

---

## Chapter 2

# Interactive sound using Pure Data

---

## 2.1 Introduction

This chapter explores interactive sound and music using the Pure Data real-time graphical program language. It gives you a solid background in computational approaches to sound and music, including music representation systems such as MIDI (Musical Instrument Digital Interface), and DSP (Digital Signal Processing). In addition, this chapter will teach you to create your own electronic music systems, and to build software that will record, play back and independently generate sound and music.

This chapter takes a broad view of music. For the purposes of this text, music is a special category of sound that can have specific acoustic features (detailed below). It follows on from the *Creative Computing 2: interactive multimedia* module (2910227), and as such assumes some knowledge of sound and acoustics. However, no specialist knowledge of music is required beyond that explained in the text. As such, this chapter does not require any high-level musical training, although knowledge of music will occasionally be advantageous.

---

## 2.2 Equipment requirements

This course does not require specialist equipment. Pure Data is compatible with any computer running recent versions of Windows, Linux or Mac OS X. At times, Musical Instrument Digital Interface (MIDI) equipment is mentioned. Access to a MIDI keyboard and interface may be of use, but is not required. Please use the alternative methods indicated if you do not have access to a MIDI device.

---

## 2.3 The Fundamentals of sound

Sound is a longitudinal vibration – a wave of varying pressure. Waves travel through any medium at various speeds, depending on both the density **and** elasticity of the medium. We tend to associate sound with longitudinal vibrations travelling through air. These waves have important properties we need to understand before we can start using digital audio.

Objects moving in air cause air molecules to move away from them. Air molecules are pushed into other air molecules (**compression**). This leaves space for new air molecules to move in (**rarefaction**). These new molecules push against the original air molecules. They in turn push more molecules, leaving more space to be filled. This starts a process of compression and rarefaction, and a pressure wave is formed that propagates through the air at a speed of approximately 340 metres per second.

This is similar to what happens when you throw a stone into a pool of water. The rock displaces the water, which pushes more water away, creating space for more water to rush in. The ripples move away from the source. This is similar (but not at all times and in all ways identical) to what happens in air.

We can make a drum by taking a section of flexible material and stretching it over the hollow end of a cylindrical object. If we strike the surface (the skin) of the drum with an object, this causes the surface to move very quickly back and forth. As this energy dissipates, the speed of this movement slows down. This causes air molecules to move either side of the drum skin in the way described above, creating sound waves. What we hear depends on the type of material, the amount of force, and the surface tension.

Waves have three important properties: amplitude, frequency and phase, discussed below.

### 2.3.1 Amplitude

The amplitude of a wave can be thought of as corresponding to the **amount of energy** in the skin of the drum. When we make a sound, for example by clapping our hands, we perceive the sound as having **loudness**. This roughly correlates to the energy being generated by our hands when they hit each other, or when we strike a drum skin, and by extension, the force with which the air molecules are pushed together. For example, if we drop a large rock in a pool, the waves will be higher than if we drop a small one. If we hit the drum skin hard, there will be more initial energy in the skin, and the sound will be louder than if we hit it softly.

We measure the amplitude of sound waves in **decibels** (dB – tenth of a Bel) after Alexander Graham Bell (1847–1922). This unit of measurement has many different scales, depending on whether we are measuring air pressure (which is also measured in **pascals**), voltages, or industry standard **Volume Units**. In sound recording, we use a scale whereby the loudest sound that can be recorded is known as 0 dBFS (decibels relative to full scale). This is equivalent to the maximum linear amplitude of 1. To calculate the amplitude value between 0 and 1 in dBFS, we use the expression:

$$x = 20 \log_{10} y$$

where  $y$  is the value in linear amplitude between 0 and 1, and  $x$  is the value in dB. Note that 0.5 is obviously not equivalent to half the perceived loudness, corresponding instead to a logarithmic amplitude of -6.0206 dBFS. For more information please refer to Section 3.4.3 in *Creative Computing 2: interactive multimedia Vol. 1* and Section 2.2 in *Creative Computing 2: interactive multimedia Vol. 2*.

### 2.3.2 Frequency

Waves propagate at particular rates. We can think of this rate as equal to the number of compression and rarefaction cycles each second. Different types of events have different numbers of cycles per second. When we strike a drum hard, there is a lot of energy in the surface, creating a great deal of tension. This causes the skin to move quickly. As the energy gets used up, the movement of the skin

gradually slows down. This causes the frequency of vibration to slow down. This creates the fast change of pitch that we associate with many types of drums.

We measure cycles per second (frequency) in **Hertz** (Hz), after Heinrich Rudolf Hertz (1857–1894). The human ear can detect frequencies between 20 Hz and 20,000 Hz, depending on age and ear function. Although air pressure waves exist way below and above these boundaries, our ears are incapable of perceiving them as sounds. More information on frequency can be found in Section 3.4.2 of *Creative Computing 2: interactive multimedia Vol. 1*, and in Sections 2.2 and 2.3 of *Creative Computing 2: interactive multimedia Vol. 2*.

### 2.3.3 Phase

Sound waves can not only be measured in terms of amplitudes and frequencies. They also have a phase. Waves are periodic – that is, they repeat themselves in cycles. They start with a steady state, go into compression, go back to a steady state, go into rarefaction, and then reach a steady state once more, where the cycle begins again.

We can measure the stages (or phases) of each cycle in degrees (or radians). We can measure the phase of a wave at any point in its cycle. A wave that begins with a phase of 0, completes half a cycle at 180 degrees ( $\pi$  radians), and completes its cycle at 360 degrees ( $2\pi$  radians). Further information on phase can be found in Section 3.4.4 of *Creative Computing 2: interactive multimedia Vol. 1*.

### 2.3.4 Musical sounds

Almost all naturally occurring sounds are the sum of many seemingly unrelated frequencies. The mathematical relationship between these different frequencies is complicated in that they are most often not easily described in terms of integer ratios. Compared to harmonic sounds, such sounds can be considered complex. Good examples of these types of sounds are the wind, the sea, bangs, crashes, thuds and explosions. As such, they do not bear much relation to what some people think of as music, despite appearing in many forms of music all the time.

Importantly, human voices can produce sounds that have far fewer combinations of frequencies and amplitudes (i.e. are non-complex). In addition, the human voice can produce sounds that maintain relatively steady frequencies and amplitudes consistently. Furthermore, the human voice can produce sounds whereby the relationships between the component frequencies approximate relationships that can be described by integer ratios relatively easily. Over time, we have learned to make instruments that produce sounds with these three qualities for the purposes of making certain types of music.

Many people associate an organised series of non-complex sounds exhibiting relative stability in terms of frequency and amplitude, and where the frequencies are mostly describable using integer ratios, with that which is commonly thought of as music. Systems exist which help us to organise these types of sounds in terms of fundamental frequency, volume, complexity and timing. For more information on these systems, you should refer to Chapter 2 of *Creative Computing 2: interactive multimedia Vol. 2*.

Music traditionally features combinations of complex and non-complex sounds, occurring at various times in relation to some form of recognisable pattern or sequence. Most importantly, in this context, frequency and amplitude are not thought of in terms of Hz (hertz) and dB (decibels). Instead, they are thought of in terms of **note-pitch** and **dynamics**.

Importantly, some modern music does not feature any non-complex, pitched sound whatsoever, instead favouring patterned combinations of complex sounds. This is a more recent phenomenon, facilitated by the development of music and sound technology that can record, sequence and play back any sound of any type. This type of music making is exemplified by the work of Pierre Schaeffer and the approach he refers to as **musique concrète** (Schaeffer 1952).

Throughout this chapter, you will be encouraged to engage with complex and non-complex sounds both in terms of engineering and creativity. You should feel free to create sound and music in any way you choose given the criteria laid out in the exercises. On occasion, you may wish to use systems of pitched sounds (scales) in relation to some sort of timing and rhythm, in which case you should refer to Chapter 2 of *Creative Computing 2: interactive multimedia Vol. 2*.

---

## 2.4 Sound, music and computers

We can measure sound using any electrical device that is sensitive to vibrations. Microphones work by converting the movements of a diaphragm suspended in an electromagnetic field into electrical signals. As the diaphragm moves in response to changes in pressure, this causes similar changes in the electromagnetic field. These variations can be passed down a conductive cable where the variations can be stored in some electrical or electronic medium.

Computers take these electrical signals and convert them into a series of numbers. This process is called analogue to digital conversion (ADC – a standard term for the digitisation of audio signals). The analogue signal is measured a set number of times per second (see Section 2.4.1 below), and the amplitude of the signal is stored as a 16 bit binary number. The bit depth is sometimes greater, for example 24 bit in professional systems, although 16 bits is considered sufficient to represent signals within a theoretical 96dB range, with each bit representing a range of 6dB. Regardless of resolution, we normally encounter each measured value as a floating-point number between 1 and -1.

Crucially, a single sinusoid at maximum amplitude with a phase of zero will be measured as having an amplitude of zero at the start of a cycle. This sounds confusing, but it is a very important to remember. What you hear as pitched information is actually cycling amplitudes, and so you should always be clear that there is no absolute way to separate these two aspects of sound waves.

At 90 degrees ( $0.5\pi$  radians), the amplitude will be 1. At 180 degrees ( $\pi$  radians), the amplitude will be zero, and at 270 degrees ( $1.5\pi$  radians) the amplitude will be -1. Finally, at 360 degrees ( $2\pi$  radians) the amplitude will again be zero. In this way, from 0–180 degrees, the phase of the wave is positive, and from 180–360 degrees, the phase of the wave is negative.

These **samples** are stored in a continuous stream either in RAM or on disk. They are played back in sequence to reconstruct the waveform. The result is a

staircased-shaped output (also called a piecewise constant output). This must be interpolated through the use of a **reconstruction filter** (see Section 2.4.2 below). The filtered signal is then converted back to an electrical signal before being sent to a speaker. The speaker cone is suspended in an electromagnetic field that fluctuates in response to the electrical signal. This makes the cone move back and forth in line with the amplitude of the waveform. The movements of the speaker cone cause air to compress and expand in front of the speaker at audible frequencies, turning the signal back into sound.

There are a number of issues that cause problems when representing analogue audio signals digitally. Two important issues are sampling frequency, and aliasing.

### 2.4.1 Sampling frequency

As can be seen above, a single sinusoid can be represented digitally by only two measurements – its maximum positive amplitude and its maximum negative amplitude. This is because at 0 and 180 degrees, the connecting line passes through zero amplitude as described above. The highest frequency that a human can hear is approximately 20,000 Hz. We can think of this as the upper boundary of the signal's **bandwidth**.

We know that to represent a wave with this frequency, two measurements are required. This means that the sampling frequency must be at least twice the bandwidth of the signal – 40,000 Hz. In practice, most digital audio systems have a sampling frequency of at least 44,100 Hz, with professional 'high-definition' systems sampling at rates as high as 192,000 Hz (commonly referred to as **over-sampling**). Crucially, if the sampling frequency is twice the bandwidth of the human ear, all audible frequencies can be represented.

### 2.4.2 Aliasing

If the sampling rate is less than twice the bandwidth of the signal, strange things happen. If a 200 Hz sinusoid waveform is sampled at a frequency of 300 Hz, 1.5 times the frequency of the signal, the resulting signal representation is lower than the original – by exactly 0.5 of its original frequency (100 Hz). The wave is effectively 'folded over' by the difference between half the sampling frequency and the frequency of the sampled signal. This is referred to as **aliasing** or **foldover**.

For these two reasons, when sound is sampled, it is usually filtered so that its bandwidth is exactly half the sampling rate. This prevents aliasing in cases where the sampling rate is not above the audible frequency range (which is not very often in modern contexts, but is important to understand). Aliasing often occurs during synthesis or sound manipulation, and filtering can be useful to help reduce its effects.

These issues are commonly referred to as the **Nyquist-Shannon Sampling Theorem**, after Harry Nyquist (1889–1976) and Claude Elwood Shannon (1916–2001). Additionally, the term **Nyquist Rate** is used to refer to a sampling rate that is twice the bandwidth of a signal, and the term **Nyquist Frequency** is always half this value. In almost all cases, the Nyquist frequency should always be above the maximum frequency you want to record, unless aliasing is a desired creative result. Please refer to Section 3.4.1 in *Creative Computing 2: interactive multimedia*

*Vol. 1* for more information.

---

## 2.5 MIDI – Musical Instrument Digital Interface

Despite being originally commercially developed in 1969 by Peter Zinovieff, David Cockerell and Peter Grogono at London's EMS studio in Putney, digital audio systems remained prohibitively expensive until the early 1980s. As a result, most electronic musical instruments were built using traditional synthesis methods, by combining simple waveforms, occasionally with small amounts of sampled sound stored in ROM.

By this time it had been realised that a unified method for transmitting musical information would be useful for allowing electronic musical instruments to talk to one another. Before this, instruments used Control Voltage to transmit pitch information, with Gate signals to switch events on and off. Other systems in use included Roland's DCB (Digital Control Bus). This allowed for basic sequencing, but had severe limitations, particularly with respect to compatibility.

MIDI was first proposed in 1981 by David Smith, and became an industry standard communications protocol for the transmission of musical information in 1983. The fact that it is still in use today is testament to its excellent design and applicability.

MIDI does not transmit audio signals of any kind. It communicates performance events, synchronisation information and some system information. The most important performance events are note-on (which occurs when a note is played), note-off (when a note is released), velocity (occurs simultaneously with note-on to denote how hard the key is pressed), pitch bend and overall volume. In addition, MIDI controllers can be assigned to a large number of sliders for a wide range of functions.

Almost every MIDI message is contained in a single byte, and uses 7 bit resolution. This means that almost all MIDI messages contain data values between 0 and 127, with 0 being equivalent to 0000000, and 127 being equivalent to 1111111. Each note is assigned a particular number between 0 and 127. Table 2.1 shows 8.176 Hz (C five octaves below middle C) as MIDI note 0, and 12,544 Hz (G five octaves above the G above middle C) as MIDI note 127. The frequency of a midi note number,  $f(p)$ , can be calculated using Equation 2.1.

$$f(p) = 2^{\frac{p-69}{12}} \cdot 440 \quad (2.1)$$

Note velocity ranges from 0 to 127, with a velocity of 0 indicating a note-off message. In addition, there are 16 MIDI channels, all of which can send and receive almost all types of MIDI data simultaneously.

There are cases where MIDI information is contained in two bytes, with a resolution of 14 bits, but these will not be used in this chapter. For more information, consult the MIDI Manufacturers Association at <http://www.midi.org/>

---

## 2.6 Using Pure Data

As it cannot be assumed that you will have had any experience with Pure Data, what follows is an extremely simple introduction to using the software, suitable for

MIDI note number	Pitch	Frequency (Hz)
0	C-1	8.176
12	C0	16.352
24	C1	32.703
36	C2	65.406
48	C3	130.813
60	C4	261.626
69	A4	440.000
72	C5	523.251
84	C6	1,046.502
96	C7	2,093.005
108	C8	4,186.009
120	C9	8,372.018
127	G9	12,543.854

**Table 2.1:** The relationship between MIDI note numbers, pitch, and frequency.

anyone who has never had any experience of programming real-time sound and music software before in PD.

Pure Data is based on the Max programming language (Puckette 1988, 2002). It is a free, open source sound, music and multimedia visual programming environment. The version used to generate these examples is PD-0.41.4-extended, and it can be obtained from <http://www.puredata.info/>

When you launch Pure Data, you will notice a window appears on the left hand side called PD. This window gives you feedback on how the PD system is running. You can use it to test functions, and to print information. In addition, any errors that occur will be posted in this window.

### 2.6.1 Testing your system

Before we begin you should check that your soundcard is correctly configured. To do this, locate the Media menu, and select 'Test Audio and Midi'. A new window will open. This is a 'patcher' window. You will notice that the patch is called 'testtone.pd'. At the left of the patch you will find some controls under the label TEST TONES (see Figure 2.1). Click in the box labelled '60'. You should hear a test tone. If you do not hear anything, check that your system volume is not muted, and that your soundcard is correctly configured before trying again. If there is still a problem, go to Audio Settings, and make sure that your soundcard is selected.

If you have a MIDI keyboard attached to your computer, press some keys down. You should see numbers appear under the section labelled 'MIDI IN'. If you do not, try checking that your MIDI device is selected under MIDI settings, and that it is correctly installed.

Now that you have tested your settings, you can close the testtone.pd patch. This is important, as PD patches will run in the background, even if you cannot see them.

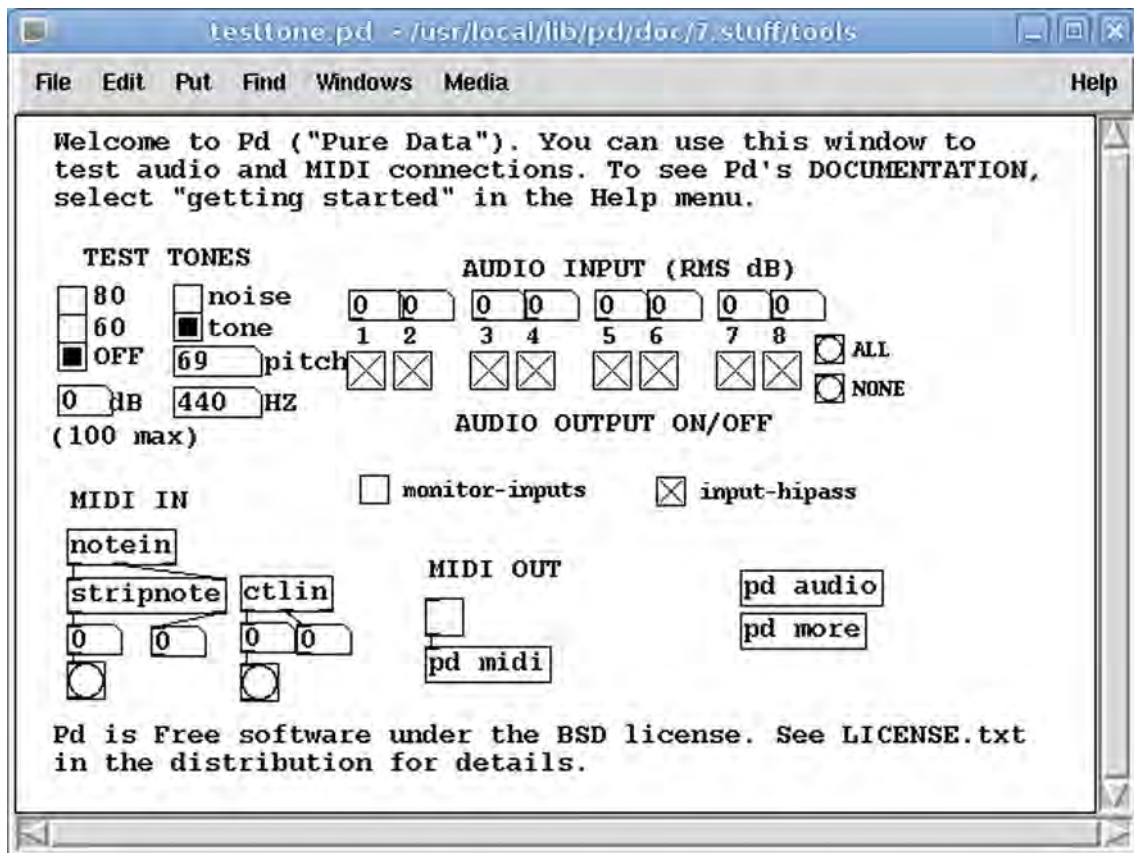


Figure 2.1: testtone.pd from Pd version 0.42-5 (Linux)

## 2.6.2 Getting started

To begin, choose file → new. This will create an empty patch. You can resize the patch window by clicking the bottom right corner and dragging the mouse.

## 2.6.3 Edit mode

You should notice that when you click the mouse within the patch area, the hand pointer appears. This means that you are in edit mode. In order to write and edit your program, you need to be in Edit mode. In order to use your program, you need to exit Edit mode. You can switch in and out of Edit mode by finding the Edit menu, and choosing the Edit mode option. When you do this, make a note of the keyboard shortcut, as you will be doing it a lot.

## 2.6.4 Number boxes

Make sure you are in Edit mode and that your patch is selected. You can do this by clicking on it.

Locate the 'Put' menu and choose 'Number'. Take a note of the shortcut key. You

should find that a box with a zero in it has appeared in your patch. This is a number box. Leave Edit mode and click in the number box with the mouse. You should find that when you drag up and down, the number in the box changes.

Now press the shift key whilst dragging. You will notice that a decimal point has appeared, and that you are now changing the numbers to the right of the decimal point. If you stop and let go of shift, you should find that when you click and drag, the whole numbers are now being changed, whilst the numbers after the decimal point are staying the same.

Finally, try typing numbers into the box. You will need to press the enter key for the numbers you have typed in to have an effect.

Number boxes (or **Atoms**) handle integers and floating point values. They can be used to enter data or to display results. If you right click on the number box and select properties, you will notice that the number box can have predefined limits. It can also have a label, send and receive messages. We will deal with this in more detail later.

### 2.6.5 Messages

Enter Edit mode and choose Put → Message, taking note of the shortcut key. You will notice that a new box has appeared, similar to the number box, but with different shaped corners on the right hand side. This is a **Message** box. Type the number 9 into the message box. When you have finished, click outside the message box.

Try grabbing the message box with the mouse. You should be able to move it around. Position it above the number box. You will notice that both the message box and the number box have small rectangular inputs and outputs on top and underneath. These are called **inlets** and **outlets**.

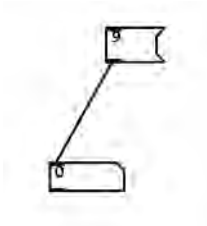
Make sure your patch is active (by clicking on it), and move your mouse over the outlet at the bottom of the message box. You should see a small black ring appear. Click and drag from the message box's outlet to the inlet on the top of the number box. You should see a thin black line appear. Let go of the mouse button just above the number boxes inlet. You should see something that looks like this (Figure 2.2).

Now leave Edit mode and click on the message box. You should see that the message has sent the data '9' to the number box. Now change the number in the number box, then click the message box again.

Messages are one way of storing and recalling data in PD. You can type long lists of messages into message boxes with the data separated by spaces. You can also separate messages inside a message box with commas.

### 2.6.6 Objects

Most of the time, you will use number boxes and message boxes to send data from place to place. However, they do not handle or manipulate data. To do something useful with data, you need to use a function. In PD, functions are performed by **objects**. PD has many hundreds of objects, each which performs different functions.



**Figure 2.2:** Message and number boxes

Objects are functions that take inputs and generate specific outputs. They can also be told how to behave through the use of messages.

Enter Edit mode and choose Put → Object, taking note of the shortcut key. You will see a square with a dashed blue outline. Position this in some whitespace and type 'print'.

Now click outside the object box. Move the [print] object underneath your message box, and above your number box.

Move your mouse until it is over the connection between the message box and the number box. You will see that a black cross has appeared. Click the mouse button. You will notice that the line has gone blue. Now, press backspace to delete your previous connection.

Make a new connection between the message box and the [print] object. Click inside the message box until the number '9' is highlighted. Type 'Hello World' and exit edit mode.

Look in the PD window and click the message box. You should see the words 'Hello World' in the PD window. You have written your first program.

Print is an object that prints any kind of text-based data to the PD window. This is useful for tracking down errors, or just finding out what things do. If you are trying to understand what an object does, it is always worth connecting a print object to it to have a look.

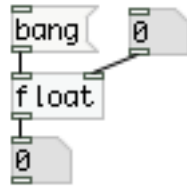
Another way of learning what an object does is to right click on it and select 'help'. This will bring up the object reference patch, which is a great way of getting extra information. You can also copy and paste patches or sections of patches. This makes building patches much easier.

We are going to start by using a few simple objects to store information and count numbers. We are also going to look at how we organise when information gets processed.

### 2.6.7 Storing and counting numbers

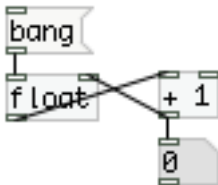
You do not need to use messages to store numbers. You can store a number in an object and recall it later. This is useful if you want to change the number every so often, something you can not do so easily with a message.

Make the patch below. Notice that 'bang' is a message, not an object.



Try changing the number box attached to the right inlet of the [float] object. Remember to exit Edit mode to interact with your patch. You should find that nothing happens. This is because in most cases, PD objects only produce output when they receive something in their leftmost inlet – the ‘hot’ outlet. Now press the ‘bang’ message. You should see the result in the number box at the bottom.

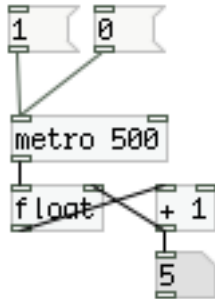
Now make the following changes to your patch until it looks like this. There is a new object here, called [+]. This adds a number to any number it receives in its hot (leftmost) inlet. In this case, it has been initialised with a value of one. This value would be changed if a number were sent to the right inlet of the [+] object.



Now click on the bang message repeatedly. Notice what happens. The number goes up by one each time you click ‘bang’. The [float] object initially produces a zero when it gets banged. This goes into the [+] object, which sends the value 1 to the number box, and also to the ‘cold’ inlet of the [float] object. The next time you click bang, the [float] object spits out the 1 that was stored in its cold inlet, and sends it to the [+] object, which adds one, and sends the result (2) to the number box and the cold inlet of [float] for storage. What would happen if a number were sent to the right (cold) inlet of the [+] box? Try it.

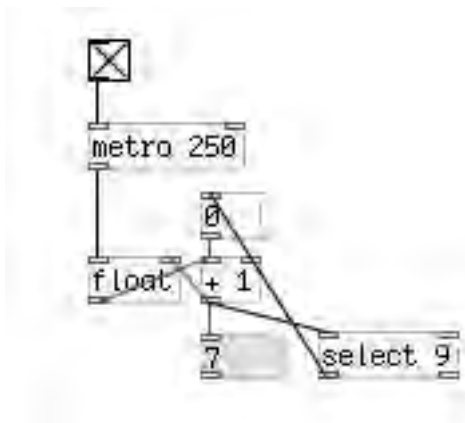
**Bang** is a special type of message that is very important in PD. Bang is so important that it even has its own user interface object – the bang button. You can create a bang button by choosing Put → Bang. Clicking on this button has exactly the same effect as clicking on a message with the word ‘bang’ in it. You can test this by connecting it to the left inlet of your [float] object and clicking it, or by using the [print] object, or both.

There is another special object called [metro] which produces bangs at regular intervals, specified in milliseconds. You can set the size of the time interval with either an initialising argument, or a number in the right inlet. [metro] starts when it receives a 1 (on) in its left inlet, and stops when it receives a 0 (off).



You can also use a special object called a **toggle** to start and stop [metro]. You can create a toggle by choosing Put → Toggle. When toggle is checked, it sends a 1 out its outlet. When it is unchecked, it sends a zero.

Another important object that sends out bangs is [select]. You can use select to make things happen at specific times, such as reset your counter. Here select sends a bang when the count reaches 9. This resets the next stored value to 1 by immediately sending 0 to the [+] object, which in turn adds 1 and sends it to the [float]. This means that the counter goes from one to eight.



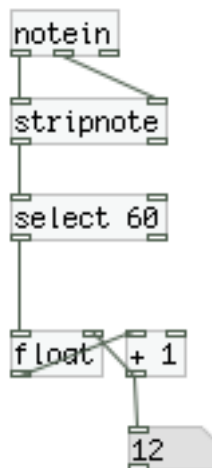
[select] can have more than one argument. This means you can use one [select] object to make lots of different things happen at different times. This is the simplest method of sequencing events in PD.

### 2.6.8 Using keyboard events

Another way of making things happen in PD is to generate events. You can grab events from the computer keyboard or from a MIDI device. The easiest way to do this is with the [key] object. This reports the key code for any key you press. You can use [select] to generate a bang when a certain key is used.

### 2.6.9 Getting MIDI events

If you have a MIDI keyboard attached, you should be able to use MIDI to send control messages to PD. [notein] receives pitch and velocity pairs when any keys are pressed on the MIDI keyboard. Be aware that MIDI note messages come in two types – note-on and note-off. If you want to use the MIDI keyboard to make things happen, you may want to strip out the note-offs with the [stripnote] object. Here is an example where the counter increments each time note number 60 is played (middle C). See if you can find it.



There are a number of important MIDI objects. In addition to [notein], the two objects which are of most use when starting out are [ctlin], which takes controller input from MIDI controller sliders, and [bendin], which converts data from the pitch-bend wheel to values between 0 and 127.

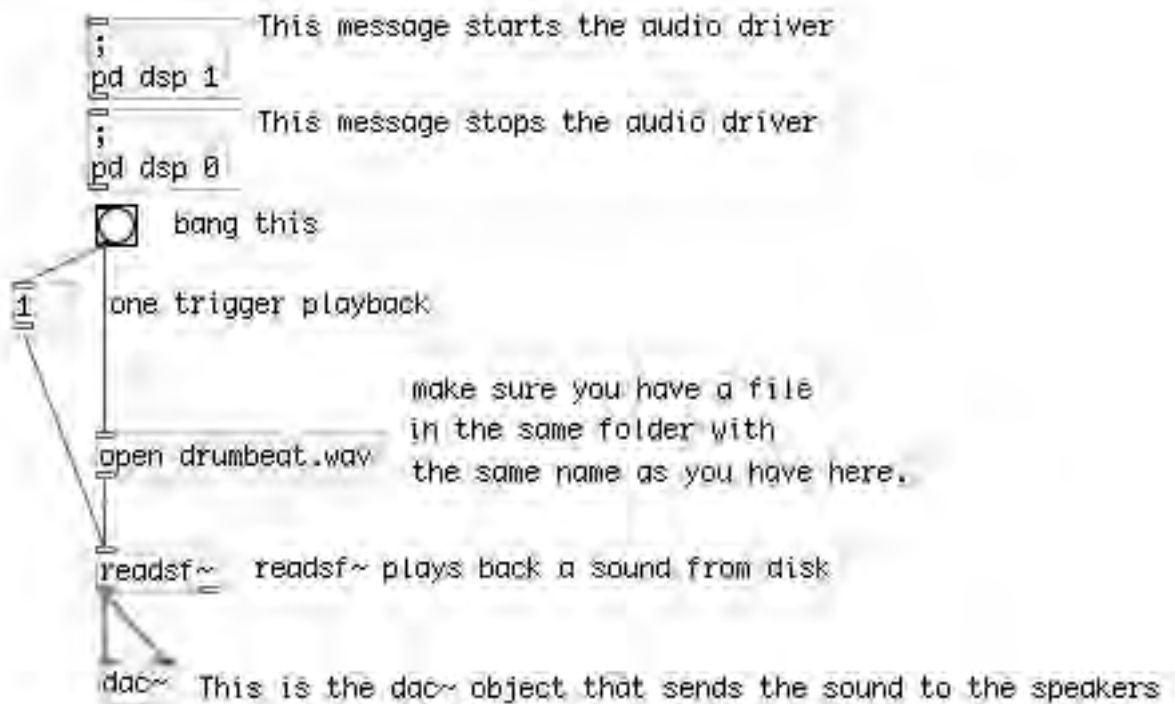
### 2.6.10 Signal objects

Basic PD objects come in two different types. Standard objects, like [metro], [select] and arithmetic objects ([+],[−],[/],[\*],[SQRT] etc.), and signal objects. Signal objects are different from standard objects in the same way that DSP and MIDI is different. Standard objects handle control data. This data represents events that are occurring. However, signal objects pass around and modify actual sound signals. In this way, PD can be used to create any number of samplers, synthesisers and effects processors that can generate and modify sound and music in real-time.

In PD, signal objects are indicated in two main ways. First, signal object names always have a tilde (~) character appended to their name. This is good to remember, as it looks like a little wave. Secondly, signal object connections are slightly thicker than standard object connections. In addition, it is worth remembering that many PD objects have signal variants that look identical apart from the tilde symbol. For example, you can multiply sounds by using the [\*~] (times tilde) object.

### 2.6.11 Playing back sounds from disk

The easiest way of playing back sounds from disk is to use the [readsf~] object. The patch below demonstrates the simplest method for doing this. In order for [readsf~] to work, you should save your patch in a folder with some uncompressed .wav files. In the example below, the patch has been saved to a folder containing a sound file called 'drumbeat.wav'.




---

#### Learning activity

Build a simple sequencer that plays back a selection of sounds from disk at different times. Use [metro], [select], [float], [+], [readsf~], plus any other objects you think you need.

Advanced exercise to test your deeper understanding of the topic:  
 Try to trigger some of the sounds from either the computer keyboard or a MIDI device.

---

---

**Learning activity**

The following are some questions for revision, that you should attempt and use to consolidate, before studying further:

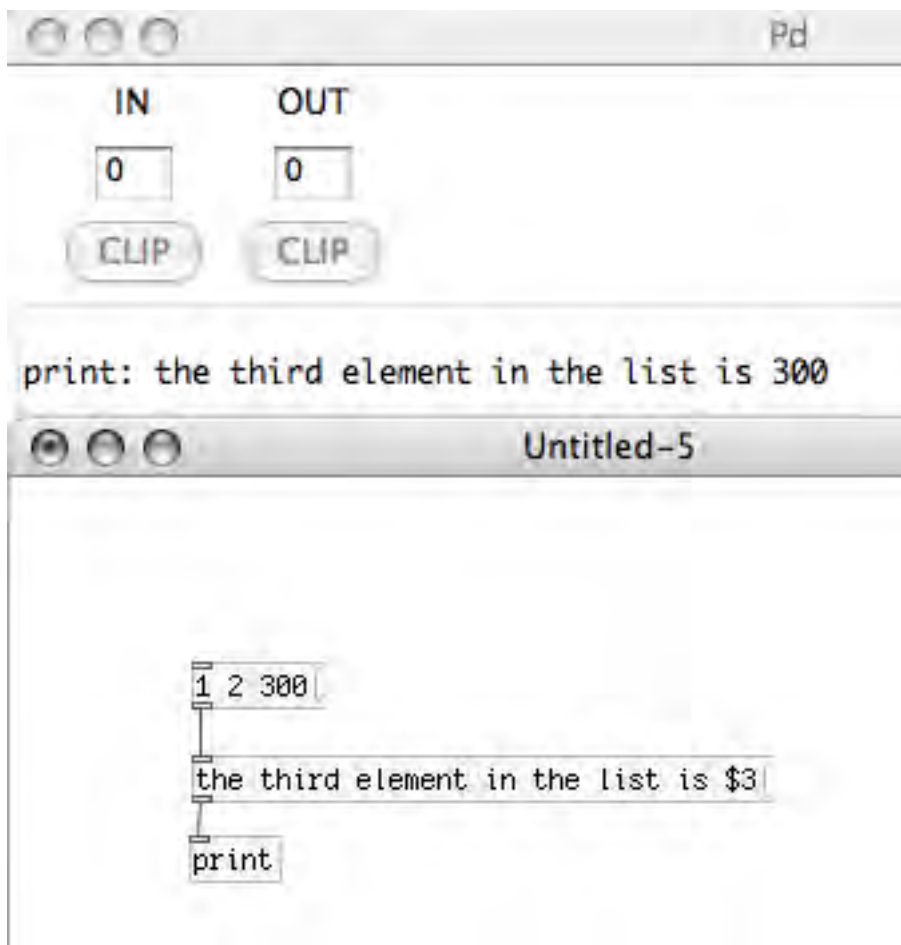
- What is a message?
  - What is an object?
  - Name two types of data
  - What does bang do?
  - What is the output of a toggle?
  - What is the difference between a hot and a cold inlet?
  - Name two fundamental properties of acoustic sound and describe how they are measured.
  - What is the difference between the Nyquist Rate and the Nyquist Frequency?
  - A frequency of 500 cycles per second is sampled at a rate of 600 samples per second. What frequency will be recorded?
- 

## 2.6.12 Lists and arrays

### Lists

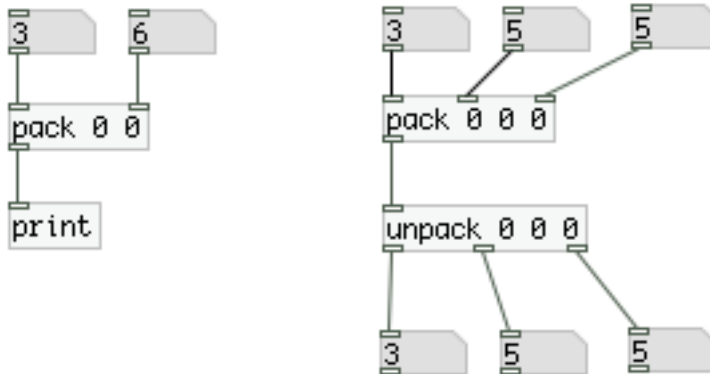
Sometimes you need to store and recall more than one number. Lists of numbers can be written into a message box. We can retrieve individual numbers from a list in many ways. The easiest way is to use dollar signs (\$).

When you place a dollar sign in a message box, followed by an index number, it acts as a placeholder. The dollar sign is then replaced by any incoming data with that index. For example, if a list has three elements, and you send the list to a message box with '\$3' written in it, it will spit out the third element in the list, as in the example below. Notice the printed output in the PD window.



This is a great method for getting numbers and words out of lists. However, it is not very efficient if you need to change numbers in a list dynamically. Luckily, there is a special object called [pack] that allows us to create dynamic lists.

[pack] takes initialising arguments that define the starting values for all elements in the list. The object [pack 2 4 6 8 10] creates a list with 5 elements that are initialised with the values 2 4 6 8 and 10 respectively. The object will therefore have 5 inlets, one for each element in the list. Importantly, **only the left inlet is hot**. This means that although data arriving at the inlets will change corresponding values, they will not be output unless inlet 1 (the leftmost inlet) either receives a number or a bang message.



You can see in the above example that lists created by [pack] can be dynamically unpacked using the [unpack] object. This is a great way of building and accessing numerical data in lists effectively. However, it must be noted that the [pack] and [unpack] objects only handle numerical data. If you want to make a list using text, you should use the dollar sign method.

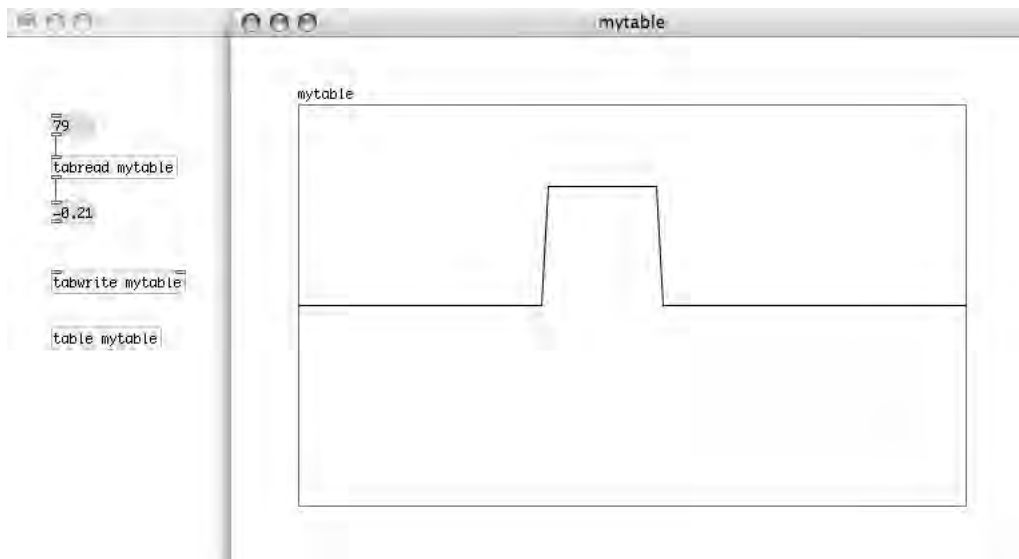
## Using arrays

You can store numbers in an array and recall them by using an index.

Arrays are stored in [table] objects, and can be filled with data by using the [tabread] and [tabwrite] objects. You can also put data in an array by clicking and dragging in the display with the mouse.

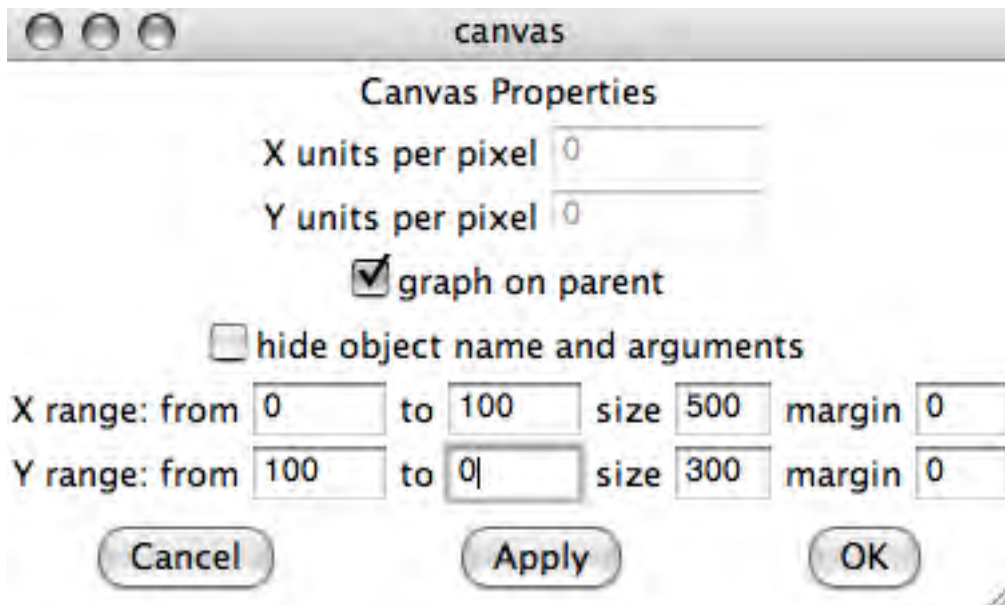
Create a [table] object, remembering to give it a name as an argument, like [table mytable]. Also create a [tabread mytable] object and a [tabwrite mytable] object. The argument names are very important. They link the [tabread] and [tabwrite] objects to your array which is now stored in the [table] object. Now create two number boxes, and connect them to both the inlet and outlet of the [tabread mytable] object. The number in the top of [tabread] can be used to ask for the value that is stored at a particular index in the array.

Now click on the [table mytable] object. This will open a window with the array inside. You can enter values into the table by clicking on the line and dragging with the mouse.



Now use the number box connected to the inlet of the [table mytable] object. You can see the result appear in the number box below.

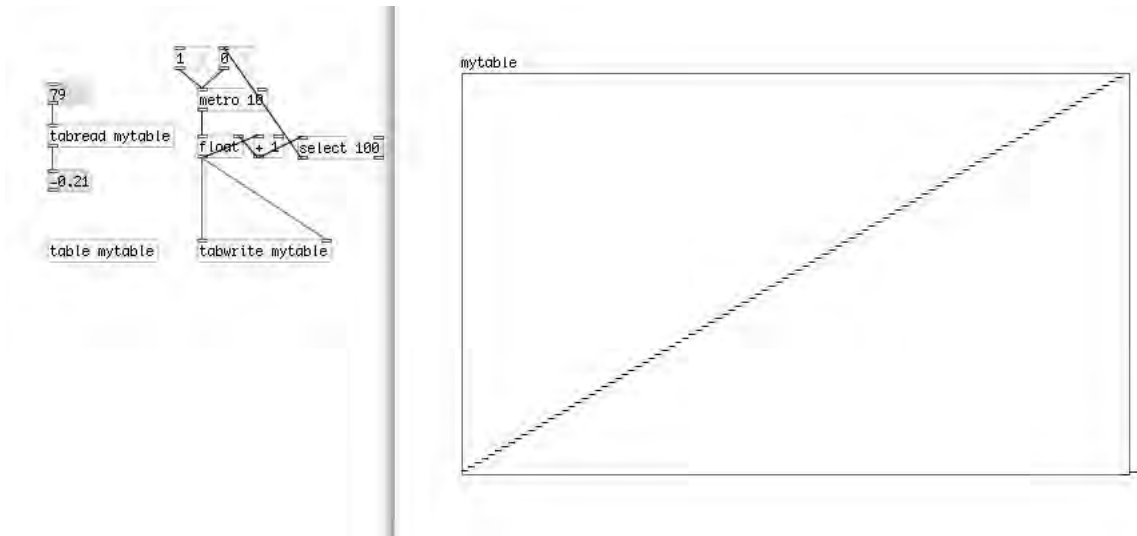
By default, arrays in a [table] object can store up to 100 floating point values on its x-axis. These numbers can be any floating point value, but by default, the graph will only contain numbers between 1 and -1. If you want to change the size of the array, you can do so easily by right clicking on it and selecting 'properties'. This brings up two windows, one called 'array' and one called 'canvas'. Using the canvas window, change the y-axis range to values between 100 and 0. It should now look like this:



Click 'Apply', and then 'OK'. Draw some values into the table with the mouse. You should find that the range of data that the table can display is now much greater.

We can fill an array with data by using the [tabwrite mytable] object. [tabwrite] has two inlets. The left inlet takes the value you want to store, and the right inlet takes the index. We can fill the array with values by using a counter. In the example

below, the counter is used to create a stream of numbers between 1 and 100. When it reaches 100, it stops. These numbers are used as both the value and the index. This creates the diagonal line we see displayed in the table.



Copy this example and press the '1' message connected to the [metro] object. You should get the same result. You will then be able to retrieve the values using [tabread].

You do not need to use [table] to create an array. If you like, you can just go to the Put menu and select 'Array'. You can set the size and properties of the array in exactly the same way, and arrays created in this manner will still work with [tabread] and [tabwrite] objects.

---

### Learning activity

Arrays are a great way of making music in PD. Can you think of a way of using an array to play back a sequence of events? Modify your previous sample player so that it uses an array to play back your sounds. Hint: you will need a counter and a [select] object, just like before, but they may need to be configured differently.

---



---

## 2.7 Sound synthesis

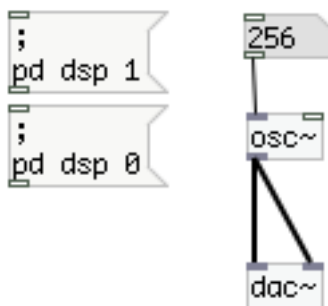
One of the best ways of making sound and music in Pure Data is through sound **synthesis**. You can use PD's basic synthesis components to carry out any number of sound creation tasks, using a variety of different methods. In this section, you will learn how to create synthesised sound of varying complexity by emulating well-known synthesis techniques, some of which have been used in sound and music for over 100 years.

## 2.7.1 Oscillators and waveforms

All sounds are made up of combinations of sinusoids (sine waves). It is therefore theoretically possible to synthesise any sound by combining sine waves with constantly varying amplitudes. This form of synthesis is known as **Fourier synthesis**, after the mathematician and physicist, Joseph Fourier (1768–1830). Fourier synthesis is often referred to as **resynthesis**, and normally requires an analysis technique known as **Fourier Analysis** to be performed first. For more information on Fourier analysis see Section 4.3.3 of *Creative Computing 2: interactive multimedia Vol. 1*, and Section 2.3.1 of *Creative Computing 2: interactive multimedia, Vol. 2*.

A far easier way of creating sounds is to start with waveforms that model the basic qualities of natural sounds. In reality this is incredibly difficult to do, as the number of frequencies and their varying amplitudes can be deceptively difficult to model. However, by combining waveforms in interesting ways, we can create a number of excellent sounds, many of which are impossible to produce in the natural world.

In PD, the most basic synthesis object is the sine wave oscillator, or [osc~]. This object creates a test tone with a frequency determined either by an argument, or by a floating-point number in its left inlet. Connect an [osc~] to both inlets of a [dac~] object to ensure that sound will come out of both speakers. Connect a number box to the left inlet of [osc~] and start the DSP using the message {;pd dsp 1}. You can just copy and paste this from your sampler patch if you want. You should end up with something like this.

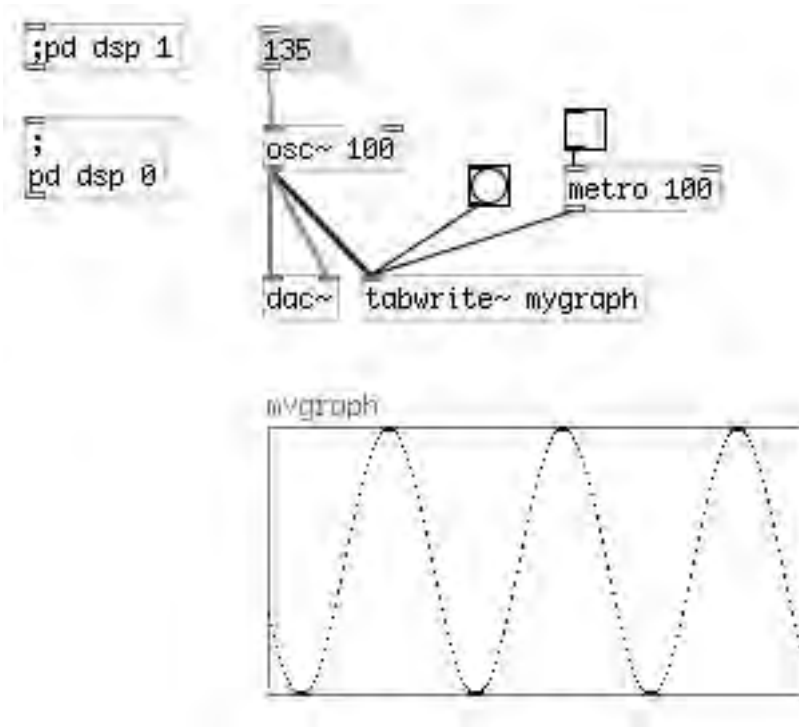


Use the number box to change the frequency and listen to the result. You should be able to tell that as the frequency increases, the tone of the oscillator gets higher in pitch.

All musical pitches relate directly to specific fundamental frequencies. For example, concert A, the note that an orchestra most often uses as a reference, is equal to 440 Hz. Similarly, middle C (MIDI note number 60), normally considered the middle of a piano keyboard and staff system, is equal to 261.626 Hz.

At this point it would be great if we could graph the sounds we are making. We can use an array to do this. All we need to do is create an array, and send our sound signals to an object called [tabwrite~]. Notice that this object is essentially the same as the [tabwrite] object, but for signals. Do not forget to name your array.

When [tabwrite~] receives a bang, it will graph the signal going into it. Create a bang button and connect it to the input of [tabwrite~]. You can automate the bangs with a [metro] object, but take care not to send the bangs every millisecond, as this will waste valuable CPU power. A timing interval of around 100 should be fine. Now your patch should look like this:



Here, I have changed the dimensions of the graph array to make it smaller. I have also increased the number of values that it displays on its x-axis from 100 to 1000. Note: you must change both the size of the canvas and also the size of the array, otherwise it will not work!

Notice that when you change the frequency of the [osc~] object, the width of the waveforms in the display changes accordingly. You can see that lower frequencies are wider than higher frequencies. This is because lower frequencies have longer wavelengths.

We will continue to use this graph throughout the rest of the examples. We may also occasionally change its parameters to allow us to view sounds differently.

## 2.7.2 Amplitude

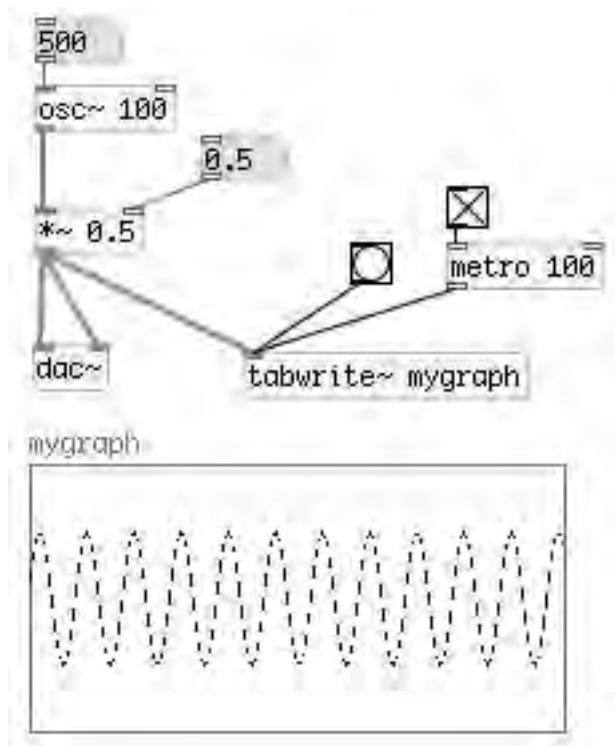
So far we have learned how to control the frequency of an oscillator and graph its output. However, we have as yet no way of controlling the amplitude of the oscillator.

By default, all oscillators output their signal at maximum possible amplitude. We can see this by looking at the graph, as we know that it is displaying values between 1 and -1, and the oscillator is peaking at the top and the bottom of the graph.

We need a method of lowering the amplitude of the signal. Most people imagine

that if we subtract a value from the signal that it will lower its amplitude. This is not the case. We could use a `[- 1]` object to do this, but this would have the function of changing the signal from an oscillation between 1 and -1, to an oscillation between 0 and -2. This can be very useful. However, it would be unwise to send such a signal to a speaker, as they are incapable of responding to signals beyond 1 and -1 for prolonged periods without being damaged.

When we want to change a signal's amplitude, we are trying to scale the values it contains. The best method of scaling a range of values is to multiply them. For this reason, we should use the `[*~]` object to reduce the amplitude of a signal, as in the example below.



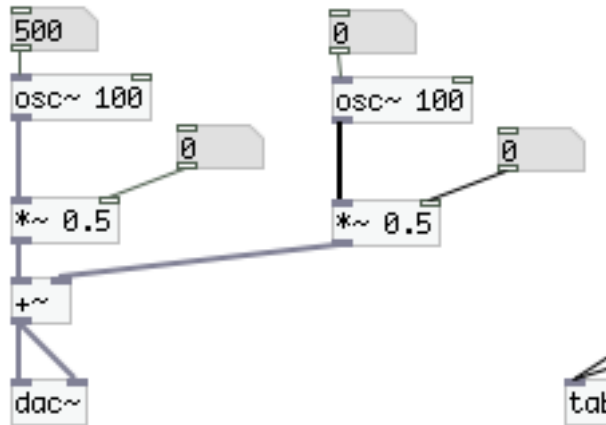
Try altering the value in the number box attached to the `[*~]` object by shift clicking and dragging. Notice what happens to the signal. At this point, you may have unintentionally multiplied the output signal by a value greater than one. This is not a great idea. If you start to panic, just hit the `{;pd dsp 0}` message.

There are times when multiplying a signal by greater than one is a good thing to do. However, you should avoid doing this to a signal that is being sent to a speaker unless it has already been turned down somewhere else. We can mitigate against this sort of problem by using the `[clip~]` object with the arguments -1 and 1 to constrain the signal between these two values.

### 2.7.3 Adding signals together

When we add two signals together, they combine in interesting ways depending on their frequency content. Add a second oscillator to the patch, but do not connect it to the `[tabwrite~]` object just yet. Try to think of a way of adding the oscillators together. If you manage to do this correctly, that section of your patch should look

something like this.



Notice that both oscillators are multiplied by 0.5 before being added together. This is important, as when we add sounds together their amplitudes are summed, so the output oscillates between 2 and -2. Therefore we must scale them first. Here we have scaled them both by the same amount, halving their values so that when we add them together with the `[+~]` object, the output oscillates between 1 and -1. Also notice that the output of the `[+~]` object is going to both inlets of the `[dac~]` so we can hear the sound in both speakers.

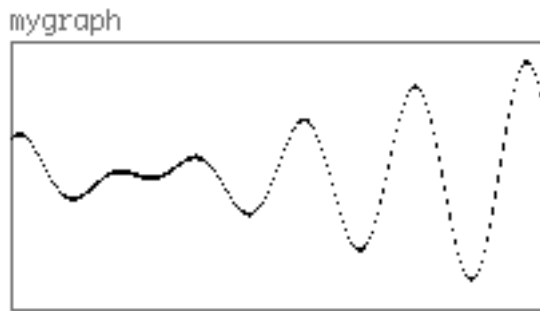
Now, change the frequencies of both oscillators to 200 Hz. You should hear one tone. Now, change one of the oscillators to a frequency of exactly 201 Hz and listen to the effect. What is happening? You may have expected to hear two separate tones, but you do not. This is because there is very little difference between a signal of 200 Hz and a signal of 201 Hz. However, there is enough of a difference to create a rather strange effect.

## Beating

You should hear that the tone is fading in and out. Look at a clock or watch with a second hand and work out how often the sound fades in and out. You should hear it happening once a second. This interval is called the 'beat frequency'. When two tones of similar frequency are combined, they 'beat' together, noticeably interacting to create a third 'beat' frequency. The beat frequency is always equal to the difference between the two summed frequencies. In this case the difference is 1 Hz – one cycle per second.

Connect the outlet of the `[+~]` object to the inlet of the `[tabwrite~]` object and take a look at the graph. You should be able to see the changing amplitude created by the frequencies beating together. Now, slowly change the frequency of the second oscillator from 221 to 222 Hz (use the shift key). The beat frequency will change from 1 Hz to 2 Hz.

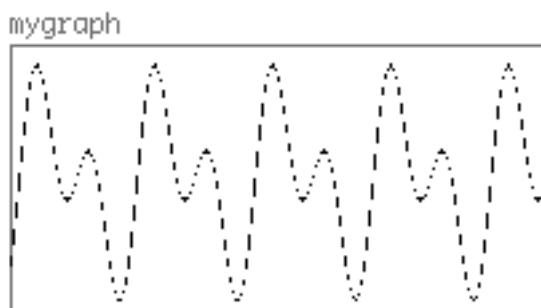
Now, slowly increase the frequency once more, 1 Hz at a time until you get to 240 Hz. You should now hear the frequency getting higher in pitch, as you would expect. However, you should also notice that the beat frequency is increasing. At 240 Hz, the beat frequency is 20 Hz, and should look something like this.



20 Hz is the lower boundary of the human hearing range. Beyond this point, depending on the quality of your speakers, you should begin to hear the beat frequency as a third audible tone increasing in pitch as the frequency rises. This is called **heterodyning**. The audible tones caused by beating are also known as **difference tones**, as they are equal to the difference between the two frequencies. Now increase the frequency further, slowing down as you approach 440 Hz.

### Harmonics

Just before you reach 440 Hz, you should be able to hear some interesting changes in the audio signal. You will hear the beating again, but this time, you will not hear much change in the amplitude. Instead, the quality of the sound changes. Stop at 439 Hz, and listen to the speed at which the changes occur. It is 1 Hz – once per second. Now look at the graph. You should be able to clearly see the changes in the oscillation of the sound. Finally, move from 439 to 440. The beating will stop, and the shape of the waveform will stabilise. It should look something like this.



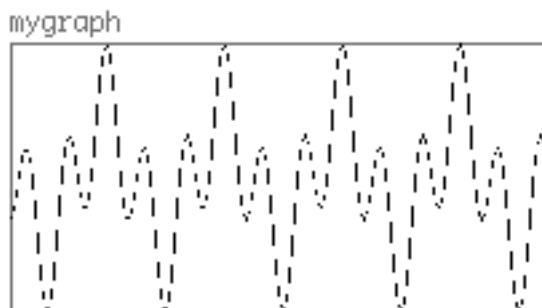
Notice that the waveform has two peaks, and that these peaks are equally spaced. This is because the second oscillator is producing a frequency that is exactly twice that of the first oscillator. They are in a mathematical relationship that can be described in terms of integer ratios, combining to produce what we call a **harmonic** sound. Furthermore, because the second frequency is a multiple of the first, we can refer to it as a **harmonic**. More specifically, as the second frequency is exactly double the first, being the first multiple, we refer to it as the **first harmonic**.

The pitches produced by both waveforms sound similar. You may not actually be able to tell that there are two pitches at all. Technically, it could be argued that they are of the same pitch class, and if we were to play them on a piano keyboard, they would both be the note A, spaced exactly one **octave** apart. In MIDI terms, they are note numbers 57 and 69. The difference between 57 and 69 is 12. This is the number of musical notes in an octave in the Western music tradition. Importantly, this is only the number of notes in this particular case, other musical traditions divide the octave into different numbers of notes.

Western musical scales do not normally feature all twelve notes of the system, instead traversing each octave in eight steps, with the eighth note being of the same pitch class as the first. Crucially, the Western music tradition is based on the premise that a doubling of frequency is equal to an octave, with the first and eighth note being of the same type. The fact that you may not be able to detect these two pitches as separate is testament to the power of the pitch class system.

### The harmonic series

If we now multiply the first frequency by three, we will get the second harmonic –  $220 * 3 = 660$ . Notice the affect this has on the waveform, and try to pay attention to the sound.

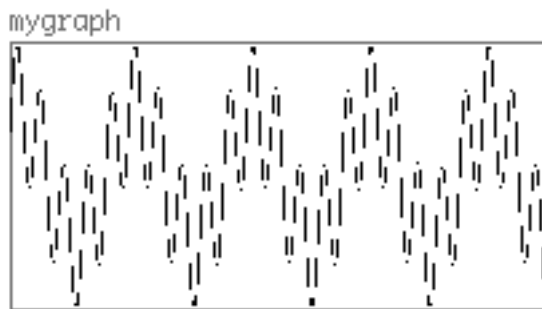


This waveform shares some characteristics with the previous one. Again, there is no beating, the sound is harmonic and the signal is stable. However, you can see from the waveform graph that the combined signal has three regular peaks. This is significant, as what we are hearing is the fundamental frequency, combined with the second harmonic. Also, you can tell that there are two distinct pitches of different types, whereas before, even though we knew there were two frequencies, it only sounded like a single pitch.

The second harmonic produces a pitch equivalent to the musical interval of a fifth above the first harmonic. In this case, the second harmonic is almost equivalent to MIDI note number 76, or the note E. From this point on, the exact mathematical relationships differ in small ways from the Western musical tuning system for reasons explained elsewhere (See Section 2.3.2 *Creative Computing 2: interactive multimedia, Vol. 2* for more information.)

If we add together the fundamental and the third harmonic, you will see that there are four peaks in the waveform, and the pitch will be two octaves above, being exactly four times the fundamental. If we add the fourth harmonic instead, the

waveform has five peaks, and the interval is almost exactly what is referred to in the Western music tradition as a **major third**.



Here you can see clearly why the peaks in the waveform increase as we ascend through the harmonic series. In this case, the fourth harmonic has a frequency exactly five times that of the fundamental, so for every cycle the fundamental makes, the fourth harmonic makes five.

Importantly, if we combine the pitch class results from the first four notes in the harmonic series inside one octave, the result is a major chord, which is found in many musical systems throughout the world. In this way, we can think of musical sounds as those that mainly contain frequencies that are close to or exactly mathematically related, and many real-world instruments create sounds that are mainly composed of harmonics. However, the effects of beating are quite beautiful in their own right, and could be described as highly musical despite their complexity.

Exercises:

Create a patch that repeatedly ascends through the first 10 harmonics of a 100 Hz Fundamental.

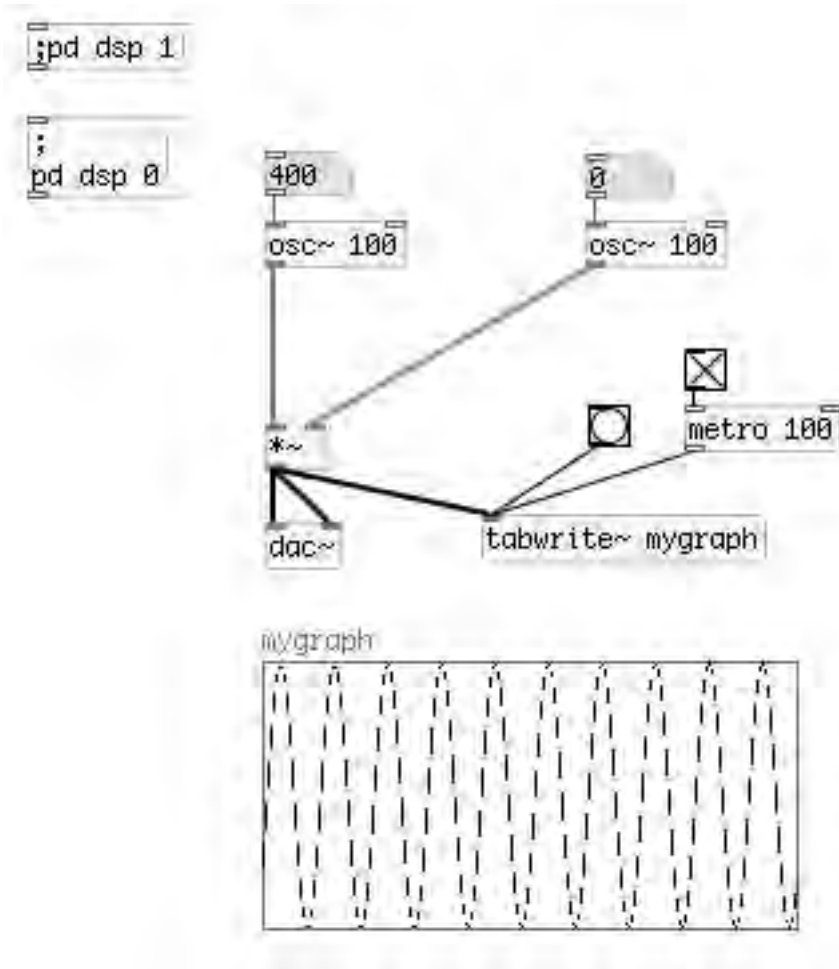
Create a patch that plays the fundamental and first four harmonics, so that the fundamental can be dynamically changed whilst the harmonics remain in ratio. Hint: use [\*] to set the values of several [osc~] objects.

Try loading sounds into [readsf~] and graphing them in your array. Look at the waveform and see if you can spot when they have mathematically related frequency content. It will be easiest to do this with recordings of solo acoustic instruments, particularly woodwind instruments like flute and/or clarinet. Alternatively, have a look at the sound of someone singing. What can you tell about it by looking at the waveform?

## 2.7.4 Multiplying signals together

Instead of adding signals together to synthesise sound, we can multiply them. Doing this is normally referred to as **modulation**, and it is used for all sorts of reasons. Create a patch with two [osc~] objects. This time, instead of scaling and then adding them together, just multiply them with a [\*~] object and send them to the [dac~] and the array so we can see the waveform. Your patch should look like

the one below.



**Amplitude Modulation** Now you should set the frequency of the first oscillator to something reasonably high, such as 400, otherwise the affect might be difficult to hear. Now, starting from 0, slowly turn up the frequency of the second oscillator. Stop at 1 Hz. You should hear that the sound is being faded in and out.

This may appear to be similar to the affect of adding sounds. However, there are two crucial differences. First, you cannot as yet hear the frequency of the second oscillator, as it is well below the audible frequency range. Secondly, the speed of the modulation is 2 Hz, not 1 Hz. This is because the first oscillator is being attenuated by the second. As the second oscillator has a positive and negative phase, this attenuation happens twice per cycle, with the original oscillator being initially modulated by the positive phase of oscillator 2, and then by the negative phase.

Now raise the frequency of oscillator 2. At first, you will hear a tremolo affect, with the speed of the tremolo increasing as you raise the frequency. As you approach the audible frequency threshold (20 Hz), you will notice that the sound begins to change. You will start to hear two separate tones, one ascending and one descending.

This is a form of amplitude modulation known as **ring modulation**. In the case of

ring modulation, the frequencies you hear are equal to the carrier frequency (oscillator 1) plus and minus the modulation frequency (oscillator 2). For example, if you ring modulate a 400 Hz sine wave with a 100 Hz sine wave, the tones you will hear will be 300 Hz and 500 Hz. In this way, ring modulation has the same properties as aliasing.

A slightly more advanced form of amplitude modulation, known as classic AM, requires that the amplitude of the modulating waveform be adjusted so that it oscillates between 0 and 1 as opposed to -1 and 1. This should be a simple task. It changes the sonic characteristics of the output considerably. Using a [`*~`] object and a [`+~`] object, implement this form of AM. Can you describe the ways in which it differs from RM?

---

### Learning activity

Ring modulation sounds especially interesting on acoustic signals. Give it a try using [`readsf~`] as the carrier. The composer Karlheinz Stockhausen (1928–2007) used the technique extensively, particularly on vocal works such as *Microphonie II*. He also claimed to have invented the technique.

Does this make aliasing occasionally creatively desirable, despite being bad practice from an engineering perspective?

---

## 2.7.5 Frequency modulation

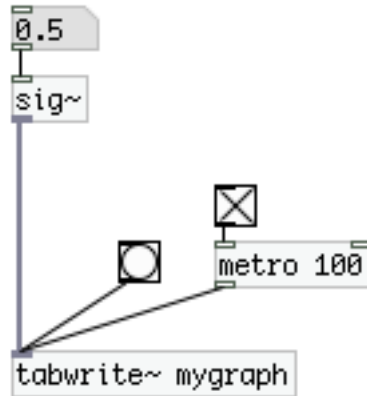
Sounds have amplitudes. They also have frequencies. We have learned that there are many creative applications for amplitude modulation. This is equally true of the following technique, **frequency modulation** (FM), discovered for musical purposes by John Chowning at Stanford University in 1967 (Chowning 1973).

To create FM, we have to modulate the frequency of oscillator 1 (the carrier), with the frequency and amplitude of oscillator 2. To do this properly, we should control the frequency of oscillator 1 with a signal, using an object called [`sig~`].

[`sig~`] takes any floating point number and turns it into a stable signal value. Have a look at the graph below. The [`sig~`] object is creating a constant signal of 0.5. It has no audible frequency, as it is not oscillating.

```
pd dsp 1 remember to turn the dsp on.
```

```
pd dsp 0
```



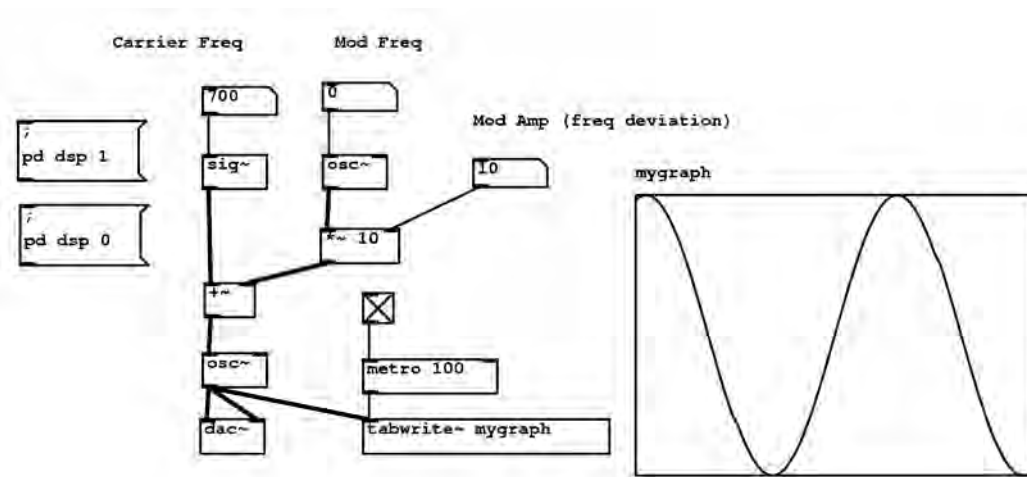
mygraph



This line is at 0.5

We can use [sig~] for all sorts of things. Most importantly, we can use it to control an oscillator. In addition, we can combine the output of [sig~] with other signals.

Create a patch with two [osc~] oscillators, both with their own amplitude controls ([\*~]). Connect oscillator 1 (the carrier) to the [dac~]. Connect oscillator 2 (the modulator) to a [+~] object that takes the output of a [sig~] as its first input. You should end up with something like this.



Now, give the carrier a frequency that you can comfortably hear, somewhere between 200 and 750 Hz depending on your speakers. Now, slowly increase the frequency of the modulator (i.e. the **modulation frequency**), until it is around 0.3 Hz (holding down shift). You should be able to hear the pitch oscillating up and down around the carrier frequency ( $\pm 10$  Hz). You should also be able to see the sine wave squeezing and stretching in the window, as the distance between each peak is modified by oscillator 2.

Click on the number box going into the modulator's amplifier ( $[*~ 10]$ ). Slowly turn the amplifier down to 0. As you reduce this value, you will hear the range of the modulation decreasing until the pitch is steady. Now slowly increase the value again until it is back at 10. You will hear the pitch modulating once more. In FM terminology, the amplitude of the modulating signal is called the peak frequency deviation (Chowning 1973, p. 527).

Now, slowly increase the modulation frequency and listen to what happens. You may be surprised by what you hear. The sound will begin to stabilise from a constantly varying pitch to a stable pitch once you reach the audible frequency threshold of 20 Hz. In addition, as you continue to increase the modulation frequency, you should find that the frequency content of the sound remains quite complex.

Finally, slowly increase the frequency deviation (the amplitude of oscillator 2) and listen to what happens. You will hear new partials becoming audible in the sound.

Now, set the carrier frequency at 400 Hz, and the modulator at 800 Hz – the first harmonic. You will realise immediately that the sound is now stable and ordered. Now adjust the frequency deviation. You should hear that the sound gets brighter or duller with the rise and fall of the frequency deviation, but the sounds remain harmonic.

In a similar way to amplitude modulation, FM synthesis produces sidebands around the carrier frequency. However, FM does not only produce a single pair of sideband frequencies, as in the case in AM, but instead produces sidebands at the carrier frequency plus and minus **multiples** of the modulator frequency:  $f_c + kf_m$  and  $f_c - kf_m$  where  $f_c$  is the frequency of the carrier,  $f_m$  is the frequency of the modulator, and  $k$  is an integer greater than zero.

The amplitude of individual partials (the carrier frequency itself as well as all sideband pairs) are calculated using the modulation index,  $i$ . The modulation index is defined as the ratio between the frequency deviation,  $d$ , and the modulation frequency:  $i = d/f_m$ . Scaling factors for partials (relative to the amplitude of the carrier) are determined using the modulation index and the set of Bessel functions  $B_n(i)$ . For example,  $B_0(i)$  determines the scaling factor for the carrier frequency,  $B_1(i)$  the scaling factor of the first pair of sidebands,  $B_2(i)$  the scaling factor for the second pair of sidebands, and so on. As a rough guide, the number of audible sideband pairs in an FM generated signal can be predicted as  $i + 1$  (Miranda 2002, pp. 26–8).

If the modulator is a harmonic of the carrier frequency, or if they are both harmonics of a lower fundamental, the sidebands will all be from the harmonic series, creating an ordered and stable sound texture. If you adjust the frequency deviation you will hear frequencies along the harmonic series being added and subtracted from the sound. Likewise, when the carrier and modulator are not harmonically related, you will experience inharmonic bell-like tones.

FM tends to be best suited to creating rich, textured sounds such as bells, and metallic noises. It is also a highly expressive synthesis method, and is very useful in a variety of electronic music contexts, particularly in combination with other synthesis approaches.

---

### Learning activity

The object [mtof] takes midi note numbers and converts them to their frequency equivalents. Build an FM synthesiser controlled by MIDI note information, either from a MIDI device, or alternatively from a series of note-number messages triggered by [key] objects.

Create three arrays and use them to alter the carrier frequency, modulation frequency and peak frequency deviation respectively. Automate this process with [metro] and a counter.

---

## 2.7.6 Playing back sampled sound with line

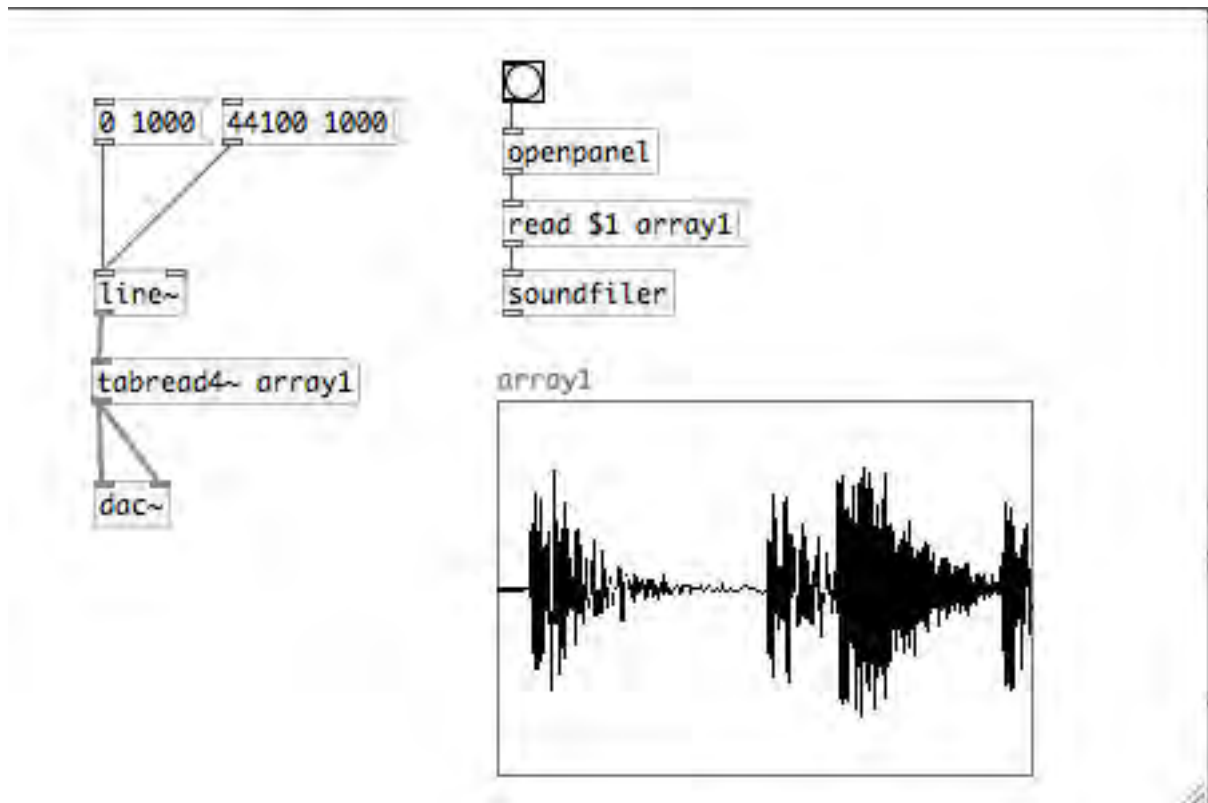
You can play back the contents of an array at signal rate by using the [tabread4 ] object and a control signal known as a **signal ramp**. You can provide the name of the array as an argument to [tabread4 ] (just like [tabread]). [tabread4 ] will allow you to treat the contents of the array as a waveform. In this way, you can use any data you like as a soundwave!

The simplest form of signal ramp in PD is called [line ]. It creates a signal ramp between its current value and a new value, over a period of time specified in milliseconds.

So, if you want to play the first second of a sample over the period of one second, you would pass the message '44100 1000' to [line ], and plug this object into [tabread 4]. If you then passed the message '0 1000' to your line object, it would play the sound backwards from its current position, to sample 0, in one second. Likewise, if you then passed the message '88200 10000', it would play the first two seconds of the sample forwards over a period of 10 seconds. As a result, it would

play back at 1/5th of the speed. Can you understand why?

You can load sampled sound from disk into an array by using an object called [soundfiler]. [soundfiler] takes a message to read samples from anywhere on your computer and passes all the data into an array. So if your array is called **array1**, all you need do is pass the message 'read (insertsamplelocation) array1' to [soundfiler].



### 2.7.7 Enveloping

You can use array data to control any signal, including the amplitudes of signals (by of course using the [\*] object). In this way, you can control the volume of your sounds over time with data in an array. This can be done even more simply by using the [line] object to create ramps between signal values of 0. and 1. You can then use these ramps to fade a sample or signal of any kind between 0. and 1. and back again over any period of time, simply through multiplying your signal ramp with your carrier signal.

Controlling the way in which a sound fades up and down like this is often referred to as envelopes. Crucially, you should not restrict the use of envelopes to the control of volume alone. As we have already seen, they can be easily deployed to control the playback position and speed of a sample stored in RAM.

### 2.7.8 Using the VCF object

Once you are used to using envelopes to control frequency and volume, you should try using them to control a filter (you will have covered the mathematics of filtering in *Creative Computing 2: interactive multimedia*).

The easiest filter for creative use in PD is called [vcf]. This allows you to attenuate certain frequencies in a signal above the cutoff frequency specified in Hz. In addition, you can selectively adjust the filter with the q function. This increases the amplitude of frequencies around the centre frequency. The value of q itself is the ratio of the centre frequency to the upper and lower frequency bands boosted to a level of at least -3db below the amplitude of the centre frequency. This means that all frequencies around the centre frequency are boosted. The q simply describes the width of the curve as a ratio. This width is adjustable with the [vcf] object, and it is this that gives it its distinctive sound. The vcf is the central component of the subtractive synthesis model, allowing you to control the frequency content of your synthesised sound.

You should now be able to create additive, wavetable-based, and subtractive synthesisers. You should also be able to sequence them and arrange them using signal rate controls.