MXB261: Week 1 Workshop - MATLAB

Dr Nadiah Kristensen

Install MATLAB before the workshop

MATLAB is a large download and the installation takes a while, so do this before you arrive, don't leave it until the workshop.

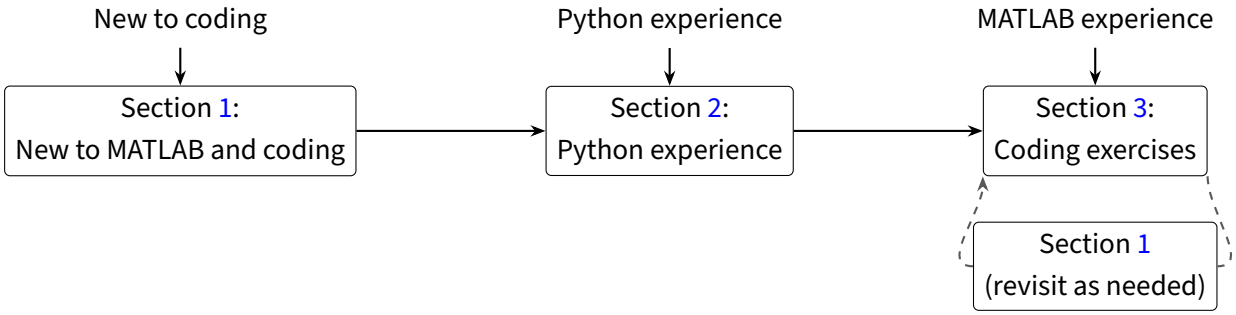
Sign up for a MathWorks account using your QUT email address:

> https://www.mathworks.com/mwaccount/account/create

This account will allow you to download and install MATLAB for free.

How to use this worksheet

This worksheet covers a lot of ground. You are **not** expected to complete it all in one workshop. It also serves as a reference to come back to throughout the course. Every question has a worked answer at the end. Pick the starting point that fits you.

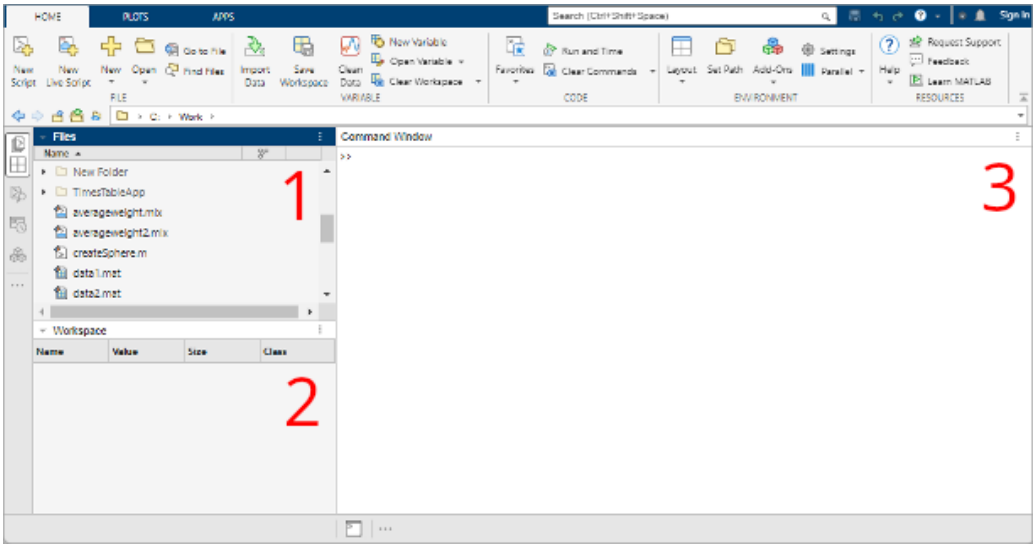


- **New to coding and MATLAB.** Start at Section 1 and work through in order, typing commands to complete exercises as needed. Aim to reach the end of Section 1.6 (Scripts) during workshop 1, and finish the rest in your own time. Use the coding exercises in Section 3 to gauge your progress. Aim to be able to complete all coding exercises before Week 3.
- **You know Python but not MATLAB.** Start at Section 2 and read to the end. Try the exercises in Section 3, dipping back into the earlier sections as needed.
- **You know MATLAB.** Start at Section 3, dipping back into the earlier sections as needed.

Contents

1 For students new to MATLAB and coding

When you start MATLAB, the desktop appears in its default layout.



1. Files panel: Access your files.
2. Workspace panel: Explore data that you create or import from files.
3. Command Window: Enter commands at the command line, indicated by the prompt

```
>>
```

1.1 Operations in the Command Window

In the Command Window, you can perform arithmetic operations:

```
>> 4 * 9
ans =
    36

>> 12 / 3
ans =
     4

>> 2^2
ans =
     4
```

When you don't assign the result to a variable, MATLAB stores it in a default variable called `ans`.

Question 1. When a semicolon `;` is put on the end of a line, e.g.,

```
>> 1 + 5;
```

what is its effect? .....

New variables can also be defined. They are listed in the Workspace Window.

```
>> x = 5; y = 10;
>> x
x =
     5

>> y
y =
    10

>> y = y + 50
y =
    60

>> temperature = 35;
>> temperature
temperature =
    35
```

Question 2. After defining `temperature` above, type just the first few letters

```
>> temp
```

and then press the `tab` key. What happens? .....

Anything that comes after a `%` is a comment; it is not read by MATLAB; comments are used to document code.

```
>> % This is a comment
>>
```

MATLAB has a number of built-in variables

```
>> pi
ans =
    3.1416
```

```
>> i
ans =
   0.0000 + 1.0000i % i is the imaginary unit, sqrt(-1)
```

Question 3. Type the following, noting the answers not shown.

```
>> pi = 3
>> clear pi
>> pi
>> i = 10
>> clear all
>> i
>> temperature
```

What does `clear` do? .....

The command `format` changes the number of decimals displayed:

```
>> format long
>> pi
ans =
   3.141592653589793

>> format short
>> pi
ans =
    3.1416
```

MATLAB has a number of inbuilt mathematical functions: `cos`, `sin`, `log`, `exp`, `rand`, `randn`.

Help for any inbuilt function can be obtained with the `help` command.

Question 4. Type the following

```
>> help mod
```

What does the function `mod` do? .....

`mod(7, 3) =` .....

Question 5. In the Command Window:

(a) Press the up and down arrows a few times. What does it do? .....

(b) Type the first few letters of something you've typed before, then press the up and down arrow keys. What does it do? .....

Question 6. Type the following

```
>> clc
```

What does it do? .....

Check if the variables you defined earlier are still in the workspace. Are they? .....

1.2 MATLAB stands for 'Matrix Laboratory'

Vectors and matrices can be defined in MATLAB using spaces or commas to separate elements along a row and semicolons to separate rows.

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4

>> x = [2; 3]
x =
     2
     3
```

When MATLAB was first created, it was not designed to be a programming language but an interactive matrix calculator. Therefore, its default behaviour is to treat operations performed on matrices and vectors as matrix operations

```
>> A * x
ans =
     8
    18

>> A * A
ans =
     7    10
    15    22
```

If the result for `A * A` is mysterious, don't worry, we will cover matrix multiplication next week. The point here is only that MATLAB does this by default.

Question 7. Type the following and note the results

```
>> A = [1 2; 3 4];
>> A .* A
>> A.^3
```

Compare the result `A .* A` to `A * A` above.

What does the 'dot' in `.*` and `.^` do? .....

Matrices can be constructed by concatenating together (glueing together) other vectors and matrices.

```
>> x = [2; 3];
>> y = [x x]
y =
     2     2
     3     3
```

Question 8. What will the following produce (check in the prompt if unsure)?

```
>> x = [2; 3];
>> z = [x; x]
```

It is possible to concatenate with an empty vector

```
>> x = []
x =
     []

>> y = [x, [3 3 3]]
y =
     3     3     3
```

Question 9. How would you quickly construct the following matrix?

```
>> D
D =
     1     2     1     2
     3     4     3     4
     1     2     1     2
     3     4     3     4
```

.....

.....

.....

.....

MATLAB has commands to quickly build certain common and useful matrices.

Question 10. Type the following and answer what type of matrix each command produces

```
>> zeros(3,2)
>> ones(2,4)
>> rand(5,6)
>> rand(5,6) % repeat it twice
>> randi([0, 2], 8, 8) % repeat it a few times
>> eye(4)
```

(a) `zeros(a, b)` .....

(b) `ones(a, b)` .....

(c) `rand(a, b)` .....

(d) `randi([imin, imax], a, b)` .....

(e) `eye(n)` .....

1.2.1 MATLAB indexes from 1, not 0

Curved brackets `()` are used for indexing as well as function calls. Unlike many other programming languages, MATLAB indexes from 1, not 0.

```
>> x = [2; 3]
x =
     2
     3

>> x(1) % first element
ans =
     2

>> x(2) % second element
ans =
     3
```

The indexing syntax for matrices is `var_name(row_nbr, col_nbr)`.

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4

>> A(1,2) % element in row 1, column 2
ans =
     2

>> A(2,1) % element in row 2, column 1
ans =
     3

>> A(2,2) % element in row 2, column 2
ans =
     4
```

Elements of a vector or matrix can be updated at a particular index.

```
>> A(1,1) = 1000;       % set the value in row 1, column 2 equal to 1000
>> A
A =
        1000           2
           3           4
```

Question 11. Enter the following matrix.

```
>> B = [1 2 3; 4 5 6; 7 8 9]
B =

     1     2     3
     4     5     6
     7     8     9
```

(a) To extract the element equal to 6, you would use `B(?,?)` .....

(b) How would you change the 6 to 50 (without rewriting the matrix)? .....

(c) How would you quickly set `B` back to its original value without having to retype it?

.....(Hint: see answer to Q5)

Enter the following and note the results.

```
>> B(:,1)
>> B(:,2)
```

(d) How would you extract the third column of `B`? .....

(e) How would you make all elements in the third column of `B` equal 0? .....

(f) How would you extract the third row of `B`? .....

Enter the following and note the results

```
>> B(1:2,:)
```

(g) How would you extract the last two columns of `B`? .....

(h) How would you set the first and second rows of `B` equal to 42? .....

(i) How would you create a new vector `x` equal to the first column of `B`? .....

(j) How would make the first column of `B` equal to the column vector (4, 5, 6)? .....

Enter the following and note the result

```
>> B(:, end)
>> B(end, end)
```

(k) What does `end` do? .....

It's often very useful to make an array with evenly spaced values between some minimum and maximum value, e.g., for defining x-axis values for plotting. There are two main ways.

The first way to make an array of evenly-spaced values is using the colon operator, which has the syntax `start:step_size:end`

```
>> x = 1:0.5:3
x =
    1.0000    1.5000    2.0000    2.5000    3.0000
```

If the `step_size` is left out, the stepsize will default to 1

```
>> y = 4:9
y =
     4     5     6     7     8     9
```

The second way to make an array of evenly-spaced values is `linspace(start, end, nbr_values)`.

```
>> linspace(1, 5, 6)
ans =
    1.0000    1.8000    2.6000    3.4000    4.2000    5.0000
```

Question 12. Consider the following array

```
x =
     1     4     7    10
```

(a) How would you construct this using the colon operator? .....

(b) How would you construct this using `linspace()`? .....

Question 13. Type the following and note the results (not shown)

```
>> A = 1:30
>> length(A)
>> B = reshape(A, 5, 6)
>> length(B)
>> C = B'
>> size(C)
>> size(C, 1)
>> size(C, 2)
>> length(C)
```

What do the following commands do?

(a) `reshape(A, 5, 6)`? .....

(b) `B'`? .....

(c) `size(C)`? .....

(d) `size(C, 1)`? .....

(e) `size(C, 2)`? .....

(f) `length()`? .....

(Hint: Use `help` if stuck)

1.2.2 Some useful functions to use on vectors

MATLAB has many built-in functions for working with a vector of numbers. Most of them do exactly what their name suggests, so the quickest way to learn them is to try them out.

Question 14. Define the vector

```
>> v = [4 1 8 3 5]
```

then type each command below and fill in what it returns. (Use `help` if a name is not obvious.)

(a) `sum(v)` .....

(b) `min(v)` .....

(c) `max(v)` .....

(d) `mean(v)` .....

(e) `prod(v)` .....

(f) `sort(v)` .....

1.2.3 Some useful functions to use on matrices

The same functions work on matrices, but here they work down each column, returning a row of answers, one per column.

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4

>> sum(A)        % sums DOWN each column -> a row of column sums
ans =
     4     6

>> mean(A)      % mean of each column
ans =
     2     3
```

When you want a single answer for the whole matrix, you can first flatten it into one long column with `A(:)` then apply the function:

```
>> A(:)          % stack all elements into one column
ans =
     1
     3
     2
     4

>> max(A(:))     % single largest element in the whole matrix
ans =
     4

>> sum(A(:))     % sum over ALL elements
ans =
    10

>> sum(A, "all") % also works for mean and prod
ans =
    10
```

Question 15. Define the matrix

```
>> M = [3 8 1; 6 2 9]
```

(a) Write a command that gives the sum of each column of `M`. .....

(b) Write a command that gives the mean over all the elements of `M`. .....

(c) Write a command that gives the single smallest element of `M`. .....

For `sum()`, `mean()`, and `prod()`, the operation can be made row-wise by specifying the `dimension` as 2:

```
>> help sum
--- help for sum ---

 sum - Sum of array elements
    This MATLAB function returns the sum of the elements of A along the
    first array dimension whose size does not equal 1.

    Syntax
      S = sum(A)
      S = sum(A,"all")
      S = sum(A,dim)
      ...

>> A
A =
     1     2
     3     4

>> sum(A, 1)    % sum down columns
ans =
     4     6

>> sum(A, 2)    % sum across rows
ans =
     3
     7
```

`dimension 1` works down the columns (the default), and `dimension 2` works across the rows.

For `min()` and `max()`, the syntax is a little different (see `help` for details)

```
>> min(A, [], 1) % min down columns
ans =
     1     2

>> min(A, [], 2) % min across rows
ans =
     1
     3
```

The empty brackets `[]` in `min(A, [], dim)` are there because `min` and `max` can also compare two arrays

```
>> min([1 8], [5 2])    % element-wise smaller of the two
ans =
     1     2
```

More generally, an empty `[]` in an argument slot is MATLAB's way of saying "use the default for this one". You will meet the same pattern with `std` and `var`, whose second argument is normally a normalisation setting:

```
>> std(A, [], 2)    % standard deviation across each row (default normalisation)
```

1.3 Relational operators

So far, our commands have produced numbers, vectors, and matrices. A relational operator instead asks a true/false question, and MATLAB answers with a `logical` value: `1` for true and `0` for false.

```
>> 3 < 5
ans =
  logical
   1

>> 3 > 5
ans =
  logical
   0
```

The full set of relational operators is

Operator	Meaning
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`~=`	not equal to

Note the double equals `==` for comparing, which is different from the single equals `=` for assigning.

```
>> x = 5        % single = ASSIGNS the value 5 to x
x =
     5

>> x == 5       % double == ASKS "is x equal to 5?"
ans =
  logical
   1

>> x == 6
ans =
  logical
   0
```

Question 16. Suppose `x` is still equal to `5`. What will each of the following return?

(a) `x ~= 5` .....

(b) `x >= 5` .....

(c) `x = 5` .....

Just like arithmetic, relational operators work elementwise on arrays, returning a logical array of the same size.

```
>> v = [1 5 3 8 2];
>> v > 3
ans =
  1x5 logical array
   0   1   0   1   0
```

Question 17. Using `v = [1 5 3 8 2]` from above, what does each return?

(a) `v == 3` .....

(b) `v ~= 5` .....

1.3.1 Combining conditions with logical operators

Two or more conditions can be combined with the logical operators `&` (*and*), `|` (*or*), and `~` (*not*). The *and* `&` is true only when *both* sides are true; the *or* `|` is true when *either* side is true.

```
>> v = [1 5 3 8 2];
>> (v > 2) & (v < 8)     % v bigger than 2 AND smaller than 8
ans =
  1x5 logical array
   0   1   1   0   0

>> (v < 2) | (v > 7)     % v smaller than 2 OR bigger than 7
ans =
  1x5 logical array
   1   0   0   1   0

>> ~(v > 3)              % NOT (v bigger than 3), i.e. flips 0s and 1s
ans =
  1x5 logical array
   1   0   1   0   1
```

The functions `any` and `all` collapse a logical array down to a single true/false answer: `any` is true if *at least one* element is true, and `all` is true only if *every* element is true.

```
>> any(v > 7)     % is ANY element bigger than 7?
ans =
  logical
   1

>> all(v > 0)     % are ALL elements bigger than 0?
ans =
  logical
   1
```

Question 18. Using `v = [1 5 3 8 2]`, predict each answer, then check at the prompt.

(a) `all(v < 8)` .....

(b) `any(v == 4)` .....

(c) Write a command that answers "are all elements of `v` even?" .....

..... (Hint: use `mod` from Q4.)

`&&` and `||` are short-circuit versions for single true/false values: they stop as soon as the answer is decided. You'll usually use these inside `if` statements (coming up later), and `&` and `|` for whole arrays.

1.3.2 Logical indexing

Logical indexing means using a logical array to pick out elements. Putting a condition inside indexing brackets keeps only the elements where the condition is true.

```
>> v = [1 5 3 8 2];
>> v > 3
ans =
  1x5 logical array
   0   1   0   1   0

>> v(v > 3)          % keep only the elements where v > 3 is true
ans =
     5     8
```

Logical indexing can also be used to assign selected elements, which is a quick way to clean up data.

```
>> v(v > 3) = 0      % set every element bigger than 3 to 0
v =
     1     0     3     0     2
```

Question 19. Start fresh with `w = [4 -1 6 -3 2 -8]`.

(a) Write a command that extracts just the negative elements of `w`. .....

.....

(b) How many negative elements does `w` have? .....

..... (Hint: combine a condition with `sum`.)

(c) Write a command that replaces every negative element of `w` with `0`. .....

.....

1.4 Character arrays and strings

MATLAB has two text types, depending on the quote type used: character arrays and strings.

```
>> 'Fizz'          % single quotes -> CHARACTER ARRAY (1x4 char), the old-school type
ans =
    'Fizz'

>> "Fizz"          % double quotes -> STRING (1x1 string)
ans =
    "Fizz"
```

Question 20. What values will the following commands return (use the prompt to check if unsure).

(a) `length('Fizz')` .....

(b) `length("Fizz")` .....

(c) `strlength("Fizz")` .....

Putting character arrays together in an array concatenates them together into a new array. But putting strings together in an array keeps them as separate elements of the array.

```
>> chars = ['Fizz', 'Buzz']
chars =
    'FizzBuzz'

>> words = ["Fizz", "Buzz"]
words =
  1x2 string array
    "Fizz"    "Buzz"
```

Question 21. Given the variables `chars` and `words` defined above, what will the following commands return (use the prompt to check if unsure).

(a) `length(chars)` .....

(b) `chars(2)` .....

(c) `length(words)` .....

(d) `words(2)` .....

The cleanest way to combine strings is using `+`. A `+` only combines strings; on character arrays it does arithmetic on the letter codes instead.

```
>> "Fizz" + "Buzz"
ans =
    "FizzBuzz"

>> "" + "Buzz"      % it is also possible to concatenate with an empty string
ans =
    "Buzz"
```

Using `+` will automatically convert a number to a string for you.

```
>> "Fizz" + 3
ans =
    "Fizz3"
```

However, if you want more control over how the number is displayed, use `sprintf`.

```
>> sprintf('%d: Fizz', 3)  % %d = whole number
ans =
    '3: Fizz'
```

Some useful format specifiers:

```
>> sprintf('%f', 3.14159)     % %f float, default 6 decimals
ans =
    '3.141590'

>> sprintf('%.2f', 3.14159)   % %.2f float, 2 decimal places
ans =
    '3.14'

>> sprintf('%e', 12345)       % %e scientific notation
ans =
    '1.234500e+04'

>> sprintf('%g', .00000000000000001) % %g trims zeros, uses sci. notn when sensible
ans =
    '1e-18'

>> sprintf('%g', 12345)
ans =
    '12345'
```

Question 22. Write `sprintf` commands that will produce the following output on the value of π (`pi`).

(a) `'pi: 3.1416e+00'` .....

(b) `'pi = 3.14'` .....

`sprintf` is useful for decorating plots:

```
k = 0.3;
title("Decay rate k = " + k)               % Decay rate k = 0.3
title(sprintf('Decay rate k = %.2f', k))   % Decay rate k = 0.30  (formatted)
```

1.5 Plotting to figure window

MATLAB's default behaviour is to plot to a new figure window:

```
>> alpha = linspace(-1, 1, 100);
>> c = 1;
>> y = alpha.^2 + c;
>> plot(alpha, y);
>> xlabel("alpha");
>> ylabel("alpha squared + c")
>> title("c = 1")
```

MATLAB reads a subset of TEX markup in plot text, so you get Greek letters, subscripts and superscripts. Some examples:

```
xlabel("\alpha")              % renders as the Greek letter alpha
ylabel("x_0")                 % subscript: x with a small 0
title("R^2 = 0.98")           % superscript: R squared
title("e^{-kt}")              % braces group multi-character super/subscripts
xlabel("\tau = " + tau + " s")   % mix TeX markup, text and a value
```

Question 23. Update the plot above so the annotations are:

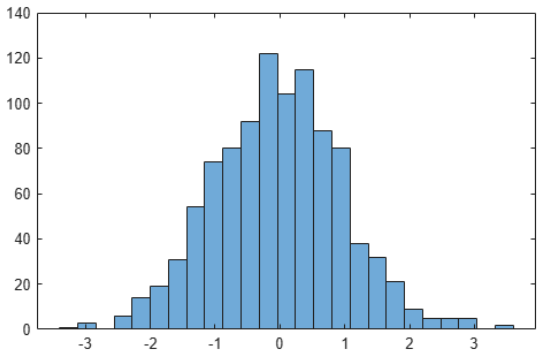
- xlabel: α
- ylabel: $\alpha^2 + c$
- title: $c = 1$

To plot more than one line on the same figure, use `hold on`

```
>> x = linspace(-pi,pi);
>> y1 = sin(x);
>> plot(x,y1)
>> hold on
>> y2 = cos(x);
>> plot(x,y2)
>> hold off
```

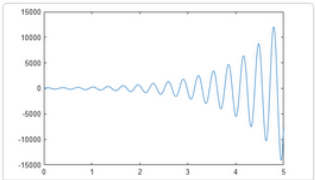
MATLAB has a wide range of plotting styles, e.g., histograms

```
x = randn(1000, 1);
nbins = 25;
h = histogram(x, nbins)
```

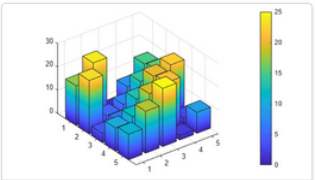


For more plotting styles and help with plotting, visit: https://au.mathworks.com/help/matlab/graphics.html

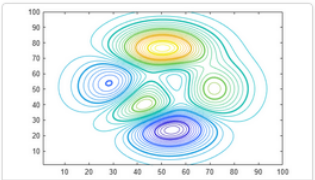
Featured Examples



Specify Axis Tick Values and Labels
Customize the tick values and labels along an axis, such as editing the tick value placement or modifying the tick label text and formatting.



Color 3-D Bars by Height
Modify a 3-D bar plot by coloring each bar according to its height.



Highlight Specific Contour Levels
Highlight contours at particular levels.

1.6 Scripts

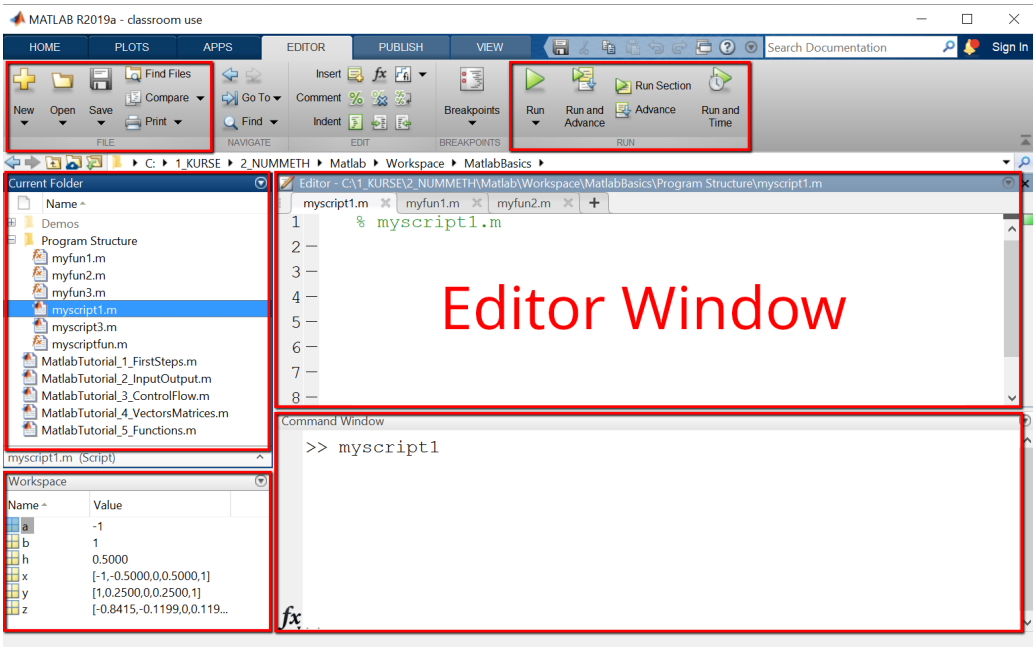
1.6.1 Writing and running scripts

The simplest type of MATLAB program is called a script. A script is a file that contains multiple sequential lines of MATLAB commands and function calls.

There are several ways to open a new script in the Editor Window:

- Click the 'New' button in the toolbar;
- Hit `ctrl + n` and type `myscript1.m`; or
- Type at the prompt: `» edit myscript`

A new Editor window should appear on your screen, which is the script file.



In the Editor Window, type the following (note: you can copy and paste this from the PDF)

```
disp("Hello world") % the command disp() displays to the Command Window
x = 1 + 1;
disp("x = " + x)
```

Save your script by clicking on the Save button on near the top left. Run your script by typing its name at the prompt:

```
>> myscript1
Hello world
x = 2
```

If you do not see the result above, call over the workshop tutor.

1.6.2 Scripts share the Command Window's scope

Two important notes about scripts.

1. Variables defined in scripts enter the workspace

Variables defined in a script become available in the workspace (Command Window).

```
>> clear all                              % clear all variables in workspace
>> x
Unrecognized function or variable 'x'.   % x unrecognised because workspace cleared
>> myscript1
Hello world
x = 2                                     % script defines value for x
>> x
x =
     2                                    % x now in workspace
```

This is useful for quick debugging, but it can be a problem if it overwrites variables you were using.

2. Variables defined in the workspace are accessible by scripts

To see this, follow these steps.

1. Define a new variable `a` in the workspace

   ```
   >> a = 5
   a =
        5
   ```

2. Add the line `disp(a)` to `myscript1.m` and save

   ```
   disp("Hello world")
   x = 1 + 1;
   disp("x = " + x)
   disp("a = " + a) % <-- new line
   ```

3. Run `myscript1`

   ```
   >> myscript1
   Hello world
   x = 2
   a = 5
   ```

   Note how the value of `a` was displayed even though it was not defined in the script.

This is rarely a useful feature. For this reason, it can be a good idea to start your script with a

`clear all`.

```
clear all       % <-- add this in all scripts so variables from the workspace don't
    enter the script
disp("Hello world")
x = 1 + 1;
disp("x = " + x)
```

1.7 Functions

1.7.1 Writing and calling functions

A function, like a script, is a file of MATLAB commands. The crucial difference is **scope**. A script shares the Command Window's workspace: the variables it defines leak out, and variables already in the workspace leak in (Section 1.6). A function is sealed off: the only way information gets in and out is through its inputs and outputs. This makes functions self-contained and reusable.

Copy the file called `triangle_area.m` from Canvas into your working directory or copy and paste from here:

```
function area = triangle_area(base, height)
% triangle_area  Area of a triangle from its base and height.
%   A = triangle_area(b, h) returns the area 0.5*b*h.

    if base < 0 || height < 0   % true if any input is negative
        error("base and height must be non-negative")
    end

    area = 0.5 * base * height;

end
```

A few things to notice about the syntax:

- The first line names the:
    1. output: `area`
    2. function name: `triangle_area`
    3. inputs: `base`, `height`
- The file must be saved with the same name as the function, i.e. `triangle_area.m`.
- There is no `return` statement. Whatever value the output variable `area` holds when the function reaches its `end` is what gets returned.

- The comment block directly under the first line is the function's help text.

Now call `triangle_area` from the Command Window:

```
>> triangle_area(4, 5)
ans =
    10

>> triangle_area(10, 3)
ans =
    15
```

Question 24. Run `triangle_area` with the following input

```
>> triangle_area(-2, 5)
```

What does it return, and why? .....

.....

Question 25. Type

```
>> help triangle_area
```

What does it return, and why? .....

.....

Because the function is sealed off, the variables it uses internally (`base`, `height`, `area`) never appear in your workspace.

```
>> clear all
>> triangle_area(4, 5)
ans =
    10

>> base
Unrecognized function or variable 'base'.
```

This is the opposite of a script, whose variables do appear in the workspace. A function cannot accidentally clobber your variables, and your variables cannot accidentally change what a function does.

Question 26. Recall the script in Section 1.6, which could "see" the workspace variable `a` even though `a` was never defined in the script.

(a) Suppose `a = 5` is in your workspace. Could `triangle_area` use `a` inside its body? Why or why not? .....

(b) What single line would you have to add to `triangle_area` so that it could use the value `5`? .

Question 27. Write your own function `celsius_to_fahrenheit(c)` that converts a temperature in Celsius to Fahrenheit using the formula

$$F = \frac{9}{5}C + 32.$$

Check `celsius_to_fahrenheit(100)` gives `212` and `celsius_to_fahrenheit(0)` gives `32`.

1.7.2 Anonymous functions

Anonymous functions are functions that are not stored in program files but as variables. They can accept multiple inputs but only return one output and execute a single statement. For example

```
sqr = @(x) x.^2;
```

Anonymous functions not only store the expression but also the values of the variables the expression uses. For example

```
>> a = 1.3;
>> b = .2;
>> c = 30;
>> parabola = @(x) a*x.^2 + b*x + c;
>> parabola(10)
ans =
   162
```

The values in `parabola` persist even if you change their values later or clear them

```
>> clear a b c
>> parabola(10)
ans =
   162
```

Anonymous functions are useful because they can be passed to another function like a variable.

For example, MATLAB has an inbuilt function `fzero` that finds the roots of a function given an initial guess. In the example below, the function `y` is passed to `fzero` as an anonymous function along with the initial guess `2`, and `fzero` calculates the more accurate estimate `2.0946`.

```
>> y = @(x) x.^3 - 2*x - 5;
>> root = fzero(y, 2)
root =
    2.0946
```

Question 28. Write the temperature-conversion function from Question 27 as an anonymous function.

.....

1.8 Loops

A loop runs the same commands many times over. MATLAB has two kinds of loops:

1. `for` loop: repeats its body once for each value in a list. Use this when you know in advance how many repetitions you want, e.g., for each number from 1 to 10.
2. `while` loop: repeats its body as long as a condition stays true, e.g., while $x < 2$. Use this when you do not know in advance how many repetitions you want, but you do know the condition that should make it stop.

Everything between the `for` or `while` line and its matching `end` is the body of the loop, and is run on each repetition. The following script, `eg_loop.m`, contains one of each:

```
clear all

counter = 0;

disp("Counting up:")
for i = 1:10
  counter = counter + 1;
  disp(counter)
end

disp("Counting down:")
while counter >= 5
  counter = counter - 1;
  disp(counter)
end
```

The script `eg_loop` produces the following output

```
>> eg_loop
Counting up:
     1
     2
     3
     4
     5
     6
     7
     8
     9
    10

Counting down:
     9
     8
     7
     6
     5
     4
```

Question 29. The factorial of a positive integer n, written $n!$, is the product of all the whole numbers from 1 up to n. For example:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Write a script containing a `for` loop that computes $10!$ (you should get `3628800`).

Question 30. A dish starts with a single bacterium, and the population doubles every hour. Write a script containing a `while` loop that finds how many whole hours pass before the population first grows to more than 1000.

1.9 If statement

An `if` statement lets a script make a decision: it runs a block of code only when a condition is true. The condition is usually a relational test (Section 1.3), which is either true (`1`) or false (`0`).

In its simplest form, an `if` pairs with an `else`. The following uses `mod` (Q4) to test whether a number is even.

```
n = 7;
if mod(n, 2) == 0
    disp("n is even")
else
    disp("n is odd")
end
```

With `n = 7`, `mod(n, 2)` is `1`, so the condition `mod(n, 2) == 0` is false; MATLAB skips the `if` block and runs the `else` block, displaying `n is odd`. Change `n` to an even number and the `if` block runs instead.

Question 31. Write a script that displays 10 randomly chosen integers between 1 and 5 inclusive, and if there are three or more 3s, it displays `wow!`.

Run it several times while checking; `wow!` will only appear on about a third of the runs.

Hints

How do I...?	**See**
randomly draw integers?	`randi()`, Q. 10
display?	`disp`
count integers in `A` equal to 3?	`sum(A == 3)`, Q. 19b

When there are more than two cases, add one or more `elseif` branches. MATLAB checks the conditions from top to bottom and runs the first one that is true, skipping the rest.

```
temp = 50;
if temp < 0
    disp("ice")
elseif temp < 100
    disp("liquid water")
else
    disp("steam")
end
```

With `temp = 50` this displays `liquid water`. The order matters: because MATLAB stops at the first true condition, the `elseif temp < 100` branch is only reached once `temp < 0` has already been ruled out.

Question 32. At a university, a final score out of 100 is converted to a grade from 3 to 7:

Score	Grade
85–100	7
75–84	6
65–74	5
50–64	4
below 50	3

Write a function that accepts a variable `score` and then uses `if` / `elseif` / `else` to return the corresponding `grade`.

Hint: Start by writing a script. That way you can use the Command Window to check variable values while you slowly build the logic, and then turn it into a function at the end.

1.10 Returning arrays and pre-allocation

So far, our loops have produced only a single value (e.g., `counter`). But more often, a loop produces one result per iteration and we want to store them all. The natural way to store them is in an array, and the loop's job becomes filling that array one entry at a time.

For example, suppose `temps` holds daily temperatures, with one row per city and one column per day.

```
temps = [
%  1  2  3  4 days
  22 31 28 35; % city 1
  18 19 25 21; % city 2
  40 38 33 36  % city 3
];
```

For each city, we want to count how many days were hotter than some `threshold`, so our result will be a column vector with one entry per city / row.

```
nbr_hot_days = [
    % count for city 1 goes here
    % count for city 2 goes here
    % count for city 3 goes here
```

```
];
```

Before the loop, we create `nbr_hot_days` at its full size filled with zeros. This is called pre-allocation, and we do it for two reasons. First, because the variable `nbr_hot_days` must exist before we can store a value in it. Second, more importantly, because it is faster. If instead we concatenated the new value onto the end of `nbr_hot_days` at each step, MATLAB would have to find a new larger block of memory and copy the whole array across at every iteration. Creating it once saves all that copying, and for long loops the speed-up can be significant.

```
clear all

temps = [
%  1  2  3  4 days
  22 31 28 35; % city 1
  18 19 25 21; % city 2
  40 38 33 36  % city 3
];

threshold = 30;

nbr_hot_days = zeros(size(temps,1), 1);     % <-- preallocated output array

for row = 1:size(temps,1)
    count = 0;
    for col = 1:size(temps,2)
        if temps(row,col) > threshold
            count = count + 1;
        end
    end
    nbr_hot_days(row) = count;              % fill array at corresponding row
end
```

Reading the loop from the outside in: `size(temps,1)` is the number of rows (3 cities) and `size(temps,2)` the number of columns (4 days). The outer loop visits each city in turn; for that city, the inner loop scans across its days, adding 1 to `count` for every day above `threshold`; the finished `count` is then written into that city's slot with `nbr_hot_days(row) = count`.

Running the script and then inspecting `nbr_hot_days`:

```
>> nbr_hot_days
nbr_hot_days =
     2
     0
     4
```

So the first city had 2 days above 30, the second had none, and the third had 4.

Question 33. Without using MATLAB's inbuilt `max()` function, write a script that takes the matrix

```
>> A = [3 9 2; 7 1 8; 4 4 6]
A =
     3     9     2
     7     1     8
     4     4     6
```

finds the largest element in each row, and stores it in a column vector `row_max`.

Hint: On a given row, store the first element as `best`. As you walk along the row, compare the next element to your current `best`; if it is larger, update `best` to equal the new larger value.

2 For students with Python experience

If you already know Python, MATLAB will feel familiar. It is a language built for numerical computing where the array is the fundamental type. It is much like `numpy` except arrays are baked into the language rather than added by a library.

The Command Window is an interactive read-eval-print loop, just like the IPython prompt: you type an expression, press enter, and see the result straight away. A script (an `.m` file) runs top to bottom just like running a `.py` script.

So most of what you know carries over; you mainly need to learn the different syntax and a handful of behaviours that differ from Python. This section is a quick orientation to those differences.

2.1 Four things most likely to trip you up

Each is explained more fully below:

1. Assignment copies; it does not alias. `b = a` gives `b` its own independent copy, so changing `b` never changes `a`.
2. Indexing starts at 1, not 0. `x(1)` is the first element. Ranges include both ends. `x(2:4)` keeps the 2nd, 3rd and 4th elements.
3. `*` `/` `^` are matrix operations by default. Put a dot in front `.*` `./` `.^` for element-wise operations on vectors and matrices.
4. Inequality is `~=`, not `!=`.

2.2 Python vs MATLAB

A more comprehensive cheat sheet can be found at https://mathesaurus.sourceforge.net/matlab-python-xref.pdf.

General behaviour:

Python	MATLAB	Description	Examples
`#`	`%`	Comment	`% hello`
`print`	Omit EOL `;`	Print output	`x`
`\`	`...`	Continue to next line	`x = 1+...2;`
`os`	`!`	Operating system command	`! echo hi`
`+ - * /`	`+ - * /`	Mathematical operators	`x = 1+2`
`**`	`^`	Exponent	`x = y^2`
`* / **`	`.* ./ .^`	Element-wise operators	`x = [1 2].* [3 4]`
`not, and, or`	`~ & |`	NOT, AND, OR logical operators	`if x<2 & x>2`
`!=`	`~=`	Inequality	`if x ~= 2`
`' "`	Not equivalent	See Sect. 1.4 for details	
`del`	`clear`	Clear variable from memory	`clear x y`
`clear`	`clc`	Clear command window	`clc`
`lambda`	`@`	Lambda / anonymous function	`sqr = @(x) x.^2;`
`len`	`numel` or `size`	`length` returns largest dimension	

Referencing:

Syntax	Purpose	Example
`( )`	Index (assignment copies; see below)	`x(1,1)`
`[ ]`	Create array	`x = [1 2 3]`
	Join arrays	`z = [x ; y]`
`{ }`	Create cell arrays	`x = {42; "hello world"}`
	Extract contents from a container	`x{1,1}`

Operators `* / ^` are linear-algebra operators by default. `A * B` is a matrix multiply and produces an error for mismatched shapes. For elementwise operations on matrices and vectors, `.* ./ .^` must be used instead.

Assignment copies. Unlike Python, where `b = a` makes `b` another name for the same list or array, in MATLAB `b = a` gives `b` its own independent copy. Changing one never affects the other.

```
>> a = [1 2 3];
>> b = a;
>> b(1) = 99;
>> a
a =
     1     2     3
```

Indexing starts at 1, and ranges include both ends. Where Python's `x[1]` is the second element and `x[1:4]` stops before index 4, MATLAB's `x(1)` is the first element and `x(2:4)` keeps the 2nd, 3rd, and 4th. Use `end` for the last.

```
>> x = [10 20 30 40 50];
>> x(1)          % first element (not the second)
ans =
    10

>> x(2:4)        % inclusive: 2nd, 3rd and 4th
ans =
    20    30    40

>> x(end)        % last element
ans =
    50
```

Functions are typically kept in separate files, but you can also define local functions within a script or function file. There are no imports/modules, and functions must be on the path.

Input arguments are captured in parentheses.

```
function z = foo(x,y)
   ...
end
```

Note there is no `return`; the value of z returned is its value when `end` is reached.

Multiple outputs are captured with square brackets.

```
function [a,b] = foo(x,y)
    ...
end
```

for loop

```
for i = 1:10
   ...
end
```

if statement

```
if x < 2
   ...
elseif x == 2    % <-- note: different to Python's elif
   ...
else
   ...
end
```

while loop

```
while x < 3
   ...
end
```

2.3 Exercise

If you feel confident, do Question 34 (FizzBuzz) in the next section and continue on.

If you have any troubles, refer back to Sect. 1 until you feel confident.

3 Coding exercises for all students

These exercises pull together everything from the workshop. Aim to be able to do them all eventually; start with FizzBuzz and work down. Each has a table of hints pointing back to the relevant sections if you get stuck.

3.1 FizzBuzz

Question 34. Write a script that displays the numbers 1 to 30, but:

- for multiples of 3, it displays "Fizz"
- for multiples of 5, it displays "Buzz"
- for multiples of both 3 and 5, it displays "FizzBuzz"

Hints

How do I...?	**See**
write and run a script?	Sect. 1.6
run through the numbers 1 to 30?	`for i = 1:30 ... end`, Sect. 1.8
check if a number is divisible by another number?	`mod`, Q. 4
condition behaviour on whether the numbers are divisible?	`if ... end`, Sect. 1.3, Sect. 1.9
handle text, e.g., checking length, concatenating text together?	Sect. 1.4
display?	`disp`

3.2 Bank history

Question 35. Write a function that accepts as input a vector of transactions `transn` and an initial balance `balance` and returns a history of balances as an output vector `history`.

The bank account is subject to an overdraft rule: any transaction that reduces the balance below zero is rejected.

Optional: display a table of transactions and balances.

Example desired output:

```
>> transn = [100 -30 -90 50 -40]; balance = 0;
>> history = bank_history(transn, balance)

transaction balance
          --    0.00
     100.00  100.00
     -30.00   70.00
     -90.00   70.00
      50.00  120.00
     -40.00   80.00

history =
   100    70    70   120    80
```

Hints

How do I...?	**See**
write and call a function?	Sect. 1.7, though start with a script.
create an output vector?	Pre-allocate `history = zeros(...)`, Sect. 1.10
step through each transaction?	`for k = 1:length(transn) ... end`, Sect. 1.8
implement the overdraft rule?	`if ... end`, Sect. 1.3, Sect. 1.9
store each new balance in the `history` array?	`history(k) = balance;`

3.3 Pascal's triangle

Question 36. Write a script that uses loops and an `if, else` statement to calculate Pascal's triangle down to 6 rows and store it in a matrix:

```
P =
     1     0     0     0     0     0
     1     1     0     0     0     0
     1     2     1     0     0     0
     1     3     3     1     0     0
     1     4     6     4     1     0
     1     5    10    10     5     1
```

The rules are:

- If the element is on an edge, i.e., in the first column or on the diagonal, then its value is `1`.
- Otherwise, its value is equal to the sum of the value above and above-left.

Hints

How do I...?	**See**
start with a 6×6 matrix of zeros to fill in?	`P = zeros(n, n)`, Sect. 1.10, Q. 10
visit every lower-triangular element, row by row?	a loop inside a loop, `for row = 1:n... for col = 1:row`, Sect. 1.8, hot-days example Sect. 1.10
tell if element `P(row, col)` is on an edge?	first column is `col == 1`, diagonal is `col == row`, Sect. 1.3, Sect. 1.9
refer to the value above and above-left?	`P(row-1, col)` and `P(row-1, col-1)`, Q. 11

3.4 Plotting two polynomials on the same figure

Question 37. Write a script that plots the two functions

$$y_1 = \alpha^2 \qquad \text{and} \qquad y_2 = \alpha^3$$

for α between -2 and 2, both on the same figure. Your plot should have:

- both curves on one set of axes;
- a legend labelling which curve is which;
- an x-axis labelled with the Greek letter α, and a y-axis label that uses a superscript;
- a title;
- and, finally, save the figure to a file `polynomials.pdf`.

Hints

How do I...?	**See**
make evenly-spaced α values?	`:` operator or `linspace`, Q. 12
plot a curve and add axis labels and a title?	`plot`, `xlabel`, `ylabel`, Q. 23
put Greek letters or superscripts in a label?	TeX markup, e.g. `\alpha` and `y^2`, Q. 23
draw a second curve on the same figure?	`hold on`, example Sect. 1.5
add a legend?	`help legend`
save the figure as a PDF?	`exportgraphics(gca, "polynomials.pdf")`

3.5 Vectorisation

Vectorisation writes loops more briefly using element-wise operations, logical indexing, and MATLAB's built-in functions. It can make code clearer to read and faster to run.

Question 38. Rewrite the following Questions and Examples with no loop:

(a) Q. 29 as a single line (Hint: `help prod`).

(b) Hot-days count from Sect. 1.10, producing the same column vector `nbr_hot_days` (Hint: Q. 19b).

(c) Q. 33, producing vector `row_max`. This time you may use `max` (Hint: final example Sect. 1.2.3).

4 Answers

4.1 For students new to MATLAB and coding

Question 1:

A semicolon prevents the answer from being displayed.

Question 2:

`tab` autocompletes.

Question 3:

`clear` removes variables from the workspace and returns predefined variables to their default values.

Question 4:

`mod` gives the remainder after division (modulo operation). `mod(7, 3) = 1`

Question 5:

(a) Up and down arrow keys cycle through previous commands typed.

(b) Typing some characters and then pressing up and down arrow keys cycles through previous commands that started with those letters.

Question 6:

`clc` clears the Command Window and moves the prompt to the top of window. The variables are still in the workspace.

Question 7:

The 'dot' in `.*` and `.^` make the operations elementwise.

Question 8:

```
z =
     2
     3
     2
     3
```

Question 9:

```
>> A = [1 2; 3 4];
>> D = [A A; A A]
```

Question 10:

(a) `zeros(a, b)` A matrix of all zeros with `a` rows and `b` columns

(b) `ones(a, b)` A matrix of all ones with `a` rows and `b` columns

(c) `rand(a, b)` A matrix with `a` rows and `b` columns of uniformly distributed random numbers between 0 and 1. Repeated calls give different values.

(d) `randi([imin, imax], a, b)` A matrix with `a` rows and `b` columns of uniformly distributed random integers between `imin` and `imax`.

(e) `eye(n)` The identity matrix of size `n` (all zeros except ones on the diagonal)

Question 11:

(a) `B(2, 3)`

(b) `B(2, 3) = 50`

(c) Type `B =` and then hit the up arrow until you reach the command you first typed.

(d) `B(:, 3)`

(e) `B(:, 3) = 0`

(f) `B(3, :)`

(g) `B(:,2:3)`

(h) `B(1:2,:) = 42`

(i) `x = B(:,1)`

(j) `B(:, 1) = [4; 5; 6]`

(k) `end` indexes to the final element.

Question 12:

(a) `x = 1:3:10`

(b) `x = linspace(1, 10, 4)`

Question 13:

(a) `reshape(A, 5, 6)` turned the array `A` into a matrix with 5 rows and 6 columns. It fills the new matrix column by column.

(b) `B'` took the transpose of matrix `B` (rows become columns and columns become rows).

(c) `size(C)` returned the size of the matrix `C`, number of rows and number of columns.

(d) `size(C, 1)` returned the number of rows in `C`.

(e) `size(C, 2)` returned the number of columns in `C`.

(f) `length()` returns the length of the longest dimension.

Question 14:

(a) `sum(v) = 21` (adds all elements).

(b) `min(v) = 1` (smallest element).

(c) `max(v) = 8` (largest element).

(d) `mean(v) = 4.2000` (the average).

(e) `prod(v) = 480` (multiplies all elements).

(f) `sort(v) = 1 3 4 5 8` (elements in ascending order).

Question 15:

(a) `sum(M)` (returns `9 10 10`, the column sums).

(b) `mean(M(:))` (returns `4.8333`).

(c) `min(M(:))` (returns `1`).

Question 16:

(a) `0` (false; `x` is equal to 5, so "not equal" is false).

(b) `1` (true).

(c) This does not compare anything, it reassigns `x` to 5 and displays `x = 5`. A single `=` is assignment.

Question 17:

(a) `0 0 1 0 0`

(b) `1 0 1 1 1`

Question 18:

(a) `0` (false; `8` is not `< 8`, so not every element qualifies).

(b) `0` (false; no element equals 4).

(c) `all(mod(v, 2) == 0)` (returns `0`, since `v` has odd elements).

Question 19:

(a) `w(w < 0)` (returns `-1 -3 -8`).

(b) `sum(w < 0)` (returns `3`; the logical `1`s are added up).

(c) `w(w < 0) = 0`

Question 20:

(a) `length('Fizz') = 4`

(b) `length("Fizz") = 1`

(c) `strlength("Fizz") = 4`

Question 21:

(a) `length(chars) = 8`

(b) `chars(2) = 'i'`

(c) `length(words) = 2`

(d) `words(2) = "Buzz"`

Question 22:

(a) `sprintf("pi: %.4e", pi)`

(b) `sprintf("pi = %.2f", pi)`

Question 23:

```
>> xlabel("\alpha")
>> ylabel("\alpha^2 + c")
>> title("c = " + c)
```

Question 24:

```
>> triangle_area(-2, 5)
Error using triangle_area
base and height must be non-negative
```

The two lines beginning `if` are an 'if statement'. If the condition is false, the code up to the matching `end` is skipped; if it is true, `error` stops the function immediately and prints the message.

Question 25:

Because you wrote a help comment, `help` works on your function just like it does on built-in ones (recall Q4):

```
>> help triangle_area
  triangle_area  Area of a triangle from its base and height.
    A = triangle_area(b, h) returns the area 0.5*b*h.
```

Question 26:

(a) No. A function has its own private workspace and can only see the values passed in as its inputs (here `base` and `height`), so it never sees `a`. This is the opposite of a script.

(b) Pass it in as an input, e.g. add `a` to the argument list:

`function area = triangle_area(base, height, a)`.

Question 27:

`celsius_to_fahrenheit.m`

```
function f = celsius_to_fahrenheit(c)
% celsius_to_fahrenheit  Convert Celsius to Fahrenheit.
    f = 9/5 * c + 32;
end
```

Question 28:

```
>> F = @(C) 9 * C / 5 + 32;
>> F(100)
ans =
   212

>> F(0)
```

```
ans =
    32
```

Question 29:

Factorial

```
result = 1;
for n = 1:10
    result = result * n;
end
disp(result)        % 3628800
```

Question 30:

Bacteria population

```
population = 1;
hours = 0;
while population <= 1000
    population = population * 2;
    hours = hours + 1;
end
disp(hours)         % 10  (the population reaches 1024 after 10 hours)
```

Question 31:

`wow.m`

```
clear all

% 10 randomly chosen integers between 1 and 5
A = randi(5, 1, 10) % leave off semicolon to display

% count the number of integers equal to 3
cnt3 = sum(A == 3);

% if three or more integers equaled 3, display "wow"
if cnt3 >= 3
  disp("wow!")
end
```

Question 32:

`if_grade.m`

```
function grade = if_grade(score)
% Convert a score out of 100 to a grade

  if score >= 85
      grade = 7;
  elseif score >= 75
      grade = 6;
  elseif score >= 65
      grade = 5;
  elseif score >= 50
      grade = 4;
  else
      grade = 3;
  end

end
```

Question 33:

Row max

```
clear all

A = [3 9 2; 7 1 8; 4 4 6];

% pre-allocate storage array
row_max = zeros(size(A, 1), 1);

for row = 1:size(A, 1)

    % assume first element is largest
    best = A(row, 1);

    % walk along row
    for col = 2:size(A, 2)

        % if another element is found that's larger, update best
        if A(row, col) > best
            best = A(row, col);
        end
```

```
    end

    % store best
    row_max(row) = best;
end
```

4.2 Coding exercises for all students

Question 34:

FizzBuzz

Method 1: character arrays

```
% for each number i from 1 to 30
for i = 1:30

  if mod(i,3) == 0
    % if i is divisible by 3, start with Fizz
    characters = 'Fizz';
  else
    % otherwise, start with an empty character array
    characters = '';
  end

  % if i is divisible by 5, concatenate 'Buzz' onto characters
  if mod(i,5) == 0
    % can concatenate onto an empty char array or 'Fizz'
    characters = [characters, 'Buzz'];
  end

  % print depending on what's in my characters vector
  if length(characters) == 0
    % if there are no characters, display the number
    disp(i)
  else
    % otherwise, display the characters
    disp(characters)
  end

end
```

Method 2: strings

```
% for each number i from 1 to 30
```

```
for i = 1:30

  if mod(i,3) == 0
    % if i is divisible by 3, start with string Fizz
    word = "Fizz";
  else
    % otherwise, start with empty string
    word = "";
  end

  % if i is divisible by 5, combine previous word with Buzz
  if mod(i,5) == 0
    word = word + "Buzz";
  end

  % print depending on what's in my string
  if strlength(word) == 0
    % if there are no characters, display the number
    disp(i)
  else
    % otherwise, display the string
    disp(word)
  end

end
```

Question 35:

Bank history

```
function history = bank_history(transn, balance)
% Returns a history of bank balances given a list of transactions
% and initial balance. The balance is subject to an overdraft rule,
% that any transaction that reduces the balance below zero is rejected.
%
% Example:
% transn = [100 -30 -90 50 -40];
% balance = 0;
% history = [100    70    70   120    80];
%

  % pre-allocate memory for history array
  history = zeros(1, length(transn));

  % fprintf is like sprintf but prints to the screen
```

```
  fprintf('transaction balance\n')
  fprintf('         -- %7.2f\n', balance)

  for k = 1:length(transn)
      if balance + transn(k) >= 0
          balance = balance + transn(k); % accept
      end                                % else: silently reject
      history(k) = balance;
      fprintf('%11.2f %7.2f\n', [transn(k), balance])
  end

end
```

Question 36:

Pascal's triangle:

```
clear all
n = 6;
P = zeros(n, n);
for row = 1:n
    for col = 1:row
        if col == 1 || col == row
            % edge
            P(row, col) = 1;
        else
            % above-left + above
            P(row, col) = P(row-1, col-1) + P(row-1, col);
        end
    end
end
```

Question 37:

Two polynomials

```
clear all

alpha = linspace(-2, 2, 100);   % evenly-spaced x values
y1 = alpha.^2;                  % note the dot: element-wise power
y2 = alpha.^3;

plot(alpha, y1)
hold on                         % keep the first curve while adding the second
```

```
plot(alpha, y2)
hold off

legend("\alpha^2", "\alpha^3")  % TeX markup: Greek letter + superscript
xlabel("\alpha")
ylabel("y = \alpha^n")
title("Two polynomials")

exportgraphics(gca, "polynomials.pdf")   % save the figure to a PDF
```

The strings such as `"\alpha^2"` are not printed literally: MATLAB reads the TeX markup, so `\alpha` becomes α and `^2` becomes a superscript. `hold on` is what lets both curves share the one set of axes.

Question 38:

(a) The product of the numbers `1:10` is exactly what `prod` does (Q. 14):

```
result = prod(1:10);        % 3628800
```

(b) `temps > threshold` is a logical matrix, with a `1` wherever a day was hot. Summing along each row then counts the hot days for each city. The second argument, `2`, tells `sum` to work along the rows (dimension 2) instead of down the columns:

```
nbr_hot_days = sum(temps > threshold, 2);   % [2; 0; 4]
```

(c) `max` accepts the same kind of dimension argument. The empty `[]` in the middle is a placeholder that tells MATLAB the `2` is a dimension to work along, not a second array to compare against:

```
row_max = max(A, [], 2);    % [9; 8; 6]
```