Introduction to *Octave* and MATLAB for Math, Science, and Engineering Students

MATTHEW F. CAMPBELL, PH.D., P.E.*

Sigay Kauyagan, Inc.
matthew.f.campbell@gmail.com
http://www.mcampbell.info/

December 4, 2018

Abstract

This tutorial teaches the basics of *Octave* software, which is an open-source computer program for performing scientific and engineering calculations whose coding language is very similar to that of MATLAB software. Only syntax that is compatible for both *Octave* and MATLAB has been included in this document, making it a good primer for both packages. Topics discussed in this manual include the *Octave* Graphical User Interface (GUI), variables, arrays, matrices, plots, scripts (including loops, logical structures, and reading/writing files), and functions.

Contents

*Mindanao State University, General Santos City, South Cotabato, 9500, Philippines.

List of Figures

List of Tables

1 Basic calculator

1.1 Introduction

This is document is a short introduction to *Octave* [1], a programming environment similar in form and function to MATLAB [2, 3] and also similarly capable to Igor Pro software [4] and Wolfram Alpha [5]. *Octave* is free, open-source software designed for Windows, Mac OS, and Linux, and can be obtained from `https://www.gnu.org/software/Octave/`.

1.2 Using the workspace

Open the *Octave* Graphical User Interface (GUI). Figure 1 shows a screenshot of the basic layout with the important parts of the window labeled.

1. Command window: Used to enter commands and read computer output.
2. Editor: Used to write scripts and functions.
3. File browser: Shows files in current directory on computer.
4. Workspace: Shows active variables and matrices.

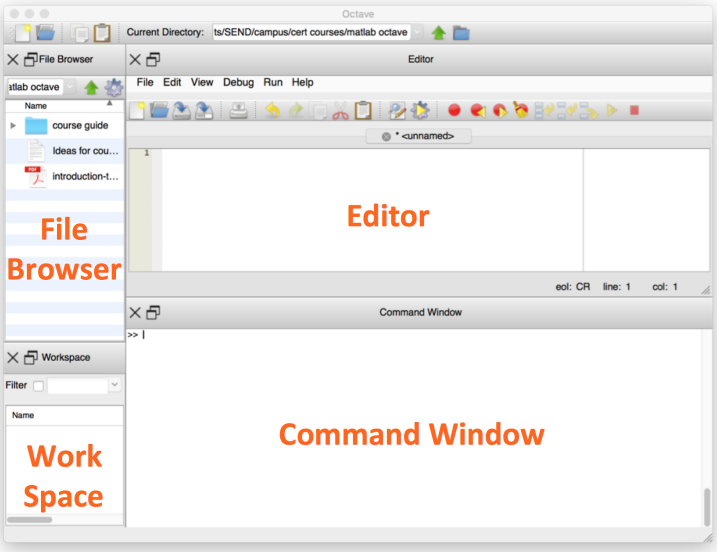

Figure 1: Parts of the *Octave* GUI window.

We can perform basic calculations like this. Press Enter or Return on the keyboard to run the command.

Command Window

```
>> 1+2*3
ans =  7
```

Notice that *Octave* assigns the output to a variable called `ans`.

To go back and re-run a previous command with modifications, we can use the up/down arrow keys (↑/↓) on the keyboard. Then we can use the left/right arrow keys (←/→) to move the cursor where we want to edit.

1.3 Variables

In *Octave*, a *variable* is an object used to store a piece of data. This data is usually a number or a string of characters (often called a *character array*). In the previous example, the output was assigned to the variable `ans`. We can also choose to assign the output to a specific variable such as `x`, and then refer to that variable in calculations too. Finally, if we want to see the value of a variable, we can type its name and press Enter or Return on the keyboard.

Command Window

```
>> x=5*6+7
```

```
x =  37
>> x/35+2
ans =  3.0571
>> x
x =  37
>> ans
ans =  3.0571
```

Octave includes several predefined constant variables, as summarized in Table 1. Try using these variables in Equations 1-5.

Table 1: Predefined constant variables in *Octave* (adapted from Houcque [6]).

Command	Name	Symbol
`pi`	Pi	π
`e`	Exponential	e
`i` or `j`	Imaginary number	$i = j = \sqrt{-1}$
`Inf`	Infinity	∞
`NaN`	Not a Number	

$$r = 5 \tag{1}$$
$$A = \pi r^2 \tag{2}$$
$$V = \frac{4}{3}\pi r^3 \tag{3}$$
$$t = \infty + 10 \tag{4}$$
$$u = (-\infty) + \infty \tag{5}$$

The input to *Octave* would look like the text below. Notice that we need to put a times (`*`) symbol to denote multiplication.

Command Window

```
>> r=5
r =  5
>> A=pi*r^2
A =  78.540
>> V=(4/3)*pi*r^3
V =  523.60
>> t=inf+10
t = Inf
>> u=(-inf)+inf
u = NaN
```

Variable names must start with a letter character (*i.e.*, `a` through `z`), but they can contain numbers (`0` through `9`) or underscores (_). Variable names are case-sensitive, so `A` is not equivalent to `a`. Avoid naming variables with function names, such as `size`, `char`, `all`, *etc.*.

Command Window

```
>> m = 1
m =  1
>> n123 = 4
n123 =  4
>> n123_abc = 5
n123_abc =  5
```

We can find if a variable exists using the `exist` command. If the answer is `0`, the variable does not exist; if the answer is `1`, the variable does exist.

Command Window

```
>> x = 25
x =  25
>> exist x
ans =  1
>> exist y
ans = 0
```

We can list the variables in our workspace by typing `who` or `whos`.

Command Window

```
>> var1 = 56/2.6
var1 =  21.538
>> var2 = 81+3/18
var2 =  81.167
>> var3 = 91/4
var3 =  22.750
>> who
Variables in the current scope:

var1  var2  var3

>> whos
Variables in the current scope:

   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        var1        1x1                          8  double
        var2        1x1                          8  double
        var3        1x1                          8  double

Total is 3 elements using 24 bytes
```

1.4 Order of operations

Octave follows standard order-of-operations rules, according to Table 2. Try to evaluate the following expressions:

$$x = \left(500 + \frac{45}{72}\right)^{\frac{2}{3}} \tag{6}$$

$$y = \frac{\left(30 \times 5 + \frac{100}{78}\right)^{4}}{\left(34 \times 2 - 36\right)^{5}} \tag{7}$$

$$z = \frac{xy}{22} \tag{8}$$

Command Window

```
>> x=(500+45/72)^(2/3)
x =  63.049
>> y=((30*5+100/78)^4)/((34*2-36)^5)
y =  15.610
>> z=(x*y)/22
z =  44.735
```

Table 2: Order of operations in *Octave* (adapted from Houcque [6]).

Precedence	Operation
1	Parentheses (inside-to-outside)
2	Exponentials (left-to-right)
3	Multiplication/division (left-to-right)
4	Addition/subtraction (left-to-right)

Now use parentheses to change the order of operations:

Command Window

```
>> a=2+(3*4)
a =  14
>> b=(2+3)*4
b =  20
```

1.5 Built-in functions

Octave includes many built-in mathematical functions for advanced scientific computing. A list of several of these is provided in Table 3.

Table 3: Several built-in scientific functions in *Octave* (adapted from Houcque [6]).

Command	Name	Function	Note
`sqrt(x)`	Square root	$\sqrt{x}$	
`cbrt(x)`	Cube root	$\sqrt[3]{x}$	
`sin(x)`	Sine	$\sin(x)$	x in radians
`cos(x)`	Cosine	$\cos(x)$	x in radians
`tan(x)`	Tangent	$\tan(x)$	x in radians
`asin(x)`	Arc sine	$\arcsin(x)$	Output in radians
`acos(x)`	Arc cosine	$\arccos(x)$	Output in radians
`atan(x)`	Arc tangent	$\arctan(x)$	Output in radians
`sind(x)`	Sine	$\sin(x)$	x in degrees
`asind(x)`	Arc sine	$\arcsin(x)$	Output in degrees
`exp(x)`	Exponential	e^x	
`log(x)`	Natural logarithm	$\ln(x)$	
`log10(x)`	Common logarithm	$\log_{10}(x)$	
`abs(x)`	Absolute value	$\lvert x \rvert$	
`inv(x)`	Inverse	$\frac{1}{x}$	
`ceil(x)`	Round up to nearest integer		
`floor(x)`	Round down to nearest integer		
`round(x)`	Round to nearest integer		
`rem(x)`	Remainder after division		

We can use these functions like this:

Command Window

```
>> sqrt(35)
ans =  5.9161
>> x = 10
x =  10
>> y = sqrt(x)
y =  3.1623
>> z = -3
z = -3
>> abs(z)
ans =  3
>> log10(x)
ans =  1
```

Try to evaluate these expressions or numerically test these equalities. The *Octave* code is given below for reference as well.

$$x = e^{i\pi} + 1 \tag{9}$$

$$y = \cos(\pi) + i\sin(\pi) \tag{10}$$

$$z = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \frac{1}{36} + \frac{1}{49} + \frac{1}{64} + \frac{1}{81} + \frac{1}{100} + \frac{1}{121} + \frac{1}{144} + \ldots = \frac{\pi^2}{6} \tag{11}$$

$$a = 10^{\log_{10}(8)} \tag{12}$$

$$b = \ln(e) \tag{13}$$

$$c = \frac{\log_{10}(\pi)}{\log_{10}(e)} = \ln(\pi) \tag{14}$$

Command Window

```
>> x=exp(i*pi)+1
x =  0.0000e+00 + 1.2246e-16i
>> y=cos(pi)+i*sin(pi)
```

```
y = -1.0000e+00 + 1.2246e-16i
>> z = 1 + 1/4 + 1/9 + 1/16 + 1/25 + 1/36 + 1/49 + 1/64 + 1/81 + 1/100 + 1/121 + 1/144
z =  1.5650
>> pi^2/6
ans =  1.6449
>> a=10^(log10(8))
a =  8.0000
>> b=log(e)
b =  1
>> c=log10(pi)/log10(e)
c =  1.1447
>> log(pi)
ans =  1.1447
```

1.6 Workspace operations

To change the appearance of decimal numbers, use the `format` command. Notice that if we want to see the value of a variable, we just need to type its name and press Enter or Return.

Command Window

```
>> format short
>> x=4536/3.2
x =  1417.5
>> format long
>> x
x =  1417.50000000000
>> format short eng
>> x
x =     1.4175e+003
>> format long eng
>> x
x =     1.41750000000000e+003
>> format short
>> x
x =  1417.5
```

The notation `x = 1.4175e+003` is called *scientific notation*; it means that $x = 1.4175 \times 10^3$.

By the way, sometimes ***Octave*** gives an answer that is very very small. In that case, it may actually be equal to zero instead. For instance, it is mathematically true that $\sin(\pi) = 0$. However, ***Octave*** says that $\sin(\pi) = 1.2246 \times 10^{-16}$, which is very very small. This is due to a round-off error in the program.

Command Window

```
>> sin(pi)
ans =    1.2246e-16
```

We can change the spacing of the lines of input/output in the command window using `format compact` (surpresses extra line blanks) or `format loose` (includes extra line breaks). Also, we can use the command `more off` to prevent the buffering of text output in the command window.

We can clear your command window (delete the text in it) easily by typing `clc` and pressing return. It's also possible for us to clear variables (remove them from our workspace), either one-by-one or all-at-once.

Command Window

```
>> x = 34
x =  34
>> y = 22
y =  22
>> z = 67
z =  67
>> clear x
>> x
error: 'x' undefined near line 1 column 1
>> clear all
>> y
error: 'y' undefined near line 1 column 1
>> z
error: 'z' undefined near line 1 column 1
```

We can use the `help` command to find information about functions.

Command Window

```
>> help clc
'clc' is a built-in function from the file libinterp/corefcn/sysdep.cc

 -- Built-in Function: clc ()
 -- Built-in Function: home ()
     Clear the terminal screen and move the cursor to the upper left
     corner.

Additional help for built-in functions and operators is
available in the online version of the manual.  Use the command
'doc <topic>' to search the manual index.

Help and information about \Octave is also available on the WWW
at http://www.\Octave.org and via the help@\Octave.org
mailing list.
```

It's possible to enter multiple commands per line, using a semicolon to separate the commands. Also, we can suppress the output of a command using a semicolon at the end of the line.

Command Window

```
>> x = 45+6/4
x =  46.500
>> x = 45+6/4; y = 22+7/3; z = 99/32;
>> x
x =  46.500
>> y
y =  24.333
>> z
z =  3.0938
>> a = x+y+z;
>> a
a =  73.927
```

If we want to continue a command on a second line (for instance, to keep our code organized), we can use three dots (`...`) like this:

Command Window

```
>> x = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + ...
10 + 11 + 12 + 13 + 14 + 15 + 16 + 17 + 18 + 19 + ...
20
x =  210
```

Sometimes it is nice to know the largest and smallest numbers that ***Octave*** can handle. To find them, use the commands `realmax` and `realmin`.

Command Window

```
>> realmax
ans =   1.7977e+308
>> realmin
ans =   2.2251e-308
```

We might also want to know the smallest number that can be distinguished from zero (sometimes called *machine precision*). To find this we can use `eps`.

Command Window

```
>> eps
ans =    2.2204e-16
```

1.7 Example: Calculating interest

Octave can be used to calculate interest. We will look at simple and compound interest here. Simple interest just accrues upon the principal amount P, whereas compound generates interest upon the accrued interest. The equation to find the total accrued amount (principal plus interest) A using simple interest is

$$A = P\,(1 + rt) \tag{15}$$

where r is the rate of interest per year and t is the time in years. What if we wanted to find the principal we would need to invest in order to have $A = 1000$ dollars in $t = 5$ years at $r = 0.01$ (1% interest). Solving Equation 15 for P gives

$$P = \frac{A}{1 + rt} \tag{16}$$

Entering the above in *Octave* gives the result of $P = 952.38$ dollars.

Command Window

```
>> A=1000
A =  1000
>> t=5
t =  5
>> r=0.01
r =  0.010000
>> P=A/(1+r*t)
P =  952.38
```

What if the interest was compounded? The equation to calculate the accrued amount A under compound interest would be

$$A = P\,(1 + r)^t \tag{17}$$

When the interest is compounded, the accrued interest can in turn generate more interest; this means that a lower principal amount P is required to generate the same accrued amount A in the same time t with the same interest rate r. Solving Equation 17 for P gives

$$P = \frac{A}{(1 + r)^t} \tag{18}$$

Entering this into *Octave* gives the new result of $P = 951.47$ dollars, which indeed is slightly less than the result under simple interest.

Command Window

```
>> P=A/((1+r)^t)
P =  951.47
```

1.8 Exercises

Try the following problems (Equations 19-26) in *Octave* and record the input/output. Just for fun, try them on paper, too!

$$s = \frac{45.34}{8 + 2 \times 50} \tag{19}$$

$$t = \frac{10^4 - 18}{100 + 22} \tag{20}$$

$$u = 121 - 15 \times 20 + \left(22 - 11 \times \frac{30}{450 - 81}\right)^{1.45} \tag{21}$$

$$v = 50(30 + 40(100 + 23(40 - 11))) \tag{22}$$

$$w = 6 \div 2(1+2) \tag{23}$$

$$x = 9 - 3 \div \frac{1}{3} + 1 \tag{24}$$

$$y = 10 \times 4 - 2 \times \left(4^2 \div 4\right) \div 2 \div \frac{1}{2} + 9 \tag{25}$$

$$z = -10 \div \left(20 \div 2^2 \times 5 \div 5\right) \times 8 - 2 \tag{26}$$

2 Arrays

2.1 Manual entry

An *array* (sometimes called a *vector*) is a list of numbers (sometimes called *values*). Arrays make it easy to do mathematical operations on large quantities of data. The naming conventions of arrays are the same as those of variables (see Section 1.3). We can create an array in several ways. Manual entry looks like this:

Command Window

```
>> format compact
>> xVec1 = [1 3 5 7 9]
xVec1 =
   1   3   5   7   9
>> xVec2 = [1, 2, 3, 4, 5]
xVec2 =
   1   2   3   4   5
>> xVec3 = [2; 4; 6; 8; 10; 12]
xVec3 =
    2
    4
    6
    8
   10
   12
```

As we can see, to separate numbers and create a *row array*, we should use spaces or commas (`,`). To separate numbers and make a *column array*, we should use semicolons (`;`).

2.2 Automatic generation

What if we wanted to have ***Octave*** generate arrays for us automatically? There are several ways, but we will just look at two for now. They are the colon operator (`:`) and a command called `linspace`. Let's try asking ***Octave*** to count by 2's from 1 until 9.

Command Window

```
>> a = 1:2:9
a =
   1   3   5   7   9
```

How about something harder, like counting by intervals of 5.6 from 3.4 until 20.2?

Command Window

```
>> b = 3.4:5.6:20.2
b =
    3.4000    9.0000   14.6000   20.2000
```

What if this time we wanted to have 13 evenly-spaced points between 3 and 15?

Command Window

```
>> c = linspace(3,15,13)
c =
    3    4    5    6    7    8    9   10   11   12   13   14   15
```

Now let's ask ***Octave*** to find 7 evenly-spaced points between 46.3 and 50.5.

Command Window

```
>> d = linspace(46.3,50.5,7)
d =
   46.300   47.000   47.700   48.400   49.100   49.800   50.500
```

So, the colon operator `vectorName=a:d:b` tells ***Octave*** to create an array starting at `a` and spacing numbers by steps of delta `d` until the last number reaches (but does not pass) `b` (this notation can be shortened to `vectorName=a:b` if the spacing delta is equal to one). The notation `vectorName=linspace(a,b,c)` tells ***Octave*** to create an array starting on `a` and ending exactly on `b` that has `c` number of evenly-spaced points. Note that if we want to have a precise step (delta) value, we need to use the colon operator, but if we want to end on an exact number, we need to use the `linspace` command. Here are some generic examples. See also Table 4 for a summary of these commands.

Command Window

```
>> startVal = 1
startVal =  1
>> endVal = 13
endVal =  13
>> delta = 3
delta =  3
>> numPoints = 5
numPoints =  5
>> xVec4 = startVal:delta:endVal
xVec4 =
    1    4    7   10   13
>> xVec5 = linspace(startVal,endVal,numPoints)
xVec5 =
    1    4    7   10   13
```

Table 4: Commands to generate arrays automatically in ***Octave***.

Command	Description
`a:d:b`	Creates an array starting at `a` and spacing numbers in steps of `d` until reaching (but not passing) `b`
`linspace(a,b,c)`	Creates an array starting at `a` and ending on `b` that has `c` number of evenly-spaced points
`logspace(a,b,c)`	Creates an array starting at 10^{a} and ending on 10^{b} that has `c` number of logarithmically-spaced points

Try to generate the following arrays automatically. The ***Octave*** code is given below for reference as well.

$$q = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \tag{27}$$
$$r = [3, 6, 9, 12] \tag{28}$$
$$s = [1.5, 3.0, 4.5, 6.0] \tag{29}$$
$$t = [10, 9, 8, 7, 6, 5] \tag{30}$$
$$u = [-10, -9, -8, -7, -6, -5] \tag{31}$$
$$v = [-1.5 - 1.0, -0.5, 0.0, 0.5, 1.0, 1.5] \tag{32}$$
$$w = [4, 1, 0, 1, 4] \tag{33}$$

Command Window

```
>> q = 1:1:10
q =
    1    2    3    4    5    6    7    8    9   10
>> r = 3:3:12
r =
    3    6    9   12
>> s = linspace(1.5,6,4)
```

```
s =
   1.5000   3.0000   4.5000   6.0000
>> t = 10:-1:5
t =
   10    9    8    7    6    5
>> u = linspace(-10,-5,6)
u =
  -10   -9   -8   -7   -6   -5
>> v = -1.5:0.5:1.5
v =
   -1.5000   -1.0000   -0.5000    0.0000    0.5000    1.0000    1.5000
>> w = [-2:1:2].^2
w =
   4   1   0   1   4
```

2.3 Index notation

We can refer to numbers in an array using *index numbers* inside parentheses. The first number in the array has index `1`, and the last number in the array can be referred to as `end`.

Command Window

```
>> yVec = 1:2:15
yVec =
    1    3    5    7    9   11   13   15
>> yVec(1)
ans =  1
>> yVec(3)
ans =  5
>> yVec(4)
ans =  7
>> yVec(end)
ans =  15
>> yVec(end-1)
ans =  13
```

We can also grab several numbers at once (called a *block*) using *colon notation* (`:`).

Command Window

```
>> zVec = 1:3:15
zVec =
    1    4    7   10   13
>> zVec(1:3)
ans =
   1   4   7
>> zVec(2:4)
ans =
    4    7   10
```

Lastly, we can delete values from arrays or insert values like this:

Command Window

```
>> x=1:10
x =
    1    2    3    4    5    6    7    8    9   10
>> x(1:3)=[]
x =
    4    5    6    7    8    9   10
>> y=11:20
y =
   11   12   13   14   15   16   17   18   19   20
>> y = [y(1:5), 1, 2, 3, y(6:10)]
y =
   11   12   13   14   15    1    2    3   16   17   18   19   20
```

2.4 Formatting and sizing

Arrays can be in *row form* or *column form*. To change an array into column form, use the prime symbol (`'`). This is called *transposing* the array.

Command Window

```
>> testVec = linspace(3,20,5)
testVec =
    3.0000    7.2500   11.5000   15.7500   20.0000
>> testVec2 = testVec'
testVec2 =
    3.0000
    7.2500
   11.5000
   15.7500
   20.0000
```

It can be useful to find the dimensions of an array. This can be done using the `length` and `size` commands. The `size` command is better-suited for matrices, which we will cover later. Its output consists of two numbers; the first is the number of rows and the second is the number of columns.

Command Window

```
x =
    1    2    3    4    5    6    7    8    9   10   11   12   13
>> length(x)
ans =  13
>> size(x)
ans =
    1   13
>> y = [1:5]'
y =
   1
   2
   3
   4
   5
>> length(y)
ans =  5
>> size(y)
ans =
   5   1
```

2.5 Operations

We can perform operations on each entry of an array using operators such as `+`, `-`, `./`, `.*`, and `.^` (see Table 5). Notice that, when performing operations involving two vectors, the multiplication (`.*`), division (`./`), and exponentiation symbols (`.^`) have periods or dots (`.`) in front of them. This tells ***Octave*** to apply the operations on each element of the arrays (this is called *elementwise notation*). Note that operations involving two arrays require that the arrays have the same number of elements (*i.e.*, the same *size* or *length*).

Table 5: Array operations in ***Octave*** (adapted from Houcque [6]).

Operation	Command
Addition	+
Subtraction	−
Multiplication	.*
Division	./
Exponentiation	.^

Command Window

```
>> uVals = 30:-4:15
uVals =
```

```
   30   26   22   18
>> vVals = 2:3:11
vVals =
    2    5    8   11
>> uVals+10
ans =
   40   36   32   28
>> uVals-2
ans =
   28   24   20   16
>> uVals*5
ans =
   150   130   110    90
>> uVals/3
ans =
   10.0000    8.6667    7.3333    6.0000
>> uVals+vVals
ans =
   32   31   30   29
>> uVals-vVals
ans =
   28   21   14    7
>> uVals.*vVals
ans =
    60   130   176   198
>> uVals./vVals
ans =
   15.0000    5.2000    2.7500    1.6364
>> uVals.^2
ans =
   900   676   484   324
```

Most built-in functions work on an *entry-by-entry* (sometimes said *element-by-element*) basis. This means that the function is applied to every value in the array. A simple example is the `sin()` function.

Command Window

```
>> xVec = linspace(0,2*pi,7)
xVec =
   0.00000   1.04720   2.09440   3.14159   4.18879   5.23599   6.28319
>> yVec = sin(xVec)
yVec =
   0.00000   0.86603   0.86603   0.00000  -0.86603  -0.86603  -0.00000
```

However, some functions work on the array as a whole. Here are some examples:

Command Window

```
xVec =
    1    2    3    4    5    6    7    8    9   10
>> xAverage = mean(xVec)
xAverage =  5.5000
>> xMedian = median(xVec)
xMedian =  5.5000
>> xTotal = sum(xVec)
xTotal =  55
>> xMaximum = max(xVec)
xMaximum =  10
>> xMinimum = min(xVec)
xMinimum =  1
```

2.6 Searching

We can select elements in an array based on their value. The resulting array shows a `1` where the condition is true, and `0` where the condition is false.

Command Window

```
>> xVec=1:1:10
xVec =
```

```
    1    2    3    4    5    6    7    8    9   10
>> yVec=10:-1:1
yVec =
   10    9    8    7    6    5    4    3    2    1
>> compareVec = [xVec>yVec]
compareVec =
   0   0   0   0   0   1   1   1   1   1
>> yVec<8
ans =
   0   0   0   1   1   1   1   1   1   1
>> xVec<8
ans =
   1   1   1   1   1   1   1   0   0   0
```

Lastly, we can ask *Octave* to look in an array to return the indices of certain elements using the `find` command.

Command Window

```
>> xVec=11:20
xVec =
   11   12   13   14   15   16   17   18   19   20
>> largeValueIndices=find(xVec>17)
largeValueIndices =
    8    9   10
>> smallValueIndices=find(xVec<14)
smallValueIndices =
   1   2   3
>> largeValues=xVec(find(xVec>17))
largeValues =
   18   19   20
>> smallValues=xVec(find(xVec<14))
smallValues =
   11   12   13
>> indexOf15=find(xVec==15)
indexOf15 =  5
```

2.7 Example: Basel problem

Remember Equation 11 above, where we added a bunch of fractions together? It's called the *Basel Problem*, and the actual equation with all the terms is

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \ldots = \frac{\pi^2}{6} \tag{34}$$

Previously we attempted this by just adding a bunch of fractions together. With our knowledge of arrays, though, we can now add a lot more terms together much more quickly (see below). Notice the use of the semicolon (`;`) to suppress output, as well as the element-by-element operators (`./` and `.^`).

Command Window

```
>> xArray = 1:1000000;
>> xArrayMod = 1./(xArray.^2);
>> xArrayModTotal = sum(xArrayMod)
xArrayModTotal =  1.64493306684877
>> actualSum = (pi^2)/6
actualSum =  1.64493406684823
```

2.8 Example: Polynomial roots

Octave can be used to find the roots of polynomial expressions. Let's say we want to find the roots of this expression:

$$0 = 2x^4 - 50x^3 + 40x^2 + 3x - 3 \tag{35}$$

We can make an array `c` to represent the coefficients of the polynomial in Equation 35, and then use the `roots(c)` command to find the roots of this polynomial.

Command Window

```
>> c = [2 -50 40 3 -3]
c =
    2  -50   40    3   -3
>> rootVals = roots(c)
rootVals =
   24.17007
    0.80859
   -0.26657
    0.28792
```

Therefore, we can say that $x = 24.17007$, $x = 0.80859$, $x = -0.26657$, and $x = 0.28792$ are all solutions to this polynomial. We can check this using the `polyval(c,a)` command, which evaluates a polynomial with coefficients in array `c` at the points in array `a`. The results are effectively equal to zero (*i.e.*, very small), indicating that ***Octave*** correctly identified the roots of Equation 35.

Command Window

```
>> polyval(c,rootVals)
ans =
   1.5790e-10
  -1.4655e-14
   1.1102e-14
   0.0000e+00
```

2.9 Example: Student enrollment

Here's another example of how we can use arrays in ***Octave***. We might know the total number of students enrolled in a certain course each year for ten years (see for example Table 6). One array, titled `year`, would have a list of years, and a second array, titled `enrollment`, would have a list of enrollment numbers in those years. We might want to find the total number of students served in those ten years, the year with the most students and the number of students in that year, or the average number of students. Also, assuming each student is required to pay dues to the department each year, and that the amount of required dues per student changes year-to-year and is recorded in a vector titled `duesPerStudent`, we might want to know the amount of money collected each year.

Table 6: Example of (fake) data that can be used in array format.

`year`	`enrollment` [students]	`duesPerStudent` [dollars per student]
1995	245	45
1996	230	45
1997	304	50
1998	286	50
1999	256	65
2000	291	60
2001	310	60
2002	283	70
2003	315	70
2004	270	75

Command Window

```
>> year = [1995:1:2004]
year =
   1995   1996   1997   1998   1999   2000   2001   2002   2003   2004
>> enrollment = [245 230 304 286 256 291 310 283 315 270]
enrollment =
   245   230   304   286   256   291   310   283   315   270
>> duesPerStudent = [45 45 50 50 65 60 60 70 70 75]
duesPerStudent =
   45   45   50   50   65   60   60   70   70   75
>> totalNumberOfStudents = sum(enrollment)
```

```
totalNumberOfStudents =  2790
>> [maxEnrollment,yearIndex] = max(enrollment)
maxEnrollment =  315
yearIndex =  9
>> yearWithMaximumStudents = year(yearIndex)
yearWithMaximumStudents =  2003
>> averageEnrollment = mean(enrollment)
averageEnrollment =  279
>> collectionEachYear = enrollment.*duesPerStudent
collectionEachYear =
   11025   10350   15200   14300   16640   17460   18600   19810   22050   20250
```

2.10 Exercises

Try to create the following vectors (Equations 36-43) automatically through either colon notation (`:`), `linspace`, or `logspace` (not using manual entry). To learn about the latter command, try `help logspace`. You may also have to do operations on the entire vector, such as squaring it (`^2`) or making it an exponent (`exp`).

$$aVec = [1, 3, 5, 7, 9] \tag{36}$$

$$bVec = [0.1, 1, 10, 100] \tag{37}$$

$$cVec = [2.0, 2.4, 2.8, 3.2, 3.6, 4.0] \tag{38}$$

$$dVec = [1, 4, 9, 16, 25, 36] \tag{39}$$

$$eVec = [1.0000, 2.7183, 7.3891] \tag{40}$$

$$fVec = [10, 8, 6, 4, 2, 0] \tag{41}$$

$$gVec = [-3, -2, -1, 0, 1, 2, 3] \tag{42}$$

$$hVec = [9, 4, 1, 0, 1, 4, 9] \tag{43}$$

3 Matrices

3.1 Manual entry

A matrix is a rectangular array of numbers that is arranged in rows and columns. In ***Octave***, matrices can be created similarly to arrays. Note that to continue a line without finishing a command we need to use three dots (`...`). Here are some examples of manual creation:

Command Window

```
>> format compact
>> x=[1 2 3; ...
4 5 6; ...
7 8 9]
x =
```

```
   1   2   3
   4   5   6
   7   8   9
>> y = [11 12 13; 14 15 16; 17 18 19]
y =
   11   12   13
   14   15   16
   17   18   19
>> z = [10 20 30 40; 50 60 70 80; 90 100 110 120]
z =
    10    20    30    40
    50    60    70    80
    90   100   110   120
```

3.2 Automatic generation

Octave can generate matrices automatically, too. The `zeros(a,b)`, `ones(a,b)`, and `rand(a,b)` commands create matrices filled with 0, 1, and random numbers between 0 and 1, respectively, that have `a` number of rows and `b` number of columns. The `reshape(c,a,b)` command changes the elements of array `c` and places them columnwise in a matrix with `a` rows and `b` columns. The `diag(d)` command extracts the diagonal of matrix `d`. Lastly, the `eye(a)` command creates an `a×a` matrix with ones on its diagonal and zeros elsewhere; it is called the *identity* matrix. These commands are summarized in Table 7.

Table 7: Commands to generate matrices automatically in ***Octave*** (adapted from Houcque [6]).

Command	Description
`zeros(a,b)`	Creates `a×b` (row-by-column) matrix filled with 0's
`ones(a,b)`	Creates `a×b` matrix filled with 1's
`rand(a,b)`	Creates `a×b` matrix filled with random numbers between 0 and 1
`eye(a)`	Creates `a×a` matrix of zeros except with 1 on the diagonal
`magic(a)`	Creates `a×a` magic square matrix
`reshape(c,a,b)`	Places elements of array `c` into `a×b` matrix

Command Window

```
>> xMat = zeros(3,5)
xMat =
   0   0   0   0   0
   0   0   0   0   0
   0   0   0   0   0
>> yMat = ones(2,4)
yMat =
   1   1   1   1
   1   1   1   1
>> rMat = rand(2,5)
rMat =
   0.88394   0.23837   0.90222   0.19985   0.61667
   0.67699   0.11758   0.29462   0.99987   0.27641
>> zVec = [1:1:12]
zVec =
    1    2    3    4    5    6    7    8    9   10   11   12
>> zMat = reshape([1:1:12],3,4)
zMat =
    1    4    7   10
    2    5    8   11
    3    6    9   12
>> mMat = rand(3,3)
mMat =
   0.416344   0.712268   0.126029
   0.570250   0.849405   0.161314
   0.097609   0.629807   0.297589
>> mDiagonal = diag(mMat)
mDiagonal =
   0.41634
   0.84940
```

```
   0.29759
>> nMat = eye(5)
nMat =
Diagonal Matrix
   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0
   0   0   0   0   1
```

Try to create the following matrices automatically. The ***Octave*** code is given below for reference.

$$q = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \tag{44}$$

$$r = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix} \tag{45}$$

$$s = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{46}$$

$$t = \begin{bmatrix} 10 & 6 & 2 \\ 8 & 4 & 0 \end{bmatrix} \tag{47}$$

Command Window

```
>> q = reshape([1:1:6],3,2)
q =
   1   4
   2   5
   3   6
>> r = ones(2,3)*3
r =
   3   3   3
   3   3   3
>> s = zeros(4,3)
s =
   0   0   0
   0   0   0
   0   0   0
   0   0   0
>> t = reshape([10:-2:0],2,3)
t =
   10    6    2
    8    4    0
```

3.3 Index notation

Just like with arrays, indices can be used to refer to elements in a matrix. We can grab individual elements or grab chunks using colon notation.

Command Window

```
>> a = reshape([1:1:20],5,4)
a =
    1    6   11   16
    2    7   12   17
    3    8   13   18
    4    9   14   19
    5   10   15   20
>> a(1,1)
ans =  1
>> a(2,3)
ans =  12
>> a(4,4)
```

```
ans =  19
>> a(5,4)
ans =  20
>> b=a(2:5,3:4)
b =
   12   17
   13   18
   14   19
   15   20
```

We can delete or insert values as follows.

Command Window

```
>> x = reshape([1:1:16],4,4)
x =
    1    5    9   13
    2    6   10   14
    3    7   11   15
    4    8   12   16
>> x(2,3) = 0
x =
    1    5    9   13
    2    6    0   14
    3    7   11   15
    4    8   12   16
>> x(2,:) = []
x =
    1    5    9   13
    3    7   11   15
    4    8   12   16
>> x(:,4) = []
x =
    1    5    9
    3    7   11
    4    8   12
>> x = [x(1,:);[0:-1:-2];x(2:3,:)]
x =
    1    5    9
    0   -1   -2
    3    7   11
    4    8   12
```

3.4 Formatting and sizing

The transpose operator (prime symbol (`'`)) works with matrices, too. In this example, the matrix `x` is transposed to make a new matrix `xTrans` where the first row becomes the first column, and the second row becomes the second column.

Command Window

```
>> x=reshape(1:1:8,2,4)
x =
   1   3   5   7
   2   4   6   8
>> xTrans=x'
xTrans =
   1   2
   3   4
   5   6
   7   8
```

Similarly, the `size` operator reveals the number of rows and columns that a matrix has.

Command Window

```
>> x=ones(3,6)
x =
   1   1   1   1   1   1
   1   1   1   1   1   1
   1   1   1   1   1   1
```

```
>> size(x)
ans =
   3   6
```

If we want to rotate a matrix 90 degrees counter-clockwise, we can use the `rot90` command. This command can be used several times to achieve more rotations.

Command Window

```
>> x=reshape(1:1:16,4,4)
x =
    1    5    9   13
    2    6   10   14
    3    7   11   15
    4    8   12   16
>> rotatedX = rot90(x)
rotatedX =
   13   14   15   16
    9   10   11   12
    5    6    7    8
    1    2    3    4
>> rotatedX = rot90(rot90(x))
rotatedX =
   16   12    8    4
   15   11    7    3
   14   10    6    2
   13    9    5    1
```

Moreover, we can flip a matrix vertically using `flipud` or horizontally using `fliplr`.

Command Window

```
>> x=reshape(1:1:16,4,4)
x =
    1    5    9   13
    2    6   10   14
    3    7   11   15
    4    8   12   16
>> xFlipVert = flipud(x)
xFlipVert =
    4    8   12   16
    3    7   11   15
    2    6   10   14
    1    5    9   13
>> xFlipHoriz = fliplr(x)
xFlipHoriz =
   13    9    5    1
   14   10    6    2
   15   11    7    3
   16   12    8    4
```

The above commands are summarized in Table 8.

Table 8: Commands to manipulate matrices in ***Octave***.

Command	Description
`'`	Transposes a matrix
`rot90`	Rotates a matrix 90 degrees counter-clockwise
`flipud`	Flips a matrix vertically
`fliplr`	Flips a matrix horizontally

3.5 Operations

We can also perform mathematical operations on all of the elements in a matrix using the same operators that worked for arrays (see Section 2.5). These include `+`, `-`, `./`, `.*`, and `.^`. Keep in mind the period/dot (`.`) for elementwise operating, and also that matrices need to have the same size (number of rows and columns) to be added, multiplied, *etc.*.

Command Window

```
>> x=reshape(1:1:15,3,5)
x =
    1    4    7   10   13
    2    5    8   11   14
    3    6    9   12   15
>> y = reshape(16:1:30,3,5)
y =
   16   19   22   25   28
   17   20   23   26   29
   18   21   24   27   30
>> x+y
ans =
   17   23   29   35   41
   19   25   31   37   43
   21   27   33   39   45
>> x-y
ans =
  -15  -15  -15  -15  -15
  -15  -15  -15  -15  -15
  -15  -15  -15  -15  -15
>> x.*y
ans =
    16    76   154   250   364
    34   100   184   286   406
    54   126   216   324   450
>> x.^y
ans =
   1.0000e+00   2.7488e+11   3.9098e+18   1.0000e+25   1.5503e+31
   1.3107e+05   9.5367e+13   5.9030e+20   1.1918e+27   1.7287e+33
   3.8742e+08   2.1937e+16   7.9766e+22   1.3737e+29   1.9175e+35
```

Most functions work as expected on matrices (*i.e.*, the same as they would on arrays). Others, however, must be given a *dimension* over which to operate. For instance, the `mean(a,b)` command finds the average value of the columns (`b`=1) or rows (`b`=2) of matrix `a`. This is because in ***Octave*** dimension 1 represents columns and dimension 2 represents rows. Don't be confused though; in the notation for automatic matrix creation, the number of rows is entered *before* the number of columns! We'll talk about dimensions more in Section 3.6.

Command Window

```
>> x=reshape(1:1:20,4,5)-10
x =
   -9   -5   -1    3    7
   -8   -4    0    4    8
   -7   -3    1    5    9
   -6   -2    2    6   10
>> columnAverages = mean(x,1)
columnAverages =
  -7.50000  -3.50000   0.50000   4.50000   8.50000
>> rowAverages = mean(x,2)
rowAverages =
  -1
   0
   1
   2
>> overallAverage = mean(mean(x))
overallAverage =  0.50000
```

Conditional operations work on matrices just like they do on arrays.

Command Window

```
>> x=reshape(-5:1:6,4,3)
x =
  -5  -1   3
  -4   0   4
  -3   1   5
  -2   2   6
>> y=zeros(4,3)
y =
   0   0   0
```

```
   0   0   0
   0   0   0
   0   0   0
>> compareMat = [x>=y]
compareMat =
   0   0   1
   0   1   1
   0   1   1
   0   1   1
>> a = rand(3,4)-0.5
a =
  -0.3887107   0.0714875   0.0048608   0.0619341
  -0.2742741  -0.3612460   0.4924967  -0.1843705
   0.0561357  -0.4229745   0.4680113   0.2996979
>> b = zeros(3,4)
b =
   0   0   0   0
   0   0   0   0
   0   0   0   0
>> c = [a>b]
c =
   0   1   1   1
   0   0   1   0
   1   0   1   1
```

3.6 Higher dimensional spaces

What if we had several matrices that were stacked on top of each other? This is where dimensions become really useful. Several important dimensions are summarized in Table 9.

Table 9: Description of some dimensions in ***Octave***.

Dimension	Representation	Analogy
1	Array	Line
2	Matrix	Plane
3	Stack of matrices	X-Y-Z space
4	Several stacks of matrices	Space-time

We can, for instance, make a stack of matrices by giving ***Octave*** a third input, or *argument*, in any of the matrix-creation commands. In this example, we made a stack of four matrices that each have two rows and three columns.

Command Window

```
>> x = ones(2,3,4)
x =
ans(:,:,1) =
   1   1   1
   1   1   1
ans(:,:,2) =
   1   1   1
   1   1   1
ans(:,:,3) =
   1   1   1
   1   1   1
ans(:,:,4) =
   1   1   1
   1   1   1
```

Here is an example with several stacks of matrices; it creates two stacks of matrices, each of which contains three matrices that each have five rows and four columns.

Command Window

```
>> y = zeros(5,4,3,2)
y =
ans(:,:,1,1) =
   0   0   0   0
```

```
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
ans(:,:,2,1) =
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
ans(:,:,3,1) =
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
ans(:,:,1,2) =
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
ans(:,:,2,2) =
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
ans(:,:,3,2) =
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
```

We can find the dimensions of an array or matrix using the `ndims` command. It may be confusing that ***Octave*** says the number of dimensions of a single number or of an array is two rather than zero or one. This is because ***Octave*** actually regards a single number as a one-row-by-one-column matrix, and it sees an array as a one-row (or one-column) matrix.

Command Window

```
>> x=1
x =  1
>> ndims(x)
ans =  2
>> xVec = ones(1,5)
xVec =
    1    1    1    1    1
>> ndims(xVec)
ans =  2
>> xMat = ones(3,5)
xMat =
    1    1    1    1    1
    1    1    1    1    1
    1    1    1    1    1
>> ndims(xMat)
ans =  2
>> x3dMat = ones(3,5,2)
x3dMat =
ans(:,:,1) =
    1    1    1    1    1
    1    1    1    1    1
    1    1    1    1    1
ans(:,:,2) =
    1    1    1    1    1
    1    1    1    1    1
    1    1    1    1    1
>> ndims(x3dMat)
ans =  3
>> x4dMat = ones(3,5,2,3)
x4dMat =
ans(:,:,1,1) =
```

```
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
ans(:,:,2,1) =
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
ans(:,:,1,2) =
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
ans(:,:,2,2) =
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
ans(:,:,1,3) =
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
ans(:,:,2,3) =
   1   1   1   1   1
   1   1   1   1   1
   1   1   1   1   1
>> ndims(x4dMat)
ans =  4
```

3.7 Example: Magic squares

Magic squares are matrices whose rows, columns, and diagonals all add up to the same value. ***Octave*** can automatically generate magic squares with size length `a` using the `magic(a)` command.

Command Window

```
>> x = magic(4)
x =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
>> columnSum = sum(x,1)
columnSum =
   34   34   34   34
>> rowSum = sum(x,2)
rowSum =
   34
   34
   34
   34
>> sum(diag(x))
ans =  34
>> sum(diag(rot90(x)))
ans =  34
```

We can have ***Octave*** randomly remove some of the values from the magic square to create a puzzle.

Command Window

```
>> y = magic(5)
y =
   17   24    1    8   15
   23    5    7   14   16
    4    6   13   20   22
   10   12   19   21    3
   11   18   25    2    9
>> z = round(rand(5))
z =
   1   0   0   0   0
   1   1   0   0   1
   1   0   1   0   0
   1   1   1   1   1
```

```
   0   1   1   0   1
>> q = y.*z
q =
   17    0    0    0    0
   23    5    0    0   16
    4    0   13    0    0
   10   12   19   21    3
    0   18   25    0    9
```

3.8 Example: Solving linear equations

Octave makes it easy to solve a system of linear equations using matrices. Let's say we wanted to solve these equations to find the values of x, y, z, m, and n.

$$4 = 1x + 3y + 45z + 6m + 3n \tag{48}$$
$$9 = 3x + 48y + 69z + 4m + 0n \tag{49}$$
$$-44 = 3x - 5y + 13z - 19m + 10n \tag{50}$$
$$23 = 34x - 8y + 8z + 0m + 0n \tag{51}$$
$$7 = 1x + 0y + 0z + 3m + 345n \tag{52}$$

It turns out that these equations can be written in matrix/vector notation with matrix A and column vectors b and v as follows.

$$A = \begin{bmatrix} 1 & 3 & 45 & 6 & 3 \\ 3 & 48 & 69 & 4 & 0 \\ 3 & -5 & 13 & -19 & 10 \\ 34 & -8 & 8 & 0 & 0 \\ 1 & 0 & 0 & 3 & 345 \end{bmatrix} \tag{53}$$

$$b = \begin{bmatrix} 4 \\ 9 \\ -44 \\ 23 \\ 7 \end{bmatrix} \tag{54}$$

$$v = \begin{bmatrix} x \\ y \\ z \\ m \\ n \end{bmatrix} \tag{55}$$

$$b = A \times v \tag{56}$$

In *Octave* we can create a matrix `A` that contains the right hand side of these equations, and a column array `b` that contains the left hand side. Then, using something called the *backslash* operator (\) we can solve for the column array `v` that contains x, y, z, m, and n from top-to-bottom. This is equivalent to stating $v = A^{-1} \times b$.

Command Window

```
>> A = [1 3 45 6 3; ...
3 48 69 4 0; ...
3 -5 13 -19 10; ...
34 -8 8 0 0; ...
1 0 0 3 345]
A =
     1     3    45     6     3
     3    48    69     4     0
     3    -5    13   -19    10
    34    -8     8     0     0
     1     0     0     3   345
>> b = [4; 9; -44; 23; 7]
b =
    4
    9
```

```
  -44
   23
    7
>> v = A\b
v =
   0.8044698
   0.3019979
  -0.2419990
   2.1971556
  -0.0011476
>> v = inv(A)*b
v =
   0.8044698
   0.3019979
  -0.2419990
   2.1971556
  -0.0011476
```

Thus, we obtain the solution:

$$x = 0.8044698 \tag{57}$$
$$y = 0.3019979 \tag{58}$$
$$z = -0.2419990 \tag{59}$$
$$m = 2.1971556 \tag{60}$$
$$n = -0.0011476 \tag{61}$$

3.9 Exercises

Try to create the following matrices (Equations 62-64) automatically.

$$aMat = \begin{bmatrix} 2 & 6 & 10 & 14 \\ 4 & 8 & 12 & 16 \end{bmatrix} \tag{62}$$

$$bMat = \begin{bmatrix} 16 & 12 & 8 & 4 \\ 15 & 11 & 7 & 3 \\ 14 & 10 & 6 & 2 \\ 13 & 9 & 4 & 1 \end{bmatrix} \tag{63}$$

$$cMat = \begin{bmatrix} 7 & 0 & 0 & 0 & 7 \\ 0 & 7 & 0 & 7 & 0 \\ 0 & 0 & 14 & 0 & 0 \\ 0 & 7 & 0 & 7 & 0 \\ 7 & 0 & 0 & 0 & 7 \end{bmatrix} \tag{64}$$

4 Plotting

4.1 Plotting basics

Plotting is important in computer programming because our minds process data visually. It can help us find problems in our code or to see new solutions if we plot the data before, during, and after our calculations. The basic command to create a plot in ***Octave*** is `plot(x,y)`, where `x` and `y` are two arrays of numbers that have the same length. Here is an example.

Command Window

```
>> x=linspace(0,2*pi,100);
>> y=sin(x);
>> plot(x,y)
```

The above code creates an array called `x` that consists of 100 evenly-spaced numbers starting on 0 and ending on 2π (note that the semicolon (`;`) surppresses the output), calculates the sine of those points and assigns them to array `y`, and finally plots `x` *vs.* `y` with `x` on the *horizontal axis* and `y` on the *vertical axis*. The plot is shown in Figure 2.

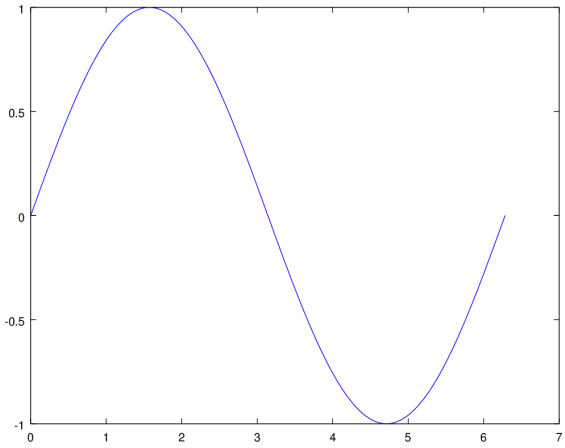

Figure 2: Figure of $y = \sin(x)$ over $0 \leq x \leq 2\pi$ with $-1 \leq y \leq 1$.

Here is another example. What if we were given the function $d = e^{-2t}\sin(10t)$ where d is displacement and t is time and we were asked to postulate what might have generated this response. We can plot the function like this, generating the graph in Figure 3.

Command Window

```
>> close all
>> t=linspace(0,5,200);
>> d=exp(-2*t).*sin(10*t);
>> plot(t,d)
```

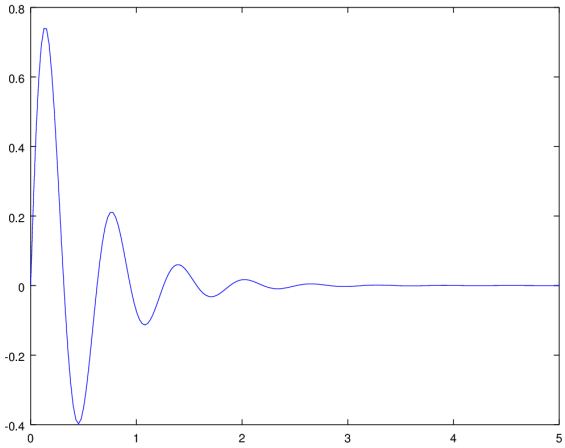

Figure 3: Graph of $d = e^{-2t}\sin(10t)$, representing the dampening of a vibrating guitar string.

Notice that the displacement decreases over time. Perhaps this represents someone putting their finger on a vibrating guitar string to dampen out the vibrations. Note that the command `close all` removes all previously open figures, such as that created in the $y = \sin(x)$ example above.

Let's now try to find out what happens when we change the `2` parameter inside the exponential term, for example by making it equal to `4`. We can do this easily by plotting the old function and the new function in the same window. First, we make a new window using the `figure` command, and then use `hold on` before plotting and `hold off` after plotting to send all

plots to the current graph window. The results are shown in Figure 4. It looks like this parameter controls the amplitude of the oscillations. Also, counterintuitively, the amplitude decreases as this number increases.

Command Window

```
>> close all
>> t=linspace(0,5,200);
>> dA=exp(-2*t).*sin(10*t);
>> dB=exp(-4*t).*sin(10*t);
>> figure(1)
>> hold on
>> plot(t,dA)
>> plot(t,dB)
>> hold off
```

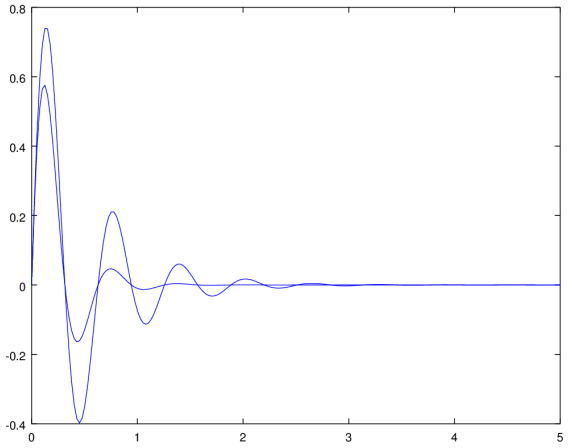

Figure 4: Comparison of two functions in the same graph window.

Try making plots of the following functions. As in the previous example, use the `figure(n)` command (where `n` is `1`, `2`, `3`, *etc.*) to create a new window and avoid overwriting the previous plot. Be careful to choose an appropriate spacing and range of the horizontal axis points so that the curves are smooth and complete. The ***Octave*** code is given below for reference.

$$y = x^3 - x^2 + x \tag{65}$$

$$b = -e^a \cos(a^2) \tag{66}$$

$$d = \frac{1}{c} \tag{67}$$

$$v = \ln\left(u \sin\left(u\right)\right) \tag{68}$$

Command Window

```
>> close all
>> figure(1)
>> x=linspace(-5,5,1000);
>> y=x.^3 - x.^2 + x;
>> plot(x,y)
>>
>> figure(2)
>> a=linspace(0,10,1000);
>> b=-exp(a).*cos(a.^2);
>> plot(a,b)
>>
>> figure(3)
>> c = linspace(-1,1,1000);
>> d = 1./c;
>> plot(c,d)
```

```
>>
>> figure(4)
>> u=logspace(-5,1,10000);
>> v=log(u.*sin(u));
>> plot(u,v)
```

4.2 Styling plots

It's possible to change the way that plots look in order to make them more clear or to emphasize certain features. We'll style one plot here to demonstrate some of the options. First, close any open figures, generate the arrays `x` and `y` to plot, create a new graph window.

Command Window

```
>> close all
>> x=0:1:10;
>> y=[1 3 5 15 18 15 16 24 28 26 32];
>> figure(1)
```

Then, plot `y` *vs.* `x` with stars (`*`) using a red (`r`) line (`-`) to connect the dots. These point/line style options go in single quotation marks (`''`) and are called additional *arguments* (also called *inputs* or *options*) to the `plot` command. More plotting options are summarized in Table 10.

Command Window

```
>> plot(x,y,'*r-')
```

Table 10: Plotting options in ***Octave*** (adapted from Houcque [6]).

Marker		Color		Line style	
Code	Description	Code	Description	Code	Description
`+`	Crosshair	`k`	Black	`-`	Solid
`o`	Circle	`r`	Red	`--`	Dashed
`*`	Star	`g`	Green	`:`	Dotted
`.`	Point	`b`	Blue	`-.`	Dashed-dotted
`x`	Cross	`m`	Magenta		
`s`	Square	`c`	Cyan		
`d`	Diamond	`w`	White		
`^`	Triangle (up)				
`v`	Triangle (down)				
`>`	Triangle (right)				
`<`	Triangle (left)				
`p`	Pentagram				
`h`	Hexagram				

Next, add x and y axis labels (using `xlabel` and `ylabel`), a title (using `title`), and a legend (using `legend`). Note that if we have multiple lines on our graph, we can label each one by putting comma-separated entries into the `legend` command. We also might want to show the grid lines for the plot in order to make it easier to read; this is done using the `grid on` command. Lastly, we can change the axis limits of the plot, too, using the `axis([xMin xMax yMin yMax])` command. Here, the numbers `xMin`, `xMax`, `yMin`, and `yMax` are the minimum and maximum values on the horizontal and vertical axes. This produces the graph shown in Figure 5.

Command Window

```
>> xlabel('X Values')
>> ylabel('Y Values')
>> title('Fun Data')
>> legend('Our Data')
>> grid on
>> axis([0 10 0 40])
```

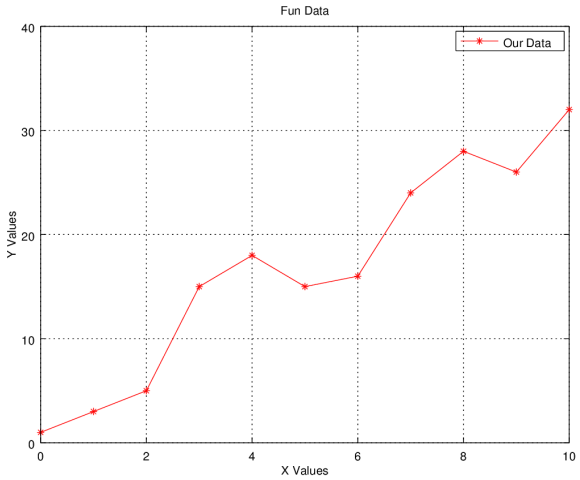

Figure 5: Graph with styling.

4.3 Advanced plots

We can make advanced plots too. Try these examples to see what the plots look like.

4.3.1 Pixel color

Use the `pcolor(x)` command to create a pixel color map using the values of two-dimensional (2D) matrix `x` (see Figure 6).

Command Window

```
>> close all
>> x=rand(10,10);
>> figure(1);
>> pcolor(x);
```

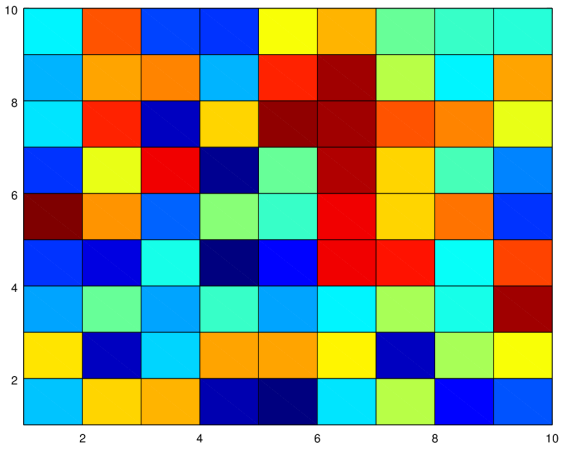

Figure 6: Pixel color graph example.

4.3.2 Pie charts

Use the `pie(x)` command to create a pie chart of the values of array `x`. The function normalizes the contents of `x` before plotting so that each slice of the pie chart is a percentage of the sum of the values of `x` (see Figure 7).

Command Window

```
>> close all;
>> x=rand(1,7)
x =
   0.698722   0.764991   0.834177   0.198607   0.619721   0.660698   0.039598
>> xNorm=x/sum(x)
xNorm =
   0.183079   0.200442   0.218570   0.052039   0.162379   0.173115   0.010376
>> figure(1);
>> pie(xNorm);
```

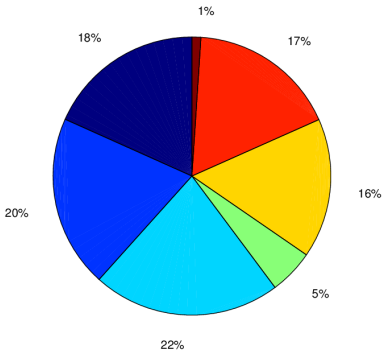

Figure 7: Pie chart example.

4.3.3 Bar charts

Similar to the `pie(x)` command, the `bar(x)` command constructs a bar chart using the elements of `x` (see Figure 8).

Command Window

```
>> close all
>> x=[10, 15, 30, 20, 25, 40, 50];
>> figure(1);
>> bar(x);
```

4.3.4 Three-dimensional plots

Use the `surf(x,y,z)` and `mesh(x,y,z)` commands to produce three-dimensional (3D) plots in which 2D matrix `z` is a function of 2D matrices `x` and `y` (see Figures 9 and 10). The `meshgrid(a,b)` command repeats arrays `a` and `b` to create two 2D matrices that repeat the contents of `a` and `b` over rows and columns, respectively. Once a plot window is open, we can click on the tools shown to rotate or translate the graph.

Command Window

```
>> close all
>> x=-10:0.5:10;
>> y=x;
>> [x2D,y2D]=meshgrid(x,y);
>> z2D=sqrt(x2D.^2+y2D.^2)+1;
```

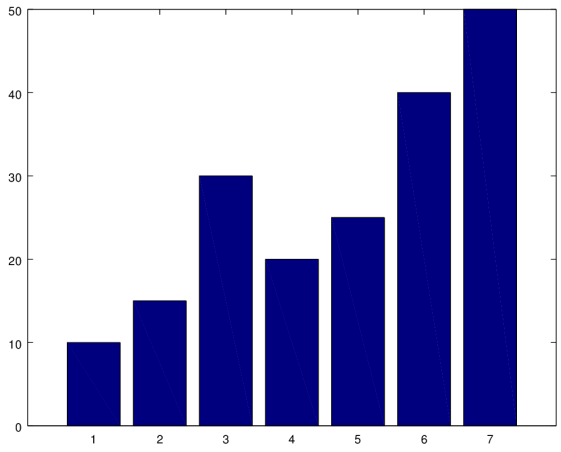

Figure 8: Bar chart example.

```
>> figure(1);
>> surf(x2D,y2D,z2D);
>> figure(2);
>> mesh(x2D,y2D,z2D);
```

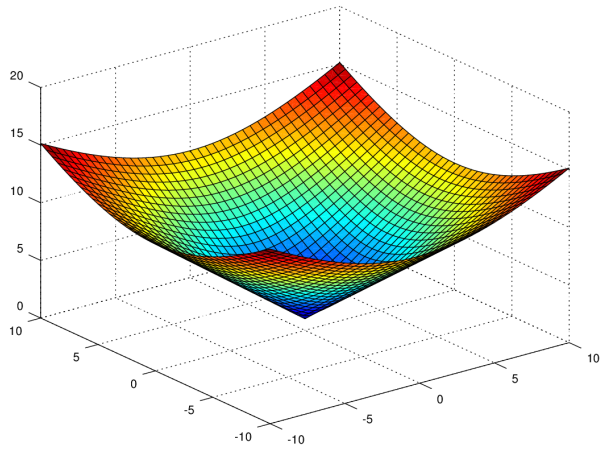

Figure 9: Surface plot example.

In the `surf(x,y,z)` and `mesh(x,y,z)` commands, `z` was a function of `x` and `y`. In other words, `z` was a *dependent variable* and `x` and `y` were *independent variables*. We can also create a plot where two coordinates are a function of just one using the `plot3(m,n,t)` command (see Figure 11). Here, `t` is an independent variable and `m` and `n` are dependent variables.

Command Window

```
>> close all;
>> t=linspace(0,10*pi,1000);
>> m=sin(t);
>> n=cos(log(t));
>> figure(1);
>> plot3(m,n,t);
```

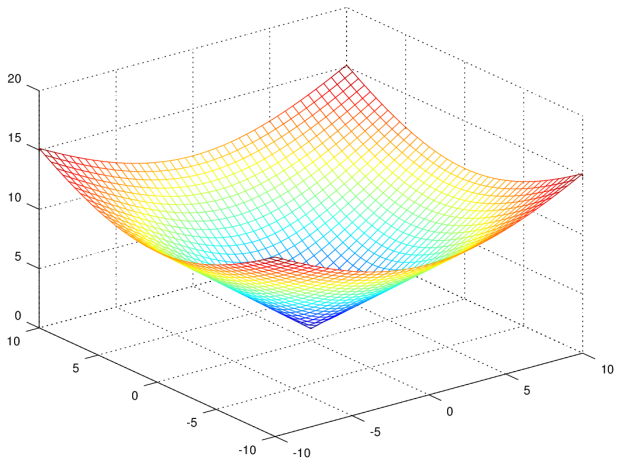

Figure 10: Mesh plot example

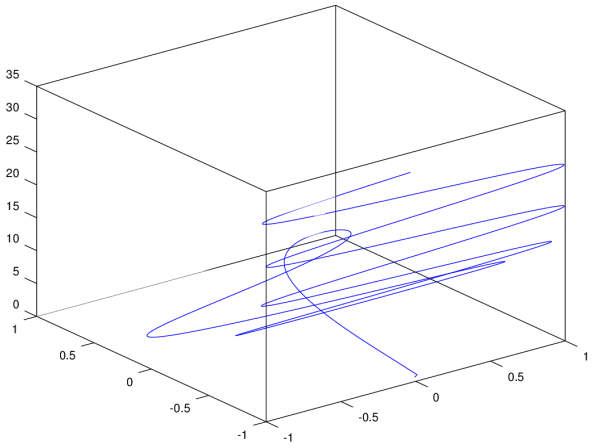

Figure 11: Example three-dimensional plot.

4.4 Exercises

Try to duplicate Figure 12. The functions in the figure are $y_1 = \cos(x^2)$ and $y_2 = \ln(x)$.

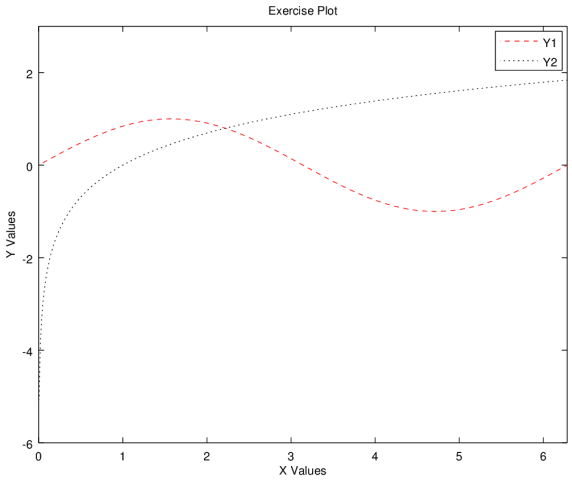

Figure 12: Figure to copy for exercise.

5 Scripts

5.1 Basic script

A *script* is a set of instructions that ***Octave*** runs sequentially. Scripts save time compared to entering commands into the command window because they allow multiple commands to be edited and re-run many times. Instead of entering each command to ***Octave*** individually in the command window, the commands are listed in the editor window in a saved file that itself is run by ***Octave*** as an entire unit. Each script is given a unique name with extension `.m`, such as `filename.m`.

Here is a simple script that plots two variants of the `sin` function. Open the editor window in ***Octave*** (either using the file menu or by typing `edit` in the command window) and type/copy in this code (if copying the code, you may need to re-type the single quotation marks (`'`) due to a copy-paste error). Then save the script with file name `plotSinScript.m`, and run the script by typing `plotSinScript` in the command window. The graph that this script produces is shown in Figure 13, and prints one line of output to the command window.

Editor

```
% plotSinScript.m
% Basic Script Example
% clear workspace
clear all
close all
format short
format compact
more off
clc
% print to command window
disp('Hello World!')
% generate arrays
x=linspace(0,2*pi,1000);
y1=30*sin(x);
y2=exp(x).*sin(x);
% graph the arrays
figure(1)
hold on
plot(x,y1,'c-')
plot(x,y2,'g-')
hold off
xlabel('X Values')
```

```
ylabel('Y Values')
legend('Function 1','Function 2')
axis([0 2*pi -180 35])
title('Variations of the Sine Function')
```

Command Window

```
Hello World!
```

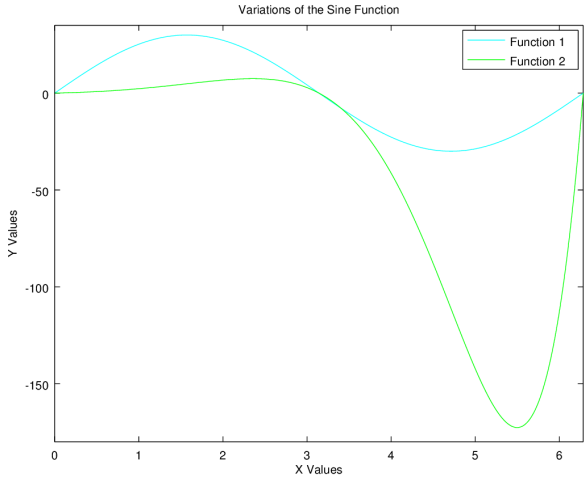

Figure 13: Plot generated by a sample script.

The first part of this script clears the workspace (command window and current variables). The next line prints `Hello World!` to the command window. Finally, the last section creates the arrays to be graphed, plots them, and styles the plot. Notice the text after the percent signs (`%`). These are called `comments` and their purpose is to place helpful information or reminders in the code.

5.2 Logical structures

5.2.1 If trees

Sometimes we need to tell ***Octave*** to choose between two alternatives or more as to what to do next in a program, depending on a set of conditions that we decide upon. This can be accomplished using *logical structures*. The primary type of logical structure is called an `if` *tree*. An example script involving an `if` tree is shown below, along with some sample output. Create a new script in the editor window, type/copy this text into it, and save it as `ifTreeExample.m`.

Editor

```
% ifTreeExample.m
% If Tree Example
% clear workspace
clear all
close all
format short
format compact
more off
clc
% multiply two random integers between 0 and 10
m=round(rand(1)*10)
n=round(rand(1)*10)
x=m*n
% decide what to do depending on the number
if(x<50)
    disp('x is closer to 0')
elseif(x>50)
    disp('x is closer to 100')
```

```
else
    disp('x is in the middle')
end
```

Command Window

```
m =  3
n =  1
x =  3
x is closer to 0
```

Here, the lines starting with `if`, `elseif`, and `else` are called *conditional statements*, the line `end` tells ***Octave*** that the `if` tree is complete, and the indented lines of code are the instructions. The indentation is done to organize the list and set apart its contents. The less than (<) and greater than (>) symbols are called *relational operators*; they ask ***Octave*** to compare two numbers. Two relational operators can be used in one conditional statement using the `and` (`&`), `or` (`|`), and `not` (`~`) commands, which are called *logical operators*. For instance, `if((x<y)&(a>b))` checks first if `x<y` and then checks if `a>b`. Also helpful are the short-circuit operators `&&` and `||`, which only evaluate the second statement if the first statement is true (`&&`) or false (`||`). For example, `if((x<y)&&(a>b))` checks first if `x<y` but only checks `a>b` if `x<y`. Several relational and logical operators are listed in Table 11.

Table 11: Relational and logical operators in ***Octave*** (adapted from Houcque [6]).

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
~=	Not equal to
&	And
|	Or
~	Not
&&	Short-circuit and
||	Short-circuit or

Just like the order of operations discussed in Section 1.4 (see also Table 2), relational and logical operational sequences also have a precedence structure. This structure is detailed in Table 12.

Table 12: Precedence structure for relational and logical operational sequences in ***Octave*** (adapted from Houcque [6]). The unary plus/minus operators are used for defining positive and negative numbers. The colon applies if/when performing logic with arrays.

Precedence	Operation
1	Parentheses (inside-to-outside)
2	Transpose (`'`), exponential (^)
3	Unary plus/minus (+/−) and logical negation (~)
4	Multiplication (⋆) and right/left division (//\)
5	Addition/subtraction (+/−)
6	Colon (`:`)
7	Relational operators (</<=/>/>=/==/~=; see Table 11)
8	And (&)
9	Or (|)

5.2.2 Switches

In situations where we have many alternatives from which ***Octave*** should chose, a `switch` statement can run more quickly and be understood by other computer programmers more readily. The following syntax compares one randomly-generated integer value to several other specific values. One sample result is given.

Editor

```
% switchExample.m
% Switch Example
% clear workspace
clear all
close all
format short
format compact
more off
clc
% pick a number and compare
x=round(10*rand(1))
switch x
    case 0
        disp('x=0')
    case 5
        disp('x=5')
    case 10
        disp('x=10')
    otherwise
        disp('Not 0, 5, or 10')
end
```

Command Window

```
x = 7
Not 0, 5, or 10
```

It is also possible to determine if a value falls into one of several ranges, as shown below.

Editor

```
% switchExampleRange.m
% Switch Example for Ranges
% clear workspace
clear all
close all
format short
format compact
more off
clc
% pick a number and compare
x=round(10*rand(1))
switch logical(true)
    case x<=1
        disp('x<=1')
    case x<=3
        disp('1<x<=3')
    case x<=7
        disp('3<x<=7')
    otherwise
        disp('7<x<=10')
end
```

Command Window

```
x = 0
x<=1
```

5.3 Loops

5.3.1 For loops

In some programs we need to do the same or a similar operation several times. For instance, we might need to ask ***Octave*** to multiply some numbers and store them in a new array. One way to do this is by using a *loop*. In particular, if the number of repetitions is known in advance, a `for` loop (sometimes called `for...end` syntax) can be used. Here is a simple script that multiplies the numbers in two arrays. Save it as `forLoopExample.m` then run it.

Editor

```
% forLoopExample.m
% For-Loop Example
% clear workspace
clear all
close all
format short
format compact
more off
clc
% create arrays
m=1:5;
n=linspace(100,-100,5);
m
n
% multiply arrays
for(k=1:length(m));
   outVecA(k)=m(k)*n(k);
   outVecB(k)=m(k).^3;
   outVecC(k)=n(k)/10;
end
% show output
outVecA
outVecB
outVecC
```

Command Window

```
m =
   1   2   3   4   5
n =
   100    50     0   -50  -100
outVecA =
   100   100     0  -200  -500
outVecB =
     1     8    27    64   125
outVecC =
   10    5    0   -5  -10
```

The loop code is contained between the `for` and `end` commands. To begin, the line `for(k=1:length(m))` tells ***Octave*** to run the loop one time for each element in array `m`, and moreover to pick a new value of `k` (`k=1, 2, 3, ...`) each iteration of the loop. The next three lines perform mathematical operations involving arrays `m` and `n` and then assign the results to the k^{th} element of arrays `outVecA`, `outVecB`, and `outVecC`. Finally, the line `end` tells ***Octave*** where the loop code is completed.

It's also possible to have two loops with one inside of the other; these are called *nested loops*. Here is an example that creates a matrix.

Editor

```
% doubleForLoopEx.m
% Double For-Loop Example
% clear workspace
clear all
close all
format short
format compact
more off
clc
% create arrays
x=(1:5).^2
y=linspace(1,10,10)
% generate matrix to fill
A=zeros(length(x),length(y))
% iterate over values of x and y to fill matrix A
for(m=1:length(x))
    for(n=1:length(y))
        A(m,n)=x(m)*y(n);
    end
end
% print matrix
A
```

Command Window

```
x =
    1    4    9   16   25
y =
    1    2    3    4    5    6    7    8    9   10
A =
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
A =
     1     2     3     4     5     6     7     8     9    10
     4     8    12    16    20    24    28    32    36    40
     9    18    27    36    45    54    63    72    81    90
    16    32    48    64    80    96   112   128   144   160
    25    50    75   100   125   150   175   200   225   250
```

In the example, the matrix `A` is created one element at a time by individually multiplying the values of `x` and `y`. The line `for(m=1:length(x))` starts the *outer loop* over the values of `x`, and the line `for(n=1:length(y))` starts the *inner loop* over the values of `y`.

Lastly, there are times when we may want to exit one cycle of a loop before it finishes or exit the loop altogether before all cycles have completed. These options are both possible using the `continue` (proceed immediately to the next loop cycle) and `break` (jump out of the loop and forgo all remaining cycles) commands.

5.3.2 While loops

In other programs, we need to do the same operation until a certain condition is met. One example is a random walk; we can tell ***Octave*** to randomly increment or decrement a variable until it "walks" out of a certain bounds. This can be accomplished using a `while` *loop*, which tells ***Octave*** to continue repeating a set of instructions until some end conditions are reached.

Editor

```
% whileLoopExample.m
% While Loop Example
% clear workspace
clear all
close all
format short
format compact
more off
clc
% set initial parameters
x=3;
k=1;
time(1)=0;
position(1)=x;
% walk x back and forth
while((x>=0)&&(x<=6)&&(k<100))
    x=x+(round(2*rand(1))-1);
    k=k+1;
    time(k)=k-1;
    position(k)=x;
end
% plot the progress
figure(1)
plot(time,position,'ks:')
xlabel('Time')
ylabel('X Position')
title('Random Walk from While Loop')
axis([0 time(end) -1 7])
% output results
time(end)
position(end)
```

Command Window

```
ans =  27
ans =  7
```

This function generates a plot like that shown in Figure 14. A random number (`-1`, `0`, or `+1`) is added to variable `x` as long as $x \geq 0$, $x \leq 6$, or the maximum permittable number of iterations is not reached (*i.e.*, $k < 100$).

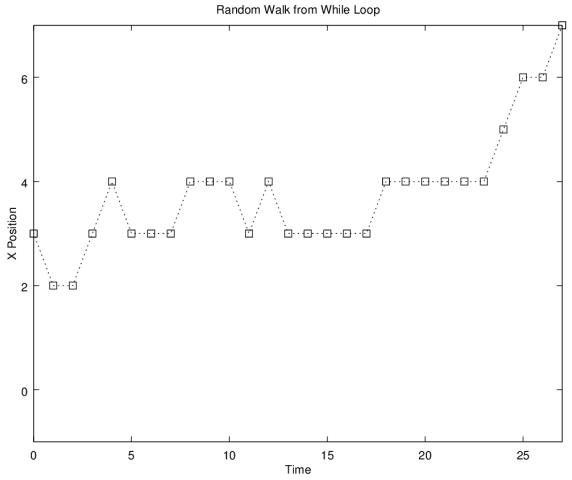

Figure 14: Random walk of a variable, generated using a loop.

5.4 Printing to the command window

It can be useful to print the output of a script to the command window. The most basic way to accomplish this is to type the name of the variable on a line and omit the semicolon (`;`) at the end. Another way is to use the `disp` command, which can either show the value of a variable (`disp(x)`) or a series of characters (`disp('some text')`). By the way, a series of characters is referred to as a *character array* or a *string*; character arrays can be stored in variables and printed as output, too. Lastly, a more advanced method is to use `fprintf`, which offers great control over the formatting and appearance of output. Try the example script below to test these methods.

Editor

```
% printingComWinEx.m
% Printing to the Command Window Example
% clear workspace
clear all
close all
format short
format compact
more off
clc
% create variables
x=pi;
y=e;
c=299792458; %m/s
% create strings
s1='Hello World!';
s2='Octave is ';
s3='awesome!';
% output without semicolon
x
y
% output using disp
disp('The value of Pi is')
disp(x)
disp(s1)
% output using fprintf
fprintf('The value of Pi is %8.6f\r\n',x)
fprintf('The value of e is %8.6f\r\n',y)
fprintf('Floating-point notation: pi=%7.2f and e=%7.2f\r\n',x,y)
fprintf('Scientific notation: %10.4e and e=%10.4e\r\n',x,y)
```

```
fprintf('The speed of light is %10.8e m/s\r\n',c)
fprintf('%10s%8s\r\n',s2,s3)
```

Command Window

```
x =  3.1416
y =  2.7183
The value of Pi is
 3.1416
Hello World!
The value of Pi is 3.141593
The value of e is 2.718282
Floating-point notation: pi=   3.14 and e=   2.72
Scientific notation: 3.1416e+00 and e=2.7183e+00
The speed of light is 2.99792458e+08 m/s
Octave is awesome!
```

Notice the syntax of the `fprintf` command. The first argument is a character array (notice the quotation marks, `''`) that specifies the formatting of the output, and the latter arguments are the numbers to be printed to the command window. Within the `%8.6f` character array in the first case, the `8` tells ***Octave*** to reserve `8` character spaces for a number, the `6` instructs ***Octave*** to print the number with `6` decimal places, and the `f` specifies that the output is a floating-point field. In later instructions, the `e` denotes scientific notation, and the `s` denotes character array output. Finally, the `\r\n` is an instruction to add a new-line character, meaning the next output statement will be printed to a new line in the command window.

We can combine `fprintf` statements within a `for` loop to print tables. The next script is an example; for each value in the `x` array it prints one row of the output table, consisting of the values in arrays `x`, `y1`, `y2`, and `y3`.

Editor

```
% printTableToComWinEx.m
% Printing Table to the Command Window
% clear workspace
clear all
close all
format short
format compact
more off
clc
% create arrays
x=linspace(0,1,10);
y1=x.^2;
y2=x.^3;
y3=x.^4;
% print header row
fprintf('  x     x^2    x^3    x^4\r\n')
% loop to print data rows
for(k=1:length(x))
    fprintf('%5.3f  %5.3f  %5.3f  %5.3f\r\n',x(k),y1(k),y2(k),y3(k))
end
```

Command Window

```
  x     x^2    x^3    x^4
0.000  0.000  0.000  0.000
0.111  0.012  0.001  0.000
0.222  0.049  0.011  0.002
0.333  0.111  0.037  0.012
0.444  0.198  0.088  0.039
0.556  0.309  0.171  0.095
0.667  0.444  0.296  0.198
0.778  0.605  0.471  0.366
0.889  0.790  0.702  0.624
1.000  1.000  1.000  1.000
```

5.5 Writing and reading text files

Another important capability of ***Octave*** is that of reading and writing text files consisting of tabular data. There are many advanced functions in ***Octave*** that can accomplish these tasks, but we will focus only on some of the basics here. Just like

writing tables to the command window, `fprintf` can be used inside a `for` loop to write text to a file. The following script writes data to a text file called `outputFile.csv` using this approach.

Editor

```
% printTableToFileEx.m
% Writing Table to a File
% clear workspace
clear all
close all
format short
format compact
more off
clc
% create arrays
x=1:1:10;
y1=x.^(1/2);
y2=x.^(1/3);
y3=x.^(1/4);
% open file
fileNameOut = 'outputFile.csv';
fid = fopen(fileNameOut,'w');
% print header row
fprintf(fid,'x,x^(1/2),x^(1/3),x^(1/4)\r\n')
% loop to print data rows
for(k=1:length(x))
    fprintf(fid,'%5.3f,%5.3f,%5.3f,%5.3f\r\n',x(k),y1(k),y2(k),y3(k))
end
% close file
fclose(fid);
```

File

```
x,x^(1/2),x^(1/3),x^(1/4)
1.000,1.000,1.000,1.000
2.000,1.414,1.260,1.189
3.000,1.732,1.442,1.316
4.000,2.000,1.587,1.414
5.000,2.236,1.710,1.495
6.000,2.449,1.817,1.565
7.000,2.646,1.913,1.627
8.000,2.828,2.000,1.682
9.000,3.000,2.080,1.732
10.000,3.162,2.154,1.778
```

Notice that the file name extension is `*.csv`, which stands for *comma-separated values*; this is the reason that the entries are all separated by commas (`,`). The `fopen` command opens the file and assigns it an identifier (`fid`). This identifier is used as an argument in the `fprintf` commands to tell ***Octave*** which file receives the output text. Also, the `'w'` argument gives ***Octave*** permission to write to the file. Lastly, the `fclose` statement closes the file.

We can then re-read this text file easily using `csvread`. This command generates a matrix consisting of the numbers in the text file. The arguments `1,0` tell ***Octave*** to skip the first row of the file but to not skip any columns.

Editor

```
% readTableFromFileEx.m
% Read Table from a File
% clear workspace
clear all
close all
format short
format compact
more off
clc
% specify file name
fileNameIn = 'outputFile.csv';
% read file to create matrix
A = csvread(fileNameIn,1,0)
```

Command Window

```
A =
    1.0000    1.0000    1.0000    1.0000
    2.0000    1.4140    1.2600    1.1890
    3.0000    1.7320    1.4420    1.3160
    4.0000    2.0000    1.5870    1.4140
    5.0000    2.2360    1.7100    1.4950
    6.0000    2.4490    1.8170    1.5650
    7.0000    2.6460    1.9130    1.6270
    8.0000    2.8280    2.0000    1.6820
    9.0000    3.0000    2.0800    1.7320
   10.0000    3.1620    2.1540    1.7780
```

5.6 Example: Printing multiplication table

By way of example, let's talk through the process we would go through to print out a multiplication table that goes up to $10 \times 10 = 100$. We could do it in one line using the vectorization capabilities of ***Octave***, but for practice we'll use a `for` loop and `fprintf` instead (the one-line version is `table=[meshgrid(1:10)].*[meshgrid(1:10)]'`). A matrix (*i.e.*, a multiplication table) generally requires a nested `for` loop, where the outer loop is usually used for the rows and the inner loop is normally for the columns. Also, we will need to print to the screen line-by-line, meaning we will have to have each row's values ready before issuing the print command. Here is what it could look like.

Editor

```
% makeMultTable.m
% Make Multiplication Table
% clear workspace
clear all
close all
format short
format compact
more off
clc
% loop over rows and columns
printArray = zeros(1,10);
for m=1:10;
    for n=1:10;
        printArray(n)=m*n;
    end
    fprintf('%4.0f%4.0f%4.0f%4.0f%4.0f%4.0f%4.0f%4.0f%4.0f%4.0f\r\n',printArray)
end
```

Command Window

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

In the above code, the `fprintf` command uses a shortcut in which the entire row vector is specified rather than each element being passed individually.

5.7 Exercises

Write scripts to accomplish the following tasks:

1. Plot five `sin` wave functions $y_n = \sin(mx)$ in the range $0 \leq x \leq 2\pi$ together on the same graph using random multipliers m spaced uniformly in the range $0 < m < 10$.
2. Print a 10-by-10 matrix of random numbers to a file named `randomNumbers.csv`.
3. List the first 100 prime numbers (using the `isprime` command).

6 Functions

6.1 Contents

A *function* is a piece of code that performs a task and/or returns a set of results, often based on a set of inputs provided by the user. At a basic level, the primary difference between scripts and functions in ***Octave*** is that functions can have input/output parameters, whereas scripts operate based on their pre-coded initial conditions and variables. Another important difference is that functions operate within their own work space, whereas scripts share the same work space as the command window; this means functions do not need to have commands like `clear all` at the top to remove all current variables, and also multiple functions can operate simultaneously while having variables with identical names but different values.

The general pattern for the contents of a function is shown below. Type/copy this text into the editor window and save it as `myFunction.m`.

Editor

```
function myOutput = myFunction(myInput)
% Return 1+myInput
myOutput = myInput + 1;
end
```

This simple function just adds `1` to whatever number was passed to it. The first line begins with the word `function`, which tells ***Octave*** that this is a function and not a script. The first line also shows that `myOuput` is the variable that the function will return, that the name of the function is `myFunction` (the function must also be saved with file name `myFunction.m`), and that the input to the function will be called `myInput`. The next line is a one-line description that is shown if you type `help myFunction` in the command window. The third line, and all subsequent lines until the `end` command, perform desired operations. Finally, the `end` statement closes the function. When the input to the function is `1`, the output is `2`:

Command Window

```
>> myFunction(1)
ans =  2
```

6.2 Input

There are several ways to provide input to a function. The most basic way is in the function's first line, as shown in the previous example. It is possible to have multiple input arguments as well, as in the following example.

Editor

```
function myOutput = multipleInputFunc(myInputA, myInputB, myInputC)
% Return myInputA+myInputB+myInputC
myOutput = myInputA + myInputB + myInputC;
end
```

This function produces the following output when the inputs are `1`, `2`, and `3`.

Command Window

```
>> multipleInputFunc(1,2,3)
ans =  6
```

Some functions don't require any inputs at all, such as this one, which returns an integer ranging from `0` to `10`.

Editor

```
function outVar = randTen
% Return random integer ranging from 0 to 10
outVar = round(10*rand(1));
end
```

Command Window

```
>> randTen
ans =  3
```

A function can also prompt the user for input information using the `input` command. This function converts kilometers to miles.

Editor

```
function mi = convertKmToMi
% Convert kilometers to miles
km = input('Enter number of kilometers: ');
mi = km*0.621371;
end
```

Command Window

```
>> convertKmToMi()
Enter number of kilometers: 42.2
ans =  26.222
```

6.3 Output

As in the case of inputs, functions can also output information in several ways. The examples above have already demonstrated the case of a single output variable (the output to the command window was assigned to the default `ans`). Functions can also produce no output variables but instead perform operations in the background, print output to a text file, or show output on the screen.

Editor

```
function printCircle(x)
% Print a circle pattern with integer x
x = abs(round(x));
while(x>=10);x=x-1;end;
fprintf('\r\n')
fprintf('%4.0f%1.0f%1.0f%1.0f%1.0f\r\n',x,x,x,x,x)
fprintf('%3.0f%6.0f\r\n',x,x)
fprintf('%2.0f%8.0f\r\n',x,x)
fprintf('%2.0f%8.0f\r\n',x,x)
fprintf('%3.0f%6.0f\r\n',x,x)
fprintf('%4.0f%1.0f%1.0f%1.0f%1.0f\r\n',x,x,x,x,x)
fprintf('\r\n')
end
```

Command Window

```
>> printCircle(7)

   77777
  7     7
 7       7
 7       7
  7     7
   77777
```

This function gives more than one output variable.

Editor

```
function [c, a] = getCircleInfo(d)
% Return the circumference 'c' and area 'a' of a circle with diameter 'd'
c=pi*d;
a=pi*(d/2).^2;
end
```

Command Window

```
>> [circ, area] = getCircleInfo(10)
circ =  31.416
area =  78.540
```

Finally, functions can also receive and return arrays or matrices. For example, using the previous function `getCircleInfo`, we can pass an array of diameters and receive back two arrays of corresponding circumference and area values.

Command Window

```
>> diameterValues = 1:5
diameterValues =
   1   2   3   4   5
>> [circValues, areaValues] = getCircleInfo(diameterValues)
circValues =
    3.1416    6.2832    9.4248   12.5664   15.7080
areaValues =
    0.78540   3.14159   7.06858  12.56637  19.63495
```

6.4 Sub-functions

One technique for keeping functions organized is to include smaller units of code, called *sub-functions*, within the larger structure. Here is a function that includes two sub-functions. It computes the square of an input value using addition, multiplication, and exponentiation.

Editor

```
function [ya, ym, ye] = testExp(x)
    % Demonstrate squaring by addition, multiplication, and exponentiation
    ya = expByAdd(x);
    ym = expByMult(x);
    ye = x^2;
    % subfunction for addition
    function ya = expByAdd(x)
        ya = 0;
        for k=1:x
              ya = ya + x;
        end
    end
    % subfunction for multiplication
    function ym = expByMult(x)
        ym = x*x;
    end
end
```

Command Window

```
>> [a, b, c] = testExp(7)
a =  49
b =  49
c =  49
```

Lastly, it can be helpful to use the `return` command in order to exit a function or sub-function before it is completed. For instance, if a function consists of many lines of code but in some special cases can be solved immediately upon executing the function, it may be useful to assign the output for these cases and then call the `return` option.

6.5 Example: Calculating current

Let's try writing a function that can compute the current (I, amperes [A]) required by an electronic device, given its power (P, watts [W]) and voltage (V, volts [V]). The equation relating these is

$$I = \frac{P}{V} \tag{69}$$

Our function must therefore obtain P and V from the user and return I. Here is one possible version.

Editor

```
function I = getIfromPV(P,V)
% Return I given P and V
I = P/V;
end
```

So, if we had a $P = 1000$ W device operating at $V = 240$ V, we could find its current consumption to be $I \approx 4.17$ A.

Command Window

```
>> I = getIfromPV(1000,240)
I =  4.1667
```

6.6 Exercises

Implement the following functions:

1. `TF=convertCtoF(TC)`: Converts Celsius temperature T_C to Fahrenheit temperature T_F according to $T_F = \frac{9}{5}T_C + 32$.
2. `tSeconds=convertDayToSeconds(tDay)`: Converts time in days t_{day} to time in seconds t_{second} according to $t_{second} = 86400 t_{day}$.
3. `[perimeter,diagonal,area]=getSquareInfo(sideLength)`: Determines the perimeter P, diagonal length D, and area A of a square given its side length L, according to $P = 4L$, $D = \sqrt{2}L$, and $A = L^2$.

7 Advanced concepts

7.1 Scalar structures

Sometimes a large program has sets of similar variables that apply to different problem states. For instance, a program that controls the air conditioning for a building with three rooms A, B, and C might have a temperature T, a pressure P, and a relative humidity ω for each room. Together, this would create $3 \times 3 = 9$ variables (*e.g.*, T_A, T_B, T_C, P_A, ..., ω_B, and ω_C). This can become rather cumbersome quite quickly (imagine a building with 100 rooms, each with 30 thermodynamic variables to control!). Instead of creating all of these variables, we can use data types called *scalar structures* to organize the information. Scalar structures group related data together in containers called *fields*. Fields can be assigned to scalar structures using dot syntax (*i.e.*, `A.T=25`) or the command `A=struct('T',25)`. These fields can be numerical variables, character arrays, numerical arrays, or other information.

Editor

```
% scalarStructureEx.m
% Scalar Structure Example
% clear workspace
clear all
close all
format short
format compact
more off
clc
% create structures
A.T = 25; % C
A.P = 101325; % Pa
A.omega = 50; % percent
B.T = A.T;
B.P = A.P;
B.omega = 40;
C.T = 30;
C.P = 101402;
C.omega = 70;
% list states
A
B
C
```

Command Window

```
A =
  scalar structure containing the fields:
    T =  25
    P =  101325
    omega =  50
B =
  scalar structure containing the fields:
    T =  25
    P =  101325
    omega =  40
```

```
C =
  scalar structure containing the fields:
    T =  30
    P =  101402
    omega =  70
```

The above example used only the numeric data type in the fields. We can also combine data types, as in the following plotting example (see Figure 15)

Editor

```
% scalarStructurePlottingEx.m
% Scalar Structure Example: Plotting
% clear workspace
clear all
close all
format short
format compact
more off
clc
% Create data
xVals=linspace(0,2*pi,50);
yVals=sin(xVals);
formatString='r-o';
titleString='Sine Wave';
xLabelString='X Values';
yLabelString='Y Values';
domainRange=[0 2*pi -1 1];
% create structure
data = struct('x',xVals, 'y',yVals, ...
'format',formatString, 'title',titleString, ...
'xLabel',xLabelString, 'yLabel',yLabelString, ...
'dr',domainRange)
% plot the data
figure(1)
plot(data.x,data.y,data.format)
xlabel(data.xLabel)
ylabel(data.yLabel)
title(data.title)
axis(data.dr)
```

Command Window

```
data =
  scalar structure containing the fields:
    x =
     Columns 1 through 7:
       0.00000   0.12823   0.25646   0.38468   0.51291   0.64114   0.76937
     Columns 8 through 14:
       0.89760   1.02583   1.15405   1.28228   1.41051   1.53874   1.66697
     Columns 15 through 21:
       1.79520   1.92342   2.05165   2.17988   2.30811   2.43634   2.56457
     Columns 22 through 28:
       2.69279   2.82102   2.94925   3.07748   3.20571   3.33394   3.46216
     Columns 29 through 35:
       3.59039   3.71862   3.84685   3.97508   4.10330   4.23153   4.35976
     Columns 36 through 42:
       4.48799   4.61622   4.74445   4.87267   5.00090   5.12913   5.25736
     Columns 43 through 49:
       5.38559   5.51382   5.64204   5.77027   5.89850   6.02673   6.15496
     Column 50:
       6.28319
    y =
     Columns 1 through 7:
       0.00000   0.12788   0.25365   0.37527   0.49072   0.59811   0.69568
     Columns 8 through 14:
       0.78183   0.85514   0.91441   0.95867   0.98718   0.99949   0.99538
     Columns 15 through 21:
       0.97493   0.93847   0.88660   0.82017   0.74028   0.64823   0.54553
     Columns 22 through 28:
       0.43388   0.31511   0.19116   0.06407  -0.06407  -0.19116  -0.31511
```

```
  Columns 29 through 35:
   -0.43388  -0.54553  -0.64823  -0.74028  -0.82017  -0.88660  -0.93847
  Columns 36 through 42:
   -0.97493  -0.99538  -0.99949  -0.98718  -0.95867  -0.91441  -0.85514
  Columns 43 through 49:
   -0.78183  -0.69568  -0.59811  -0.49072  -0.37527  -0.25365  -0.12788
  Column 50:
   -0.00000
 format = r-o
 title = Sine Wave
 xLabel = X Values
 yLabel = Y Values
 dr =
    0.00000   6.28319  -1.00000   1.00000
```

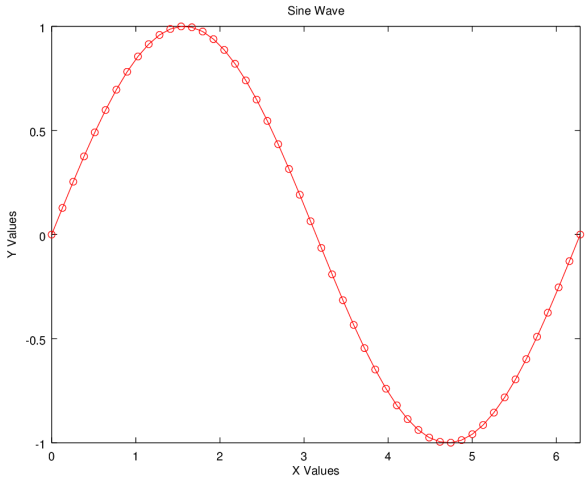

Figure 15: Figure created using structure data organization.

7.2 Character array manipulation

Octave can accomplish very complicated tasks on character arrays (sometimes called *strings*). This section will introduce a few of these capabilities; more information can be found using the `help` feature or the online documentation. As shown previously, a character array can be created using single quotation marks, as in `a='char array'`. Two character arrays can be combined using brackets `[]`, and a character array can be split using parenthesis `()`.

Command Window

```
>> a = 'this is a character array'
a = this is a character array
>> b=' also known as a string'
b =  also known as a string
>> c = [a b]
c = this is a character array also known as a string
>> d = c(11:19)
d = character
```

Searching within character arrays is possible using `strfind`. This arguments to this command are first the character array to be searched and second the character array to be found within the first character array; the function returns the index of the first character of the found string, or, if multiple instances are found, it returns a numerical array with these indices.

Command Window

```
>> x = 'ok lets go'
x = ok lets go
>> y = strfind(x,'go')
y =  9
```

```
>> z = strfind(x,'not')
z = [](0x0)
>> m = strfind(x,'o')
m =
    1   10
```

Strings can be converted to numbers and numbers to strings using the `str2num` and `num2str` commands, respectively.

Command Window

```
>> marathonDistance = str2num('26.2')
marathonDistance =  26.200
>> marathonDistanceChar = num2str(26.2)
marathonDistanceChar = 26.2
```

7.3 Cell arrays

Recall the scalar structures introduced above, which were able to contain different data types using fields. While scalar structures are very useful, they are in a sense limited because they cannot be indexed; in other words, the fields cannot be referred to by an array-like index (*e.g.*, `k=1`, `2`, `3`, ...). *Cell arrays* address this problem in that their containers, called *cells*, can not only contain any data type but also can be indexed. One cell array may contain, for instance, a character array, a vector, an integer, and a boolean (true/false) value. Cell arrays are created using curly brackets `{}`, and their contents can be accessed using index notation. In turn, contents of the cell contents (*i.e.*, values stored within a numeric array within a cell array) can be accessed using a combination of curly brackets `{}` and parenthesis `()`.

Command Window

```
>> x={'character array',[1 3 5 7],24,true}
x =
{
  [1,1] = character array
  [1,2] =
     1   3   5   7
  [1,3] =  24
  [1,4] =  1
}
>> x{1}
ans = character array
>> x{2}(2)
ans =  3
>> x{2}(4)
ans =  7
```

One particularly useful feature of cell arrays is their ability to function as arrays of character arrays, or effectively as string arrays.

Command Window

```
>> y={'this','is','a','cell','array','containing','strings'}
y =
{
  [1,1] = this
  [1,2] = is
  [1,3] = a
  [1,4] = cell
  [1,5] = array
  [1,6] = containing
  [1,7] = strings
}
```

7.4 Debugging

Problems with a piece of code can be discouraging and time-consuming. Fortunately, there are several best practices for debugging code systematically and efficiently.

The most important tool in debugging code is the comment. If a script or function does not run, begin by commenting out large sections of code until at least some of the code (even the first few lines) run smoothly. A single line can be commented

(using `%`) or a block of text can also be commented (using `%{` and `}%`). Once a minimal amount of code works, uncomment lines or segments of code and run the script/function until the erroneous lines have been identified (*i.e.*, the bug resurfaces). Comments are also useful for leaving notes in your program to yourself or others about how certain lines work. If a section is particularly complicated, it helps to have a guide as to how to understand it.

Another important debugging technique is outputting data (variables, arrays, text, *etc.*.) to the command window or plotting it in a figure window. Seeing actual values produced by the code is much more useful than guessing at what a section of code might be doing. Moreover, a graphical representation of data can allow the mind to quickly pinpoint pattern breaks or irregularities caused by code glitches. Values can be printed or plotted easily by modifying the code (*e.g.*, by deleting a trailing semicolon (`;`) so output to the command window is not suppressed) or by using the `keyboard` command. The `keyboard` command can be inserted on a blank line in a script or function; it causes ***Octave*** to pause and gives control of the cursor to the command window such that variables or other data in the `*.m` file can be examined.

Lastly, once bugs have been identified, it can be useful to use the find/replace features of the editor window to ensure that all instances of the problem have been corrected. Along the same lines, prior to making any large change to a piece of code (fixing a bug or implementing a new feature), it is wise to save a separate copy of the program. This way, if the change fails or produces a bug, it can be readily undone.

A Bug workarounds

This appendix lists some known bugs for ***Octave*** and their workarounds.

1. Command window cannot scroll up. If this happens, make a fake dialogue box using the code `msgbox('');` "msgbox(”);"and you will be able to scroll until you click the OK button in the window. See more information here: `https://savannah.gnu.org/bugs/?52496`

References

[1] John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. *GNU Octave version 4.0.0 manual: A high-level interactive language for numerical computations*. 2015. http://www.gnu.org/software/octave/doc/interpreter.

[2] MathWorks. *MATLAB*. The MathWorks, Inc., 3 Apple Hill Drive Natick, MA 01760-2098, R2011a edition, 2011. https://www.mathworks.com/products/matlab.html.

[3] Wikibooks. MATLAB programming - Wikibooks, the free textbook project, 2018.

[4] WaveMetrics, Inc. Igor Pro Version 6.34A. Technical report, WaveMetrics, Inc., Lake Oswego, OR, USA, 2015. https://www.igorpro.net/.

[5] Wolfram Alpha LLC. *Wolfram Alpha Computational Knowledge Engine*, 2018. https://www.wolframalpha.com/.

[6] David Houcque. *Introduction to MATLAB for Engineering Students*. Northwestern University, 1.2 edition, 2005. https://www.mccormick.northwestern.edu/documents/students/undergraduate/introduction-to-matlab.pdf.