MAT5OPT MATLAB Notes

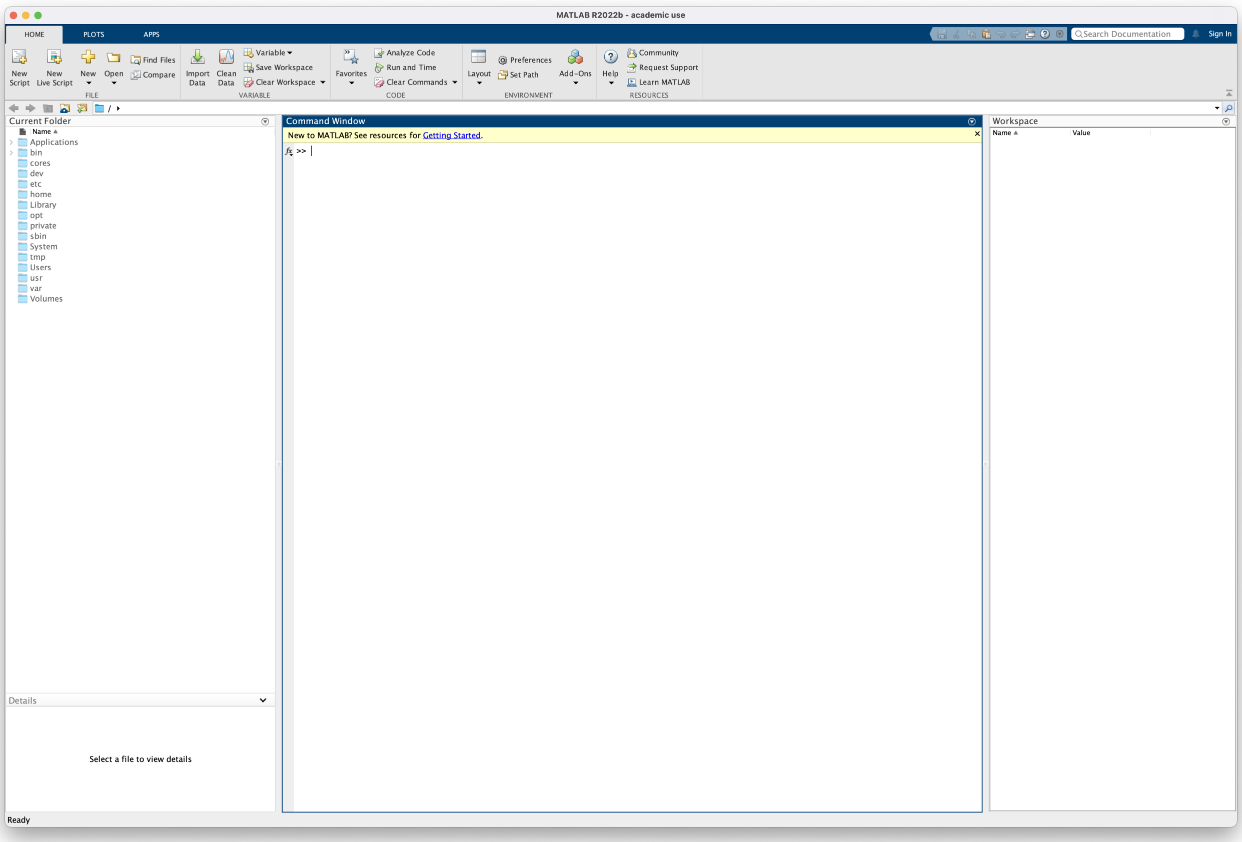
Contents

Chapter 1

Introduction to MATLAB

1.1 Getting started

Learning MATLAB is not the main objective of this subject, and hence we will not explore its full functionality. These notes we will introduce to you the basics, and extra features will be picked up along the way. Let's start at the beginning. When you open MATLAB, you are faced with a window like this:



New to MATLAB? See resources for Getting Started.

```
>>
```

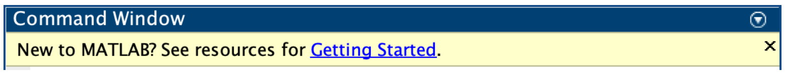
Select a file to view details

Ready

Figure 1.1: The MATLAB window as it appears on Mac OS 13.3.1. It may appear slightly different depending on your operating system, but the contents of the window should be largely the same.

Inside this window you see three tabs (Home, Plots and Apps), a lot of buttons you can click on, and three smaller windows, namely: (from left to right) the current folder (where MATLAB looks for files), the command window (where you enter code) and the workspace (which keeps track of all the variables that you define).

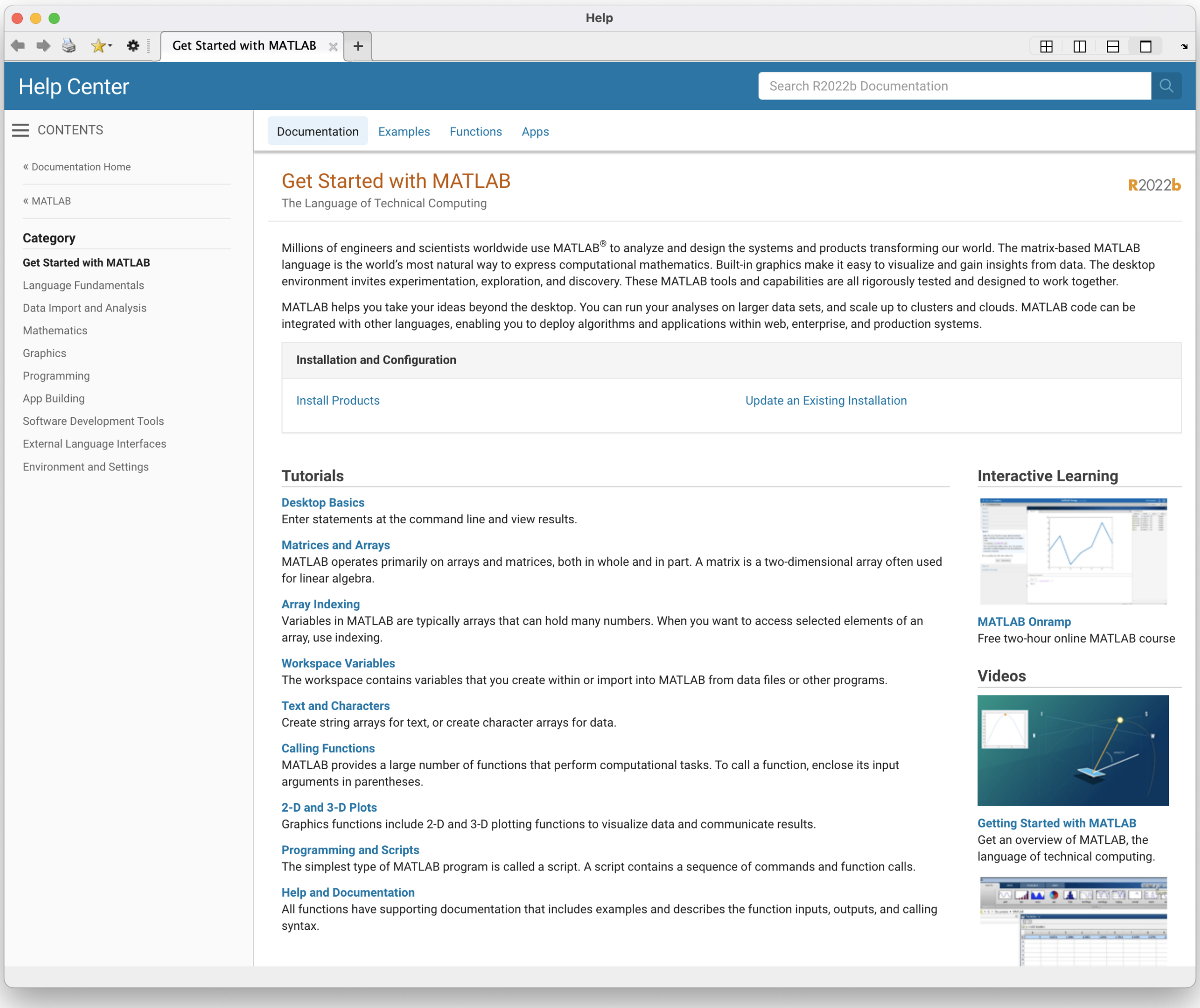
In the Command Window, in yellow you'll find the question *New to MATLAB?*



Command Window

New to MATLAB? See resources for Getting Started.

Clicking on the Getting Started link will bring up the MATLAB documentation.



Help

Get Started with MATLAB

Help Center

Search R2022b Documentation

CONTENTS

« Documentation Home

« MATLAB

Category

Get Started with MATLAB

Language Fundamentals

Data Import and Analysis

Mathematics

Graphics

Programming

App Building

Software Development Tools

External Language Interfaces

Environment and Settings

Documentation Examples Functions Apps

Get Started with MATLAB

R2022b

The Language of Technical Computing

Millions of engineers and scientists worldwide use MATLAB® to analyze and design the systems and products transforming our world. The matrix-based MATLAB language is the world's most natural way to express computational mathematics. Built-in graphics make it easy to visualize and gain insights from data. The desktop environment invites experimentation, exploration, and discovery. These MATLAB tools and capabilities are all rigorously tested and designed to work together.

MATLAB helps you take your ideas beyond the desktop. You can run your analyses on larger data sets, and scale up to clusters and clouds. MATLAB code can be integrated with other languages, enabling you to deploy algorithms and applications within web, enterprise, and production systems.

Installation and Configuration

Install Products Update an Existing Installation

Tutorials

Desktop Basics
Enter statements at the command line and view results.

Matrices and Arrays
MATLAB operates primarily on arrays and matrices, both in whole and in part. A matrix is a two-dimensional array often used for linear algebra.

Array Indexing
Variables in MATLAB are typically arrays that can hold many numbers. When you want to access selected elements of an array, use indexing.

Workspace Variables
The workspace contains variables that you create within or import into MATLAB from data files or other programs.

Text and Characters
Create string arrays for text, or create character arrays for data.

Calling Functions
MATLAB provides a large number of functions that perform computational tasks. To call a function, enclose its input arguments in parentheses.

2-D and 3-D Plots
Graphics functions include 2-D and 3-D plotting functions to visualize data and communicate results.

Programming and Scripts
The simplest type of MATLAB program is called a script. A script contains a sequence of commands and function calls.

Help and Documentation
All functions have supporting documentation that includes examples and describes the function inputs, outputs, and calling syntax.

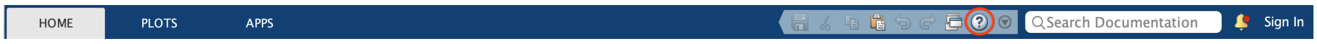
Interactive Learning

MATLAB Onramp
Free two-hour online MATLAB course

Videos

Getting Started with MATLAB
Get an overview of MATLAB, the language of technical computing.

You can also access the documentation online via `https://www.mathworks.com/help/index.html`, or in MATLAB by clicking on the circled question mark in the menu bar, or by clicking the Help menu and selecting Documentation.



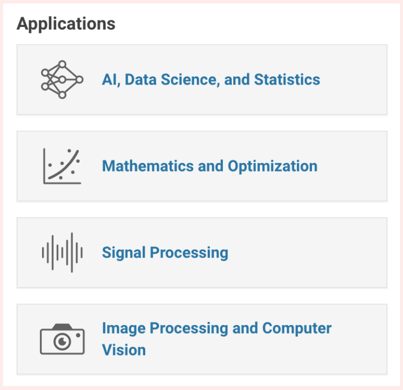
Each of the blue topics can be clicked and will lead you to a short tutorial on that topic. On the right you'll find some useful videos. On the left, you'll find categories with more detailed documentation on MATLAB's features.

You may like to take some time to familiarise yourself with some MATLAB topics. In particular, if you have done any programming before, you may want to acquaint yourself with MATLAB's syntax and structure.

The documentation will always be a useful reference, so be sure to refer to it when necessary. Meanwhile, in this text, the basics will be covered, but most of the material is also covered in the above mentioned tutorials and videos. Be aware that this only the tip of the iceberg; MATLAB can do much more!

Activity: learn about optimisation in MATLAB

In the menu on the left, click the *Documentation Home* button and find the *Applications* section.


Applications
AI, Data Science, and Statistics
Mathematics and Optimization
Signal Processing
Image Processing and Computer Vision

Explore the ways MATLAB can be used for optimisation and other areas of data science.

1.2 The command window

We will start by typing commands in the command window. In the command window, you will see a sign that looks like this:

```
>>
```

You can then write commands next to it, for example:

```
>> 3.5 + 2
```

The output will show

```
ans =

    5.5000
```

In this case, `ans` is a *variable*. The variable `ans` is regularly updated to contain the result of the most recent computation performed in the command window. Writing `ans` in the command window will show

```
>> ans

ans =

    5.5000
```

The next thing you need to know is how to get MATLAB's help: press F1 or click on ?. This will bring up a screen with possible commands.

Otherwise, if there is a function you want to know more about, you can type something like

```
>> help sqrt
```

or

```
>> doc sqrt
```

It is an important skill to know how to access and read the MATLAB documentation for new or unfamiliar functions.

1.3 Operations, functions and variables

MATLAB understands the usual arithmetic operations. One just needs to know what the syntax is for each operation.

Operation	Syntax
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Powers	`^`

For example, to encode $3.5 + 2 \times \frac{3}{4-1}$, you would write `3.5 + 2*3/(4-1)`. Note the careful bracketing.

Since `ans` is a variable, it can also partake in operations. If you now write

```
>> (ans + 3)/5 + 2.5*3.5
```

then you will find the result

```
ans =

    10.4500
```

Note that `ans` now contains the value `10.4500`.

There are a number of ways to input a number. MATLAB accepts many different forms of input for numbers. All of the following are equivalent

```
>> 100
>> 100.0
>> 1e2
```

where the latter is to be considered as scientific notation, as is on the calculator, meaning 1×10^2.

By default, numbers are stored as *64-bit double-precision floating point* values. We will not dwell on what exactly that means, but will note some important properties:

- The smallest number that can be represented is -1.79769×10^{308}, which can be written as `-1.79769e308`.
- The largest number that can be represented is 1.79769×10^{308}, which can be written as `1.79769e308`.
- Computations that result in a number smaller or larger than these values will result in `-Inf` or `Inf`, respectively. For example, `2 * 1.79769e308` will result in `Inf`.
- Equality comparisons can be risky! See the box below.

!

Floating point numbers are not generally suitable for situations where exactness is a concern. We haven't yet described the relational operators, but as a preview, using `==` between two numbers will return `1` if the numbers are equal and `0` otherwise. For example,

```
>> 3.5 + 2 == 5.5
```

will result in an output of `1`. You might expect the next piece of code to output `1` as well since, arithmetically, $3 \times 0.2 = 0.6$ is true:

```
>> 3*0.2 == 0.6
```

But it will output `0` (false)!

The reason for this is similar to the fact that we can't write $\frac{1}{3} \approx 0.3333$ as a finite decimal expansion in base 10. In binary, the number 0.2 requires infinitely many digits to be represented exactly.

Instead, the underlying representation of 0.2 in a double-precision floating point number amounts to the calculation

$$\left(1 + \frac{2702159776422298}{2^{52}}\right) \times 2^{-3}$$

This is correct up until the 16th decimal place, which is just enough to cause a rounding error. Consequently, the internal representation of 3×0.2 is the number 0.6000000000000001, even though it does not display that way.

MATLAB also has built in complex numbers. A complex number $a + bi$ is written as `a+bi` in MATLAB. (the multiplication symbol `*` does not need to be used explicitly for the constant i). But for operations between complex numbers, explicit multiplication is necessary:

```
>> i*(1+5i)

ans =

  -5.0000 + 1.0000i
```

Another built in constant is π, obtained by writing `pi`.

```
>> pi

ans =

      3.1416
```

!

Want to display more decimal places? Type this to set MATLAB to display numbers in long format:

```
>> format long
```

Alternatively, change it by clicking Preferences in the menubar, finding *Command Window* in the choices on the left, and then set *Numeric format* to *long*. Return to the default behaviour by typing `format short`.

You can also remove the lines between output by writing

```
>> format compact
```

Alternatively, change it by visiting the Command Window preferences as above and change *Line spacing* from *loose* to *compact*. Return to the default behaviour by typing `format loose`.

The long, compact format will be used through the remainder of these notes.

You can store values by assigning a number to a variable. Afterwards, you can apply the usual operations to that variable. A valid variable name must start with a letter (upper or lower case), and can be followed by letters, digits or underscores. Define a variable by using a single equals sign.

```
>> x = 3*pi
x =
    9.424777960769379
>> A_new_variable = (x - 1)/2
A_new_variable =
    4.212388980384690
```

Defining another variable will *not* modify the contents of the `ans` variable.

MATLAB also understands trigonometric functions, logarithms, exponentials and many other standard functions. To the left of >> you'll find a little symbol fx. If you click it, a menu opens which contains more functions than you can dream of. Functions can be evaluated by enclosing the arguments of the function in brackets.

```
>> sin(3)
ans =
     0.141120008059867
>> x = pi/4
x =
    0.785398163397448
>> sin(x)
ans =
     0.707106781186547
```

!

MATLAB will allow you to assign variables over function names, such as

```
>> sin = 3
sin =
    3
>> sin(pi)
Array indices must be positive integers or logical values.
```

Be cautious about the variable names you use.

If you accidentally overwrite a built in function, you can use the `clear` keyword to remove your definition. For the above example, `clear sin` will restore the default functionality of `sin`.

Some useful built-in functions to familiarise yourself with are listed in the table below.

Name	Function	Description
Exponential	`exp`	`exp(x)` calculates e^x
Square root	`sqrt`	`sqrt(x)` calculates $\sqrt{x}$
Logarithms	`log`, `log2`, `log10`	`log(x)` calculates $\ln(x)$, `log2(x)` calculates $\log_2(x)$ and `log10(x)` calculates $\log_{10}(x)$.
Absolute value	`abs`	`abs(x)` calculates $\|x\|$
Modulus	`mod`	`mod(x,i)` calculates the remainder of x after dividing by i
Trigonometric functions	`sin`, `cos`, `tan`	`sin(x)` calculates $\sin(x)$, `cos(x)` calculates $\cos(x)$, and `tan(x)` calculates $\tan(x)$.

Activity: practice operations and variables

Use MATLAB to perform each of the following.

1. Calculate the value of $|-5+12i|$.
2. Calculate the value of $\cos(3\pi+2)$ and store it in a variable named `x`. Calculate the value of $e^{2\pi}$ and store it in a variable named `y`. Finally, display the value of $x+2y$.
3. Calculate the value of $\sqrt{5}$ and store it in a variable named `a`. Calculate the value of e^a and store it in a variable named `b`. Calculate the value of $\log_{10}(b)$ and store it in a variable named `c`. Finally, display the value of `c`.

1.4 Script files

Once an operation becomes too large, it is a hassle to type in all the operations over and over again. We may use files to store sequences of commands for MATLAB to use and reuse, and most importantly, for us to easily modify and run. There are two types of file you will be required to use in MAT5OPT:

- Scripts: script files contain a sequence of operations to be performed in order.
- Functions: function files define a new operation that can be used in other scripts.

To define a script file, click on *New Script* in the menubar. MATLAB has an editor which will allow you to open, save and run files, and also includes various debugging tools. Using this editor is recommended in this subject.

Activity: create a script

1. Open a new script as described above.
2. Observe that after opening the new script, some new tabs appear above the menubar.
3. Copy and paste the following code into the script editor:

```
tally = 0;
count = 1;
while count <= 100
    tally = tally + count;
    count = count +1;
end
tally
```

   Note that many lines include semicolons. This has been deliberately omitted from the last line.
4. In the menubar, click *Run*. This option appears only if the Editor tab is selected. MATLAB will prompt you to save the file first. After saving the script file, it will run in the command window.
5. To run it again, type the name of the script in the command window. Any script that is in the *Current Folder* shown on the left hand side of the main window can be run in this way.

What happens if you add a semicolon to the end of the last line? What happens if you remove the semicolons from each line? Try it and see.

You can also create a script file from the command history window. The command history window can be opened by pressing the up arrow on the keyboard whilst the cursor is active in the command window.

Selecting a set of commands, right-clicking and selecting *Create Script* will create a script file that contains the chosen commands, and can then be used in the same manner as earlier. (On Mac OS, you may need to hold control while clicking rather than right clicking.)

Another type of file you can create allows you to save the contents of one or more variables. The activity below will guide you through this.

Activity: save and restore

1. Make sure you have completed the previous activity.
2. Check the contents of the workspace window on the right hand side of the screen. Observe the values of the variables `count` and `tally`.
3. Hold down the ctrl key and click on the `count` and `tally` variables in the workspace window to select them. (On Mac OS, you will need to hold down the command key instead).
4. Right click on the selection and choose *Save As...*. (On Mac OS, you may need to ctrl+click instead).
5. Save the file to the current directory with the name `answers.mat`.
6. Type `clear tally` in the command window or select the `tally` variable in the workspace and press the Delete key on your keyboard. Then type `count = -10` in the command window. Observe the changes in the workspace window.
7. In the Current Folder on the left side of the window, double click on `answers.mat`. Alternatively, type `load('answers.mat')` in the command window.

You can also use the `save` command to save variables:

- `save variablename` will save that variable to a file. For example, `save tally`.
- `save [variable1, variable2, ...]` will save a list of variables to a file. For example, writing `save [count,tally]` will save both of those variables to the file.
- `save mydata` will save all variables into a file called `mydata.mat`.

1.5 Arrays

In MATLAB, an array is a multidimensional list of data elements. The most common types we will use are one-dimensional and two-dimensional arrays.

- A one-dimensional array is best thought of as a list.
- A two-dimensional array is best thought of as a matrix.

We will look at matrices (two-dimensional arrays) in the next chapter of this text.

MATLAB has a somewhat different way of thinking of variables. It will assume all variables are arrays or matrices. This includes the case of a single number: it is treated as a 1×1 array, i.e., a list with one element.

To define a list of numbers, simply surround the numbers by square brackets. You can separate the individual entries by spaces or commas:

```
>> A = [1 2 4 8 12]
A =
        1       2       4       8      12

    >> A = [1,2,4,8,12]
    A =
        1       2       4       8      12
```

Individual elements are accessed by using round brackets. Note that MATLAB uses 1-based indexing, so that the first element of an array is referred to using the number 1. The last entry in an array can be obtained using the keyword `end`.

```
    >> A(1)
    ans =
        1
    >> A(3)
    ans =
        4
    >> A(end)
    ans =
        12
```

Since scalars are also treated as arrays, it makes sense to ask for the first element of a variable when the variable is just a single number. E.g.:

```
    >> x = 15;
    >> x(1)
    ans =
       15
```

More generally, the index used when accessing an array can be any list of numbers. For example:

```
    >> A = [1 2 4 8 12];
    >> A([end, 1, 1, 3, end])
    ans =
       12       1       1       4      12
```

Sequences of numbers can be written using colons. For example, the array `[2 3 4 5 6 7]` can be conveniently created by typing

```
    >> 2:7
```

and, for an arithmetic sequence with common difference 3,

```
    >> 2:3:15
    ans =
        2       5       8      11      14
```

This permits easy access to a range of entries in an array.

```
    >> A = [4 14 3 13 9 15 2 7 2 15];
    >> A(2:7) % obtain entries 2 to 7
    ans =
       14       3      13       9      15       2
    >> A(1:2:end) % obtain every second element up to the end
    ans =
        4       3       9       2       2
```

Now consider the following piece of code:

```
>> x = [1 2 3];
>> y = [9 8 7];
>> z = [x y];
```

What do you think is contained in the variable `z`? A natural guess would be that `z` contains `[1 2 3 [9 8 7]]`, i.e., the fourth element of `z` is an array with three elements. However, this is not the way MATLAB behaves. In MATLAB, an array will not contain other arrays; instead, `[x y]` has the effect of concatenating the two arrays:

```
>> z
z =
     1     2     3     9     8     7
```

This is useful behaviour, and can be used easily combine multiple arrays. For instance, an array with the first 5 positive even numbers followed by the first 5 positive odd numbers:

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

There are also some useful built-in functions that act on arrays, summarised in the table below.

Function	Description
`length`	returns the number of elements in an array
`sum`	sums together the elements of the array
`min`	determines the smallest element of the array
`max`	determines the largest element of the array
`sort`	sorts the array from smallest to largest

Activity: practice array operations

The function `randi` can be used to construct an array of random integers:

```
>> randi(20, 1, 10)
ans =
     1    16    17    18     2     8     6    17     9    19
```

The first argument (20) is the maximum number that can be generated; the second argument (1) is because we want a 1-dimensional array; the third argument (10) is the number of elements in the array.

A single random integer can be generated using, e.g., `randi(20)`. In this case, it means a random positive integer less than or equal to 20.

Nesting these means we can create an array of integers with a random size:

```
>> x1 = randi(20, 1, randi(20))
```

Use this to generate two random arrays with a random number of elements (say `x1` and `x2`) and then write code to perform each of the following:

1. Determine the number of elements of each array.
2. Construct the array obtained by concatenating `x1` and `x2`.
3. Construct the array obtained by concatenating `x1` and `x2` and sort them from smallest to largest.
4. Extract every second element in `x1` (starting at index 2).
5. Extract every third element in `x2` (starting at index 1).

6. Construct an array which first has the odd index elements of `x1`, then the even index elements of `x2`, then the even index elements of `x1`, then the odd index elements of `x2`, in that order.
7. Construct an array containing 5 random elements from `x1` and 3 random elements from `x2` (repeats are allowed).

1.6 Writing functions

We have already seen some basic use of MATLAB's built-in functions, but more important is the ability to define your own. In MATLAB, functions are defined using a function file.

To define a function file, in the menubar, click on *New* and then select *Function*. MATLAB will automatically create a function file with the following (or something similar):

```
function [outputArg1,outputArg2] = untitled(inputArg1,inputArg2)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

which can then be edited to accept the required input and output.

We will start by breaking down the syntax of a MATLAB function definition. Generally, the first line will look something like

```
function [y1,y2,...,yN] = functionName(x1,x2,...,xM)
```

This works as follows:

- The word `function` tells MATLAB that you are defining a function.
- The variables in square brackets `[y1,y2,...,yN]` are the output variables. There can be any number of output variables, including none.
- The equals sign is a necessary part of the syntax to define a function with output variables.
- The word `functionName` is the name of the function, which can be modified to any name you like. **However, when defining a function file, the name of the function must be exactly the same as the name of the file.**
- The variables in round brackets `(x1,x2,...,xN)` are the input variables. There can be any number of input variables, including none.

Note that, if you want to define a function that has no output variables, the syntax requires omitting the equals sign:

```
function noOutputFunction(x1, x2)
% code that does not produce output variables
end
```

! It bears repeating that, when defining a function file, the name of the function must be the same as the name of the file!

In the default function, the lines that begin with percent signs (`%`) are *comments*, which are lines that are not parsed by MATLAB and are instead included for human readability. Next, in the *function body*, the default function assigns some values to the output variables. These output variables will be automatically returned by the function.

Finally, the word `end` indicates the end of the function definition.

Here is an example that simply takes two inputs, `x` and `y`, and adds them together.

```
function [z] = Addition(x,y)
z = x+y;
end
```

We save this file as `Addition.m`, because the name of the function is `Addition`.

Observe the structure of the code:

- The input variables are `x` and `y`.
- There is just one output variable, namely `z`.
- The line of code `z = x+y;` calculates `x+y` and stores it in the variable `z`.

Output variables must be assigned in the body of the function (as is done by the line `z = x+y;`) and are then automatically given as outputs when the function runs.

The function can then be used like any other built-in function. For example:

```
>> Sum_of_3_and_4 = Addition(3,4)
Sum_of_3_and_4 =
     7
```

It is worth noting that when the function has a single output (as the example above does), the square brackets are optional. So we can equivalently define the function as follows:

```
function z = Addition(x,y)
z = x+y;
end
```

For simple functions that have exactly one output and a single executable statement, one can instead use anonymous functions. Another equivalent way to define the `Addition` function is:

```
Addition = @(x,y) x+y;
```

The syntax is as follows:

- The `@` symbol indicates the start of an anonymous function.
- The variables in parentheses are the input variables.
- The code that follows is evaluated and output by the function.
- This function is stored in a variable called `Addition`.

When writing anonymous functions in this manner, all variables not included in the input variables are evaluated at the moment the function is defined. Consider the following code:

```
>> a = 1.5;
>> b = 0.4;
>> c = 2.9;
>> y = @(x) a*x^2 + b*x + c;
>> y(1)
ans =
     4.800000000000000
>> a = 10000;
>> y(1)
ans =
     4.800000000000000
```

Although the variable `a` was changed between evaluations, the function is not aware of this.

A more robust version of this might be as follows, which defines the function $y(x) = ax^2 + bx + c$ for arbitrary a, b and c.

```
>> quadratic = @(x,a,b,c) a*x^2 + b*x + c;
>> quadratic(1, 1.5, 0.4, 2.9);
ans =
     4.800000000000000
>> quadratic(1, 10000, 0.4, 2.9)
ans =
     1.000330000000000e+04
```

Activity: practice writing functions

The volume V and surface area S of a cylinder of radius r and height h are given by the formulas

$$V = \pi r^2 h$$
$$S = 2\pi r h + 2\pi r^2$$

Write functions `cylinderVolume` and `cylinderSurfaceArea` that take r and h as inputs and output the volume and surface area, respectively.

1.7 Control flow

Control flow statements are used to define the order in which statements are processed. For example, branching based on a given condition, or executing the same statement repeatedly until another condition is met.

Control flow is often based on boolean expressions. A *boolean expression* is an expression that evaluates to either true or false. In MATLAB, true and false are represented by `1` and `0` respectively. In fact, any non-zero number will be interpreted by MATLAB as true.

The following list of the most common conditions, or *relational operators*, may be used to form boolean expressions. They are interpreted in the usual mathematical manner, resulting in `1` if the expression is true and `0` otherwise.

Boolean operator	Syntax
Less than	`<`
Greater than	`>`
Less than or equal	`<=`
Greater than or equal	`>=`
Equal	`==`
Not equal	`~=`

For example:

```
>> 4 >= 4
ans =
  logical
   1
>> 4 > 4
```

```
ans =
  logical
   0
```

MATLAB will display the word `logical` to indicate that the output should be interpreted as true or false.

In addition to the above, we may also combine boolean expressions into new ones using the following *boolean operators*:

Boolean operator	Syntax
And	`&`
Or	`|`
Not	`~`

They are defined as follows:

- `x & y` is true only if *both* `x` and `y` are true, and is false otherwise.
- `x | y` is true only if *at least one of* `x` and `y` are true, and is false otherwise.
- `~x` is true only if `x` is false, and is false otherwise.

For example:

```
>> ~(4 > 3)
ans =
  logical
   0
>> 4 > 3 & 4 > 5
ans =
  logical
   0
>> 4 > 3 | 4 > 5
ans =
  logical
   0
```

A convenient feature of the boolean operators is that they operate elementwise on arrays. Suppose you wanted to extract the elements of an array which are greater than 10.

```
>> A = [4 14 3 13 9 15 2 7 2 15];
>> A > 10
ans =
  1×10 logical array
   0   1   0   1   0   1   0   0   0   1
```

Observe that the previous code results in a 1 for each element that is strictly greater than 10, and a 0 for each element less than or equal to 10. This can then be combined to extract those elements directly:

```
>> greater_than_10 = A > 10;
>> A(greater_than_10)
ans =
    14    13    15    15
```

Or more succinctly:

```
>> A(A > 10)
ans =
    14    13    15    15
```

The find function can be used to extract the indices instead of the elements.

```
>> find(A>10)
ans =
     2     4     6    10
```

This tells us that the elements larger than 10 are at indices 2, 4, 6 and 10.

Activity: practice boolean operators

Start by revisiting *Activity: practice array operations* from Section 1.5 if needed. Then generate a random array called A with a random size and then write code to complete the following:

1. Find the elements in A which are strictly between 3 and 10.
2. Find the index of the elements in A which are strictly between 3 and 10.
3. Find the elements in A which are less than 2 or greater than or equal to 5.
4. Find the index of the elements in A which are less than 2 or greater than or equal to 5.
5. The mean of an array A can be computed by writing mean(A). Construct two arrays, one which contains all elements less than or equal to the mean of A, and the other which contains all elements which are greater than the mean of A.
6. Construct two arrays, one which contains all the even elements of A and one which contains all the odd elements of A. Note that to decide if a number is even, we have to compute its remainder after dividing by 2, which can be done using the mod function.

Boolean expressions are often combined with *if statements*. An if statement allows you to selectively apply an operation depending on whether a boolean expression evaluates to true or false. They can also be evaluated in the command window, but note that the enter button will no longer immediately evaluate what you have asked for. It will just continue to ask for directions until you type in end.

```
>> if 4 > 2
x = 0
else
x = 1
end
x =
     0
```

Now what if we wanted to do the same thing 100 times? One could write 100 lines of code that does this, however, this is not the most efficient way of doing this. The best way to do this is by utilizing a loop. The two loops we use are *for* loops and *while* loops.

A for loop evaluates statements for a given list of values. The syntax is as follows:

```
for variable = expression
    % insert statements as required
end
```

The expression should evaluate to an array (usually a list of numbers). Recall that i:j constructs a list of numbers between i to j inclusive. Thus, a common way to repeat the same thing n times is to use a for loop of the following form:

```
for i = 1:N
    % do something
end
```

For instance, to add together the first 100 numbers:

```
tally = 0;
for count = 1:100
   tally = tally + count;
end
tally
```

A similar structure is the while loop. A while loop repeats an operation so long as a given boolean expression evaluates to true. As soon as it evaluates to false, the while loop stops. Here is a piece of code that does exactly the same thing using a while loop instead:

```
tally = 0;
count = 1;
while count <= 100
   tally = tally + count;
   count = count +1;
end
tally
```

Note, we have ended each line within the loops with semicolons (`;`), which means to suppress the output for that line. It is good to suppress output in loops as the output can often slow down operations by several orders of magnitude.

Activity: practice if statements

Although the absolute value is a built-in function, for this activity you will define your own version of it. The formal definition of $|x|$ is:

$$|x| = \begin{cases} x, & \text{if } x \geqslant 0, \\ -x, & \text{otherwise.} \end{cases}$$

Write a function `myabs` which takes a single input `x` and outputs the absolute value using the given definition.

Activity: practice looping structures

Write a function `countEvens` that takes a single array `A` as input and returns the number of even numbers in that array. Write three different versions:

1. One that uses a for loop.
2. One that uses a while loop.
3. One that uses no looping structures, and instead uses built in functions that act on arrays.

Remember that, to decide if a number is even, we have to compute its remainder after dividing by 2 by using the `mod` function.

1.8 More on writing functions

Let's now combine our knowledge from the previous parts to write functions that perform more involved calculations. We will start by rewriting the `tally` computation from earlier into a function that allows arbitrary choice of the first and last numbers to add. Start by clicking on *New* in the menubar, and then select *Function.* The function will have one output variable, `tally` and two input variables, `first` and `last`. So our function should look something like this:

```
function [tally] = sumOfNumbers(first,last)
% Calculates the sum of all integers from first to last.
tally = 0;
for i = first:last
    tally = tally + i;
end
```

Remember that you need to save the function file with the same name given to the function.

Actually, there is a simpler way to define this function by using anonymous functions and the built-in `sum` function:

```
sumOfNumbers = @(first, last) sum(first:last);
```

Let's change things up a bit: now, if the number is even, we will add it as usual. But if the number is odd, we will square it before adding it to the tally. This is where we would use an if statement.

Note that to decide if a number is even, we have to compute its remainder after dividing by 2, which can be done using the `mod` function.

```
function [tally] = sumOfNumbers2(first,last)
% Calculates the sum of all integers from first to last,
% but squares it first if it is an odd number.
tally = 0;
for i = first:last
    if mod(i,2) == 0
        tally = tally + i;
    else
        tally = tally + i^2;
    end
end
```

Recall that MATLAB functions may have more than one output. For example, you might want to write a function that has a separate tally: one for odd numbers, and one for even numbers.

```
function [oddSum, evenSum] = sumOddsAndEvens(first,last)
% Calculates the sum of all integers from first to last,
% but keeps a separate tally for odd and even numbers.
oddSum = 0;
evenSum = 0;
for i = first:last
    if mod(i,2) == 0
        evenSum = evenSum + i;
    else
        oddSum = oddSum + i;
    end
end
```

However, note that when calling the function only the first output is obtained:

```
>> sumOddsAndEvens(1,100)
ans =
        2500
```

To get all the output, one must give them separate names:

```
>> [odds, evens] = sumOddsAndEvens(1,100)
odds =
        2500
evens =
        2550
```

This how MATLAB suggests to deal with “Function with Multiple Outputs” (see MATLAB documention). However, here is another way of dealing with this.

```
function [sums] = sumOddsAndEvens(first,last)
% Calculates the sum of all integers from first to last,
% but keeps a separate tally for odd and even numbers.
oddSum = 0;
evenSum = 0;
for i = first:last
    if mod(i,2) == 0
        evenSum = evenSum + i;
    else
        oddSum = oddSum + i;
    end
end
sums = [oddSum, evenSum];
end
```

Now we can simply call

```
>> sumOddsAndEvens(1,100)
ans =
        2500        2550
```

Activity: practice writing functions with more than one output

Write a MATLAB function `practiceFunction` which, takes a single array `A` as input, and then returns all of the following:

- the sum of numbers in that list
- the result of adding the even numbers in the list and subtracting the odd numbers in the list
- the number of even numbers in the list
- the number of odd numbers in the list

For example, it should work as follows:

```
>> A = [4 14 3 13 9 15 2 7 2 15];
>> [total, addEvenSubOdd, numEven, numOdd] = practiceFunction(A)
total =
    84
addEvenSubOdd =
   -40
numEven =
     4
numOdd =
     6
```

1.9 Solutions to activities

Activity: learn about optimisation in MATLAB

No solutions needed—simply explore the documentation window to your leisure.

Activity: practice operations and variables

1.
```
>> abs(-5+12i)
ans =
    13
```
2.
```
>> x = cos(3*pi+2);
>> y = exp(2*pi);
>> x + 2*y
ans =
     1.071399457886076e+03
```
3.
```
>> a = sqrt(5);
>> b = exp(a);
>> c = log10(b)
c =
   0.971111983788723
```

Activity: create a script

Following the instructions precisely will complete this activity. Note that the effect of the semicolons is to supress output: ending a line with a semicolon will prevent MATLAB from outputting the result of that line.

Activity: save and restore

Following the instructions precisely will complete this activity.

Activity: practice array operations

The following code can be run in a script file:

```
% Generate the arrays:
x1 = randi(20, 1, randi(20));
x2 = randi(20, 1, randi(20));

% Output the number of elements:
length(x1)
length(x2)

% Concatenate x1 and x2:
[x1 x2]

% Concatenate x1 and x2 and then sort:
sort([x1 x2])

% Every second element of x1 (starting at index 2):
```

```
x1(2:2:length(x1))

% Every third element of x2 (starting at index 1):
x2(1:3:length(x2))

% Part 6:
oddx1 = x1(1:2:length(x1));
evenx1 = x1(2:2:length(x1));
oddx2 = x2(1:2:length(x2));
evenx2 = x2(2:2:length(x2));
[oddx1 evenx2 evenx1 oddx2]

% Part 7:
% We want to choose 5 random numbers, the maximum possible
% being the length of each array.
% Then we obtain the elements with those indices.
% 5 random elements of x1:
r1 = x1(randi(length(x1), 1, 5));
% 3 random elements of x2:
r2 = x2(randi(length(x2), 1, 3));
% Concatenate the arrays:
[r1 r2]
```

String formatting from Chapter 2 can be used to make the output display more context.

Activity: practice writing functions

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

```
function V = cylinderVolume(r, h)
% Calculates the volume of a cylinder of radius r and height h
V = pi * r^2 * h;
end
```

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

```
function V = cylinderSurfaceArea(r, h)
% Calculates the surface area of a cylinder of radius r and height h
S = 2*pi*r*h + 2*pi*r^2;
end
```

These can also be defined more succinctly (in the command window or in a script) by using anonymous functions:

```
>> cylinderVolume = @(r, h) pi * r^2 * h;
>> cylinderSurfaceArea = @(r, h) 2*pi*r*h + 2*pi*r^2;
```

Activity: practice boolean operators

The following code can be run in a script file:

```
A = randi(20, 1, randi(20));

% Part 1:
part1 = A(A > 3 & A < 10);
```

```
% Part 2:
part2 = find(A > 3 & A < 10);

% Part 3:
part3 = A(A < 2 | A >= 5);

% Part 4:
part4 = find(A < 2 | A >= 5);

% Part 5
m = mean(A);
lesser = A(A <= m);
greater = A(A > m);

% Part 6:
evens = A(mod(A,2) == 0);
odds = A(mod(A,2) == 1);
```

Activity: practice if statements

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

```
function result = myabs(x)
% Computes the absolute value of x
if x >= 0
    result = x
else
    result = -x
end
end
```

Activity: practice looping structures

The three versions of the function should be included in a file called `countEvens.m`.

Using a for loop:

```
function count = countEvens(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
```

Using a while loop:

```
function count = countEvens(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
```

Without using loops:

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

The last one can even be written using anonymous functions:

```
countEvens = @(A) sum(mod(A,2) == 0);
```

Activity: practice writing functions with more than one output

One possible solution (which needs to be saved in a file called `practiceFunction.m`) is as follows:

```
function [total, addEvenSubOdd, numEven, numOdd] = practiceFunction(A)
% Calculates the sum of numbers in the array A (total),
% the sum of evens minus sum of odds (addEvenSubOdd),
% the number of even numbers (numEven),
% and the number of odd numbers (numOdd).
total = 0;
addEvenSubOdd = 0;
numEven = 0;
numOdd = 0;
for i = A
    total = total + i;
    if mod(i,2) == 0
        addEvenSubOdd = addEvenSubOdd + i;
        numEven = numEven + 1;
    else
        addEvenSubOdd = addEvenSubOdd - i;
        numOdd = numOdd + 1;
    end
end
```