Chapter 2

More on MATLAB

2.1 Strings and formatting

Comments and output is the analogue of good mathematical communication for coding. Be very careful with how you write comments and format output for marked assessments as communication is part of the marking scheme.

It is important to comment your code: communication is important, not just for the person reading your code, but for yourself to know what you were doing when you revise for your assessment. Comments are written using `%` signs. Such as

```
% This is a comment
```

To see how we should write comments, consider an earlier piece of code with suitable comments:

```
% We initially set the tally to be 0.
% At each point in the loop, tally represents a running total.
tally = 0;
% count is a variable that keeps track of the current number.
% We loop until count reaches 100
count = 1;
while count <= 100
    % At each iteration of the loop, we add count to the tally,
    % and then iterate count.
    tally = tally + count;
    count = count +1;
end
fprintf('The sum of integers from 1 to 100 is %d\n', tally);
```

Indenting commands within for loops, while loops and if statements is not completely necessary, but is good coding practice as it allows you to keep track of where control structures begin and end.

The last line of code above uses the `fprintf` function to produce formatted string output. Before talking about `fprintf`, we will talk about text data.

MATLAB has two kinds of data structures for storing text:

- character arrays, which are defined using single quotes (e.g. `str1 = 'abcdefgh'`),
- strings, which are defined using double quotes (e.g. `str2 = "abcdefgh"`).

We will not dwell on the underlying differences between the two, but it should be noted that they can and do behave differently in different circumstances. See the box below for more detail.

These distinctions will not be relevant in MAT5OPT, but are written for your own interest (and in case you encounter unexpected behaviour).

In MATLAB, a character array (using single quotes) is treated literally as an array of characters. The following two lines of code are equivalent:

```
>> str1 = 'abcde'
>> str1 = ['a', 'b', 'c', 'd', 'e']
```

Consequently, using single quotes allows easy access to individual characters. For example, `str1(3)` will output the letter c. This also has an advantage of making concatenation straightforward. Concatenation of character arrays can be performed using array concatenation, and the `length` command gives the expected output:

```
>> [str1, 'fgh', 'zyx']
ans =
    'abcdefghzyx'
>> length(str1)
ans =
     5
```

This can also misdirect you if you think you're working with an array of strings!

On the other hand, a string (using double quotes) is treated as a single object in its own right. So the objects `"abcde"` and `["a", "b", "c", "d", "e"]` are distinct. Evaluating `length("abcde")` will output 1, because it is an array containing a single string. The `strlength` function can be used instead.

This treatment of strings also permits arrays of strings, e.g., `["str1", "str2", "str3"]`.

In MAT5OPT, we will blur the distinction between them and commonly refer to them both as *strings*. For our purposes, the underlying difference in behaviour will not be relevant.

As for output, there are various ways it can be approached. Typing the name of a variable (without a semicolon) will output the contents of the variable, including the usual `var =` prefix. A simple way to output a single variable (including strings) without the prefix is by using the `disp` function:

```
>> a = 10;
>> a
a =
    10
>> disp(a)
    10
```

More often, we will want to assign some context to the output. The usual method is to use the `fprintf` command. The first argument to the `fprintf` function is a *format specification*, which specifies the format of the text to be output. An example"

```
a=5;
fprintf('I found the variable a to be equal to %d\n', a);
```

The format specification is the string `'I found the variable a to be equal to %d\n'`. This includes the following two substrings:

- the substring `%d`, a *specifier*, which will be replaced with a **d**ecimal number,
- the substring `\n`, s *special character*, which will place a line break at the end of the text.

The line break from `\n` is necessary when using `fprintf`, as otherwise all text will be displayed on a single

line.

After the format specification are the elements used to replace the specifier. In our example, the variable `a` that follows is substituted for the specifier `%d`, and output as an integer or using scientific notation.

More generally, specifiers are replaced in the order they appear.

```
fprintf('I found these variables: a, which is %d; b, which is %d, and
        c; which is %d\n',a,b,c);
```

To get control over the number of digits you can use the specifier `%f` The specifier `%f` can be given more detail with the following syntax:

```
%[width].[precision]f
```

The width allocates a minimum number of characters used to display the number (which can be useful for aligning text) and can be omitted. The precision specifies the number of decimal places. Some examples for you to try:

```
fprintf('To 3 decimal places, pi is %.3f\n',pi)
fprintf('To 10 decimal places, pi is %.10f\n',pi)
fprintf('To 5 decimal places and extra space, pi is %10.5f\n',pi)
```

The most common form you will want to use is, e.g., `%.3f`, which will output the number to 3 decimal places.

Also, it is possible to store formatted text in variables by replacing `fprintf` with `sprintf`:

```
output = sprintf('To 4 decimal places, pi is %.4f', pi)
disp(output)
```

Note that in this case, it was not necessary to use `\n` to introduce a line break—this was done automatically by `disp`.

A final note: outputting a variable without any context is bad practice, and will be marked accordingly.

Activity: practice string formatting

Write a function `myDisplay` which takes two arguments, `x` and `y`, then calculates $e^x + 2\cos(y)$, and then prints three lines of text stating the values of `x` and `y` as well as the result of the computation, to 2 decimal places. It should behave as follows:

```
>> myDisplay(3.123, pi)
To 2 decimal places, x is 3.12.
To 2 decimal places, y is 3.14.
To 2 decimal places, e^x+2cos(y) is 20.71.
```

2.2 Local functions

In the previous chapter, we described how to define function files. A more precise description of a function file would be as follows:

- The function whose name matches the file is called a *main function*, and is accessible anywhere in MATLAB (e.g. in the command window or in other scripts).
- Other functions can be defined in any script file (including function files), so long as they are written at the end of the file. Such a function is called a *local function*, and can be accessed only within the file they are written.

Sometimes, it is more convenient to write a script file with multiple local functions instead of writing several separate function files.

Copy the code below into a script file called `displayTemperature.m`:

```
function displayTemperature(celsius)
% Takes as input a temperature in degrees Celsius, and displays the
% temperature in Celsius, Fahrenheit and Kelvin.
fahrenheit = celsiusToFahrenheit(celsius);
kelvin = celsiusToKelvin(celsius);
fprintf('Temperature in Celsius: %.2f°C\n', celsius);
fprintf('Temperature in Fahrenheit: %.2f°F\n', fahrenheit);
fprintf('Temperature in Kelvin: %.2fK\n', kelvin);
end

function fahrenheit = celsiusToFahrenheit(celsius)
fahrenheit = (celsius * 9/5) + 32;
end

function kelvin = celsiusToKelvin(celsius)
kelvin = celsius + 273.15;
end
```

Because `displayTemperature` is the main function, it can be used in the command window. However, the functions `celsiusToFahrenheit` and `celsiusToKelvin` will be unavailable.

```
>> displayTemperature(30)
Temperature in Celsius: 30°C
Temperature in Fahrenheit: 86°F
Temperature in Kelvin: 303.15K
>> celsiusToKelvin(30)
Unrecognized function or variable 'celsiusToKelvin'.
```

This can also be utilised in generic script files without a main function, permitting multiple function definitions to be used when it is not necessary to use them in other scripts.

You can copy the following code into a script file and name it anything you like, then run it to plot the graph of three separate functions. An explanation of the commands used is given in the subsequent sections.

```
% Plot three functions: f1, f2 and f3.
xc = linspace(-3,3,100);
y1 = f1(xc);
y2 = f2(xc);
y3 = f3(xc);
hold on;
plot(xc,y1);
plot(xc,y2);
plot(xc,y3);

function y = f1(x)
y = x.^2 + 2*x + 3;
end

function y = f2(x)
```

```
y = sin(x)+cos(x);
end

function y = f3(x)
y = abs(x);
end
```

Activity: practice local functions

Using Pythagoras's theorem, the length H of the hypotenuse of a right-angled triangle in terms of the other two side lengths, x and y, is:

$$H = \sqrt{x^2 + y^2}.$$

Then, the distance between two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ in 2D space is the length of the hypotenuse of the right-angled triangle with other two side lengths $|x_1 - x_2|$ and $|y_1 - y_2|$.

Write a main function `distance` that takes two arguments, `p1` and `p2`, which are arrays containing two elements (representing points in 2D space), and uses a local function `hypotenuse` to calculate the distance between the two points.

E.g., to evaluate the distance between the points $(1, 1)$ and $(2, 2)$, it should run as follows:

```
>> distance([1,1], [2,2])
ans =
   1.414213562373095
```

2.3 Matrices

If we wish to input a matrix, such as

$$M = \begin{pmatrix} 1 & 0 & 5 \\ 0 & -4 & 7 \\ 0 & 0 & 2 \end{pmatrix}$$

we use spaces (or commas) to separate columns, and semicolons to separate rows:

```
>> M = [1 0 5; 0 -4 7; 0 0 2]
M =
     1     0     5
     0    -4     7
     0     0     2
```

MATLAB has various built-in functions to construct matrices with certain properties, such as `ones`, `zeros`, `magic` and `rand`. Read the documentation (using the `help` keyword) to learn more about them.

The entry in row i, column j is obtained using `M(i,j)`. For example, to access the element in row 2, column 3:

```
>> M(2,3)
ans =
     7
```

As we did with arrays, we can employ lists and the `end` keyword to extract multiple elements simultaneously. A single colon can be used to indicate the entire row/column is desired. To extract all of row 1:

```
>> M(1,:)
ans =
```

```
     1     0     5
```

To extract all of column 2:

```
>> M(:,2)
ans =
     0
    -4
     0
```

Recall that the matrix minor M_{ij} denotes the matrix obtained by removing row i and column j. Here are several different ways to obtain the matrix minor M_{12} (i.e., remove row 1 and column 2 from M).

Firstly, an explicit description of the desired columns:

```
>> M([2 3],[1 3])
ans =
     0     7
     0     2
```

Another method is to make a copy of the matrix and then delete the rows and columns:

```
>> M12 = M;
>> M12(1,:) = [];
>> M12(:,2) = []
M12 =
     0     7
     0     2
```

For instance, by writing `M12(1,:) = []` we are deleting that row from the matrix.

Another method uses the fact that boolean operators and relational operators act elementwise on arrays, so the following code works by requesting only row indices which are not equal to 1, and column indices which are not equal to 2:

```
>> M(1:end~=1, 1:end~=2)
ans =
     0     7
     0     2
```

Sometimes it can be convenient to treat two-dimensional arrays as one-dimensional arrays. This is called linear indexing, and is possible as MATLAB actually stores a matrix as a list of columns.

```
>> M(5:7)
ans =
    -4     0     5
```

Note that this was obtained by reading down the columns of M. Try to predict the output of `M(:)` and then confirm it by evaluating it in MATLAB.

The operations of multiplication, division and powers in MATLAB are based on how they act on matrices.

Operation	Syntax
Matrix addition	`+`
Matrix subtraction	`-`
Matrix multiplication	`*`
Solve a system of linear equations	`/`
Matrix powers	`^`

The symbol `*` performs matrix multiplication by default, and both `/` and `^` act on matrices.

Writing `x = B/A` in MATLAB when A and B are matrices will attempt to solve the system of linear equations $xA = B$, if possible. The system $Ax = B$ can be solved in MATLAB using `x = A\B` instead.

The power symbol `^` will compute matrix powers, e.g. M^2 is computed using `M^2`.

Furthemore, MATLAB has the usual matrix operations such as the size, determinant, inverse and transpose. The corresponding MATLAB commands are `size`, `det`, `inv` and `'`.

```
>> M'
ans =
     1     0     0
     0    -4     0
     5     7     2
>> size(M)
ans =
     3     3
>> det(M)
ans =
    -8
>> (M*[1 1 1]')'
ans =
     6     3     2
```

Note that `size` outputs a list where the first entry is the number of rows and the second entry is the number of columns.

The entries of the inverse of a matrix M will be a double-precision floating point. To get the exact result one can use `sym`

```
>> inv(M)
ans =
   1.000000000000000                   0  -2.500000000000000
                   0  -0.250000000000000   0.875000000000000
                   0                   0   0.500000000000000
>> sym(inv(M))
ans =
[ 1,    0, -5/2]
[ 0, -1/4,  7/8]
[ 0,    0,  1/2]
```

You may need to install the *Symbolic Math Toolbox* for this to work. In that case, MATLAB will prompt you to install it.

The relational operators and most functions will act elementwise on matrices. For example, to detect which elements have absolute value smaller than 3:

```
>> abs(M)<3
ans =
     1     1     0
     1     0     0
     1     1     1
```

You can find the indices of the entries that satisfy a condition:

```
>> find(abs(M)>3)'
ans =
```

```
     5     7     8
```

This uses linear indexing. The two-dimensional indices can also be found by requesting more than one output:

```
>> [r,c] = find(abs(M)>3);
>> r'
ans =
     2     1     2
>> c'
ans =
     2     3     3
```

This tells us that the elements in M with absolute value larger than 3 are in entries $(2, 2)$, $(1, 3)$ and $(2, 3)$.

Activity: practice matrix operations

Read the documentation on the `randi` function to determine how to construct a matrix of random integers. Then, construct a randomly generated 5×5 matrix `A`, a randomly generated 5×1 matrix `B` and a randomly generated 2×5 matrix `C`, with maximum possible value 15. Use those matrices to compute the following:

1. All possible products of two matrices chosen from the three matrices.
2. All possible matrix determinants for the three matrices.
3. All possible matrix inverses for the three matrices.

2.4 Elementwise operations

An important feature of MATLAB is that many operations act *elementwise* on lists. This means that when a function is applied to a list, the assumed behaviour of that function is that it applies that function to each element in the list. This is different behaviour to languages such as Python, where a function may throw an error if it is given a list as input. For example, suppose you wanted to calculate the natural logarithm of all numbers from 1 to 100. Your first thought might be to use a for loop, but instead, you can do the following:

```
>> x = 1:100;
>> log(x)
```

The output will show the result of applying the natural logarithm to each element of the array `x`.

This behaviour extends to the binary operations of addition, subtraction, multiplication, division and powers. However, since `*`, `/` and `^` refer to *matrix operations*, a special syntax is used for elementwise multiplication, division and powers.

Operation	Syntax
Addition (elementwise)	`+`
Subtraction (elementwise)	`-`
Multiplication (elementwise)	`.*`
Division (elementwise)	`./`
Powers (elementwise)	`.^`

Try to predict the output of the following operations before running them in MATLAB.

```
>> M = [1 0 5; 0 -4 7; 0 0 2]
>> N = ones(3,3)
>> M + N
>> M - N
>> M * N
>> M .* N
>> M / N
>> M ./ N
>> M ^ 2
>> M .^ 2
```

The important thing to notice is that the elementwise operations (`+`, `-`, `.*`, `./` and `.^`) apply the operation to elements in matching indices, whereas the non-elementwise operations (`*`, `/` and `^`) apply a special operation to the matrices.

This behaviour is particularly important for plotting. Observe that in Section 2.2, the local function `f1` used the elementwise power operation:

```
function y = f1(x)
y = x.^2 + 2*x + 3;
end
```

The effect of this is that the function can take *one or more arguments* arbitrarily. Consequently, by choosing a set of points to apply the function to, one obtains a representative sample of the function evaluated at several points.

Here we use an inline version of the function to demonstrate the behaviour. The function `f1` will succeed on one input and on many inputs, whereas `f2` will succeed only on one input.

```
>> f1 = @(x) x.^2 + 2*x + 3;
>> f2 = @(x) x^2 + 2*x + 3;
>> f1(3)
ans =
    18
>> f2(3)
ans =
    18
>> f1([1,2,3])
ans =
     6    11    18
>> f2([1,2,3])
Error using ^
Incorrect dimensions for raising a matrix to a power. Check that
the matrix is square and the power is a scalar. To operate on
each element of the matrix individually, use POWER*) (.^) for
elementwise power.
```

If you are experienced with programming in other languages, you may often be inclined to think of using a for loop to repeatedly apply an operation, but sometimes it is useful to be aware that elementwise operations may be a more effective solution.

Activity: practice elementwise operations

Generate two random arrays with 10 elements, `A1` and `A2`, and then:

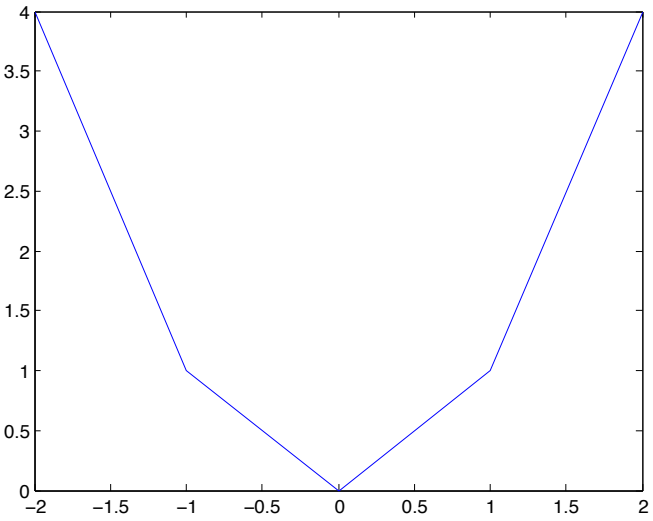
1. Compute the elementwise sum of `A1` and `A2`.
2. Square every element of `A2` and then subtract the natural logarithm of the corresponding element of `A1`.
3. Construct an array `B` whose elements are are defined as follows:
   - if `A1(i)` is even, then `B(i)` is the square of `A1(i)`,
   - otherwise, if `A2(i)` is even, then `B(i)` is `-A2(i)`,
   - otherwise, `B(i)` is `A1(i) + A2(i)`.

2.5 Plotting

Plotting in MATLAB is performed using the `plot` command. The `plot` command requires two equally sized lists. One list will contain the x-coordinates, while the other contains the corresponding y-coordinates. For example, a first attempt at plotting $f(x) = x^2$ over the interval $[-2, 2]$.

```
>> xc = -2:2;
>> yc = xc.^2;
>> plot(xc,yc);
```

Observe the use of the elementwise power when computing `yc`. This opens up a new window which contains the following figure:



We chose the points `[-2 -1 0 1 2]` as x-coordinates, and the corresponding points in the plane are joined by line segments. A more faithful representation of $f(x) = x^2$ will require more points in the interval. The simplest way to choose a nice set of points is by using the `linspace` function. Evaluating `linspace(a,b,N)` will generate `N` equally spaced points between `a` and `b`. To create a nice set of x-coordinates that splits the interval $[-5, 5]$ into 100 pieces, the following would do Alternatively, one can use `linspace(a,b,N)`.

A nicer version of the plot might proceed as follows:

```
>> xc = linspace(-2,2,100);
>> yc = xc.^2;
>> plot(xc,yc);
```

You can embed the plot window in the main window by using the *Dock Figure* button, which is an arrow that appears on the right side of the menubar:

You can also add labels, a title, a legend, and other graphical features using the *Insert* menu in the window that contains the plot. They can also be added from the command window using commands such as `xlabel`, `ylabel`, `legend`, `title`, etc. Use the `help` function to read more about them.

The default behaviour is that each plot overwrites the previous one. You can add to a plot by executing `hold on`, which is usually followed by `hold off`. An example:

```
>> f = @(x) x.^2 + 2*x - 3;
>> df = @(x) 2*x + 2;
>> x = linspace(-5,5,200);
>> hold on
>> plot(x, f(x));
>> plot(x, df(x));
>> title('A function and its derivative.');
>> legend('f(x)', 'f''(x)')
>> xlabel('x')
>> ylabel('y')
>> hold off
```

Note: because strings are delimited using quotes, to include a quote inside a string requires writing a double quote. So `legend('f(x)', 'f''(x)')` will format the second string as `f'(x)`.

Additional windows can be created using, e.g., `figure(2)`.

You can add a grid using `grid on` and remove it using `grid off`, and zoom in using the `axis` function:

```
>> axis([3.4 3.6 12 13])
```

Moreover, selecting `Generate Code` from the *File* menu of the plot window will generate the code that creates your plot, which might come in handy if you have to hand in MATLAB-code for an assessment...

Activity: practice plotting

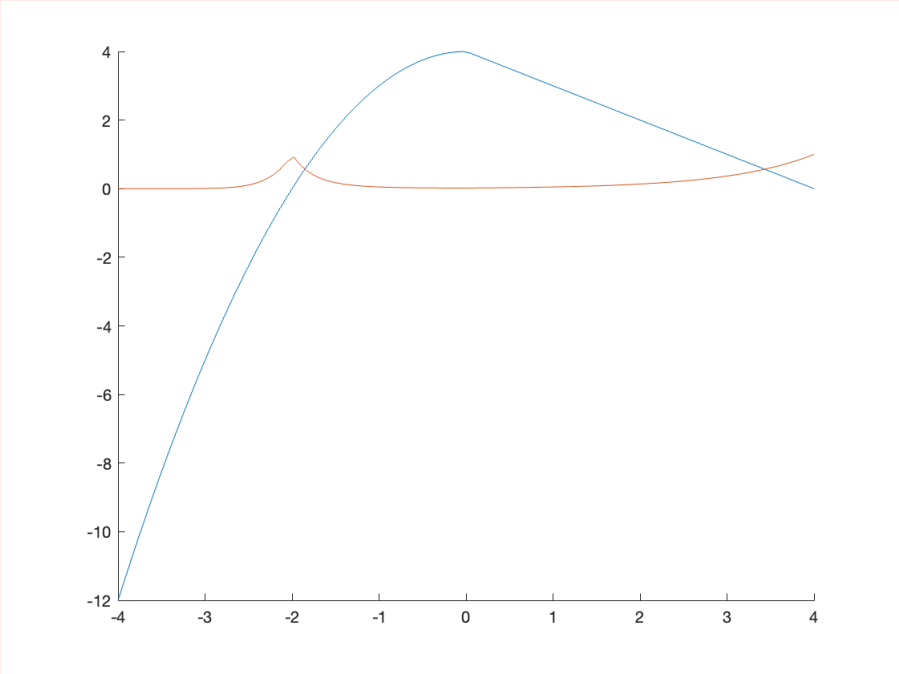
Consider the functions $f_1 \colon \mathbb{R} \to \mathbb{R}$ and $f_2 \colon \mathbb{R} \to \mathbb{R}$ given by

$$f_1 = \begin{cases} 4 - x^2, & \text{if } x \leqslant 0, \\ 4 - x, & \text{if } x > 0 \end{cases} \quad \text{and} \quad f_2 = e^{-|f_1(x)|}.$$

Define two MATLAB functions `f1` and `f2` as defined defined above, and then plot them on the same axis over the interval $[-4, 4]$.

A warning: to ensure f_1 correctly acts elementwise, using an if/else statement may not be suitable.

Your final plot should look something like this:



2.6 Timing output

Sometimes functions are defined recursively. For example, the Fibonnaci numbers are defined, for non-negative integers n, by the recurrence relation

$$\begin{aligned} F(0) &= 0, \\ F(1) &= 1, \\ F(n) &= F(n-1) + F(n-2), \text{ for } n > 1. \end{aligned}$$

The first ten Fibonacci numbers are then 0, 1, 1, 2, 3, 5, 8, 13, 21 and 34.

One way to program a function that takes n as input and outputs the n-th Fibonacci number is

```
function [F] = Fibonacci(n)
% This function is implemented recursively.
% It outputs the n-th Fibonacci number.
if n==0 | n==1
    F=n;
else
    F=Fibonacci(n-1)+Fibonacci(n-2);
end
end
```

To find the 40^{th} Fibonacci number,

```
>> Fibonacci(40)
ans =
      102334155
```

one has to be a little patient. Indeed, here the recursive definition is certainly not the most *efficient* algorithm. The following is much faster:

```
function [F] = FastFibonacci(n)
% This function uses a for loop.
```

```
% It outputs the n-th Fibonacci number much faster than Fibonacci.
if n==0 | n==1
    F=n;
else
    %We introduce two variables Fp(revious) and Fp(revious)p(revious)
    %set them to F(0) and F(1) initially
    %and update them n-1 times, to find F(2), ..., F(n)
    Fpp=0;
    Fp=1;
    for i=2:n
        F=Fp+Fpp;
        Fpp=Fp;
        Fp=F;
    end
end
end
```

The performance of a procedure can be quantified using `tic` and `toc`. Hold down the shift key while pressing enter to create a new line in the command window, or run this as a script instead.

```
>> tic
Fibonacci(40)
toc
ans =
   102334155
Elapsed time is 3.860232 seconds.
>> tic
FastFibonacci(40)
toc
ans =
   102334155
Elapsed time is 0.013059 seconds.
```

Activity: practice timing operations

Use `tic` and `toc` to compare the performance of the three functions in *Activity: practice looping structures* from Section 1.7.

2.7 Solutions to activities

Activity: practice string formatting

The following should be included in a file called `myDisplay.m`:

```
function myDisplay(x,y)
% Outputs x, y and e^x + 2*cos(y) to 2 decimal places.
z = exp(x) + 2*cos(y);
fprintf('To 2 decimal places, x is %.2f.\n', x);
fprintf('To 2 decimal places, y is %.2f.\n', y);
fprintf('To 2 decimal places, e^x + 2cos(y) is %.2f.\n', z);
end
```

Activity: practice local functions

The following should be included in a file called `myDisplay.m`:

```
function result = distance(p1, p2)
% Calculates the distance between the points p1 and p2
% using a local function
x1 = p1(1);
y1 = p1(2);
x2 = p2(1);
y2 = p2(2);
% taking the absolute value on the next line is not needed
% since the two numbers will be squared anyway
result = hypotenuse(x1-x2, y1-y2);
end

function H = hypotenuse(x,y)
% Calculates the length of the hypotenuse of a right-angled triangle
% whose other two side lengths are x and y
H = sqrt(x^2 + y^2);
end
```

Activity: practice matrix operations

The following code can be run in a script file:

```
A = randi(15, 5);
B = randi(15, 5, 1);
C = randi(15, 2, 5);

% The possible matrix products are:
A*A
A*B
C*A
C*B

% Only square matrices have a determinant, so we have
det(A)

% Only matrices with nonzero determinant have an inverse.
% This will depend on the randomly generated matrix
% So long as det(A) ~= 0, the matrix A has an inverse
inv(A)
%or
sym(inv(A))
```

Activity: practice elementwise operations

```
A1 = randi(20, 1, 10);
A2 = randi(20, 1, 10);

% Part 1:
```

```
A1 + A2

% Part 2:
A2.^2 - log(A1)

% Part 3:
% Start by making an array of 0's of suitable size
B = zeros(1,10);
% When A1 is even, use the square of A1
condition1 = mod(A1,2) == 0;
B(condition1) = A1(condition1).^2;
% Otherwise, if A1 is not even and A2 is even:
condition2 = ~condition1 & mod(A2,2) == 0;
B(condition2) = -A2(condition2);
% Otherwise, B is A1 + A2
condition3 = ~condition1 & ~ condition2;
B(condition3) = A1(condition3) + A2(condition3);
B
```

Activity: practice plotting

Here we draw attention to an important fact that the if/then control structure does not act elementwise. Consider the following script:

```
f1([-1,2,3])
f1_incorrect([-1,2,3])

function y = f1(x)
% We first create an array containing only zeroes
y = zeros(size(x));
% Then for the values of x which are <= 0,
% update y using the first formula.
y(x <= 0) = 4 - x(x <= 0).^2;
% Then for the values of x which are > 0,
% update y using the second formula.
y(x > 0) = 4 - x(x > 0);
end

function y = f1_incorrect(x)
if x <= 0
   y = 4 - x^2;
else
   y = 4 - x;
end
end
```

At first glance, it may appear that `f1([-1,2,3])` and `f1_incorrect([-1,2,3])` should both give the same output. However, the line `if x <= 0` will perform an *elementwise comparison* on `x`. It will then apply the first formula only if *all values* in `x` satisfy the condition. Making it apply elementwise requires some careful thought.

The code to produce the required plot is as follows:

```
x = linspace(-4,4,100);
y1 = f1(x);
y2 = f2(x);
hold on
plot(x,y1);
plot(x,y2);

function y = f1(x)
% We first create an array containing only zeroes
y = zeros(size(x));
% Then for the values of x which are <= 0,
% update y using the first formula.
y(x <= 0) = 4 - x(x <= 0).^2;
% Then for the values of x which are > 0,
% update y using the second formula.
y(x > 0) = 4 - x(x > 0);
end

function y = f2(x)
y = exp(-abs(f1(x)));
end
```

Activity: practice timing operations

We use local functions, so that the following script can be run in a single file:

```
% Generate a large array A
A = randi(30, 1, 100);


% Time each of the functions
tic
countEvensForLoop(A);
toc

tic
countEvensWhileLoop(A);
toc

tic
countEvensNoLoop(A);
toc


function count = countEvensForLoop(A)
% Counts the number of even elements in the list A
% using a for loop
count = 0;
for element = A
    if mod(element, 2) == 0
        count = count + 1;
```

```
    end
end
end

function count = countEvensWhileLoop(A)
% Counts the number of even elements in the list A
% using a while loop
count = 0;
index = 1;
while index <= length(A)
    if mod(A(index), 2) == 0
        count = count + 1;
    end
    index = index + 1;
end
end

function count = countEvensNoLoop(A)
% Counts the number of even elements in the list A
% using list functions.
count = sum(mod(A,2) == 0);
end
```