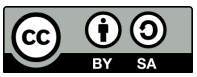
INTRODUCTION TO MATLAB (OR OCTAVE) BY APPLICATION TO MUSICAL ACOUSTICS

© JONATHAN A. KEMP 2012

University of St Andrews, Beethoven Lodge, 65 North Street, St Andrews, Fife, KY16 9AJ, UK



Contents

E-mail address: jk50@st-andrews.ac.uk.
Date: August 28, 2012.

Introduction

MATLAB is a computer program that lets you calculate things in a way that is very intuitive. You can use it as a simple calculator, but the best thing about MATLAB is the fact that you can start without knowing any computer programming techniques and in no time you will be writing programs to do numerical computation. All computer-programming languages involve learning some fiddly commands, but MATLAB keeps this to the absolute minimum, meaning that you have more time for trying out different ideas.

While MATLAB is a commercial piece of software, Octave is a free competitor featuring much of the same functionality. In this work the commands described can be applied successfully in both MATLAB and Octave. If you download and install Octave, just pretend the word MATLAB is replaced with Octave for the remainder of the work unless you are told otherwise.

Why is everything not written in MATLAB? The simple answer is that more traditional programming languages, such as C, C++, Java and so on may be optimized to run complicated tasks faster once a program has been written. If you are beginning a project and want to maximize the time spent doing things that get you results that can be put into a report, though, then MATLAB is for you.

This work assumes a basic working knowledge of high school mathematics but nothing more and acts as an introduction to computer programming by illustrating how it could be used to create results for a project report. In particular, the field of Musical Acoustics is used as an example wherever possible. This area has the advantages of being scientifically interesting and relatively cheap to run experiments in. Computer programming, mathematics and music (as well as physics and digital signal processing) are all topics that will seem more accessible through working through this booklet.

1. The Basics: MATLAB as a calculator

Lets begin by opening MATLAB or Octave and considering it to be a glorified calculator. You will see the command prompt which looks like:

```
>>
```

in MATLAB or

```
octave-3.2.3:1>
```

in Octave. I will show commands that should be typed into the command line with

```
>>
```

in front but all should work on Octave as well as on MATLAB. Try typing 1+1 into the command prompt and hitting return. We should see:

```
>>1+1
ans =  2
```

As well as addition using +, we can do subtraction using −, multiplication using ∗, division using / and to the order of using ^ (which may be obtained using Shift 6 on the computer keyboard).

Try typing the following text just after the command prompt and check you get the same answers:

```
>>9 - 6
ans = 3

>>3 * 2
ans = 6

>> 28/4
ans = 7

>>2^3
ans = 8
```

1.1. **Example: Using the speed of sound.** As an example of practical interest lets think about sound travelling through the air. A drummer, standing 85 metres away, beats a drum. How long after we see the drummer's stick striking the drum do we hear the beat?

We will assume here that the speed of sound is $c = 340$ m/s (although in real life it depends on the temperature and humidity etc.). Assuming that light travels instantly, we can work out t, the time taken for the sound to travel using:

$$t = \frac{d}{c} \tag{1}$$

where $d = 85$m is the distance between us and the drummer. In MATLAB can put in the numbers and do the calculator work in the command line using:

```
>> 85/340
ans = 0.2500
```

Note that, as with a calculator, there are no units shown in MATLAB. On paper we would write the answer as:

$$t = \frac{d}{c} = \frac{85}{340} = 0.25\text{s}$$

This means that it takes 0.25 seconds, or a quarter of a second, for sound to travel from the drum to us. We hear the beat a quarter of a second after we see the drummer strike the drum.

Exercises 1. 1.1). We are standing 34 m from the flat wall of a castle and we sing a brief note. How long after we start the sound do we hear the echo? You should assume that the speed of sound is 340 m/s. Hint: Consider how sound has to travel from us the the castle and back again.

1.2). We see a guitarist strumming a chord and hear the chord 0.3 seconds later. How far away is the guitarist? Hint: Rearrange equation (1).

1.3). A gong is sitting against one wall of a corridor. When the gong is struck echoes are heard repeating 180 times a second. How far apart are the walls? Hint: The time taken for echoes in seconds must be 1 divided by the number of echoes per second.

1.4). An echo unit works by recording a sound on magnetic tape at one location and then playing it back at another location, 5 cm away. If the magnetic tape travels at a speed of 2.5 cm/s, how long will it take for echoes to appear? Hint: Use the speed that the tape is moving at instead of the speed of sound in air.

1.2. **Brackets.** You can also use brackets. The commands are interpreted in the usual way for mathematics which may be remembered using the mnemonic BODMAS (calculations inside Brackets are calculated first, to the Order of command are calculated next, followed by Division and Multiplication, then followed by Addition and Subtraction).

As an example try:

```
>>(80 + 18)/(2*3 + 1)^2
ans =  2
```

which is the same as:

```
>>98/(7^2)
ans =  2
```

This would mean we could calculate the answer to Exercise 1.1 above using

```
>>(34+34)/340
ans = 0.2000
```

to give us the answer of 0.2 s.

1.3. **Using sqrt, the built in function for square roots .** MATLAB features a large number of built in functions. Our first example of a function is called "sqrt" and calculates the square root of a number. As an example try the following:

```
>> sqrt(9)
ans = 3
```

Note that we have to put in the brackets around the number that we are taking the square root of and "sqrt 9" without the brackets gives an error message.

Square roots can be used where we have to work out distances involving measurements taken at right angles using Pythagorus' Theorem. For instance, if a choir is standing in a rectangular grid of four rows, each of 6 singers, how long does sound take to travel between the the furthest away choir members? Let's assume that the singers are separated by 70 cm from nearest neighbours in both the sideways and forwards directions. The distances along the along the rows and front to back can be worked out in meters using:

```
>> 4*0.7
ans = 2.8000
>> 6*0.7
ans = 4.2000
```

giving 2.8 m and 4.2 m respectively. We apply Pythagorus' Theorem to give the distance between furthest performers as:

```
sqrt(2.8^2 + 4.2^2)
ans = 5.0478
```

so rough 5.05 m. In order to calculate the time taken we have to take this number and divide it by the speed of sound as in equation (1). Rather than typing out the answer again, we can type "ans" in its place to give:

```
>> ans/340
ans = 0.0148
```

So the sound takes just less than 0.015 s or 15 milliseconds to travel between the furthest members of the choir. Musical ensembles of this size and bigger will

usually have a conductor beating time and part of the reason is to prevent timing differences between performers becoming noticeable.

2. Storing numbers: Variables

It is often useful to store number for use later. We have already taken advantage of number storage in MATLAB because when we use MATLAB as a calculator the answer can be reused using by typing "ans". In computer programming terminology we would say that when we use MATLAB as a calculator, it automatically creates a variable called "ans". Variables let us store numbers within computer programs and it is possible to assign number to letters in using the equals sign in a similar way to the use of letters in algebra.

In mathematics the quantity on the left of the equals sign and the quantity on the right of the equals sign must be the same. The left hand side and the right hand side may be swapped over without changing the meaning of an equation. Equals signs in computer programming are subtly different in that the item on the left hand side of an equals sign is set to contain the value calculated on the right hand side when the line is computed. In order to see how this works lets store the speed of sound in a variable name called c by typing $c = 340$ into the command line and hitting return. This results in the display showing:

```
>> c = 340
c = 340
```

We could also have a calculation done on the right hand side and a variable name on the left of the equals sign. For instance, to assign the distance calculation in section 1.3 to a variable called d we would type $d =$ then the calculation to give:

```
>> d = sqrt(2.8^2 + 4.2^2)
d = 5.0478
```

and check what variables we have stored by typing the command "who" and pressing enter. You should see something along the lines of:

```
>> who
Your variables are:
ans  c    d
```

showing that c and d are variables that have been defined. In Octave the text says "Variables in the current scope" rather than "Your variables are" but the command and the results are identical otherwise. We can then work out the time taken and store it is a variable called t by dividing the variables we have defined earlier by entering:

```
>> t = d/c
t = 0.0148
```

If we wish to remove all the variables we have stored in memory then we can do this using the clear command:

```
>> who
Your variables are:
ans  c    d    t
>> clear
>> who
```

After the clear command has been entered there are no variables left and the who command doesn't return us any text.

3. Script files

So far we have entered MATLAB commands one at a time using the command line. MATLAB can also run commands stored in separate files called script files. These files contain the text for the commands you wish to run and should be saved with a name ending in ".m" (for instance "myscript.m").

If you are using MATLAB then we can begin writing a script file simply by typing "edit" and pressing return (don't include the quotation marks). This makes MATLAB's built in text editor pop up. In Octave it may be necessary to use open a separate text editing program (many free ones are available).

After opening the editor, enter the commands:

```
clear
c = 340
d = sqrt(2.8^2 + 4.2^2)
t = d/c
```

into a new empty text document and save the file, with file being named "myscript.m" (again, don't include the quotation marks). Note that it is helpful to make your file names quite specific so that they don't clash with an existing functions built in to MATLAB (i.e. don't name it sqrt.m).

Now go back to the command line and type "ls" (which is short for list the contents of the current folder). If you have saved the file myscript.m in the appropriate place then the output should be:

```
>> ls
myscript.m
```

Type "myscript" into the command line (making sure you don't include the quotation marks or the ".m" on the end of the file name). The command line should then read:

```
>> myscript
c = 340
d = 5.0478
t = 0.0148
```

If you achieve this then well done, you have successfully run your first script file. Note that the command line output doesn't show the "sqrt" command that was used to compute the value of d, just the numerical result.

3.1. **Files and folders (or directories).** If there has been a problem running a script file because you are not in the correct folder or the file name is wrong then you will see an error message along the lines of:

```
>> myscript
??? Undefined function or variable 'myscript'.
```

if you are using MATLAB or

```
octave-3.2.3:10> myscript
error: 'myscript' undefined near line 10 column 1
```

if you are using Octave. Type "pwd" then enter and we should see something along the lines of:

```
>> pwd
ans = /Users/Jonathan/Documents/MATLAB
```

for the user called Jonathan on a Mac OS X computer, but the answer will of course depend on you and your system. The way to remember the command "pwd" is by remembering the acronym "Present Working Directory" where the word "directory" just means "folder". If, on the other hand, we know that we have stored the myscript.m file in "/Users/Jonathan/Documents/MATLAB" and the "pwd" command actually returns:

```
>> pwd
ans = /Users/Jonathan
```

then we know we need to change folders. This can be done using the "cd" command (where "cd" stands for "change directory") as follows:

```
>> pwd
ans = /Users/Jonathan
>> cd Documents/MATLAB
>> pwd
ans = /Users/Jonathan/Documents/MATLAB
```

or you can specify the full name of the folders. On a Windows computer this may be done with something along the lines of:

```
>> cd C:/Users/Jonathan/Documents/MATLAB
ans = C:/Users/Jonathan/Documents/MATLAB
```

where the "C:" is used by Windows as the "drive letter" for the hard drive used to store the data.

If you need to get out of a folder we can type "cd .." and press enter (which is known as changing directory to go into the "parent" directory) and if you want to go back to the directory that you started with just type "cd" and press enter (this is known as going to the home directory). If, on the other hand you are not sure where you saved the file "myscript.m" then go back to the editor and try selecting "Save As..." from the File menu. The main thing is that typing "myscript" into

the command line should work if the file "myscript.m" is in the present working directory.

3.2. **Semicolons to prevent printing results on the screen.** In computer programs we often want to make calculations several lines of computation happen to arrive at a single answer. In this case we don't want every line printing results on the screen. We can do this by typing the semicolon character ";" at the end of every line where we wish to hide the output. As an example, change the contents of myscript.m to be:

```
clear
c = 340;
d = sqrt(2.8^2 + 4.2^2);
t = d/c
```

Typing myscript into the command line will now give:

```
>> myscript
t = 0.0148
```

where the answer is now given with the output from the lines for c and d suppressed because of the semicolons on the end of the appropriate lines.

3.3. **Commenting code.** The text in a script file is a form of computer "code" and well written code should be easy to read and as easy to understand as is possible. It certainly helps if you make sensible variable names, but an essential part of making your code easy to understand is including comments that are actually ignored by the computer when the code runs but which explain in words what the other lines are for. In MATLAB anything written to the right hand side of a percentage sign "%" is ignored by the computer when the script is run. For instance try changing the contents of myscript.m to read:

```
%Clear all variables that have been stored before:
clear
%Store the speed of sound (measured in m/s) in a variable called c:
c = 340;
%Calculate the distance in meters using Pythagorus's Theorem and
%store this in a variable called d:
d = sqrt(2.8^2 + 4.2^2);
%Calculate the time taken for sound to travel distance d and store
%this in a variable called t and display the results on the screen:
t = d/c
```

Typing myscript into the command line will now give exactly the same answer:

```
>> myscript
t = 0.0148
```

but the code is now much easier for anyone to interpret.

4. Using sin, cos and tan, the built in functions for trigonometry

Trigonometry in MATLAB is done by typing sin, cos or tan as appropriate followed by angle values contained by a pair of brackets. The calculations are all performed assuming radians (rather than degrees) so that a full rotation is 2π radians. In MATLAB we can also access the value of π simply by typing "pi".Try the following examples:

```
>> sin(0)
ans = 0
>> sin(pi/4)
ans = 0.7071
>> sin(pi/2)
ans = 1
>> cos(0)
ans = 1
>> cos(pi/2)
ans = 6.1232e-17
>> cos(pi)
ans = -1
>> tan(pi/4)
ans = 1.0000
```

Note in particular that the exact answer for $\cos(\pi/2)$ is zero. Why then is MATLAB showing a value other than zero? The answer is that MATLAB is designed to do calculations numerically. There is a finite number of decimal points in the approximation of the value of π and a finite number of accurate decimal points in calculating the cosine function. The value of 6.1232e-17 (or 6.1232×10^{-17}) corresponds to the typical accuracy that MATLAB is working to. If MATLAB was prioritised to do symbolic computations instead of numerical computations then it could get the exact answer, but in practice tiny rounding errors like this are seldom a problem.

4.1. **Example: Diffraction for stereo loudspeakers.**

As an example of using trigonometric calculations inside a script, consider two loudspeakers in a stereo system are separated by a distance of L. Let's assume that a listener very far away from these speakers (in comparison to the distance between the two speakers) and is off to one side so that the sound from the speakers arrives out of phase. The distance between the listener and the two speakers will have a length difference of:

$$d = L\sin(\theta) \tag{2}$$

where θ is the angular position of the listener (in radians) with respect to the line of equal distance from the speakers (which is perpendicular to a line drawn between the speakers). Create a script, starting with the clear command, followed by definitions of the variables for the distance between the source (in metres) of

$d = 1.1$ and an angular position of listener (in radians) of $\theta = pi/4$ as well as the speed of sound (in m/s) of $c = 340$ to calculate the difference in the time taken for sound to travel from the two loudspeakers to a distant listener.

Creating a script for this involves both equations (1) and (2), for example:

```
%Clear any old variables (in case we use them by accident):
clear
%Speed of sound
c = 340;
%Anglular position (radians) of listener from line
%of equal distance from speakers:
theta = pi/4;
%Distance between speakers (metres):
L = 1.1;
%Calculate difference in distance travelled (metres):
d = L*sin(theta);
%Time difference (seconds) calculated using difference in distance:
t = d/c
```

Exercises 4. 4.1). Create a MATLAB script which calculates the distance between the loudspeakers if a distant listener is at an angular position of $\theta = \pi/8$ and the time difference between the loudspeakers is $t = 1$ millisecond.

5. Vectors

We have seen how variables are powerful containers for numerical values MATLAB. Experimental data, however, often comes in the form of lists of values, whether these are data points for a graph or a sampled sound wave. In order to store such information in MATLAB we need to use vectors. We can define a vector in MATLAB using square brackets and commas. As an example, lets manually type in a vector consisting of a ramp of values from 0 to 1 in steps of 0.1 and store it as x as follows:

```
>> x = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
x =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 8 through 11
    0.7000    0.8000    0.9000    1.0000
```

We can then use the letter x access any of the values stored in the vector using ordinary brackets as follows:

```
>> x(1)
ans = 0
>> x(2)
ans = 0.1000
```

```
>> x(7)
ans = 0.6000
>> x(11)
ans = 1
```

Note that we are referring to the first entry in x as entry number 1 and this has a value of 0.

5.1. **Guitar string fundamental mode shape.** The simplest form of vibration for a guitar string has a standing wave shape consisting of half a period of a sine wave. This is known as the fundamental mode of vibration for the string and the shape can be approximated using:

$$y = \sin(\pi x) \tag{3}$$

where x is a vector with values between 0 to 1. We will plot the shape of this vibration as follows:

```
>> y = sin(pi*x)
y =
  Columns 1 through 7
         0    0.3090    0.5878    0.8090    0.9511    1.0000    0.9511
  Columns 8 through 11
    0.8090    0.5878    0.3090    0.0000
>> plot(x,y)
```

I have assumed here that we still have the vector for x created in the previous section and if this has been done correctly a plot showing an arch should have popped up in the screen. Most functions in MATLAB will return a vector y with the same size as the input x and sin does indeed behave in this way. The arch will have a slightly rough appearance due to the fact that there are only 11 data points used.

5.2. **Using the colon character for specifying ranges within vectors.** The plot we produced in section 5.1 would be smoother, and more accurate, if we use a larger number of data points. It would be tedious to make a vector containing, say, 100 terms by manual typing. Instead we want to use the colon character ":" to do the job of creating a vector of an arithmetic series for us. Try typing the following:

```
>> 1:10
ans =
     1     2     3     4     5     6     7     8     9    10
```

This is an example of how the notation a:b creates a vector going from a to b in steps of 1. If we include an extra colon character then we can specify the step size for a more general arithmetic series. Try the following:

```
>> 1:2:10
ans =
     1     3     5     7     9
```

We have specified a starting point of 1, a step size of 2 and the range and a maximum value of 10. The last value in the vector is actually 9 because 11 would the next value in the series and this would exceed the maximum value.

If we want to create the vector x defined in section 5 using this notation it could be done by specifying a starting value of 0, a step size of 0.1 and a maximum value of 1 as follows:

```
>> x = 0:0.1:1
x =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 8 through 11
    0.7000    0.8000    0.9000    1.0000
```

To increase the number of points in the range, or "resolution", we could specify a smaller step size, say of a hundredth instead of a tenth. This is done in the next line, with a semicolon character at the end of the line so that the long vector isn't shown on the screen:

```
>> x = 0:0.01:1;
```

Now if we type in the following to recreate the plot from section 5.1, but with better resolution:

```
>> y = sin(pi*x);
>> plot(x,y)
```

Another use of the colon character is to look at a range within a vector. Say we wish to show the first 6 values in x and y, this can be achieved using:

```
>> x(1:6)
ans =
         0    0.0100    0.0200    0.0300    0.0400    0.0500
>> y(1:6)
ans =
         0    0.0314    0.0628    0.0941    0.1253    0.1564
```

It is interesting to note that the first values in $y = \sin(\pi x)$ are approximately equal to πx:

```
>> pi*x(1:6)
ans =
         0    0.0314    0.0628    0.0942    0.1257    0.1571
```

6. Mode shapes for guitar string harmonics

There are a whole series of sinusoidal standing wave shapes possible on a guitar string. Physically we are assuming that the guitar string is fixed at both ends. This is known as an "ideal string" because it is an idealisation. Real strings will always move their supports a little, but the ideal string is a very useful first approximation. The mode shapes are all sinusoidal and all share a value of 0 at both ends of the string so that the shape is given by:

$$y = \sin(m\pi x) \tag{4}$$

where m can be any integer (i.e. $m = 1, 2, 3, ...$) and is known as the mode number.

We can plot the mode shape for $m = 2$ using the code:

```
>> x = 0:0.01:1;
>> m = 2;
>> y = sin(m*pi*x);
>> plot(x,y)
```

If you don't see a graph popping up at this point it is possible that the graph might be hidden behind other windows. If this is the case then try typing the text "shg" (without the quotes) into the command line, followed by enter, to force the graph to be shown. We can also plot more than one mode shape on the same graph. This can be done by using a list of vectors in the plot command as follows:

```
>> x = 0:0.01:1;
>> m = 1;
>> y_1 = sin(m*pi*x);
>> m = 2;
>> y_2 = sin(m*pi*x);
>> plot(x,y_1,x,y_2)
>> shg
```

Note that we have redefined the value of m here and used vector names y_1 and y_2. In general we can give a variable or vector a name with continuous text of any length as long as it doesn't feature a number as the first character. It is best to give variables names which make the code easy to understand. This may encourage long variable or vector names. Try to keep names to a maximum length of five characters or so because long names can lead to problems with spelling mistakes.

Exercises 5. 5.1). Create a script to plot the shape given in equation (4) of the mode $m = 3$.

5.2). The mode shapes for velocity standing waves in a cylinder may be approximated by the equation:

$$y = \cos(m\pi x) \tag{5}$$

Create a script to plots this shape of the mode $m = 5$.

5.3). Create a script to plots these shapes for the modes $m = 1$, $m = 2$ and $m = 3$ all on the same graph.

5.4). Create a script to plots the mode shape for $m = 1$ with different spacings for the x range (for instance by defining xlow = 0:0.1:1; and xhigh = 0:0.01:1;).

7. Row vectors, column vectors, and matrices

So far we have looked at row vectors (which consist as a horizontal list of numbers). It is also possible to make column vectors which feature a list of numbers stacked on top of each other, and grids of numbers. A two dimension grid of numbers is called a matrix (the plural of which is matrices). In order to make a column vector manually we use the square brackets as before, but use the semicolon instead of the comma between each entry as follows:

```
>> xcol = [0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1]
xcol =
         0
    0.1000
    0.2000
    0.3000
    0.4000
    0.5000
    0.6000
    0.7000
    0.8000
    0.9000
    1.0000
```

The way to remember this is to think of the semicolon character as meaning "new line" whereas the comma character means "carry on with the same line". If we have a column and want to change it to a row then we can use the transpose command as follows:

```
>> transpose(xcol)
ans =
  Columns 1 through 7
         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000
  Columns 8 through 11
    0.7000    0.8000    0.9000    1.0000
```

Plotting works the same way for column vectors and row vectors in MATLAB.

You can create a matrix manually in MATLAB using a combination of commas and semicolons as follows:

```
>> A = [1, 2, 3; 4, 5, 6]
A =
     1     2     3
```

```
     4     5     6
```

It is conventional to use capitals for matrices. Note that typing "a" is will not get you the contents of "A" (i.e. MATLAB is case-sensitive). You can also make a matrix using multiple vectors within square brackets, as long as the vectors are the same length. For instance we can make a matrix, Y, consisting of two row vectors (y_1 and y_2) using:

```
>> x = 0:0.01:1;
>> m = 1;
>> y_1 = sin(m*pi*x);
>> m = 2;
>> y_2 = sin(m*pi*x);
>> Y = [transpose(y_1), transpose(y_2)];
>> plot(x,Y)
>> shg
```

Here we have transposed the row vectors to make column vectors and put them alongside one another using the comma. Note that the plot command in MATLAB plots each column in a vector with a different colour, so the plot should look the same as the one produced in section 6.

Ranges can be selected in a similar way to vectors. To show the first 6 rows of the two columns in Y we can type:

```
>> Y(1:6,1:2)
ans =
         0         0
    0.0314    0.0628
    0.0628    0.1253
    0.0941    0.1874
    0.1253    0.2487
    0.1564    0.3090
```

We can check the size or dimension of a matrix, vector or variable using the size command as follows:

```
>> size(m)
ans =
     1     1
>> size(y_1)
ans =
     1   101
>> size(y_2)
ans =
     1   101
>> size(transpose(y_1))
ans =
```

```
   101     1
>> size(Y)
ans =
   101     2
```

The size command thus returns a vector whose first entry is the vertical height of the matrix (or number of rows) and whose second entry is the horizontal width of the matrix (or number of columns) so that Y has dimensionality 101×2 in this case. Note that vectors can be though of as matrices with a size of one along one dimension and variables (like $m = 2$) can be thought of as matrices with dimensionality 1×1.

Exercises 7. 7.1). Create a script to plot a matrix containing the shapes for the guitar string modes $m = 1, 2, 3, 4$ and 5 on the same graph.

8. Frequency

You will notice that the different standing waves mode shapes on a guitar string have different wavelengths. Next we will discuss the concept of frequency in order to work towards constructing a wave which follows a sine wave profile over time which we can the listen to. The frequency of a wave is defined as the number of cycles per second for a wave, and we will set this equal to a variable called f. Initially we will try 440 samples per second, and this is expressed in scientific units as 440 Hz.

If we have a frequency (or number of cycles per second) then we can work out the time taken for a single cycle, known as the period time using the equation:

$$T = \frac{1}{f} \tag{6}$$

In the case of a 440 Hz vibration the period time will be:

```
>> f = 440;
>> T = 1/f
T = 0.0023
```

so one cycle of a sine wave at 440 Hz will take 0.0023 seconds, or 2.3 milliseconds.

8.1. **Sampling frequency.** In order to create a sound on a computer we need to first set a number of samples per second for approximating the wave shape and put this into a variable called fs. Initially we will use 44100 samples per second which can also be expressed as 44100 Hz or 44.1 kHz. This is the standard sample rate for CD quality audio.

We can then compute the time between samples using equation (6) which will be $ts = 1/fs$. Try checking this value in the command line as follows:

```
>> fs = 44100;
>> ts = 1/fs
ts = 2.2676e-05
```

8.2. **Making a sine wave plot.** The total time duration of our sine wave can be defined as one second and stored in a variable called t_dur. We will initially set this equal to the period time of the wave, so that one cycle will fit into our time duration. Calculating the total number of samples in the sound is then a matter of dividing the total time duration by the number of seconds between each sample. As an example, in 1/440 of a second the number of samples, N, will be calculated as follows:

```
>> fs = 44100;
>> t_dur = 1/440
t_dur = 0.0023
>> N = t_dur/ts
N = 100.2273
```

The should be just over 100 samples in one cycle of a sine wave of frequency 440 Hz at a sample rate of 44100 Hz. It doesn't make any sense to have a fraction of a sample on the end so we have to round the answer to the nearest integer using the "round" function which is built in to MALTAB:

```
>> N = round(t_dur/ts)
N = 100
```

Following this we create a vector of N time points. The vector of time points should have a first value of zero, a second value of ts, a third sample of $2ts$ and so on, so the Nth sample will have a value of $(N-1)ts$. This can be achieved by first making a vector of N integers, then multiplying by the sampling period.

```
>> fs = 44100;
>> ts = 1/fs;
>> n = [0:N-1];
>> t = ts*n;
```

Checking the first 6 values in the time vector should now give:

```
>> t(1:6)
ans =
   1.0e-03 *
         0    0.0227    0.0454    0.0680    0.0907    0.1134
```

Now we can make a vector containing the sine wave of frequency f and plot the results against time as follows:

```
>> p = sin(2*pi*f*t);
>> plot(t,p)
>> shg
```

8.3. **Making a sine wave sound.** Making a sound file of the sine wave vector, p, can be done using the built in "wavwrite" function. This function interprets columns of numbers as audio signals. In order to make our row vector sine wave

into a column we must use the transpose command, replacing p with the column vector version as follows:

```
>> p = transpose(p);
```

Next we write the column vector using the wavwrite function:

```
>> wavwrite(p, fs, 'mysinewave.wav')
```

Make sure you do include the single quotation marks this time when you type in this command. Now type "ls" (without quotes) into the command line to list the contents of the directory you are in and you should see mysinewave.wav is one of the files present. This is a wav file; an uncompressed sound file that can be played back in any audio software. Note that we are making files which have the potential to sound piercing. Turn down the volume on your audio equipment or remove your headphones before playing sounds for the first time to prevent unpleasant experiences involving loud, piercing sounds causing ringing in the ears (tinnitus) or hearing damage.

Try opening the file (using the Finder if you are on Mac or using Start > Documents on Windows, then double clicking on the file). You should hear a very short sound. While we set a frequency of 440 Hz, the duration was so short that the pitch will not be clear. If we want to hear a clear pitch we must do this process with longer duration (say t_dur = 1 for one second duration).

Exercises 8. 8.1). Make a script to create a two second sine wave of frequency 110 Hz and store it in a wav file.

8.2). Make a script to create a one second sine wave of frequency 220 Hz and store it in a wav file (with a different name).

8.3). Make a script to create sine wave of frequency 330 Hz that lasts for half a second and store it in a wav file (with a different name again).

8.4). Have a listen to the three files you have created above. The 220 Hz sine wave should sound higher in pitch than the 110 Hz sine wave. This frequency ratio of 2:1 corresponds to a difference in perceived pitch which is known as an "octave". The 330 Hz sound will sound higher again. This time the ratio of 3:2 with the 220 Hz will produce a pitch difference known as a "perfect fifth".

8.4. **Making a function.** We have created several very similar scripts now to do very similar jobs but with slightly different input definitions (for frequency and duration in this case). It is often useful to create a single function which works for different input values. For instance we can create a MATLAB function to create a sine wave of a particular sample rate, duration and frequency by creating a file with the name mysine.m in a text editor and typing the following code inside it:

```
function [p, t] = mysine(fs, f, t_dur)
%fs is the sample rate (Hz)
%f is the frequency of the sine wave (Hz)
%t_dur is the duration of the sound (seconds)
```

```
%p is the vector of the sine wave
%t is the vector of times

%ts is the sampling period (seconds):
ts = 1/fs;
%N is the number of samples in time duration:
N = round(t_dur/ts);
%n is a vector of integers with N entries:
n = [0:N-1];
%t is the vector of times:
t = ts*n;
%p is the vector of the sine wave:
p = sin(2*pi*f*t);
%Transpose p to make it a column vector so wavwrite can be used later:
p = transpose(p);
```

The start of the code above begins with the word "function" and this tells MATLAB that it is to behave like a function rather than a script. MATLAB functions have inputs inside brackets (fs, f and t_dur in this case) and output arguments (p and t in this case) and you cannot access any of the variables we may have stored in memory so far from inside a function unless you include them in the list of inputs (as the variables we have created so far are what are known as local variables in that they only exist in the command line or in scripts). Likewise any variables created inside the function can't be accessed from the command line after the function has been used unless they are included in the list of outputs.

Next type the following into the command line:

```
>> fs = 44100;
>> f = 440;
>> t_dur = 1/440;
>> [p, t] = mysine(fs, f, t_dur);
>> plot(t, p)
>> shg
```

If we have saved the file mysine.m in the directory we are in correctly then we should see one period of a sine wave on the screen. Note that the input variables in the function mysine can be changed to whatever we want, such as to create a wav file contain a one second duration sine wave with sample rate 44100 Hz and frequency 880 Hz we would type:

```
>> fs = 44100;
>> f = 880;
>> t_dur = 2;
>> [p,t] = mysine(fs, f, t_dur);
>> wavwrite(p, fs, 'mysine880.wav')
```

8.5. **Clipping.** You may have seen a warning about the file being "clipped" in completing section 8.4 above. Sine waves have a minimum and maximum of precisely -1 and 1 but the wav file format can only store values between roughly -1 and 1 with a finite precision (or "bit resolution") and the slight discrepancy produces the warning. Assuming the file mysine.m has been created successfully using the code above then we can check the minimum and maximum of the vector, p, as follows:

```
>> fs = 44100;
>> f = 880;
>> t_dur = 2;
>> [p,t] = mysine(fs, f, t_dur);
>> min(p)
ans = -1.0000
>> max(p)
ans = 1.0000
```

As the maximum amplitude is indeed 1 there won't be audible effects due to the clipping.

If on the other hand we create a waveform and multiply by a factor of, say, 4 before writing the wav file, then we should hear audible differences. Try the following:

```
>> fs = 44100;
>> f = 220;
>> t_dur = 1;
>> [p, t] = mysine(fs, f, t_dur);
>> wavwrite(p, fs, 'mysine220.wav')
>> wavwrite(4*p, fs, 'myclipped220.wav')
```

This time the wave 4*p is producing an input with a range of -4 and 4 meaning that the wav file format (which can only store numbers in the range between -1 and 1) will "clip" the top and bottom of the waveforms, causing a sort of distortion called "clipping". This is similar to the concept of "saturation" in amplifiers where the output "clips" or "saturates" when the desired output exceeds the operating range of the amplifier (which in turn depends on the amplifier power supply). Have a listen to the two files you have produced. You will notice that, as well as sounding louder, the file myclipped220.wav has a brighter tone quality while retaining the same pitch.

We can read the contents of the wav file back into MATLAB as a variable called p4 using the "wavread" function, and this can then be compared to 4*p using a plot command:

```
>> [p4, fs] = wavread('myclipped220.wav');
>> plot(t, 4*p, t, p4)
>> shg
```

The plot shows how the wav file has been clipped outside the range -1 and +1 and this is especially clear if we limit the x axis range to just show one period using the "xlim" command:

```
>> xlim([0, 1/f]);
>> shg
```

8.6. **Using the fft command for Fourier transforms.** We can analyse the frequency spectrum of sounds in MATLAB using Fourier transforms with the fft command. Assuming that the wav files have been created correctly from section 8.5 then a plot of the frequency spectrum should be produced using the following code:

```
>> [p, fs] = wavread('mysine220.wav');
>> N = length(p);
>> n = [0:N-1];
>> fvec = n*fs/N;
>> pfft = fft(p);
>> plot(fvec,abs(pfft))
```

Here we have used three new functions: the "fft" command which computes a vector containing the frequency spectrum of the input vector, the "length" command which returns the number of entries in the input vector, and the "abs" command which returns the magnitude (or absolute value) of a complex number and this is necessary because the "fft" command returns complex numbers. Note that the left-hand side of the resulting plot shows one peak and this corresponds to a frequency of 220 Hz. This indicates that the sound stored in the wav file was indeed a sine wave at 220 Hz. The right-hand side of the plot shows a mirror image of the left-hand side because frequencies greater than half of the sample rate cannot be measured uniquely. We can label the axes using the "xlabel" and "ylabel" commands and limit the range of the plot to show the frequency peak in more detail as follows:

```
>> xlabel('Frequency (Hz)')
>> ylabel('Amplitude')
>> xlim([200, 240])
```

Next we can try plotting the frequency spectrum of the clipped file as follows:

```
>> [p4, fs] = wavread('myclipped220.wav');
>> p4fft = fft(p4);
>> plot(fvec,abs(p4fft))
>> xlim([0, 2000])
```

This time we can see multiple peaks and these are at 220 Hz, 3×220 Hz $= 660$ Hz, 5×220 Hz $= 1100$ Hz, $7 \times 220 = 1540$ Hz and 9×220 Hz $= 1980$ Hz. These peaks are called "odd harmonics" in that they are odd integers multiplied by the fundamental frequency of 220 Hz. The clipping generated by the wavwrite process

is an example of what is called harmonic distortion. This creates a change in the tone quality of the sound without changing the perceived pitch. Hi-fi loudspeakers and amplifiers are designed to minimise the distortion of waveforms, but some distortion will always occur and their specification charts typically show the total harmonic distortion (THD) as a percentage. Guitar amplifiers and distortion effect units on the other hand are designed to produce larger amounts of harmonic distortion. If distortion is used subtlety it may add "warmth" to the sound produced (as is often the case in blues guitar playing), and if larger amounts of harmonic distortion are produced the sound can be more aggressive (as is often the case in heavy metal style guitar playing).

Exercises 8 continued. 8.5). Create a script in which you create a vector called p containing a sine wave with frequency 330 Hz, duration 1 second and sample rate 44100 Hz using the function "mysine.m" described in section 8.4. Multiply the resulting vector by a factor of 10 and save the result in a wav file. Listen to the sound of the wav file and check that the sound is heavily distorted due to clipping distortion. Plot the frequency spectrum of the wav file as shown in section 8.6.

8.6). Create a script in which you create a vector called p containing a sine wave with frequency 330 Hz, duration 1 second and sample rate 44100 Hz as above. Multiply the resulting vector by a factor of 2 and save the result in a wav file. Check that the clipping distortion is more subtle in both the sound and when you plot the frequency spectrum of the wav file. Hint: The harmonics should have lower amplitudes in the frequency spectrum plot.

8.7). Create a script in which you create a vector called p containing a sine wave with frequency 330 Hz, duration 1 second and sample rate 44100 Hz as above. Apply processing using the formula:

```
p = 2*(p + 0.25);
```

Save the result in a wav file. Check that the clipping distortion features "even harmonics" at $2 \times 330 = 660$ Hz, $4 \times 330 = 1320$ Hz and so on (as well as the "odd harmonics" seen in previous plots) in the frequency spectrum of the wav file. Check that the sound has a different tone quality in comparison to the wav file created in the previous exercise.

8.8). Create a script which reads in the wav files created in exercises 8.6) and 8.6), and then plot their frequency spectra on the same graph.

9. Wavelength

If a sine wave is produced in air, say by loudspeaker then the sound travels away from the source as a travelling wave. There will be one period time $T = 1/f$ between one peak of the wave being produce and the next peak of the wave being produced for a frequency of f. The first peak will have travelled a distance of

$$\lambda = cT = \frac{c}{f} \tag{7}$$

where c is the speed of wave propagation (and $c \approx 340$ m/s is the speed of sound in air). The value λ is called the wavelength of the wave. If, on the other hand, we somehow know the wavelength of a sound we can work out the frequency using a rearrangement of equation (7):

$$f = \frac{c}{\lambda} \tag{8}$$

9.1. **Travelling waves, standing waves and reflection.** So far we have encountered standing waves, such as those on a guitar string, and travelling waves, such as the waves which travel from a source to a listener. The relationship between these two forms of sound are useful to study. When a sound wave collides with wall it is reflected in a similar way to light being reflected from a mirror. The wall doesn't act as a perfect "acoustic mirror" in that sound with low frequency bends around objects due to a process called diffraction and different surfaces absorb different frequencies to different extents.

Once a wave is reflected, we get two travelling waves on top of each other: one going towards the wall and one going away from the wall. If the two waves have the same amplitude then the result is alternating areas of constructive and destructive interference. The wave vibrates in the areas of constructive interference while appearing to not vibrate in the areas of destructive interference. This is a what we mean by a standing wave. For the moment the main thing to note is that the wavelength and frequency of both standing waves and travelling waves obey equation (8) as standing waves are simply two travelling waves superimposed.

When a positive acoustic pressure bounces off a wall in a room then the reflection will also have a positive pressure. We say that the wave is reflected in phase.

Consider waves on a string fixed at both ends. A travelling wave featuring an upwards motion of the string hitting a fixed end will supply an upwards force to the support which are fixing the end of the string. The string will receive an equal and opposite force from the support (and this is an example of Newton's third law). The motion on the string will then show a downwards motion for the reflected travelling wave. We say that the wave is reflected out of phase in this case.

9.2. **The frequencies of the modes of a string with fixed ends.** As waves are reflected out of phase from the ends, the standing wave shapes must be zero at the ends. This is exactly what we saw in section 6. We will now consider the wavelength of these modes, define the speed of wave propagation on a string and calculate the frequency of vibration of the modes. If we define the string length as L then the fundamental mode will feature half a wavelength so

$$L = \frac{\lambda_1}{2}, \lambda_1 = 2L \tag{9}$$

where λ_1 is the wavelength of the fundamental or $m = 1$ mode from equation (4). If we consider the $m = 2$ mode from equation (4) we get

$$L = \lambda_2 \tag{10}$$

and for $m = 3$:

$$L = \frac{3\lambda_3}{2}, \lambda_3 = \frac{2L}{3} \tag{11}$$

so there are m half wavelengths in the length L in general, giving a formula of for the wavelength, λ_m of the mth mode on a string of:

$$\lambda_m = \frac{2L}{m}, m = 1, 2, 3 \ldots \tag{12}$$

If we know the speed of wave propagation on the string then we can now work out the frequency of the mth mode on a string using equation (8). It is important to note that the speed of wave propagation on a string is not the same things as the speed of sound in air and depends on μ, the mass per unit length of the string (measured in kg/m) and T, the tension of the string (measured in Newtons) according to the formula:

$$c = \sqrt{\frac{T}{\mu}} \tag{13}$$

Now to give:

$$f_m = \frac{mc}{2L}, m = 1, 2, 3 \ldots \tag{14}$$

The modes of vibration on a string are therefore "harmonics" of frequency f_1, $2f_1$, $3f_1$, $4f_1$ ...

9.3. **Waves in pipes.** When a positive acoustic pressure bounces of the closed end of a cylinder the wave is reflected in phase (i.e. with a positive acoustic pressure) so that the closed end behaves in the same way as a wall in a room. If a travelling waves arrives at the open end of the tube the opposite happens in that the wave is, to a first approximation, reflected out of phase back down the cylinder. This is, initially, rather a surprising statement. If we walk down a corridor and out of an open door at the end then we do not experiencing a force sending us back in the other direction upside-down!

Wave motion is strange and sometimes unintuitive. The fact that a positive acoustic pressure is reflected from an open end as a negative acoustics pressure is due to changes in the area that the pressure wave may occupy. Waves in a pipe will reflect whenever the cross-section area changes, for instance. The reflection is in phase if the cross-section decreases and the reflection is out of phase if the cross-section increases. While this is very hard to understand, study of this phenomenon in acoustics in strongly related to the subject of quantum mechanics where the

Schrödinger equation shows how very small particles behave in very strange, wave-like manner. Electrons inside atoms behave in this way, occupying standing wave states. The reason we find wave-like behaviour strange is because we are not small enough! Acoustics is just one of the subjects that can help educate us.

9.4. **Using a for loop to show travelling waves.** We will now create a series of plots so that travelling waves are displayed on the screen. Make a script file and put the following code into it:

```
%Clear any old variables (in case we use them by accident):
clear
%Plotting updates per second (Hz):
fs = 25;
%Number of points in the plot along x axis:
Lx = 100;
%Frequency (Hz):
f = 0.5;
%Duration of movie (seconds):
t_dur = 5;
%Wavelength:
lambda = 2*Lx/5;

%Number of time points:
Nt = t_dur*fs;
%Spatial vector:
x = 0:Lx;
%Wavenumber (radians per sample):
k = 2*pi/lambda;
%Angular frequency (radians per second):
w = 2*pi*f;

%for loop:
for ind = 0:Nt-1
    %Time now:
    t = ind/fs;
    %Travelling wave in forward direction:
    pfor = sin(w*t - k*x);
    %Do plot:
    plot(x, pfor)
    %Bring graph to top of the screen if necessary:
    shg
    %Delay for ts seconds so plot moves at (roughly) the correct speed:
    pause(1/fs)
```

```
end
```

This script plots a travelling wave that travels to the right.

Exercises 9 continued. 9.1). Create a script to plot a travelling wave which travels to the left. Hint: Use a positive sign inside the bracket as follows:

```
pback = sin(w*t + k*x);
```

9.2). Create a script to plot a travelling wave going to the right, a travelling wave of the same frequency going to the left and the sum of the two, all on the same graph. The sum of the two should be a standing wave.

9.3). Create a script to plot travelling waves and their sum when we have integer number of quarter wavelengths within the plot window.

9.4). Create a script to plot a travelling waves and their sum when the backward going wave has the opposite phase. Hint: The backward going wave should be preceeded by a negative sign as follows:

```
pback = - sin(w*t + k*x);
```

10. Class, strings and measurements

Using MATLAB for measurements involves making a wav file with a test signal, importing this into some audio software (such as Cockos REAPER) and then playing the sound back using a loudspeaker while recording the sound that results using a microphone. In this section we will try making sine waves at 125 Hz, 250 Hz, 500 Hz, 1000 Hz and 2000 Hz all with a duration of 1 second and a variety of sample rates (44100 Hz, 96000 Hz and 192000 Hz). In order to do this is is helpful to use the name of the file to store information on the sound. This can be done by taking a script and altering the number on the lines beginning with f (where we set the frequency) and also the line beginning with wavwrite (where we name the file we are creating). It is much better to we are making have a working knowledge of how MATLAB uses text so that we only have to change the line beginning with f, and then use the variable f as part of the file name automatically.

Programming languages hold text in a different format to numbers as each character in text needs to have its own independent digital code. Technically we say that number and text are stored as different data "class" with numbers usually stored as members of the class "double" (this just means that the computer uses 64 bits to store the number instead of the older "single" precision storage which uses 32 bits) while "strings" of text are stored as vectors which are members of the class "char" (which is short for character).

In order to get MATLAB to use the variable name f within the name of the wav file we need to convert the number stored in f from a double precision number into a string of characters using the command num2str. Try typing:

```
>> f = 250;
>> num2str(f)
```

```
ans = 250
```

This may not look like it has achieved much at first inspection, but we can check what class has been created using the command whos (which is different from the command who in that it shows you information on the variables in long rather than short form):

```
>> who
Your variables are:
ans  f
>> whos
  Name      Size            Bytes  Class     Attributes
  ans       1x3                 6  char
  f         1x1                 8  double
```

So we can see that the answer is a member of the class char and this means that it can be used as part of a file name. We can use square brackets to contain a vector with a combination of quotes and num2str commands separated by commas as follows:

```
>> ['sine',num2str(f),'.wav']
ans = sine250.wav
```

Using this technique we write a wav file using the wavwrite command incorporating the variables f and fs in the file name by editing the function we created in section 8.4 with the new bottom line in the following code:

```
function [p, t] = mysine(fs, f, t_dur)
%fs is the sample rate (Hz)
%f is the frequency of the sine wave (Hz)
%t_dur is the duration of the sound (seconds)
%p is the vector of the sine wave
%t is the vector of times

%ts is the sampling period (seconds):
ts = 1/fs;
%N is the number of samples in time duration:
N = round(t_dur/ts);
%n is a vector of integers with N entries:
n = [0:N-1];
%t is the vector of times:
t = ts*n;
%p is the vector of the sine wave:
p = sin(2*pi*f*t);
%Write wav file:
wavwrite(p,fs,['mysine_',num2str(f),'_',num2str(fs),'.wav'])
```

Creating a wav file can then be achieved using:

```
mysine(44100, 250, 1);
```

resulting in a one second file called mysine_250_44100.wav with the sample rate 44100 Hz and containing a sine wave of frequency 250 Hz and so on. The great flexibility here is that we can then simply change one number and the name of the file that is created reflects this automatically.

Try creating a set of files containing one second sine waves at 125 Hz, 250 Hz, 500 Hz, 1000 Hz and 2000 Hz at sample rates of 441000 Hz, 96 kHz and 192 kHz. Insert these files into audio software such as Cockos REAPER

11. Answers

Answers to Selected Exercises. 1.1). We need to work out the distance to the castle and back:

```
>> 34+34
ans = 68
```

so $d = 68$ m and then work out the time using $t = d/c$ which is:

```
>> 68/340
ans = 0.2000
```

so the time before we hear the echo is 0.2 s.

1.2). Rearranging equation (1) we get $d = ct$ so:

```
>> 340*0.3
ans = 102
```

gives the answer as 102 m.

1.3). The time between echoes is:

```
>> 1/180
ans = 0.0056
```

so 0.0056 seconds or 5.6 milliseconds. The distance travelled by sound in between each echo must be $d = ct$:

```
>> 340*0.0056
ans = 1.9040
```

so the distance between the walls must be half this (as the sound must travel to the other wall and back to make each echo:

```
>> 1.9040/2
ans = 0.9520
```

so the distance between the walls is 0.9520 m or 95.2 cm.

1.4). Here the tape speed is $c = 2.5$ cm/s and the taken for echoes to appear is $t = d/c$ so using $d = 5$ cm we have:

```
>> 5/2.5
ans = 2
```

so echoes appear after 2 seconds.

4.1). A suitable script would be:

```
%Clear any old variables (in case we use them by accident):
clear
%Speed of sound
c = 340;
%Anglular position (radians) of listener from line
%of equal distance from speakers:
theta = pi/8;
%Time difference (seconds):
t = 0.001;
%Distance difference (metres):
d = c*t;
%Distance between speakers (metres):
L = d/sin(theta)
```