
Chapter 3

Signals

Essential reading

Eaton, J.W. *GNU Octave Manual*. (Bristol: Network Theory, 1996) [ISBN 0954161726]. (This is also available online in HTML form at <http://www.gnu.org/software/octave/doc/interpreter> and in texinfo source format in the Octave source code distribution.)

Recommended reading

Oppenheim, A.V. and A.S. Willsky with S. Hamid Nawab *Signals and Systems*. (Upper Saddle River, N.J.; London: Prentice Hall, 1997) [ISBN 0138147574] Chapter 1.

3.1 Introduction

In this chapter you will learn about the fundamental concepts of signals as they are understood by the engineering profession. To assist in the understanding of signals, and in the next chapter, 'Systems', it is useful to get some direct experience of constructing and manipulating them.

Digital multimedia systems are built upon a class of signal and system building blocks called discrete-time signal processing, or *digital signal processing (DSP)*. DSP is a branch of engineering that is concerned with the analysis and design of signals and systems for everyday applications, such as: radar, satellite communication, seismic monitoring, rocket guidance systems and, the subject of this guide, digital multimedia.

Signals are built out of fundamental units that are combined to make more complicated signals using basic mathematical operations. Therefore, this chapter introduces the fundamental building blocks of DSP and methods to construct and manipulate signals.

3.2 Octave

Octave is an open-source mathematics and engineering tool that was written by John Eaton and it is maintained as part of the free software foundation's GNU project; as such, it will be available to use for free well into the future. Octave is useful for the construction, manipulation and visual display of signals. It can also be used for displaying images, audio and video, that are manipulated using DSP techniques. In later chapters you will learn how to perform signal manipulations in

Octave; in the current chapter you will learn how to install and run Octave, and how to start constructing signals out of fundamental DSP units.

3.2.1 Installing Octave

You will first need to obtain the latest version of Octave. As an open-source project, Octave is freely available on the Internet. If your operating system has a package manager (such as fink on Mac OSX or Synaptic Package Manager in Ubuntu Linux) then you should use the package manager to install the latest version of Octave. Otherwise you can download Octave directly from:

<http://www.gnu.org/software/octave/>

At the time of writing, the latest version of Octave is 3.0.1; you should download or install at least this version or a later (stable) release if available.

In addition to Octave you will need to install GnuPlot and ImageMagick for plotting and image graphics. Again, if you have a package manager it is likely that the additional packages were automatically installed when installing Octave. However, if you do not have a package manager you can obtain both of these packages freely from the Internet:

<http://www.gnuplot.info>
<http://www.imagemagick.org>

3.2.2 Installing for different operating systems

Octave's primary support is for the Linux operating system, so if you are using that, it is likely that your installation will be straightforward.

Some versions of Windows present problems for Octave, in which case you are recommended to download Cygwin – a version of Linux that runs on Windows systems – and run Octave within that.

For students using Mac, you are advised to go to <http://pdb.finkproject.org> if you encounter any problems with your installation.

Please also note that the plotting examples might look different from the ones in this guide, depending on which version of Octave you are using. What is important is that your plots illustrate to you the concepts, and that you become competent in using Octave to test them out.

3.2.3 Running Octave

Once you have installed Octave you can test it by opening a terminal window and typing the Octave command.

```
shell%1>octave
```

Octave will start up and will print some information about the version and the copyrights of the software. For example, you might get the following message:

```

GNU Octave, version 3.0.1 (i486-pc-linux-gnu).
Copyright (C) 2006 John W. Eaton.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.
Additional information about Octave is available at
http://www.octave.org.
Please contribute if you find this software useful.
For more information, visit http://www.octave.org/help-wanted.html
Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful
report)

```

You are now presented with Octave's interactive shell which allows you to enter commands and display the results of mathematical operations. The interactive nature of Octave is an advantage over languages that require compilation, such as Java, because the response to entering your code is immediate.

3.2.4 Using Octave

Octave can manipulate numbers that are organised into convenient containers called vectors and matrices. In Octave, all numbers are actually matrices, but the user doesn't know this until they need to use matrices.

Scalars

A scalar is a single numeric quantity, such as the numbers 3, -6.3 and the irrational pi. When you type these values at the Octave prompt, they will be evaluated as expressions and their values returned as answers:

```

octave:> 3
ans = 3
octave:> -6
ans = -6
octave:> pi
pi = 3.1416

```

Note that the last of these three inputs evaluated a pre-defined constant: pi. Just as in the other programming languages that you have used in your studies, we can define a variable to contain a value. Octave is not a typed system, so variables do not have to be declared and assigned types explicitly; we can simply define a variable by assigning it a value:

```

octave:> a = -100
a = -100
octave:> b = pi
b = 3.1416
octave:> aLongVariableName = 10
aLongVariableName = 10

```

Scalar operations

The same mathematical operations that you have used in Java and other programming languages are also available in Octave. There common operations of addition, subtraction, multiplication and division can be entered directly at the command prompt and Octave will evaluate them:

```
octave:> 4.5 + 9.6 * 2
ans = 23.7
octave:> ( 4.5 + 9.6 ) * 2
ans = 28.2
octave:> ( 4.5 + 9.6 ) / 2
ans = 7.05
octave:> ( 4.5 + 9.6 ) ^ 2 / 2
ans = 99.405
```

The order of precedence of operators is similar to that of Java, with multiplication taking precedence over addition, and division taking precedence over multiplication. The order of operation is altered with the use of parentheses as in the above examples. In the last example the \wedge operator was used to perform exponentiation; raising the expression in the parentheses to the power 2 thus taking its square.

Mathematical operations can also be performed on variables in the same manner. In the following examples the variables defined above are used in mathematical expressions:

```
octave:> ( a*a + b ^3 ) / aLongVariableName

ans = 1003.1
octave:> 2*pi
ans = 6.2832
octave:> pi/2
ans = 1.5708
```

Mathematical functions

Octave provides a comprehensive set of mathematical functions such as `sin()`, `cos()`, `tan()`, `exp()`, `asin()`, `acos()`, `atan()`, `min()`, `max()`, `mean()`. Functions are expressions that return a value given one or more input arguments. The following examples illustrate some of the more common functions:

```
octave:> cos(0)
ans = 1
octave:> sin(pi)
ans = 1.2246e-16
octave:> sin(pi/2)
ans = 1
octave:> exp(1)
ans = 2.7183
octave:> tan(pi/2)
ans = 1.6332e+16
```

The functions `cos()`, `sin()` and `tan()` are the familiar trigonometric functions that

you may have used either in Java or on a calculator. They accept a scalar argument in the range $[0 \dots 2\pi]$ and return the value of the function for the given argument. Note that the values for $\sin(\pi)$ and $\tan(\pi/2)$ are not exact: the mathematically correct values of these functions for the given arguments are 0 and *Infinity* (or ∞) respectively. Here, as with all mathematical operations, Octave reports the value of the function to within the floating-point numerical accuracy of the host system. The values are not exact because of finite floating-point precision; the same is true in the Java programming language.

The `exp()` function is the exponential function which raises the natural exponent e , often called the Euler number, to its argument. Thus, `exp(1)` returns the identity of the natural exponent, to a fixed number of decimal places, in this case 4 decimal places (2.7183). To see more of the decimal places of this irrational number use the `format long` command:

```
octave:> format long
octave:> exp(1)
ans = 2.71828182845905
```

To return to the default short format use the command `format short`.

All of the functions in octave have built-in help available. To access the help use 'help *functionname*', for example:

```
octave:>help sin
```

Relational and conditional operators

Just like Java, Octave also has basic programming constructs such as *relational* (`<`, `>`, `<=`, `>=`, `==`, `,`), *conditional* (`if(...)` `endif`; `switch..case`) and *control* operations (`for(...)` `endfor`; `while(...)` `endwhile`;. We may use Octave by writing a script, storing it in a file, and calling the script by name to access its functionality.

Learning activity

Type the following into your text editor; for example *gedit*, *wordpad* or *emacs*:

```
for k = 10:-1:0
  if(k>0)
    100/k
  else
    printf('What will happen if we divide by zero?')
    100/k
  endif
endfor
```

Do not worry about the syntax for now; but most of this code should look familiar to you. It is very similar to Java except it does not use braces '`{}`' to delimit code blocks. Instead, blocks of code are delimited by keywords such as `for (...)` `endfor` and `if ...endif` as in many Unix-style scripting languages.

Now save the file using the name `myscript.m` in a CC227 working directory, say, `~/CC227/octave`. The symbol `~` means your HOME directory. The extension `.m` is used by Octave to locate script and function files.

You will first need to make a directory to store your scripts. Do this either from the desktop of your operating system, from a terminal or even in Octave:

```
octave:> mkdir('~foo/bar')
ans = -1
```

Only one directory at a time can be created; here, since `~/foo` did not exist an error is returned (`-1`) when trying to make the `bar` directory within it.

Instead, we must create each new directory separately:

```
octave:> mkdir('~ /CC227')
ans = 0
octave:> mkdir('~ /CC227/octave')
ans = 0
```

The return value, 0, indicates that all is well with the new directories. When you have saved `myscript.m` you will need to tell Octave where to find it so that you can use it. To do this, use the `addpath` command:

```
octave:>addpath('~ /CC227/octave')
```

Octave's `path` command will display all the directories that Octave will look in to find scripts and data files that you might request at the command prompt.

```
octave:>path
```

Octave's search path contains the following directories:

```
~/CC227/octave
.
/usr/lib/octave/2.1.73/site/oct/i486-pc-linux-gnu//
/usr/lib/octave/site/oct/api-v13/i486-pc-linux-gnu//
/usr/lib/octave/site/oct/i486-pc-linux-gnu//
/usr/share/octave/2.1.73/site/m//
/usr/share/octave/site/api-v13/m//
/usr/share/octave/site/m//
/usr/lib/octave/2.1.73/oct/i486-pc-linux-gnu//
/usr/share/octave/2.1.73/m//
/usr/local/share/octave/site-m//
```

Do not worry about the long list of directories, you are most interested in the ones at the top. The single `."` refers to the current working directory which can be displayed using the command `pwd`:

```
octave:> pwd

/home/mkc/CC227/octave
```

If all has gone well, you should now be able to execute your script from the command line:

```
octave:>myscript
ans = 10
```

```

ans = 11.111
ans = 12.500
ans = 14.286
ans = 16.667
ans = 20
ans = 25
ans = 33.333
ans = 50
ans = 100
What will happen if we divide by zero?
warning: division by zero
ans = Inf

```

What happened at the end of the script?
Is the last value a numeric?

Octave supports scripts, as in the example above, which accept no arguments and do not **return** a value. A script can output a value to the screen, such as in your example above.

To support input arguments and return values Octave also allows user-defined **functions**. Functions are, essentially, scripts with an extra keyword to define a function, and they are able to accept and return arguments.

Make a new text file in your Octave directory called myfunction.m:

```

function y = myfunction(n)
  for k = n:-1:0
    if(k>0)
      y = 100/k
    else
      printf('What will happen if we divide by zero?')
      y = 100/k
    endif
  endfor
endfunction

```

The Octave path was already set above; so you may now type the following at the command line:

```

octave:>myReturnValue = myfunction(5)
y = 20
y = 25
y = 33.333
y = 50
y = 100
What will happen if we divide by zero?
warning: in /home/mkc/src/octave/myfunction.m near line 7, column 8:
warning: division by zero
y = Inf
myReturnValue = Inf

```

Explain why the number contained in myReturnValue is the last value computed by myfunction.m.

Vectors

So far, the scalar operations that you have executed in Octave have been very similar to the numerical computations that you have used in other programming languages. The power of Octave comes in combining many scalar values into ordered lists, called vectors and matrices. Octave's operations upon vectors and matrices make programming for signals and systems much easier than it would be in languages such as Java.

A vector is simply a list of scalar values such as `[2 4 6 8 10]` and `[exp(1) 23.7 42 -pi]`. A vector is defined with the use of the square brackets `[` and `]`. The following are examples defining vectors in Octave:

```
octave:> [exp(1) 23.7 42 -pi]
ans =
2.7183 23.70 42.00 -3.1416
octave:> [pi pi/2 pi/3 pi/4 pi/5 pi/6 pi/7 pi/8 pi/9]
ans =
Columns 1 through 8:
3.14159 1.57080 1.04720 0.78540 0.62832 0.52360 0.44880 0.39270
Column 9:
0.34907
```

In the first example, a vector was constructed using both scalar literals and functions. The functions are evaluated and the results inserted into the vector. The vector is then a fixed set of numbers. In the second example, the pre-defined constant `pi` was used to construct every element of the vector. The result is a vector of nine elements with values that spill over two lines. When this happens, during printing to the screen, Octave informs the user which **columns** appear on each line.

The term **column** refers to the number of elements that the vector contains if it is oriented as a **row** vector as in the examples above. Vectors have either one row and multiple columns of values, or one column and multiple rows. The **orientation** of a vector determines whether its values are organised in rows or columns.

To find out whether a vector is a row or a column vector you can use Octave's `size()` command:

```
octave:> size([-1 0 1 2 3 4 5])
ans =
1 7
```

The `size` command returns the number of rows and columns in a vector. In this example, the command has resulted in a response of `1 7` for row and column respectively, indicating that the resulting vector is a row vector.

To make a column vector you can enter the values one row at a time:

```
octave:> [
> -1
> 0
> 1
> 2
> 3
> 4
> 5]
```

```
ans =
-1
0
1
2
3
4
5
```

Notice that the resulting values are now listed as a single column.

Conveniently, Octave keeps the last computed value in an automatic variable called **ans**; in this case a column vector. The `size` function can be called with **ans** as an argument to find out the number of rows and columns of the last computed vector:

```
octave:> size(ans)
ans =
7 1
```

Now Octave reports that there are 7 rows and 1 column; so this is a column vector.

The orientation of a vector is very important. The process of changing orientation is a fundamental operation in Octave called **transposition**; transposition of a vector is obtained with a special operator “ ’ ”.

In the following examples we transpose a row vector into a column vector, and a column vector into a row vector. Just as with scalars, you can assign vectors to variables. There is nothing special that needs to be done; Octave treats scalars and vectors in the same manner for variable assignment:

```
octave:> a = [-1 0 1 2 3 4 5]

a =
-1  0  1  2  3  4  5

octave:> b = [-1 0 1 2 3 4 5]'
b =
-1
0
1
2
3
4
5

octave:> aSize = size( a )
aSize =
1 7

octave:> bSize = size( b )
bSize =
7 1
```

Notice how the number of rows and columns is exchanged with the use of the ’ operator.

Another special operator enables vectors to be defined as sequences of numeric

values, without having to list the entire sequence. The colon operator `:` was used above in the `myfunction.m` function; it is an iterator that generates a list of values between a start value and an end value:

```
octave:> [1:10]
ans =
 1 2 3 4 5 6 7 8 9 10
octave:> [10.5:20.4]
ans =
10.5 11.5 12.5 13.5 14.5 15.5 16.5 17.5 18.5 19.5
```

In the above two examples, the start and end values are traversed in order in step sizes of 1.0, the default increment for colon operator iteration.

Learning activity

In the second example above, the value 20.4 is not reached. Can you see why?

To change the step size of the iteration Octave has a syntax using two colon operators:

```
octave:> [10.5:.9:20.4]
ans =
Columns 1 through 10:
10.5 11.4 12.3 13.2 14.1 15.0 15.9 16.8 17.7 18.6
Columns 11 and 12:
19.5 20.4
```

Learning activity

Here the value 20.4 is reached. Can you see why?

In this example, the direction of the iterator is reversed by the use of a negative increment:

```
octave:> myVector = [10:-1:1]
myVector =
10
 9
 8
 7
 6
 5
 4
 3
 2
 1
```

The increment argument to the colon operator can have any real numeric value. In this example the resulting vector was transposed to a column vector using the transpose operator `'`.

Learning activity

Try reversing the positions of 10 and 1 in the above example. Explain the output that you see.

Vector arithmetic operations

The same mathematical operations are available for vectors as for scalars. First, we can apply the basic operations of addition, subtraction, multiplication and division between vectors and scalars:

```
octave:> aVec = [ 0 : 0.1 : 1 ]
aVec =
Columns 1 through 8:
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7
Columns 9 through 11:
0.8 0.9 1.0
octave:> aVec + 1
ans =
Columns 1 through 10:
1. 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
Column 11:
2.0
octave:> aVec * 10
ans =
Columns 1 through 8:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0
Columns 9 through 11:
8.0 9.0 10.0
octave:> aVec - 1
ans =
Columns 1 through 8:
-1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3
Columns 9 through 11:
-0.2 -0.1 0.0
octave:> aVec / 0.1
ans =
Columns 1 through 8:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0
Columns 9 through 11:
8.0 9.0 10.0
```

The following examples consist of arithmetic operations between two vectors:

```
octave:> bVec = [ 1 : -0.1 : 0 ]
bVec =
Columns 1 through 8:
1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3
Columns 9 through 11:
0.2 0.1 0.0
octave:> aVec + bVec
ans =
```

```

1 1 1 1 1 1 1 1 1 1
octave:> aVec - bVec
ans =
Columns 1 through 8:
-1.0 -0.8 -0.6 -0.4 -0.2 0.0 0.2 0.4
Columns 9 through 11:
0.6 0.8 1.0

```

The operations of vector addition and vector subtraction simply perform the scalar operations of addition and subtraction independently on each pair of vector elements taken in order.

Learning activity

What happens if you add, or subtract, two vectors of different lengths? Why does this happen?

What happens if you transpose one of two vectors of the same length and perform addition or subtraction?

What happens if you transpose both vectors? Can you explain the result?

The rule for adding, or subtracting, pairs of vectors is that they must have the same **dimensionality**. That is, they must both be either row vectors or column vectors and they must have the same number of elements. Before performing vector addition or subtraction it is often useful to use the `size()` command, to make sure that the vectors are compatible with the rules of vector arithmetic.

Vector indexing

Octave provides access to the individual elements of a vector variable using parentheses: `()`. The ordinal position of the element that is required is supplied as an argument, as if the vector variable were itself a function. For example:

```

octave50>a = [1 1 2 3 5 8 13 21 34]
>a(4)
ans = 3
>a(9)
ans = 34
>a(0)
error: invalid vector index = 0
>a(10)
error: invalid vector index = 10

```

Note that the indexing is one-based, not zero-based as it is in Java. This can lead to a lot of confusion when moving from Java to Octave, so it is best to take extra care when writing index code in Octave.

The argument supplied for vector indexing can be another vector. The elements of the inner index vector are the positions in the outer vector to access:

```

octave:> a([2 4 6 8])
ans =
1 3 8 21

```

```

octave:> a([3 5 7 9])
ans =
 2 5 13 34
octave:> a([1:2:9])
ans =
 1 2 5 13 34

```

In the last example, the colon notation was used to construct an index vector starting at 1 with increment 2 and ending at 9. This example demonstrates how vectors can take on different roles depending on how they are used. Mastery over vectors and their notation in Octave will be necessary for understanding the concepts of signals and systems and how they can be implemented in Octave.

Learning activity

For each of the following, find at least three ways to construct the vector: (Hint: you can use direct construction, the colon operator and arithmetic operators.)

- even numbers from 22 to 56
 - odd numbers from -17 to -39
 - the Sine function for values $\pi/8$, $2\pi/8$, $3\pi/8$, $4\pi/8$
 - the eighth through 10th powers of 2
 - a single vector consisting of the integers 1 through 5 forwards and then backwards.
-

Vector multiplication

You should already be familiar with vector and matrix multiplication; here we revisit the principles of vector multiplication in the context of Octave.

Given the above examples it may seem intuitive that two vectors should be multiplied in the same way as addition. Try multiplying two vectors that are of the same dimensionality:

```

octave:> aVec * bVec
error: operator *: nonconformant arguments (op1 is 1x11, op2 is
1x11)
error: evaluating binary operator '*' near line 50, column 6

```

What has happened? The dimensionality of the vectors is the same; yet Octave has given a **nonconformant arguments** error.

The operation of vector multiplication is not the same as element-wise scalar multiplication. Instead, it has a special meaning that is unique to vectors. When two vectors are multiplied, the elements are taken from the columns of the first vector and the rows of the second vector. In other words, the two vectors must be in different **orientations**. Let us see what happens when we multiply two vectors of equal lengths – i.e. the same number of elements – but transpose the second vector to be a column vector:

```
octave:> aVec * bVec'
ans = 1.65
```

The result is a single number: 1.65. This number is obtained by first multiplying each element of the first vector with each element of the second and then summing all the values to get a single number. This is a fundamental operation of Linear Algebra, which is the branch of mathematics that Octave uses to represent numerical computations.

The operation of multiplying two vectors produces either a scalar or a whole set of new vectors, called a matrix, depending on the orientation of the two arguments. This algebra is non-commutative; this means that if we change the order of the multiplication we usually get a different answer. This is different from scalar multiplication which is commutative, so changing the order of a multiplication does not affect the result and the same value is obtained.

Learning activity

There are some instances where matrix multiplication can be commutative. Give some examples where this is the case.

Some examples of vector multiplication in Octave follow:

First, let us define two vectors of the same **length** but in different orientations.

```
octave:> a = [0 1 2]
a =
0 1 2
octave:> b = [3 4 5]'
b =
3
4
5
octave:> size(a)
ans =
1 3
octave:> size(b)
ans =
3 1
```

The above example defines a row vector and a column vector. The `size()` function is used to discover the dimensions of each of the vectors: `size(a) == [1 3]` and `size(b) == [3 1]`. If we take the number of rows of the first vector, `a` and the number of columns of the second vector, `b`, the two resulting values inform us of the dimensionality of the output: in this case it will be `[1 1]`, which means one row and one column, which is simply a scalar.

Each column of the first vector is multiplied by each row of the second and the resulting answers are summed to yield the vector multiplication result. Using vector indexing we can generate the result manually:

```
octave:> a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
ans = 14
```

Here, each pair of elements was taken from the two vectors in turn and added to make a final value, 14. This is the value that we obtain if we multiply these two vectors:

```
octave:> a*b
ans = 14
```

A second rule for multiplication is that the second dimension of the first vector must match the first dimension of the second vector: in this case these dimensions are 3 and 3 so all is well. If we transpose one of the vectors then the second dimension of the first vector and first dimension of the second will not match.

However, if we transpose both of the vectors then we get two vectors that have dimensionalities $[3 \ 1]$ and $[1 \ 3]$ respectively. The first dimension of the first vector and the second dimension of the second vector forms a new object that has dimensionality $[3 \ 3]$. By the second rule of vector multiplication, the second dimension of the first vector matches the first dimension of the second vector, 1 and 1, respectively, so the operation is permitted. What results, then, is a matrix that is formed of three column vectors. These column vectors are the first vector $[0 \ 1 \ 2]'$ scaled by each of the elements of the second vector $[3 \ 4 \ 5]$:

```
octave:> a' * b'
ans =
0 0 0
3 4 5
6 8 10
```

Note that Octave uses the `*` symbol for vector and matrix multiplication.

3.3 What are signals?

For the work that follows, we take the view that a signal is a measurement that is done at regular time intervals. Examples of signals are the temperature reading from a thermometer taken at hourly intervals, the hours of daylight for each day of the year, your height measured annually or the number of sounds that you hear between successive clock ticks. Creative computing is concerned with signals that are organised for our senses.

You will find it very helpful to look at the introductory chapters of the recommended reading, or any other book on signal processing, to clarify the basic concepts that follow.

3.3.1 One-dimensional signals

The examples given above are all one-dimensional signals. That is, they are functions of one independent variable, **time**.

Let us construct a real signal using Octave.

Learning activity

You will need a clock or watch with a second hand for this activity.

Open Octave, type `sig1 = [` without hitting return.

Now, over a ten second period; count the number of sounds you hear between each clock tick and type the number into Octave followed by a space.

The first click is time 0. The second clock tick will be the first count – i.e. at time index 1. The third clock tick will yield the second count, at time index 2.

When you reach time index 10, type the last count and then type a `]` and hit return.

Octave now has a variable defined called `sig1` that provides information about the level of **sound activity** in your neighbourhood. There should be 10 samples in your signal. To verify, use Octave's `length` command:

```
octave:>length(sig1)
ans=10
```

Now you can display your signal using Octave's `stem` command:

```
octave:>stem(sig1,'*')
```

This command makes a plot that shows the height of your signal at each discrete time instant.

If all is well you should see your **sound activity** signal; it should look like Figure 3.1.

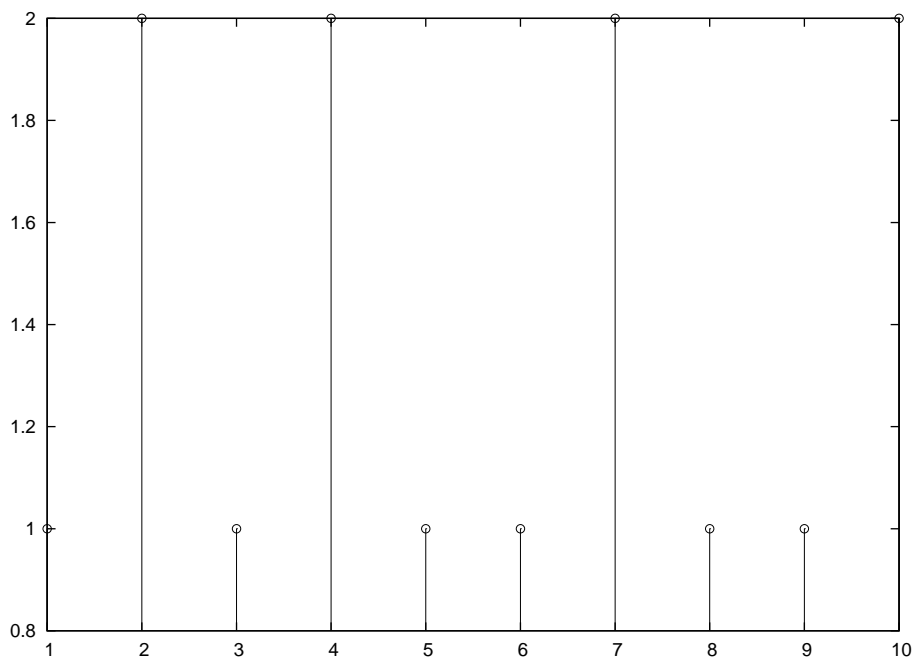


Figure 3.1: A stem plot of the sound activity signal.

For samples 2 through 9 it is easy to read the values. But the samples at time 1 and time 10 are not easy to see because they sit on top of the axis lines. To make the axis lines move outside of the range of our signal we can use Octave's `axis` command.

```
octave:>axis([0 11 0 3])
```

This takes a vector consisting of four values: `xmin`, `xmax`, `ymin`, `ymax`. Notice that the minimum value of the y-axis is located at the bottom of the graph; this is the reverse of graphical representations in Java-like languages. It is also often desirable to place grid lines on our plots so that it is easier to read the heights of the values. We can do this using Octave's `grid` command:

```
octave:>grid on
```

With these two commands executed you should now see a graph that looks like Figure 3.2.

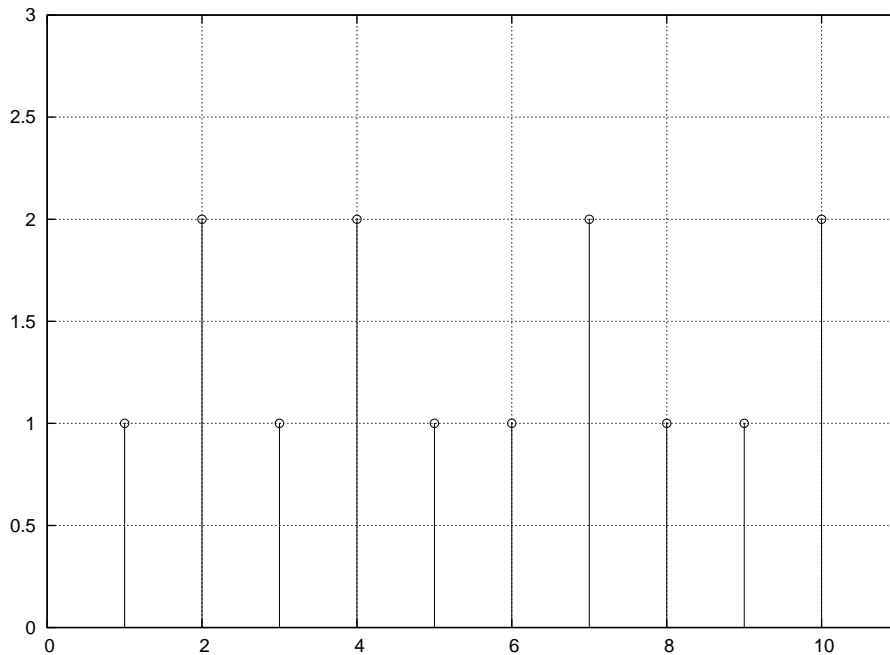


Figure 3.2: A stem plot of the same sound-activity signal but with the position of the axes changed so as not to overlap with the signal, and with grid lines added.

3.3.2 Octave representation of discrete-time signals

We can inspect the **sound activity** measurement for each of the time instants, $1 \dots 10$, in the above signal using the time point (in seconds) as an index:

```
octave:>sig1(1)
ans=1
octave:>sig1(2)
ans=2
octave:>sig1(10)
ans=2
```

For this signal we are fortunate that it does not have a meaningful value at time point 0. What would happen if we tried to access the signal's value at time point 0?

Learning activity

Try the above. what does Octave do?

Most signals have time indices that do not conveniently line up with Octave's vector indexing. The time index must be mapped into a vector index.

To do this we must choose a sample in our signal to act as the zero-time reference index. This could be vector index 1, or vector index 100, depending on the signal. Time point zero is often used to mean **now**, rather than simply the beginning of a signal. The meaning of 0 is determined by the application.

Once a zero-time reference index is chosen we may represent signals that have negative, zero and positive time index values. In this notation, the zero time index represents **now**; negative time indices represent **past** values of the signal and positive values represent **future** values of the signal. This is not to be taken literally; the concept of the zero time index is simply a reference point for various uses, such as to relate the time positions of other signals.

Figure 3.3 shows a cosine function that is sampled at nine equally-spaced positions. The spacing of the positions is $\pi/8$, but the time index is discrete, running from 0 to 8.

Here is the signal's construction in Octave:

```

octave:>t = 0 : pi/8 : pi
t =
  Columns 1 through 8:
    0.0  0.39270  0.78540  1.17810  1.57080  1.96350  2.35619  2.74889
  Column 9:
    3.14159
octave:>sigcos = cos( t )
sigcos =
  Columns 1 through 5:
    1.e+00    9.2388e-01    7.0711e-01    3.8268e-01    6.1230e-17
  Columns 6 through 9:
   -3.8268e-01  -7.0711e-01  -9.2388e-01  -1.e+00
octave:>stem(0:length(sigcos)-1, sigcos, '*')
octave:>axis([-1 length(sigcos) -1.5 1.5] )
octave:>sigcos( 0 + 1 )
ans = 1
octave:>sigcos( 8 + 1 )
ans = -1

```

In this example a number of convenient variables are used. The first is the definition of a vector of time values, t , at which to sample the cosine function. These are the time points which are distinct from the discrete-time index which runs from $0:\text{length}(\text{sigcos})-1$. In this case, the time points are related to the discrete-time index by a scalar multiplication by $\pi/8$:

```

octave:>t
t =
  Columns 1 through 8:

```

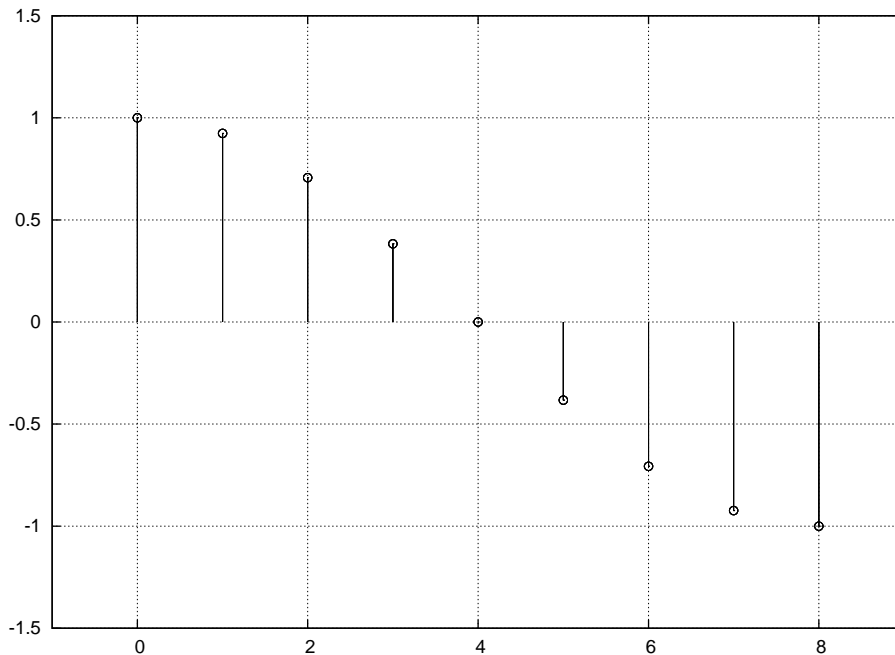


Figure 3.3: A cosine signal plotted using Octave’s `stem()` function. The time interval between samples is $\pi/8$ but the time index is discrete, running from 0 to the length of the signal minus 1. This **discrete time index** is a requirement in digital signal processing.

```
0.0 0.39270 0.78540 1.17810 1.57080 1.96350 2.35619 2.74889
Column 9:
3.14159
octave:>[ 0 : length(sigcos)-1 ] * pi/8
ans =
Columns 1 through 8:
0.0 0.39270 0.78540 1.17810 1.57080 1.96350 2.35619 2.74889
Column 9:
3.14159
```

This is the most common form of **discrete time index** used in signal processing. Finally, when requesting values from the signal vector, the lookup is performed using the discrete time index mapped to Octave’s vector indexing (`1:length(sigcos)` in this case). To do this we add the zero-time reference index 1 to the discrete-time index. To look at the value of `sigcos` at time-position 0, the vector index $0 + 1$ is used, and to access the value of `sigcos` at time-position 8 the vector index $8 + 1$ is used.

Of course, we did not **need** to write the vector index as $0 + 1$ and $8 + 1$ respectively, but we do this to remind ourselves that we map from the discrete-time index 0 and 8 to Octave’s vector index 1 and 9 using the zero-time reference index of 1. Figure 3.4 shows the same signal, but with the discrete-time signal elements individually labelled as $x[0], x[1], \dots, x[8]$. When we notate signals we must include the square brackets containing either a number or a variable to show that the value of the signal depends on a discrete-time index. In general we represent a signal when we write it down as $x[n]$; thus showing that it is a function of the independent discrete-time index n .

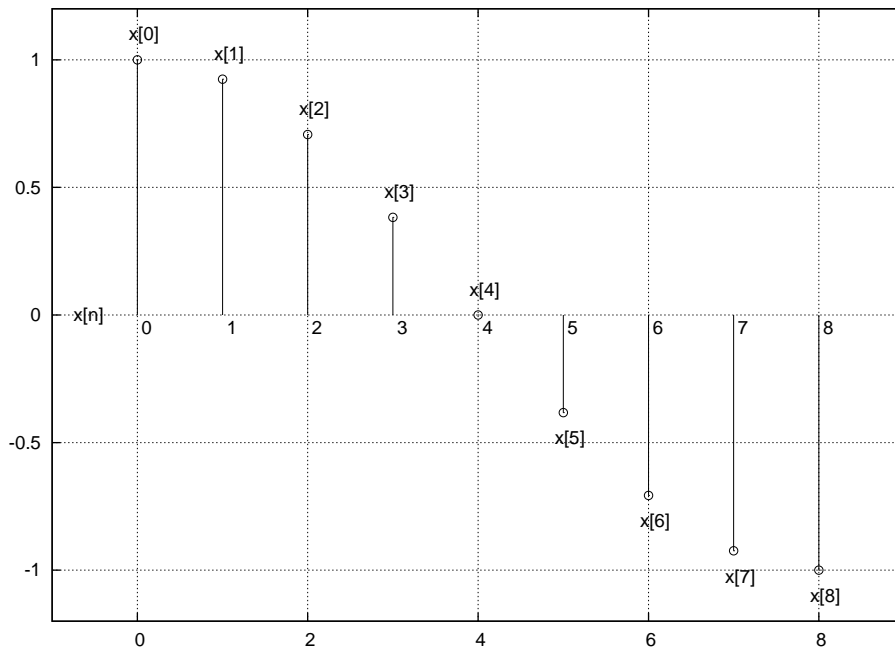


Figure 3.4: A cosine signal sampled at intervals of $\pi/8$ from 0 to π yielding a vector of nine elements. While the sampling interval of the nine samples is $\pi/8$, the signal has a discrete time index in the range 0 to 8. Each time index represents a sampling interval of $\pi/8$.

Figure 3.5 shows a stem plot of a different signal. This time the signal has negative, zero and positive discrete-time indices. The vector indices are in the range 1 to 10; but the discrete-time index runs from -4 to $+5$. When plotting the signal, the discrete-time index $[-4 : 5]$ is provided to Octave's `stem()` function.

Learning activity

Use the following Octave code, but fill in the correct values in the vector in the first line, to produce the signal in Figure 3.5.

```
octave:>sig2=[ ]
>stem(-4:5, sig2, '*');
>axis([-5 6 -6 5])
```

Here, the discrete-time index 0 of the signal corresponds with vector index 5. Therefore, vector-index 5 is the zero-time reference index in this example. To access values of the `sig2` vector in Octave by their discrete-time index we must add the zero-time vector reference index 5.

Learning activity

What is the effect of including the `;` in the above example? What would happen if you did not have it?

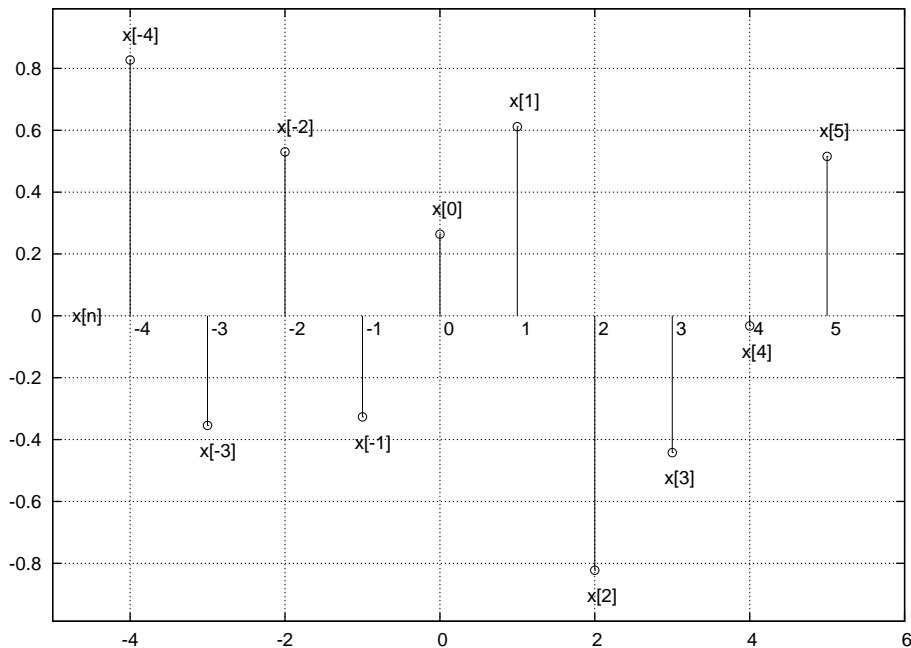


Figure 3.5: A discrete-time signal; the x-axis is the independent variable, usually representing time. The y-axis is the signal's value at each discrete-time instant. A vector, x , holds the values of the signal; the individual samples of the signal are accessed by vector indexing, $x[1]$, $x[2]$, etc.

```
octave:>sig2( -4 + 5 )
ans = 3
octave:>sig2( 1 + 5 )
ans = 2
```

The concept of zero-time reference index is independent of the actual size of the vector. We can use a time index that is anywhere in the range of integers that can be represented on a given machine architecture. For example, a signal consisting of 10 samples makes a vector of length 10. However, the zero-reference time index might be -10. The range of discrete-time indices would then be -10:-1 with zero-reference time index 11, which is a greater value than the length of the signal vector.

It does not matter that the zero-reference time index is greater than the length of the signal vector if we only want to access values in the discrete-time range $-10 : -1$. For example,

```
octave:>sig3 = [ 2 : 2 : 20 ]
sig3 =
2 4 6 8 10 12 14 16 18 20
octave:> sig3( -10 + 11)
ans = 2
octave:> sig3( -1 + 11)
ans = 20
octave:> sig3( 3 + 11)
error: invalid vector index = 14
```

In the last command the vector index was out of range. This is because we tried to

access the signal using a discrete-time index that was out of range of the signal which we determined would represent time indices $-10 : -1$. Familiarity with discrete-time indexing and its relationship to vector indexing in Octave will be needed for working with discrete-time signals and systems.

Learning activity

Construct the following signals in Octave; for each signal, write down the zero-time reference vector index and use it to access the first and last values of the signal vector using the **discrete-time index**:

- the odd numbers from 23 to 3 with discrete-time index $-11 : -1$
 - sine function sampled at intervals of $\pi/4$ from 0 to $2 * \pi$ with discrete-time index $0 : 8$
 - even numbers from 100 to 120 with discrete-time index $10033 : 10053$.
-

3.3.3 The unit impulse

The fundamental building block of discrete-time signal processing is an entity called the unit impulse. We can think of the unit impulse as being the number 1 represented as a signal at discrete-time index 0. The remainder of the signal is zero. The unit impulse is often called the **delta** function so it is a signal called $\delta[n]$, where n is the independent variable.

We can construct a function `imp` in Octave that returns the unit impulse:

```
octave:1> function result = imp(length,position)
> zeros = zeros(1,length)
> zeros(position)=1
> result = zeros
> endfunction
```

This user-defined function takes two arguments: the first is the length of the signal to generate and the second is the position of the 1 within the signal. This position is the zero-time reference index for the vector. We can now use the function, as shown below:

```
octave:>imp(11,6) % Make a unit impulse

zeros =

  0  0  0  0  0  0  0  0  0  0  0

zeros =

  0  0  0  0  0  1  0  0  0  0  0

result =

  0  0  0  0  0  1  0  0  0  0  0

ans =
```

```
0 0 0 0 0 1 0 0 0 0 0
```

```
stem(-5:5,imp(11,6),'*')
```

In this example the impulse is centred in a signal of length 11. The position of the impulse is the zero-time index 6. Figure 3.6 shows the unit impulse stem plot labelled with the signal's common name: $\delta[n]$; the 1 occurs at time 0 denoted by $\delta[0]$. Note that it is possible to define the function in such a way that it does not output the values of zeros, etc., while it is being executed, by using appropriate semi-colons at the end of relevant lines.

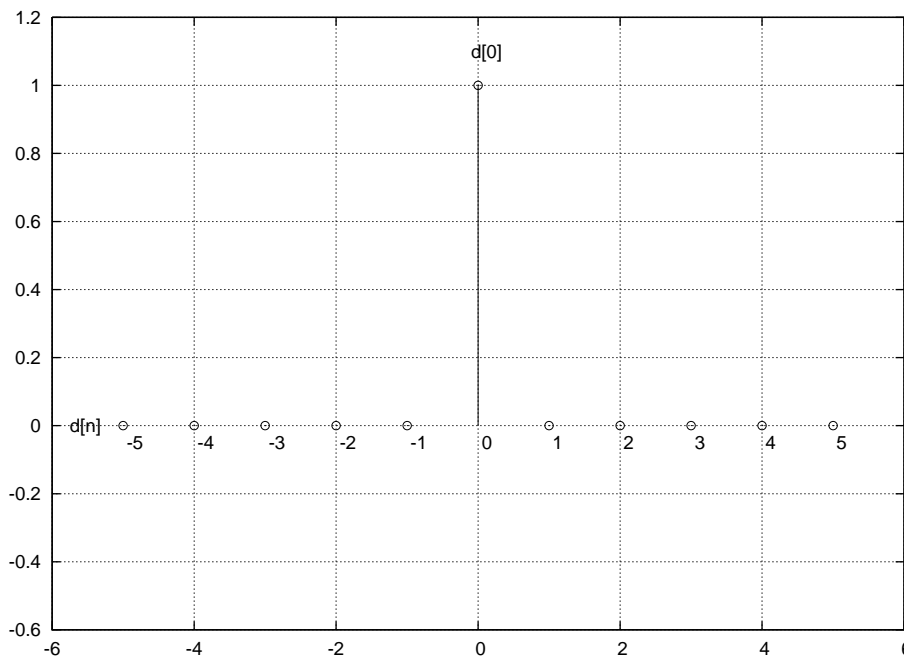


Figure 3.6: The unit impulse, or delta, is a fundamental building block of discrete-time signal processing. It consists of a single non-zero sample with value 1 at time index 0.

We shall see in the next chapter that the unit impulse is the basis for every signal and signal operation in DSP.

3.3.4 The unit step

The unit step signal is one that is all ones for time indices zero and higher, and zeros for negative time indices. It is named $u[n]$ because it is a fundamental building block of discrete-time signal processing. The unit step signal can be constructed in Octave using two convenient functions: `zeros()` and `ones()`:

```
octave:> u = [ zeros(1,10) ones(1,11) ]
u =
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
```

Here the zero-time reference is vector index 11. The `zeros()` function accepts two arguments which are the number of rows and columns of zeros to generate.

Similarly the `ones()` function also takes the same arguments but generates ones instead of zeros. Putting these two functions together allows construction of signals containing runs of ones and/or zeros as in the example above. Figure 3.7 shows the unit step signal for discrete-time index $-10 : 10$.

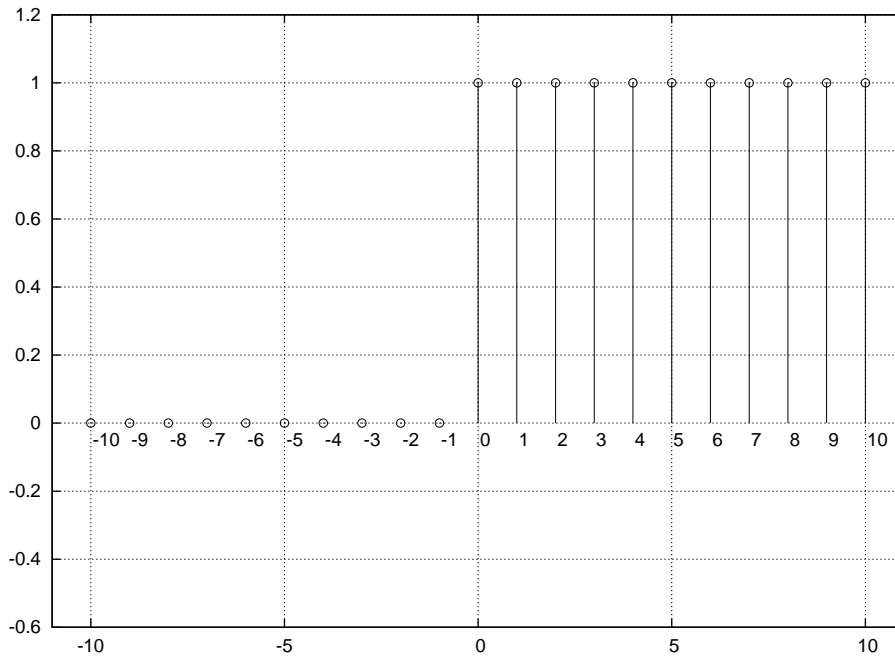


Figure 3.7: The unit step is also a fundamental building block of discrete-time signal processing. It consists of zeros for negative discrete-time indices and all ones for indices zero and higher.

Octave has a built-in function that can differentiate signals. This means that it subtracts the value at each time point from the value of the previous time-point to generate a new signal. For example:

```
octave:> a = [ 0 1 3 6 10 15 21 28 36 45 55 ]
a =
0 1 3 6 10 15 21 28 36 45 55
octave:> diff(a)
ans =
1 2 3 4 5 6 7 8 9 10
```

What happens when you differentiate the unit step signal above? Which signal does this produce?

3.3.5 The unit delay

Another of the fundamental building blocks of DSP systems is the unit delay. This signal is exactly the same as the Delta signal $\delta[n]$, but it has its non-zero sample at time-index 1 instead of time-index 0.

Figure 3.8 shows a stem plot of the unit delay signal. The concept of delay is fundamental to much DSP processing as we shall see in the next chapter. The unit

delay is the simplest possible representation of the concept of delay. It has the same name as the delta signal, which was notated $\delta[n]$, but to represent the delay the time index is denoted $n - 1$, so the unit delay is notated as $\delta[n - 1]$.

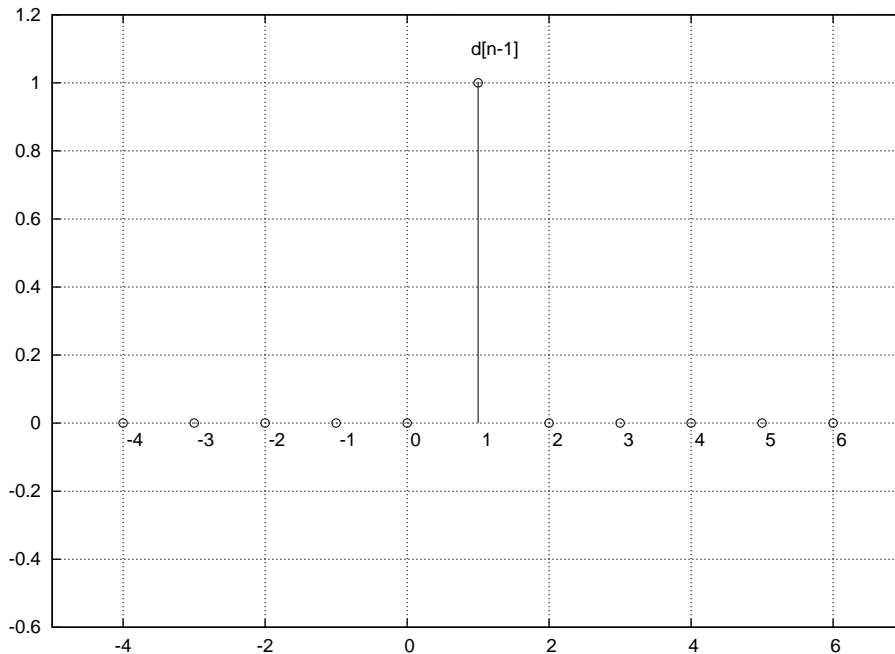


Figure 3.8: The unit delay signal is a time-delayed delta signal. The delay is one sample, which means the time index of the non-zero value is 1 compared with 0 of the delta signal. The impulse has been delayed by one sample to occur later in time.

This notation indicates that a delay is a subtraction operation that acts on the **discrete-time index** of the signal rather than on the signal itself. This process means that the Delta signal should be looked-up one sample before, -1 , the given discrete-time index, n . Putting this together, the unit delay is the Delta signal **transformed** to a new signal by performing an arithmetic operation on its time index.

A signal can be delayed by any number of samples using this notation. Consider Figure 3.9. Here the Delta signal has been delayed by two samples. The notation is now $\delta[n - 2]$ to show that the signal is constructed by looking-up the Delta signal two samples prior to the given time-index.

Just as we can delay a signal by a given number of samples, we can also advance it in time. Figure 3.10 shows the Unit Advance signal. This is also constructed out of the Delta signal, but the notation is now $\delta[n + 1]$ to indicate that we lookup the Delta signal one sample ahead of the given time index. A signal may be advanced by any number of samples; furthermore the relationship between time-advance and time-delay is now obvious since they are both simply arithmetic operations upon the time index of a signal. A negative time-delay is equivalent to a time advance; can you see a mathematical reason why this is so?

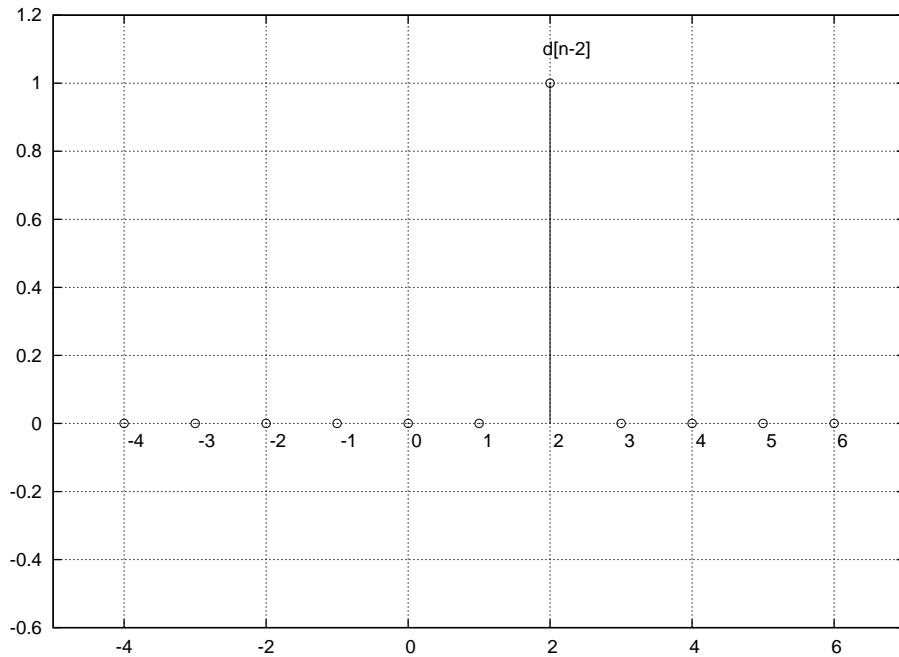


Figure 3.9: The Delta signal delayed by two samples. The notation for delaying the delta signal, $\delta[n]$, by two samples is $\delta[n - 2]$ indicating that the Delta signal is to be read at a given time-index, n , minus two samples. If we do this, the value 1 is produced at time-index $n = 2$, because $\delta[2 - 2] = \delta[0] = 1$ later in time.

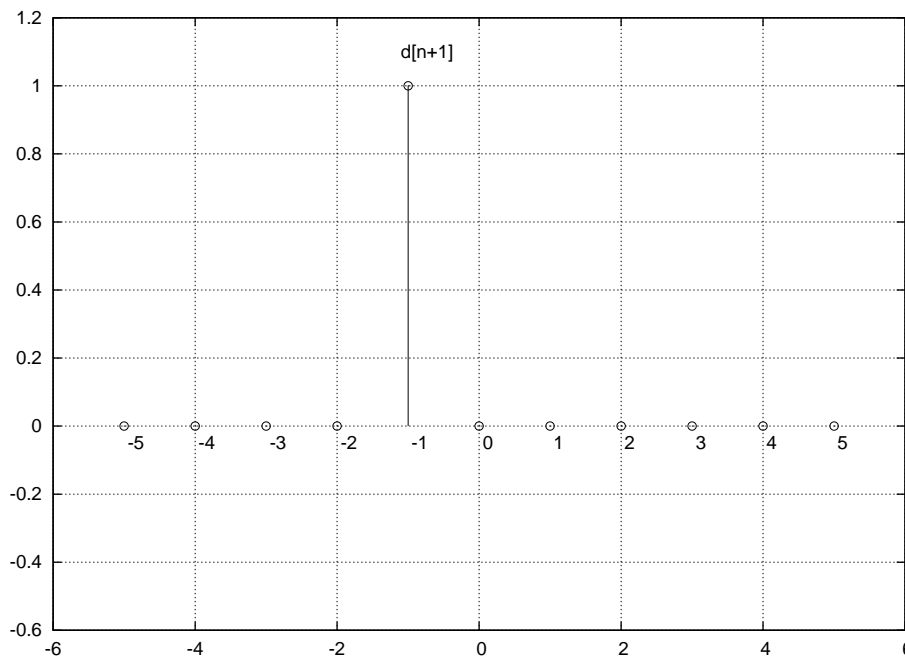


Figure 3.10: The unit advance signal is a time-advanced delta signal. The advance is by one sample, which means the time index of the non-zero value is -1 compared with 0 of the delta signal. The impulse has been advanced by one sample to occur earlier in time by one sample.

3.3.6 Delay operations in Octave

Delay operations in Octave can be implemented by careful use of the discrete-time index, the zero-time reference vector index and the time-index arithmetic implementing delays as discussed above. We must ensure that there is enough space inside our vector to perform the given delay operation; we do this by padding the signal at the beginning and end with zeros, making note of the zero-time vector index reference.

Consider the Delta signal padded with ten zeros on either side of time-index zero giving zero-time vector reference index 11:

```
octave:> d = [ zeros(1,10) 1 zeros(1,10) ]
d =
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
octave:> d( 0 + 11 )
ans = 1
```

The Delta function can be delayed by one sample, making it a unit delay, by subtracting one from the discrete-time index:

```
octave:> d( 0 - 1 + 11 )
ans = 0
octave:> d( 1 - 1 + 11 )
ans = 1
```

To understand this, we have now used three values for indexing the signal vector. The first value is the discrete-time index, the second is the amount of delay to introduce into the signal and the third value is the zero-reference vector index (because Octave uses indices that run from 1 to the length of the signal). We can perform time advances as either an additive value to the discrete time index, or equivalently, as a negative delay, thus subtracting a negative number from the discrete-time index.

```
octave:> d( 0 - -1 + 11 )
ans = 0
octave:> d( 1 - -1 + 11 )
ans = 0
octave:> d( -1 - -1 + 11 )
ans = 0
```

Here the only value of the discrete-time index that outputs the value 1 is time -1. Hence the discrete-time index is advanced by one sample before looking up the value in the delta signal. Fortunately, Octave provides a more convenient method to perform time-delays and time-advances on signals than that of using three different indices. The function is called `shift()`:

```
octave:> d
d =
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
octave:> shift(d,1) ans =
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
octave:> shift(d,-1)
ans =
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
```

We now have enough knowledge about signals to start to use them to represent sound and images for creative computing. The remainder of the Chapter is devoted to the representation and display of sound and image as signals.

3.4 Audio signals

The ear/brain system interprets certain types of motion in air as sound. The conditions under which movement in air is heard as sound are complex. When an object in air undergoes a displacement, the **air pressure** changes a small amount, near the object, for a short period of time. When a back-and-forth motion is introduced into the air by an object, called an oscillation, the displacement of air is also in a back and forth motion causing oscillations in the air pressure.

In general, sounds are caused by objects in the world that vibrate. The vibrations that the ear is sensitive to are those that occur between the range of 20 cycles per second and 20,000 cycles per second. A cycle is a single back-and-forth motion, so objects that move back and forth more than twenty times per second, but less than 20,000 times per second, cause pressure changes in the air that can be heard as sound.

Air is a gas, so when some of the air molecules are displaced they, in turn, displace air molecules near them forming a chain reaction in air pressure changes that is called a pressure wave. The change in air pressure can be detected at a remote point in space because it is transmitted via the air molecules to that point.

Audio signals represent the displacement of air at a single point in space as measured by a microphone membrane. Displacement of the membrane generates an electrical signal that is proportional to the displacement. The process of digital sampling is that of recording the level of the electrical signal at fixed time intervals called the sampling period. The measured value is recorded in digital memory as either an integer or a floating point number for each sample.

To play back an audio signal the digitally sampled signal must first be converted back to an analogue electrical signal and then this signal used to displace the cone, or membrane, in a loudspeaker or headphones. The operation of moving the air using a loudspeaker generates movement in air that is similar to the movement that we measured with a microphone.

Again, reference to a relevant textbook will be helpful for you, as these are standard audio signal processing concepts.

3.4.1 Sampling

As mentioned above, sampling is the process of measuring the value of some phenomenon at regular intervals, called the sampling period. If measured in seconds, the sampling period is inversely proportional to the sampling frequency, also called the sample rate: SR . To capture the variation in the measured value of a phenomenon, such as sound, we must know in advance what the maximum frequency will be. Then we make sure that we measure the source often enough to capture the variation at this maximum frequency.

The maximum frequency that is audible by humans is in the region of 20kHz. So we

must be able to capture variation in a signal that is close to 20kHz. To do this we must use a sampling rate that is at least **twice** the maximum frequency that we want to measure. This is because the maximum frequency that can be detected by sampling is **half** the sample rate: $SR/2$. This latter frequency is called the **Nyquist frequency** named after the engineer Harry Nyquist who first recognised its importance. If we make the Nyquist frequency the maximum frequency we want to detect, such as 20kHz in the case of audio, then the sampling rate will be double this maximum frequency: 40kHz.

The most commonly used sample rate used for digital audio is 44.1kHz which is just slightly above that required for detecting frequencies at 20kHz. Other common sample rates are 22.05kHz, 11.025kHz, 96kHz, 48kHz, 32kHz, 16kHz and 8kHz. What are the Nyquist frequencies for each of these sampling rates?

Learning activity

Show why if you do not sample at twice the frequency, you can lose vital information about the wave.

Digital audio

Digital audio is the application of discrete-time signal construction and signal processing to applications using sound. As such, the sine and cosine functions, $\sin()$ and $\cos()$, are often used as sound atoms; they are the building blocks of digital audio.

The definition of the sine and cosine functions comes from the coordinates of the unit circle. At each point on the unit circle, the angle of the line from the origin to that point has the cosine value of the point's x-coordinate and the sine value of the point's y-coordinate.

Figure 3.11 illustrates the relationship between angle, a point on the unit circle and the $\sin()$ and $\cos()$ functions. To construct a sine wave in time, we consider a point starting at a given angle on the unit circle, and imagine it travelling at a constant speed around the circumference of the unit circle. At each moment in time, the height of the point above or below the x-axis makes the sine wave through time.

Audio occurs in time and is due to oscillation in air. A fundamental form of oscillation is a sine wave; which is the $\sin()$ function through time. When played as a sound a sine wave is heard as a pure tone, such as the sustained part of a tuning fork sound.

3.4.2 Frequency

The frequency of a sine wave is the number of oscillations, or cycles, per second. That is, the number of complete revolutions around the unit circle tracing out the height of a point travelling on its circumference. The faster the point travels around the unit circle, the higher the frequency.

From elementary mathematics, the circumference of a circle is $2\pi r$ with r the radius. The unit circle has radius 1 so the circumference has length 2π . For a given point on

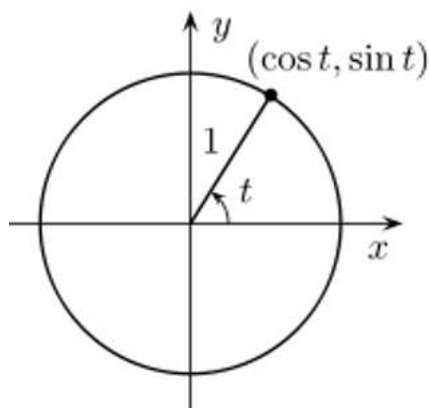


Figure 3.11: A circle of radius 1. For angle t , a line from the origin meets the circle at the point $(\cos(t), \sin(t))$

the unit circle, the distance around the circumference of the unit circle to that point from the intersection with the positive x-axis is the angle in **radians**.

For example, a point that is a quarter of the distance around the unit circle, measured counter clockwise, has an angle of $2\pi/4 = \pi/2$ radians. The sine function is the height (as measured on the y-axis) of the point at this angle, which is 1. So $\sin(\pi/2)$ has the value 1.

To return to the frequency of a sine wave, a point that is travelling at 1Hz frequency around the unit circle traverses a distance of 2π once every second. A point travelling at 50Hz traverses the unit circle 50 times every second, so it travels a distance of 2π in $\frac{1}{50}$ of a second. Therefore, the frequency, f , of a sine wave expressed as cycles per second, or Hertz, is the distance travelled per second which is $2\pi f$. For $f = 50\text{Hz}$ the distance is $2\pi f$ radians per second. So we often call the frequency of a sinusoid its angular frequency because it is expressed in radians.

Now to construct a sine wave, all that is needed is the angular frequency, in radians, and a time index. The time index is a vector of time points, in seconds, at which to sample the sine wave. By the Nyquist theorem we know that we must sample a waveform at a rate of at least twice the highest frequency that we want to measure. A sine wave has exactly one frequency, its angular frequency. So we must be sure to sample the sine wave at at least twice this angular frequency. However, in practice it is best to make sure that the sampling rate is much greater than this, to avoid possible boundary effects near the limit. As an exercise, you should try to establish what happens with very low sample rates, to see what is meant by the possibility of boundary effects.

For digital audio, typical sampling rates are: 8000Hz, 16000Hz, 32000Hz, 44100Hz, 48000Hz, 96000Hz. The sample rate is measured in Hertz (Hz) because it is a fixed number of samples per second.

As an example, let us construct one second of a sine wave with angular frequency 400 Hz at a sample rate of 8kHz (8000Hz).

We first generate a time index that samples the time interval 0s to 1s in steps of $1/8000$ s. Then we construct a sine wave using an angular frequency and the sampled time intervals.

```
octave:>t = [ 0 : 1/8000 : 1 ] ;
octave:> x = sin( 2 * pi * 400 * t ) ;
octave:>stem(t(1:20), x(1:20), '*')
```

Note that this example uses a semicolon at the end of each line that generates a vector. A semicolon in Octave is used as a command separator, but it has another function, that is to suppress the printing of results of operations to the terminal. The vectors for audio signals are very long – in this case 8001 samples – so we want to suppress Octave’s default behaviour of printing vectors to the screen. When you have made your plot of the sine wave you should see a figure like that of Figure 3.12.

We computed 8001 values for one second of the sine wave. The extra sample is because we sampled up to and including the 1s boundary, which made the example easier to read. To generate exactly 8,000 samples, i.e. exactly one second of audio, we should use: $t = [0 : 1/8000 : 1 - 1/8000]$ or, equivalently, $t = [0 : 7999] / 8000$.

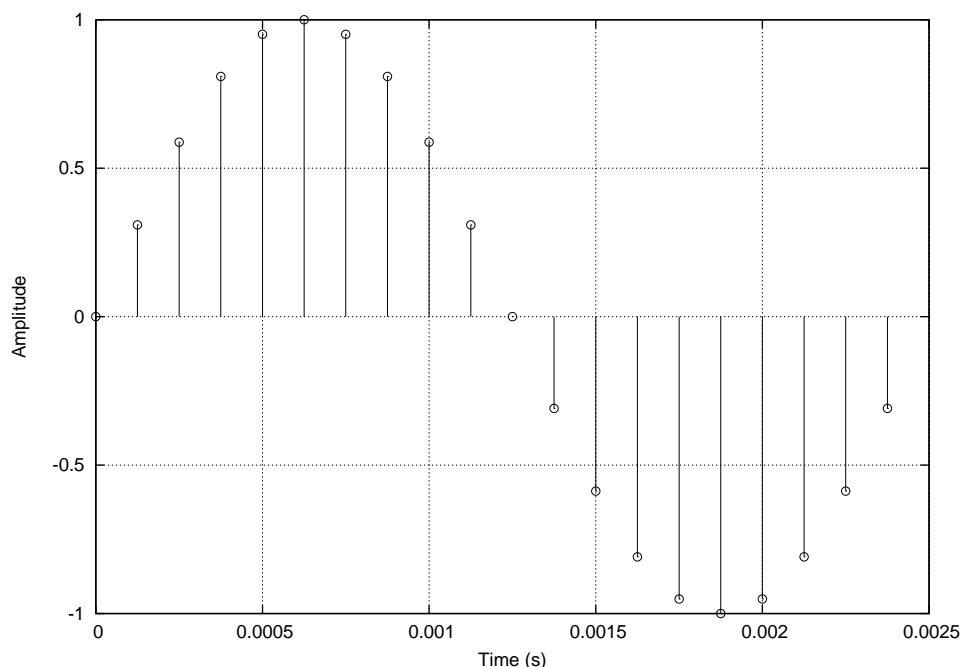


Figure 3.12: One complete cycle of a 400Hz sine wave sampled with a sampling rate of 8000Hz (8kHz). The time index is in seconds.

The example above plotted only 20 samples of the signal instead of the entire signal. This is for two reasons: the first is that 8,000 samples is a lot of data, and plotting it would produce an unreadable graph. Secondly, this number of samples was chosen to plot because it is exactly one complete cycle out of the 400 cycles in the one second signal that we generated.

Plotting an audio signal is useful to check that we have performed the correct operations. We see an oscillating waveform that has a maximum height of 1, and that the duration is 0.0025 seconds, which is $1/400$ s; that is, the amount of time that a single cycle of a 400Hz signal occupies. But plotting the signal is not the same as **hearing it**, so let us listen to the entire signal.

We can play digital audio in Octave using the `sound()` function. Sound takes two

arguments, a signal and a sampling rate:

```
octave:>sound(x, 8000)
```

When you hit the return key, you should hear a pure tone that lasts for one second. You should also see Octave print out some information about the sound:

```
Input File      : '-' (au)
Sample Size    : 16-bit (2 bytes)
Sample Encoding: signed (2's complement)
Channels       : 1
Sample Rate    : 8000
Time: 00:00.12 [4473:55.33] of 4473:55.46 ( 0.0%) Output Buffer: 5.80K
play sox: 'resample' clipped 4 samples; decrease volume?
play sox: alsa: output clipped 2 samples; decrease volume?
Done.
```

If you do not see information like this (it will be different on different operating systems) but instead see an error then you must check your computer's audio settings. It is best to check things in the following order:

- Is the volume turned up? (Find the audio control panel on your computer and check this.)
- Is there a cable plugged into the headphone socket on your computer?
- If yes, then is the cable connected to headphones, speakers or an external audio device?
- If speakers and/or external audio device, are they switched on?
- Double check your setup by playing a music file from **iTunes** (Mac) or **WindowsMediaPlayer** (Windows) or **RhythmBox/mplayer** (Linux).
- If you cannot hear sound from a media player then you may need to purchase a sound card for your computer; or there may be something wrong with the audio facility on your computer.
- If you can hear sound from a media player but not in Octave, check the internet for information about your operating system's version of Octave: search for Octave Audio **Windows, Mac or Linux**. You may need to install a software package for audio support: such as **sox**, <http://sox.sourceforge.net/>, which is also free software.

Learning activity

Make the following sine waves and play them using the `sound()` command in Octave: (Note: If you cannot get the sound command to work in Octave, but your media player works, then you can save your sound using the `wavwrite()` command and then open it in your media player to hear it.)

- a 1s 1000Hz sine wave at 8000Hz sample rate
- a 1s 50Hz sine wave at 8000Hz sample rate
- a 1s 8000Hz sine wave at 32000Hz sample rate
- a 1s 11000Hz sine wave at 44100Hz sample rate.

For each of the sine waves above:

- make a stem plot of one cycle of the sine wave: (the number of samples in one cycle is SR/f where SR is the sample rate and f is the angular frequency of the sine wave)
 - play the sound of one cycle of the sine wave (instead of one second): what do you hear?
 - play the sound of five cycles of the sine wave (instead of one second): what do you hear?
 - play the sound of twenty cycles of the sine wave (instead of one second): what do you hear?
-

3.4.3 Amplitude

The maximum height of the waveform is its **amplitude** or **peak amplitude**. In Octave, sounds are represented by floating-point numbers in the range -1.0 to 1.0 ; any sound that has a greater range than this is **scaled** so that it fits within this range.

The sine waves we generated above have amplitude 1; so they are at the maximum range for sound in Octave.

The perceived intensity of the sound is relative to its amplitude. To change the perceived intensity of a sound the waveform is divided (or multiplied) by a scalar. For example, dividing the sound by 10 will result in an amplitude that is 10 times smaller. However, the sound that is heard will be perceived as approximately **half** as loud. This is due to the way the ear responds non-linearly to changes in a sound's amplitude.

A common measure of a sound's level, or loudness, is decibels (dB). Decibels can be computed from a sine wave's amplitude using the following expression:

$$\text{decibels} = 20 * \log_{10}(\max(\text{sig}))$$

For a sine wave at full volume the loudness – relative to the gain in the speakers – is at 0dB . This is called the reference; all other loudness levels are measured with respect to this reference loudness.

We might be able to amplify the output of Octave using speakers so that the loudness is perceived to be the same as normal conversation. Given this loudness reference, Table 3.1 is a guide for relative loudness of different sounds at different decibel levels:

dB	Amplitude factor	Sound	Perceived loudness
-0	1	Normal conversation	1
-10	0.3162	Quiet suburban outdoors	1/2
-20	0.1	Buzz of mosquito	1/4
-30	0.03162	Watch ticking	1/8
-40	0.01	Quiet house interior	1/16
-50	0.003162	Rustling leaves	1/32
-60	0.001	Threshold of hearing (silence)	1/64

Table 3.1: Table of relative loudness levels for sounds across the entire range of human hearing.

A change of -10dB represents half the perceived loudness; and a change of -20dB is a change of half of a half which is a quarter of the loudness. Therefore each loudness step in Table 3.1 is a perceived change of a factor of two in loudness.

To change the loudness of a sine wave we multiply the signal by a scalar. To do this, first turn the speakers up to the reference loudness of 0dB and then **attenuate** the signal by multiplication with a scalar < 1.0 to the desired loudness level relative to the 0dB reference.

```
octave:>sound( x, 8000 ) % reference level of 0dB
octave:>sound( 0.31623 * x, 8000 ) % -10dB = half the loudness of 0dB
octave:>sound( 0.1 * x, 8000 ) % -20dB = quarter of the loudness of
0dB
octave:>sound( 0.031623 * x, 8000 ) % -30dB = eight of the loudness of
0dB
octave:>sound( 0.01 * x, 8000 ) % -40dB = sixteenth of the loudness of
0dB
```

As well as sine waves, we can generate noise very easily in Octave. Noise is simply uniform randomness which in Octave is generated by the `rand()` function. This function takes two arguments: the number of rows and columns of the vector that it generates. It generates uniform random real numbers in the range 0 to 1; to make them occupy the full audio range we multiply the output of the `rand()` function by 2 and subtract 1. Now the random numbers will be in the range -1 to 1 as in the following example.

```
octave:>x = rand(1,8000)*2-1;
octave:>sound(x,8000);
```

When you play this example you should hear a loud noise that lasts for one second. Noise is another fundamental unit for constructing audio signals and it is very often used to make synthetic percussion sounds such as snare drums, cymbals and other instruments. Noise has a very interesting property with regard to its frequency content. Recall that a sine wave has exactly one frequency present, while noise has **all** frequencies present in the signal (at random amplitudes). This remarkable fact makes uniform random noise a very useful signal. Figure 3.13 shows a stem plot of a randomly-generated signal.

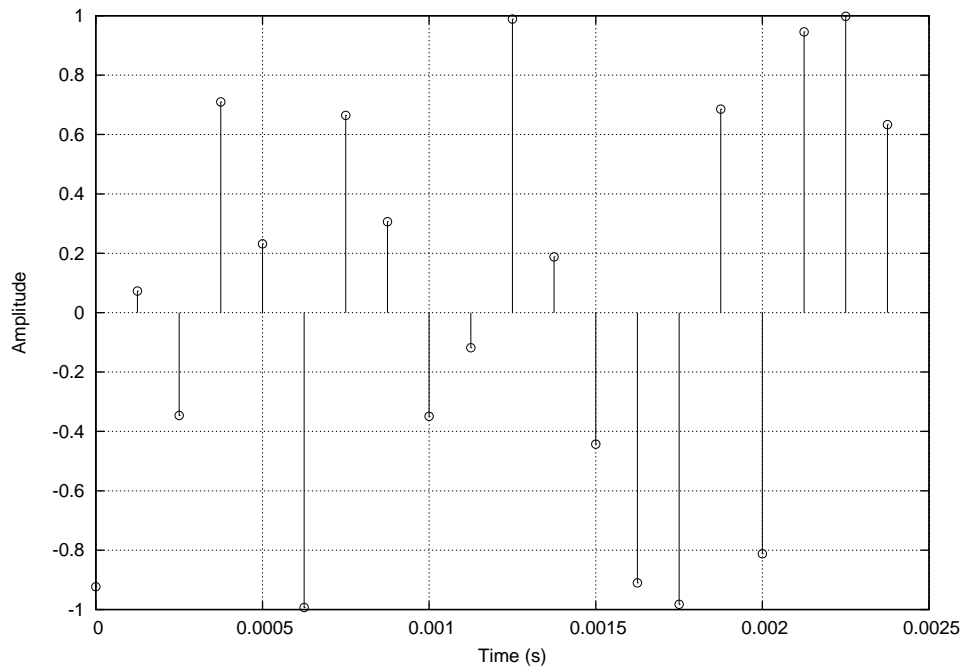


Figure 3.13: Twenty samples of a random signal sampled at 8000Hz. It is not possible to select a single cycle of a random signal because **all** frequencies are present. The signal is generated in the range -1 to 1 so it has a peak amplitude of 1 .

Learning activity

Using the noise signal, x , generated above, play back the sound at the following different loudness levels:

- -10dB
- -20dB
- -30dB
- -40dB
- -50dB
- -60dB .

At what level can you no longer hear the sound?

3.4.4 Phase

Other than amplitude, angular frequency and a time index, there is another parameter available for sine waves; that is the phase offset. A phase offset occurs when the sine wave starts at an angle (in radians) other than 0 . Figure 3.14 shows a sine wave with the same frequency and amplitude as Figure 3.12 but that starts, at time 0 , with a phase offset of $\pi/8$ radians.

Instead of starting with a height of 0 at time 0, this sine wave starts with a value of 0.38268 at time 0. This corresponds to the value of a sine wave at $\pi/8$ radians. The remaining values continue the sine function for the given angular frequency but they are all offset by $\pi/8$ radians.

To implement a phase offset in Octave we add the phase offset to the **instantaneous phase** argument of the `sin()` function:

```
octave:>x = sin( 2 * pi * 400 * t + pi/8);
octave:>stem(t(1:20),x(1:20))
```

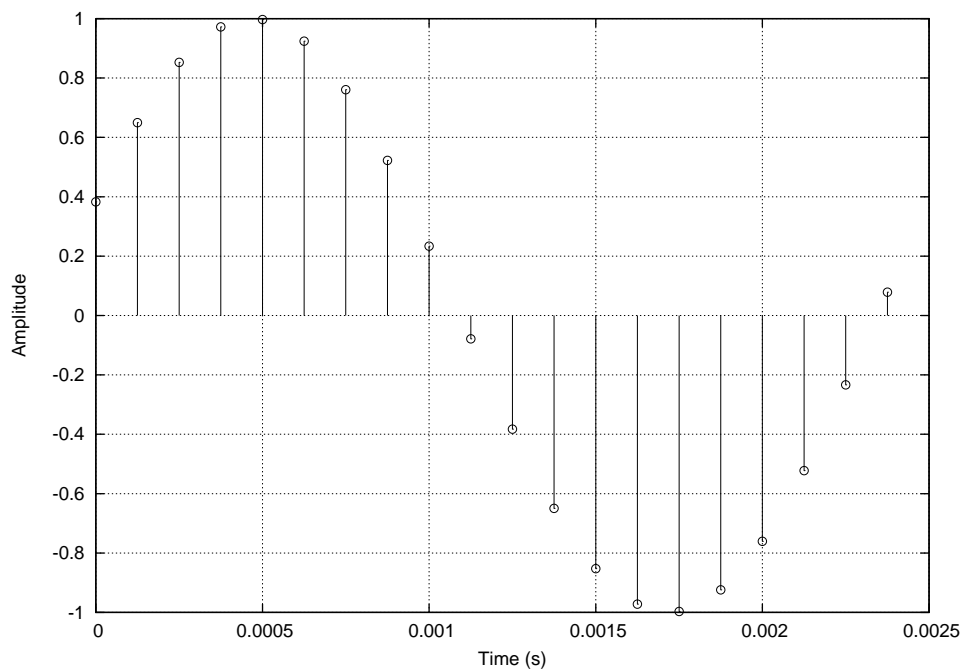


Figure 3.14: One complete cycle of a 400Hz sine wave with phase offset of $\pi/8$.

We can offset a sine wave by any angle. For example, let's see what happens when we offset the sine wave by an angle of $\pi/2$ radians:

```
octave:>x = sin( 2 * pi * 400 * t + pi/2);
octave:>stem(t(1:20),x(1:20))
```

One cycle of the resulting signal is shown in Figure 3.15. What is the value of this signal at time 0? Contrast this with the value of the signal with phase offset of 0. Which mathematical function has been generated by a sine wave with a phase offset of $\pi/2$?

Additive synthesis

A sinusoid is a sine wave that can have any amplitude, frequency and phase offset; i.e. `sin()` and `cos()` are both sinusoids; they are instances of the same curve but they are offset by $\pi/2$. It is a remarkable fact about sinusoids that a number of them can be summed to make **any** signal. There must be the correct number with correct amplitudes, frequencies and phase offsets, but it is a fact that they can be

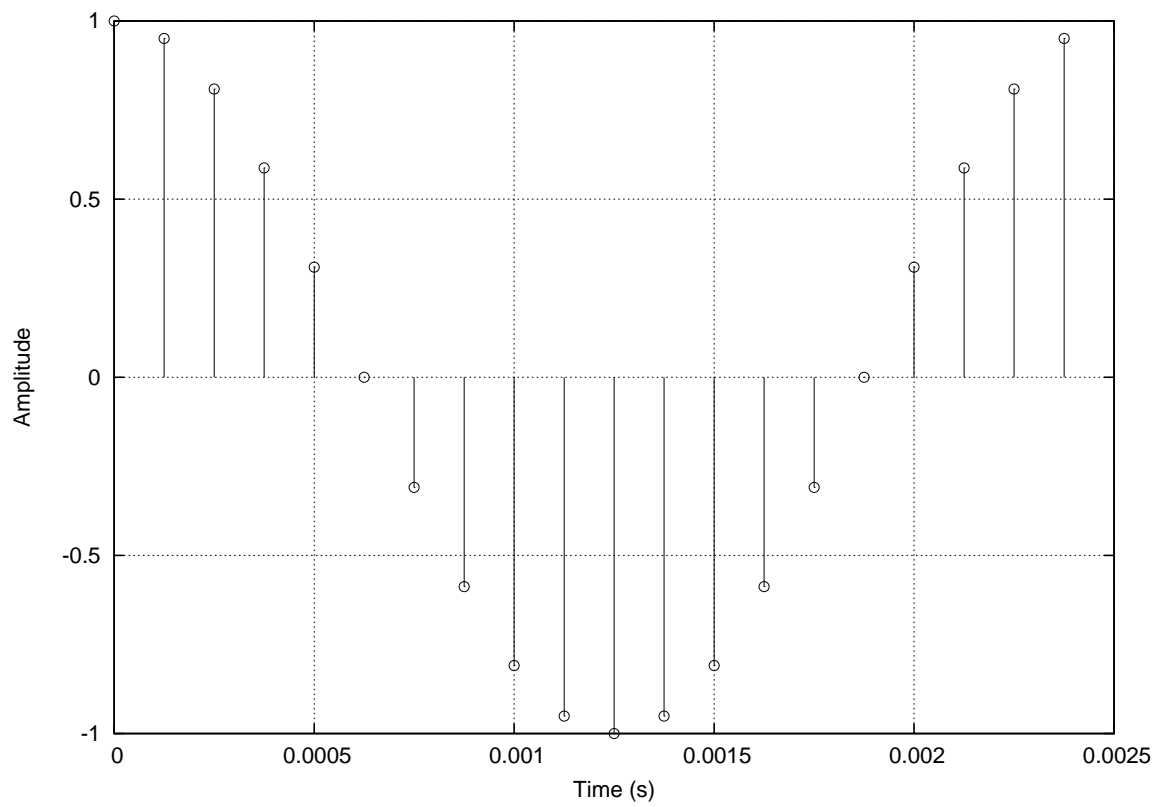


Figure 3.15: One complete cycle of a 400Hz sine wave with phase offset of $\pi/8$.

seen as the fundamental elements of **all** signals. Therefore, sinusoids form a basis over signals; they are the sound atoms from which signals can be constructed.

To see how sinusoids can be used to make complex waveforms we consider the simple case of adding sinusoids with different amplitudes and frequencies. First, let us add two sinusoids, where the second has double the frequency of the first:

```
octave:> x1 = sin( 2 * pi * 400 * t);
octave:> x2 = sin( 2 * pi * 800 * t);
octave:> complexSig = ( x1 + x2 ) / 2;
octave:> sound( complexSig * 0.3162, 8000)
octave:> stem( t(1:20), complexSig(1:20) )
```

Figures 3.16 to 3.18 show successive plots of the signals after adding one more sinusoid to the signal obtained from the previous one. Work through all of these examples successively. How do the sounds compare with previous and subsequent tones?

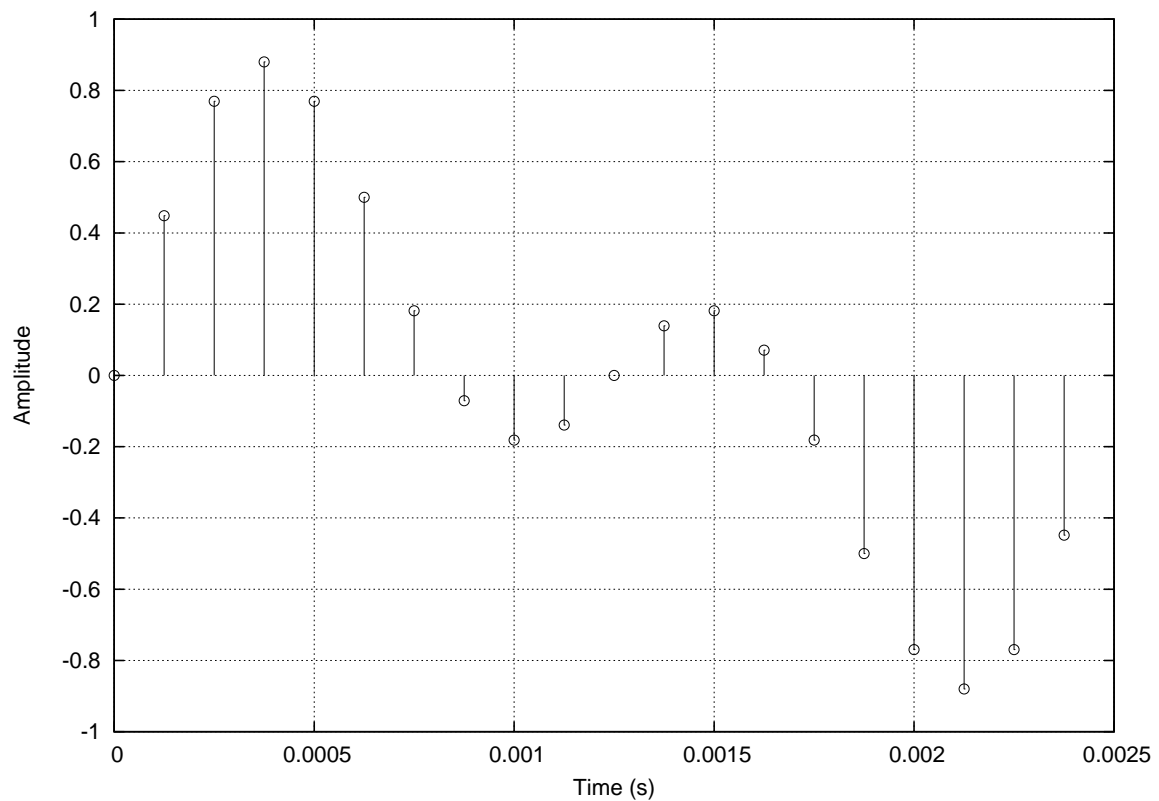


Figure 3.16: One complete cycle of a complex signal formed by summing two sinusoids with frequencies 400Hz and 800Hz. The amplitudes of the sinusoids were halved after summing so that the amplitude remained at 1.

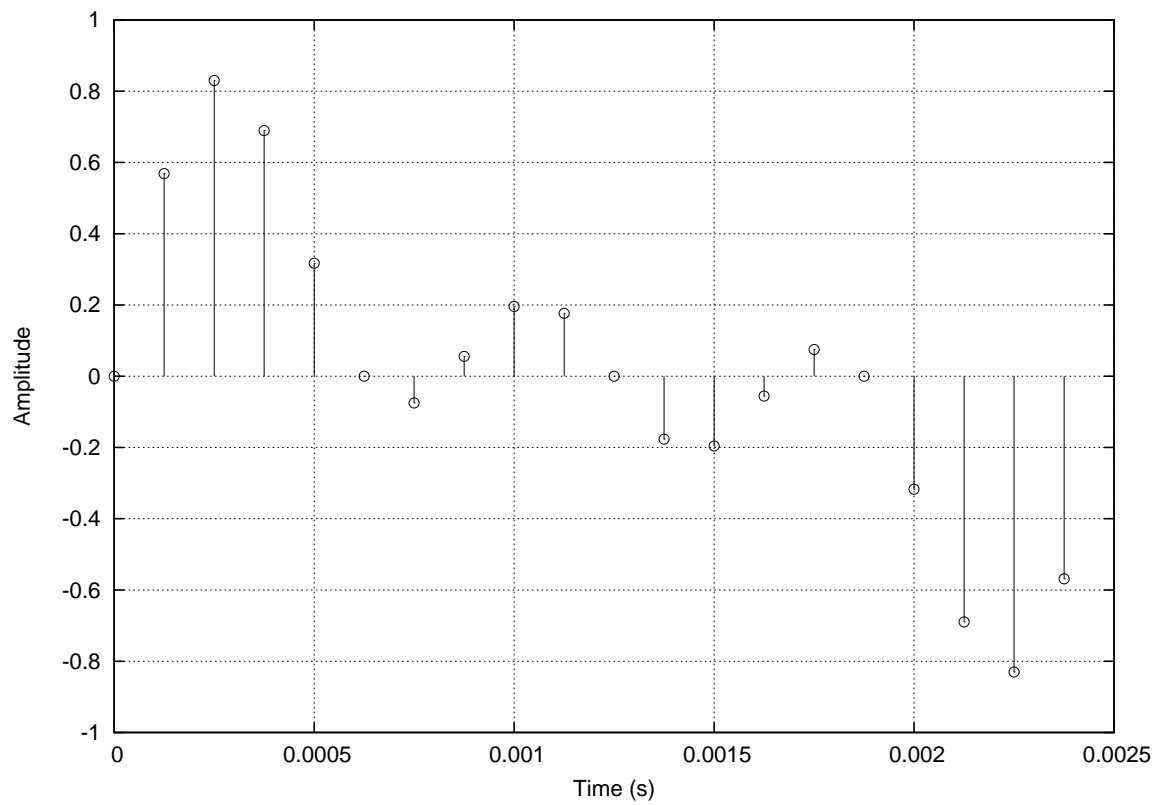


Figure 3.17: One cycle of a complex signal formed by summing three sinusoids with frequencies 400Hz, 800Hz and 1200Hz. The amplitudes were divided by 3 giving a peak amplitude of 1 in the resulting complex signal.

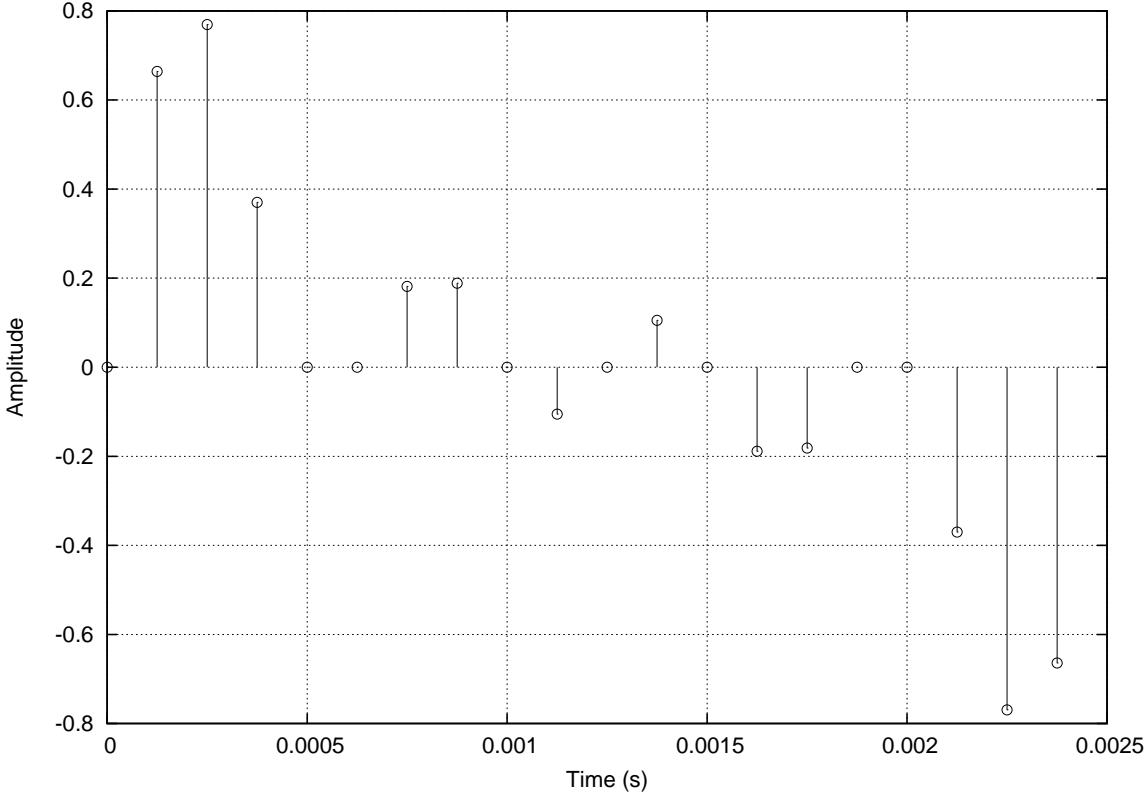


Figure 3.18: One cycle of a complex signal formed by summing four sinusoids with frequencies 400Hz, 800Hz, 1200Hz and 1600Hz. The amplitudes were divided by 4 giving a peak amplitude of 1 in the resulting complex signal.

3.5 Summary and learning outcomes

This chapter provides an introduction to basic signal processing, through the use of the Octave software. It introduced some important signals, namely the unit impulse, the unit step, and the unit delay. It then went on to discuss sampling of audio signals, and described how changes in frequency, amplitude and phase affect sounds.

With a knowledge of the contents of this chapter and its directed reading and activities, you should be able to:

- use Octave to perform basic signal processing tasks, as well as generate plots of signals
- describe what a signal is, and in particular what a one-dimensional signal is
- discuss the significance of the unit impulse, the unit step, and the unit delay in signal processing
- perform various delay operations using Octave
- discuss what the Nyquist frequency is, and how it relates to sound and digital processing
- describe frequency, amplitude and phase as properties of signals.

3.6 Exercises

1. Type the following lines into Octave. This code makes a vector of sinusoid samples x , plots one cycle of the sinusoid and plays the sinusoid as audio at a sample rate of 11.025kHz. This sample rate is the default setting for Octave. You can use the command `setaudio` to change the sample rate in Octave.

```
amp = 0.3162; % sinusoid amplitude (-10dB)
freq = 110; % frequency in Hertz (cps)
dur = 2.0; % duration in seconds
sr = 16000; % the audio sample rate
phase= 0; % initial phase (phase offset)
n = [ 0 : dur * sr ]; % sample index vector
x = sin( 2 * pi * freq / sr * n + phase ); % The sinusoid generator
plot( [ 0 : ceil(sr/freq) - 1 ] / sr , x( 1 : ceil(sr/freq) )) % Plot one
    %cycle grid add grid markings to the plot
sound( x, 16000 ) % play the sound
```

Experiment with the interactive command-line by making sinusoids of 220Hz, 440Hz and 130.81Hz, 261.63Hz and 523.25Hz; these correspond to the musical pitches A3, A4 and C2, C3 and C4. Play the audio for these sinusoids; can you hear the musical scale relationships?

- (a) Plot your sinusoids on the same graph by typing `hold on` in Octave.
 - (b) Choose a melody and construct it out of a set of sinusoids by playing them one after the other. The melody could be a favourite song, or your national anthem, for example.
2. The interactive command line is useful but a bit slow. To speed up making sinusoids we can make a function. Use a text editor to type in the following function and save it as an Octave function in a file called `sinusoid.m`.

```

function [y, n] = sinusoid(A, f, dur, sr, phi)
% y = sinusoid(A, f, d, sr, phi)
%
% A - amplitude
% f - frequency in Hertz
% d - duration in seconds (1.0)
% sr - sample rate in Hertz (44100)
% phi - initial phase n Radians (0)

if nargin<5, phi=0; end % default phase
if nargin<4, sr=16000; end % default sr
if nargin<3, dur=1; end % default dur
if nargin<2, f=440; end % default freq
if nargin<1, A=0.3162;end % default amplitude (-10dB)

n = [0:ceil(dur*sr)-1]./sr; % sample index
y = A * sin ( 2*pi*f*n + phi ); % sinusoid
p = sum(y.^2) / length(n); % average power

```

To use the function you will need to type

`addpath('/Volumes/myDisk/myPath/')`. This function returns a vector of sinusoid samples as its first return argument, and a vector of time instants as its second return argument. For example:

```

octave:>[x,t] = sinusoid(100, 440, 2, 11025, pi/2);
octave:>stem(t(1:20),x(1:20),'*')

```

(a) Using your newly created function, plot sinusoids of the following frequencies:

- 110Hz
- 220Hz
- 330Hz
- 440Hz.

3. So far we have made simple tones. Helmholtz studied the composition of complex tones. We make complex tones by summing simple tones (sinusoids) of different amplitudes, frequencies and phases.

We can efficiently compute the **partials** for a harmonic series using a for loop in Octave.

```

amp = 0.3162; % amplitude
freq = 220; % frequency
dur = 2; % duration
phase = 0; % phase
sr = 44100; % sample rate
N=10; % Number of partials
x = zeros( 1, ceil(dur*sr)); % initial empty vector
for k=1:N
x = x + (1/N)*sinusoid( amp, k * freq, dur, sr, phase);
end
stem([0:length(x(1:ceil(sr/freq)))]/sr,*x(1:ceil(sr/freq)) % Automatically plot one cycle
sound(x*.01) % scale audio by number of partials

```

If Octave does not graph your function then the graphics may need re-setting. Use the command `close all` to close all open graphics windows and reset the axis scaling.

- (a) Use a `for`-loop, similar to the example above, to construct each of the following waveforms with a sample rate of 44.1kHz:
- ten harmonics with amplitude $(1 - k/10) * amp$
 - ten harmonics with amplitude $k/10$
 - ten harmonics with amplitude $exp(0.5 * -k) * amp$
 - ten harmonics with amplitude $exp(0.5 * k) * amp$
 - the fundamental and even harmonics up to ten harmonics [1 2 4 6 8 10]
 - the fundamental and odd harmonics up to ten harmonics [1 3 5 7 9].
- (b) Listen to each of your waveforms. Describe the sounds that you hear.
4. Using the sinusoid function from above, make two sinusoid vectors called `x` and `y`, each with amplitude 0.3162 and frequency 220 Hertz, but `x` should have a phase offset of 0 and `y` should have a phase offset of π (pi in Octave and PI in MATLAB).
- (a) What is the difference between these waveforms? Is there an easy way to derive `y` from `x` and vice-versa?
- (b) Now make the following waveforms:
- ten harmonics with amplitude $(1 - k/10) * amp$ with alternating phases $0, pi$
 - ten harmonics with amplitude $exp(0.5 * -k) * amp$ with alternating phases $0, pi$
 - the fundamental and odd harmonics up to ten harmonics [1 3 5 7 9] with alternating phases $0, pi$.
- (c) Plot these waveforms against the corresponding waveforms from above. What is the difference?
- (d) Now listen to each of these waveforms against the corresponding waveforms from above. What is the difference between the two?
5. Make a cosine wave using a sine wave with a phase offset of $\frac{\pi}{2}$ (pi/2). Play both of the signals using the `sound()` function. Is there an audible difference between the sine wave and cosine wave? Explain your answer.

