.3
Audio settings pane.

you are using Jack (recommended), then check that Jack audio is running with `qjackctl` on Linux or `jack-pilot` on MacOSX. Sample rate is automatically taken from the sound card.

SECTION 9.2

How Does Pure Data Work?

Pure Data uses a kind of programming called *dataflow*, because the data flows along connections and through objects which process it. The output of one process feeds into the input of another, and there may be many steps in the flow.

Objects

Here is a box ☐. A musical box, wound up and ready to play. We call these boxes *objects*. Stuff goes in, stuff comes out. For it to pass into or out of them, objects must have *inlets* or *outlets*. Inlets are at the top of an object box, outlets are at the bottom. Here is an object that has two inlets and one outlet: ☐. They are shown by small "tabs" on the edge of the object box. Objects contain processes or procedures which change the things appearing at their inlets and then send the results to one or more outlets. Each object performs some simple function and has a name appearing in its box that identifies what it does. There are two kinds of object, *intrinsics* which are part of the core Pd program, and *externals* which are separate files containing add-ons to the core functions. Collections of externals are called *libraries* and can be added to extend the functionality of Pd. Most of the time you will neither know nor care whether an object is intrinsic or external. In this book and elsewhere the words *process*, *function*, and *unit* are all occasionally used to refer to the object boxes in Pd.

Connections

The connections between objects are sometimes called *cords* or *wires*. They are drawn in a straight line between the outlet of one object and the inlet of another. It is okay for them to cross, but you should try to avoid this since it makes the patch diagram harder to read. At present there are two degrees of thickness for cords. Thin ones carry message data and fatter ones carry audio signals. *Max/MSP* and probably future versions of Pd will offer different colours to indicate the data types carried by wires.

Data

The "stuff" being processed comes in several flavours: video frames, sound signals, and messages. In this book we will only be concerned with sounds and messages. Objects give clues about what kind of data they process by their name. For example, an object that adds together two sound signals looks like ☐. The + means that this is an addition object, and the $\sim$ (tilde character) means that its object operates on signals. Objects without the tilde are used to process messages, which we shall concentrate on before studying audio signal processing.

Patches

A collection of objects wired together is a *program* or *patch*. For historical reasons *program* and *patch*[2] are used to mean the same thing in sound synthesis. Patches are an older way of describing a synthesiser built from modular units connected together with patch cords. Because inlets and outlets are at the top and bottom of objects, the data flow is generally down the patch. Some objects

---

2. A different meaning of patch from the one programmers use to describe changes made to a program to remove bugs.

have more than one inlet or more than one outlet, so signals and messages can be a function of many others and may in turn generate multiple new data streams. To construct a program we place processing objects onto an empty area called a *canvas*, then connect them together with wires representing pathways for data to flow along. At each step of a Pure Data program any new input data is fed into objects, triggering them to compute a result. This result is fed into the next connected object and so on until the entire chain of objects, starting with the first and ending with the last, have all been computed. The program then proceeds to the next step, which is to do the same thing all over again, forever. Each object maintains a state which persists throughout the execution of the program but may change at each step. Message-processing objects sit idle until they receive some data rather than constantly processing an empty stream, so we say Pure Data is an *event-driven system*. Audio-processing objects are always running, unless you explicitly tell them to switch off.

A Deeper Look at Pd

Before moving on to make some patches consider a quick aside about how Pd actually interprets its patches and how it works in a wider context. A patch, or dataflow graph, is navigated by the interpreter to decide when to compute certain operations. This *traversal* is *right to left* and *depth first*, which is a computer science way of saying it looks ahead and tries to go as deep as it can before moving on to anything higher, and moves from right to left at any branches. This is another way of saying it wants to know what depends on what before deciding to calculate anything. Although we think of data flowing down the graph, the nodes in figure 9.4 are numbered to show how Pd really thinks about things. Most of the time this isn't very important unless you have to debug a subtle error.

Pure Data Software Architecture

Pure Data actually consists of more than one program. The main part called **pd** performs all the real work and is the interpreter, scheduler, and audio engine. A separate program is usually launched whenever you start the main engine, which is called the **pd-gui**. This is the part you will interact with when building Pure Data programs. It creates files to be read by **pd** and automatically passes them to the engine. There is a third program called the **pd-watchdog**, which runs as a completely separate process. The job of the watchdog is to keep an eye on the execution of programs by the engine and to try to gracefully halt the program if it runs into serious trouble or exceeds available CPU resources. The context of the **pd** program is shown in figure 9.5 in terms of other files and devices.

Your First Patch

Let's now begin to create a Pd patch as an introductory exercise. We will create some objects and wire them together as a way to explore the interface.
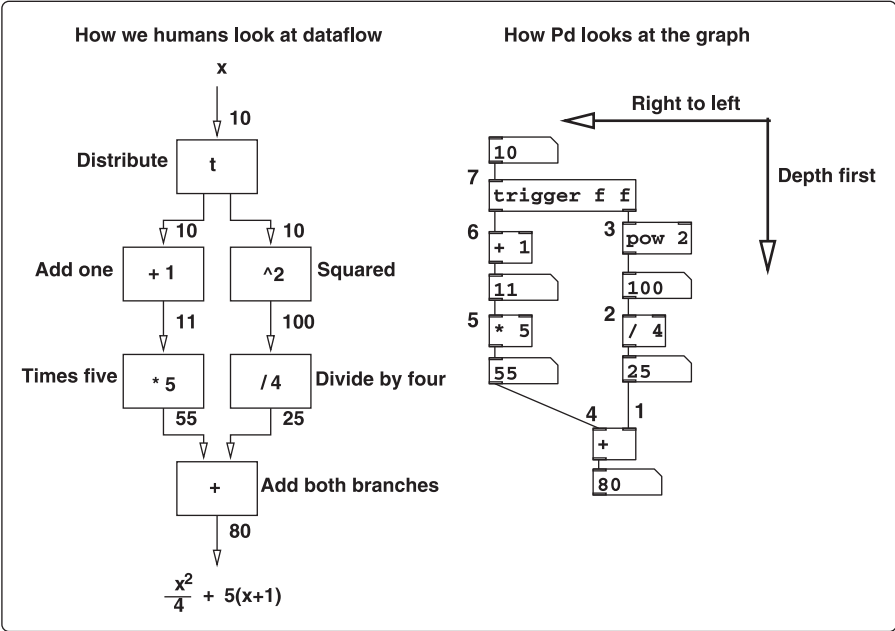
How we humans look at dataflow

x

10

Distribute   t

10   10

Add one   + 1   ^2   Squared

11   100

Times five   * 5   / 4   Divide by four

55   25

+   Add both branches

80

$$\frac{x^2}{4} + 5(x+1)$$

How Pd looks at the graph

Right to left

Depth first

10

7   trigger f f

6   + 1   3   pow 2

11   100

5   * 5   2   / 4

55   25

4   1

+

80

Figure 9.4
Dataflow computation.

pd–watchdog

pd (main engine)   pd–gui

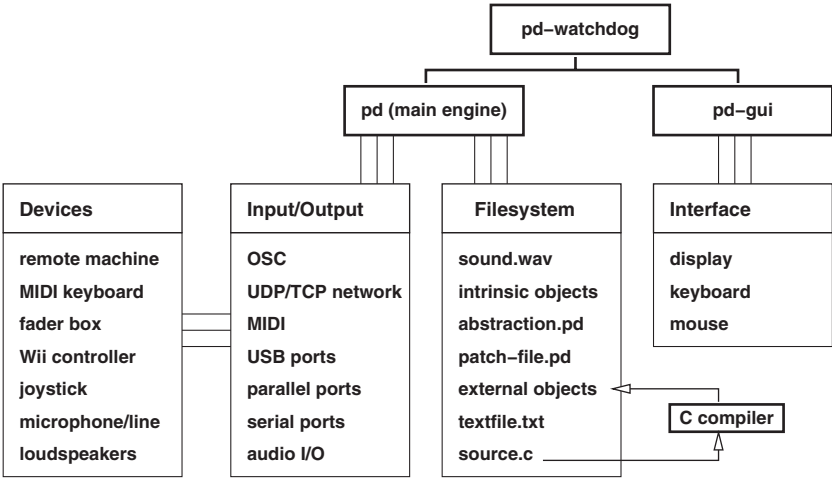| Devices	Input/Output	Filesystem	Interface
remote machine	OSC	sound.wav	display
MIDI keyboard	UDP/TCP network	intrinsic objects	keyboard
fader box	MIDI	abstraction.pd	mouse
Wii controller	USB ports	patch–file.pd	
joystick	parallel ports	external objects	
microphone/line	serial ports	textfile.txt	
loudspeakers	audio I/O	source.c	

C compiler

Figure 9.5
Pure Data software architecture.

Creating a Canvas

A *canvas* is the name for the sheet or window on which you place objects. You can resize a canvas to make it as big as you like. When it is smaller than the patch it contains, horizontal and vertical scrollbars will allow you to change the area displayed. When you save a canvas, its size and position on the desktop are stored. From the console menu select File→New or type CTRL+n at the keyboard. A new blank canvas will appear on your desktop.

New Object Placement

To place an object on the canvas, select Put→Object from the menu or use CTRL+1 on the keyboard. An active, dotted box will appear. Move it somewhere on the canvas using the mouse and click to fix it in place. You can now type the name of the new object, so type the multiplication character * into the box. When you have finished typing, click anywhere on the blank canvas to complete the operation. When Pure Data recognises the object name you give, it immediately changes the object box boundary to a solid line and adds a number of inlets and outlets. You should see a [*] on the canvas now.

Figure 9.6
Objects on a canvas.

Pure Data searches the paths it knows for objects, which includes the current working directory. If it doesn't recognise an object because it can't find a definition anywhere, the boundary of the object box remains dotted. Try creating another object and typing some nonsense into it; the boundary will stay dotted and no inlets or outlets will be assigned. To delete the object, place the mouse cursor close to it, click and hold in order to draw a selection box around it, then hit delete on the keyboard. Create another object beneath the last one with an addition symbol so that your canvas looks like figure 9.6.

Edit Mode and Wiring

When you create a new object from the menu, Pd automatically enters edit mode, so if you just completed the instructions above you should currently be in edit mode. In this mode you can make connections between objects or delete objects and connections.

Figure 9.7
Wiring objects.

Hovering over an outlet will change the mouse cursor to a new "wiring tool." If you click and hold the mouse when the tool is active you will be able to drag a connection away from the object. Hovering over a compatible inlet while in this state will allow you to release the mouse and make a new connection. Connect together the two objects you made so that your canvas looks like figure 9.7. If you want to delete a connection, it's easy; click on the connection to

select it and then hit the `delete` key. When in edit mode you can move any object to another place by clicking over it and dragging with the mouse. Any connections already made to the object will follow along. You can pick up and move more than one object if you draw a selection box around them first.

Initial Parameters

Most objects can take some initial parameters or *arguments*, but these aren't always required. Objects can be created without these if you are going to pass data via the inlets as the patch is running. The ⊞ object can be written as ⊞ 3 to create an object which always adds 3 to its input. Uninitialised values generally resort to zero, so the default behaviour of ⊞ would be to add 0 to its input, which is the same as doing nothing. Contrast this with the default behaviour of ⊞, which always gives zero.

Modifying Objects

You can also change the contents of any object box to alter the name and function, or to add parameters.

* 5
+ 3

In figure 9.8 the objects have been changed to give them initial parameters. The multiply object is given a parameter of 5, which means it multiplies its input by 5 no matter what comes in. If the input is 4 then the output will be 20. To change the contents of an object click on the middle of the box where the name is and type the new text. Alternatively, click once, and then again at the end of the text to append new stuff, such as adding 5 and 3 to the objects shown in figure 9.8.

Figure 9.8
Changing objects.

Number Input and Output

10
* 5
+ 3
53

One of the easiest ways to create and view numerical data is to use *number boxes*. These can act as input devices to generate numbers, or as displays to show you the data on a wire. Create one by choosing `Put→Number` from the canvas menu, or use `CTRL+3`, and place it above the ⊞ object. Wire it to the left inlet. Place another below the ⊞ object and wire the object outlet to the top of the number box as shown in figure 9.9.

Figure 9.9
Number boxes.

Toggling Edit Mode

Pressing `CTRL+e` on the keyboard will also enter edit mode. This key combination toggles modes, so hitting `CTRL+e` again exits edit mode. Exit edit mode now by hitting `CTRL+e` or selecting `Edit→Edit mode` from the canvas menu. The mouse cursor will change and you will no longer be able to move or modify object boxes. However, in this mode you can operate the patch components such as buttons and sliders normally. Place the mouse in the top number box, click and hold, and move it upwards. This input number value will change, and it will send messages to the objects below it. You will see the second number

box change too as the patch computes the equation $y = 5x + 3$. To reenter edit mode hit `CTRL+E` again or place a new object.

More Edit Operations

Other familiar editing operations are available while in edit mode. You can cut or copy objects to a buffer or paste them back into the canvas, or to another canvas opened with the same instance of Pd. Take care with pasting objects in the buffer because they will appear directly on top of the last object copied. To select a group of objects you can drag a box around them with the mouse. Holding `SHIFT` while selecting allows multiple separate objects to be added to the buffer.

- `CTRL+a` Select all objects on canvas.
- `CTRL+d` Duplicate the selection.
- `CTRL+c` Copy the selection.
- `CTRL+v` Paste the selection.
- `CTRL+x` Cut the selection.
- `SHIFT` Select multiple objects.

Duplicating a group of objects will also duplicate any connections between them. You may modify an object once created and wired up without having it disconnect so long as the new one is compatible the existing inlets and outlets, for example replacing ⬚ with ⬚. Clicking on the object text will allow you to retype the name and, if valid, the old object is deleted and its replacement remains connected as before.

Patch Files

Pd files are regular text files in which patches are stored. Their names always end with a **.pd** file extension. Each consists of a *netlist*, which is a collection of object definitions and connections between them. The file format is terse and difficult to understand, which is why we use the GUI for editing. Often there is a one-to-one correspondence between a patch, a single canvas, and a file, but you can work using multiple files if you like because all canvases opened by the same instance of Pd can communicate via global variables or through `send` and `receive` objects. Patch files shouldn't really be modified in a text editor unless you are an expert Pure Data user, though a plaintext format is useful because you can do things like search for and replace all occurrences of an object. To save the current canvas into a file select `File→Save` from the menu or use the keyboard shortcut `CTRL+s`. If you have not saved the file previously a dialogue panel will open to let you choose a location and file name. This would be a good time to create a folder for your Pd patches somewhere convenient. Loading a patch, as you would expect, is achieved with `File→Open` or `CTRL+o`.

SECTION 9.3

Message Data and GUI Boxes

We will briefly tour the basic data types that Pd uses along with GUI objects that can display or generate that data for us. The message data itself should

not be confused with the objects that can be used to display or input it, so we distinguish messages from boxes. A *message* is an event, or a piece of data that gets sent between two objects. It is invisible as it travels down the wires, unless we print it or view it in some other way like with the number boxes above. A message can be very short, only one number or character, or very long, perhaps holding an entire musical score or synthesiser parameter set. Messages can be floating point numbers, lists, symbols, or pointers which are references to other types like datastructures. Messages happen in *logical time*, which means that they aren't synchronised to any real timebase. Pd processes them as fast as it can, so when you change the input number box, the output number box changes instantly. Let's look at some other message types we'll encounter while building patches to create sound. All GUI objects can be placed on a canvas using the `Put` menu or using keyboard shortcuts `CTRL+1` through `CTRL+8`, and all have *properties* which you can access by right-clicking them while in edit mode and selecting the `properties` pop-up menu item. Properties include things like colour, ranges, labels, and size and are set per instance.

Selectors

With the exception of a bang message, all other message types carry an invisible *selector*, which is a symbol at the head of the message. This describes the "type" of the remaining message, whether it represents a symbol, number, pointer, or list. Object boxes and GUI components are only able to handle appropriate messages. When a message arrives at an inlet the object looks at the selector and searches to see if it knows of an appropriate *method* to deal with it. An error results when an incompatible data type arrives at an inlet, so for example, if you supply a symbol type message to a `delay` object it will complain:

```
error: delay: no method for 'symbol'
```

Bang Message

This is the most fundamental and smallest message. It just means "compute something." Bangs cause most objects to output their current value or advance to their next state. Other messages have an implicit bang so they don't need to be followed with a bang to make them work. A bang has no value; it is just a bang.

Bang Box

A bang box looks like this ◻ and sends and receives a bang message. It briefly changes colour, like this ◼, whenever it is clicked or upon receipt of a bang message to show you one has been sent or received. These may be used as buttons to initiate actions or as indicators to show events.

Float Messages

"Floats" is another name for numbers. As well as regular (integer) numbers like 1, 2, 3 and negative numbers like -10 we need numbers with decimal points like

−198753.2 or 10.576 to accurately represent numerical data. These are called *floating point numbers*, because of the way computers represent the decimal point position. If you understand some computer science then it's worth noting that there are no integers in Pd; everything is a float, even if it appears to be an integer, so 1 is really 1.0000000. Current versions of Pd use a 32-bit float representation, so they are between −8388608 and 8388608.

Number Box

For float numbers we have already met the number box, which is a dual-purpose GUI element. Its function is to either display a number or allow you to input one. A bevelled top right corner like this ▣ denotes that this object is a number box. Numbers received on the inlet are displayed and passed directly to the outlet. To input a number click and hold the mouse over the value field and move the mouse up or down. You can also type in numbers. Click on a number box, type the number and hit `RETURN`. Number boxes are a compact replacement for faders. By default it will display up to five digits including a sign if negative, –9999 to 99999, but you can change this by editing its properties. Holding `SHIFT` while moving the mouse allows a finer degree of control. It is also possible to set an upper and lower limit from the `properties` dialogue.

Toggle Box

Another object that works with floats is a toggle box. Like a checkbox on any standard GUI or web form, this has only two states, on or off. When clicked a cross appears in the box like this ⊠ and it sends out a number 1; clicking again causes it to send out a number 0 and removes the cross so that it looks like this ▢. It also has an inlet which sets the value, so it can be used to display a binary state. Sending a bang to the inlet of a toggle box does not cause the current value to be output; instead it flips the toggle to the opposite state and outputs this value. Editing `properties` also allows you to send numbers other than 1 for the active state.

Sliders and Other Numerical GUI Elements

GUI elements for horizontal and vertical sliders can be used as input and display elements. Their default range is 0 to 127, nice for MIDI controllers, but like all other GUI objects this can be changed in their `properties` window. Unlike those found in some other GUI systems, Pd sliders do not have a step value. Shown in figure 9.10 are some GUI objects at their standard sizes. They can be ornamented with labels or created in any colour. Resizing the slider to make it bigger will increase the step resolution. A radio box provides a set of mutually exclusive buttons which output a number starting at zero. Again, they work equally well as indicators or input elements. A better way to visually display an audio level is to use a VU meter. This is set up to indicate decibels, so it has a rather strange scale from −99.0 to +12.0. Audio signals that range from −1.0 to +1.0 must first be scaled using the appropriate object. The VU is one of the few GUI elements that acts only as a display.
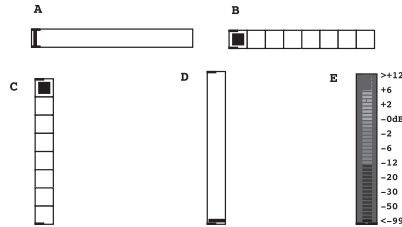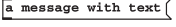
Figure 9.10
GUI Objects A: Horizontal slider. B: Horizontal radio box. C: Vertical radio box. D: Vertical slider. E: VU meter.
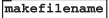
General Messages

Floats and bangs are types of message, but messages can be more general. Other message types can be created by prepending a *selector* that gives them special meanings. For example, to construct lists we can prepend a *list* selector to a set of other types.

Message Box

These are visual containers for user-definable messages. They can be used to input or store a message. The right edge of a message box is curved inwards like this ⬚⟨, and it always has only one inlet and one outlet. They behave as GUI elements, so when you click a message box it sends its contents to the outlet. This action can also be triggered if the message box receives a bang message on its inlet. Message boxes do some clever thinking for us. If we store something like `5.0⟨` it knows that is a float and outputs a float type, but if we create `a message with text⟨` then it will send out a list of symbols; so it is type aware, which saves us having to say things like "float 1.0" as we would in C programs. It can also abbreviate floating point numbers like 1.0 to 1, which saves time when inputting integer values, but it knows that they are really floats.

Symbolic Messages

A *symbol* generally is a word or some text. A symbol can represent anything; it is the most basic textual message in Pure Data. Technically a symbol in Pd can contain any printable or nonprintable character. But most of the time you will only encounter symbols made out of letters, numbers, and some interpunctuation characters like a dash, dot, or underscore. The Pd editor does some automatic conversions: words that can also be interpreted as a number (like 3.141 or $1e + 20$) are converted to a float internally (but $+ 20$ still is a symbol!). Whitespace is used by the editor to separate symbols from each other, so you cannot type a symbol including a space character into a message box. To generate symbols with backslash-escaped whitespace or other special characters inside, use the `makefilename` symbol maker object. The `openpanel` file dialogue object preserves and escapes spaces and other special characters in filenames,

too. Valid symbols are *badger*, *sound_2*, or *all_your_base*, but not *hello there* (which is two symbols) or *20* (which will be interpreted as a float, 20.0).

Symbol Box

For displaying or inputting text you may use a `symbol` box. Click on the display field and type any text that is a valid symbol and then hit `ENTER/RETURN`. This will send a symbol message to the outlet of the box. Likewise, if a symbol message is received at the inlet it will be displayed as text. Sending a bang message to a symbol box makes it output any symbol it already contains.

Lists

A list is an ordered collection of any things, floats, symbols, or pointers that are treated as one. Lists of floats might be used for building melody sequences or setting the time values for an envelope generator. Lists of symbols can be used to represent text data from a file or keyboard input. Most of the time we will be interested in lists of numbers. A list like {*2 127 3.14159 12*} has four elements; the first element is 2.0 and the last is 12.0. Internally, Pure Data recognises a list because it has a *list selector* at the start, so it treats all following parts of the message as ordered list elements. When a list is sent as a message all its elements are sent at once. A list selector is attached to the beginning of the message to determine its type. The selector is the word "list," which has a special meaning to Pd. Lists may be of mixed types like {*5 6 pick up sticks*}, which has two floats and three symbols. When a list message contains only one item which is a float it is automatically changed (cast) back to a float. Lists can be created in several ways, by using a message box, or by using `pack`, which we will meet later, to pack data elements into a list.

Pointers

As in other programming languages, a *pointer* is the address of some other piece of data. We can use them to build more complex data structures, such as a pointer to a list of pointers to lists of floats and symbols. Special objects exist for creating and dereferencing pointers, but since they are an advanced topic we will not explore them further in this book.

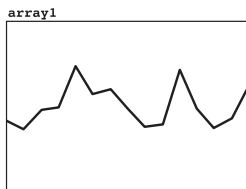## Tables, Arrays, and Graphs

Figure 9.11
An array.

A *table* is sometimes used interchangeably with an *array* to mean a two-dimensional data structure. An array is one of the few invisible objects. Once declared it just exists in memory. To see it, a separate *graph* like that shown in figure 9.11 allows us to view its contents.

Graphs have the wonderful property that they are also GUI elements. You can draw data directly into a graph using the mouse and it will modify the array it is attached to. You can see a graph of **array1** in figure 9.11 that has been drawn by hand. Similarly, if the data in an array changes and it's attached to a visible

graph then the graph will show the data as it updates. This is perfect for drawing detailed envelopes or making an oscilloscope display of rapidly changing signals.

To create a new array select Put→Array from the menu and complete the dialogue box to set up its name, size, and display characteristics. On the canvas a graph will appear showing an array with all its values initialised to zero. The Y-axis range is −1.0 to +1.0 by default, so the data line will be in the centre. If the save contents box is checked then the array data will be saved along with the patch file. Be aware that long sound files stored in arrays will make large patch files when saved this way. Three draw styles are available: points, polygon, and Bezier, to show the data with varying degrees of smoothing. It is possible to use the same graph to display more than one array, which is very useful when you wish to see the relationship between two or more sets of data. To get this behaviour use the in last graph option when creating an array.
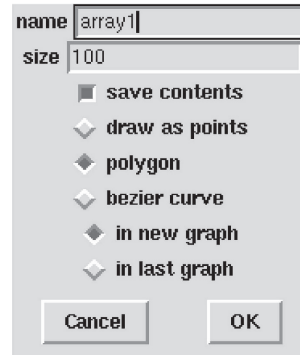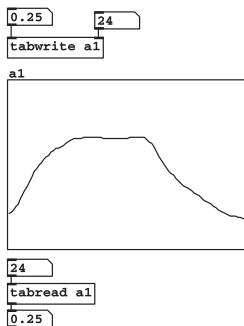
Figure 9.12
Create array.

Data is written into or read from a table by an index number which refers to a position within it. The index is a whole number. To read and write arrays, several kinds of accessor object are available. The `tabread` and `tabwrite` objects allow you to communicate with arrays using messages. Later we will meet `tabread4~` and `tabwrite~` objects that can read and write audio signals. The array **a1** shown in figure 9.13 is written by the `tabwrite` object above it, which specifies the target array name as a parameter. The right inlet sets the index and the left one sets the value. Below it a `tabread` object takes the index on its inlet and returns the current value.

Figure 9.13
Accessing an array.

Getting Help with Pure Data

At <**http://puredata.hurleur.com/**> there is an active, friendly forum, and the mailing list can be subscribed to at **pd-list@iem.at**.

Exercises

Exercise 1

On Linux, type `pd --help` at the console to see the available startup options. On Windows or MacOSX read the help documentation that comes with your downloaded distribution.

Exercise 2

Use the `Help` menu, select `browse help`, and read through some built-in documentation pages. Be familiar with the `control examples` and `audio examples` sections.

Exercise 3

Visit the online `pdwiki` at <http://puredata.org> to look at the enormous range of objects available in `pd-extended`.

References

Arduino I/O boards: http://www.arduino.cc/.

Puckette, M. (1996). "Pure Data." *Proceedings, International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 269–272.

Puckette, M. (1996). "Pure Data: Another integrated computer music environment." *Proceedings, Second Intercollege Computer Music Concerts*, Tachikawa, Japan, pp. 37–41.

Puckette, M. (1997). "Pure Data: Recent progress." *Proceedings, Third Intercollege Computer Music Festival.* Tokyo, Japan, pp. 1–4.

Puckette, M. (2007). *The Theory and Technique of Electronic Music.* World Scientific Press.

Winkler, T. (1998). *Composing Interactive Music: Techniques and Ideas Using Max.* MIT Press.

Zimmer, F. (editor) (2006). *Bang—A Pure Data Book.* Wolke-Verlag.