Chapter 2

Basics

2.1 Expressions

Let's dig in to writing expressions in the Command Window. The following table gives examples of MATLAB syntax for common operations.

Operation	Example	MATLAB
Addition	$5+7$	5 + 7
Subtraction	$42-50$	42 - 50
Multiplication	3×8	3 * 8
Division	$64\div 8$	64 / 8
Exponentiation	3^4	3^4

The constant π is so common and useful that it is built-in. Try typing `pi` in the Command Window and you will see an answer of 3.1416. Note that the value displayed is (confusingly) not the value actually stored behind the scenes. By default, you will not see the full decimal expansion of a number, but a rounded value.

The constant i is also built-in and MATLAB can evaluate complex expressions.

```
>> (3 - 2*i) * (5 + i)
 ans =
   17 - 7i
```

2.1.1 Explicit Operations

You must be explicit about operations such as multiplication. Computers only do what we tell them to do, so we have to be clear about what we are telling them. In mathematics we might write

$$5(4-1)$$

where the 5 is implicitly multiplied (not explicitly writing a multiplication symbol). Most programming languages will not interpret 5(4-1) as a valid expression. In MATLAB, we would write

```
>> 5*(4-1)
```

to multiply 5 with $(4-1)$.

2.2 Errors

Errors are useful! They are feedback about something that has gone awry. One of the most common errors when getting started is to forget to be explicit with operations. You might see something like:

```
        >> 5(x+2)
        5(x+2)
         |
        Invalid expression.  When calling a function or indexing a variable,
use parentheses.  Otherwise, check for mismatched delimiters.

        Did you mean:
        >> 5*(x+2)
```

which helpfully suggests what we probably meant!

Another common error is trying to use a variable we haven't yet defined. For example,

```
        >> num / 7
        Unrecognized function or variable 'num'
```

Some bugs won't result in an error at all and are harder to catch. This could include code that is technically correct but miscalculates, or commands that are out of order. When we start to write longer scripts we will explore debugging and good coding practices.

2.3 Built-In Functions

A function is a relation between input and output. In a program, a function defines how to determine the output given an input.

We will see how to write our own functions in Chapter 6, but for now it is helpful to know some of the built-in functions.

The following table provides examples of several functions. Where there is a mathematical notation, it is listed.

Function	Example	MATLAB
Square root	$\sqrt{2}$	`sqrt(2)`
Absolute Value	$\|x\|$	`abs(x)`
Round	$\lfloor 2.713 \rceil$	`round(2.713)`
Ceiling (round up)	$\lceil 3.14 \rceil$	`ceil(3.14)`
Floor (round down)	$\lfloor 3.14 \rfloor$	`floor(3.14)`
Modulus	$17 \mod 5$	`mod(17,5)`
GCD	$\gcd(12, 68)$	`gcd(12, 68)`
LCM	$\text{lcm}(13, 22)$	`lcm(13, 22)`
Prime Test	·	`isprime(57)`
Factor	·	`factor(157)`

Try running each of these examples in MATLAB. Change values and explore the behavior of the outputs. Are you familiar with all of these functions, or are some new to you?

Remember that detailed documentation is available at https://www.mathworks.com/help/. Help for all of these functions is also available in the command window through the `help` command. Let's find out more about that interesting `ceil` function!

```
>> help ceil
 ceil - Round toward positive infinity
    This MATLAB function rounds each element of X to the nearest integer
    greater than or equal to that element.
    Syntax
      Y = ceil(X)

      Y = ceil(t)
      Y = ceil(t,unit)

    Input Arguments
      X - Input array
        scalar | vector | matrix | multidimensional array | table |
        timetable
      t - Input duration
        duration array
      unit - Unit of time
        'seconds' (default) | 'minutes' | 'hours' | 'days' | 'years'

    Examples
      Round Matrix Elements Toward Positive Infinity
      Round Duration Values Toward Positive Infinity

    See also fix, floor, round

    Introduced in MATLAB before R2006a
    Documentation for ceil
    Other uses of ceil
```

There is a consistent formatting in MATLAB's documentation. The first section is always a plain text description of what the function does. In the case of `ceil`, it "rounds each element of X to the nearest integer greater than or equal to that element". In other words, it rounds up. After the description is examples of the syntax, or what it looks like to call the function.

2.3.1 Trigonometry

MATLAB's trigonometric functions use radians by default, but there are also degree versions.

To find the sine of 45°, we can use either of

```
>> sin(pi/4)
```

```
>> sind(45)
```

where `sind` is the sine function using degrees. Similarly, if we want cos(120°) we can use either of

```
>> cos(2*pi/3)
```

```
>> cosd(120)
```

The table below has examples of the basic trigonometric functions. Note that there are two different arctan functions, `atan` and `atan2`.

Function	Example	MATLAB
Sine	$\sin(0.2)$	`sin(0.2)`
Cosine	$\cos\left(\frac{\pi}{6}\right)$	`cos(pi/6)`
Tangent	$\tan(x)$	`tan(x)`
Arcsine	$\sin^{-1}(0.707)$	`asin(0.707)`
Arccosine	$\cos^{-1}(0.866)$	`acos(0.866)`
Arctangent	$\tan^{-1}\left(\frac{-4}{-3}\right)$	`atan(-4/-3)`
Arctangent	$\arctan 2(-4,-3)$	`atan2(-4,-3)`

For those unfamiliar with `atan2`, it is a modified arctan function that uses the sign information of the two arguments to determine the four-quadrant inverse (as opposed to the two-quadrant inverse of `atan`).

2.3.2 Exponential and Logarithm

The MATLAB `log` function returns the natural log, i.e., `log(8.7)` in MATLAB is equivalent to $\ln(8.7)$. To obtain an alternative base logarithm, divide `log()` by the `log` of that base (see the example in the table below).

To calculate an exponential with Euler's constant e, we use `exp()`. The value of $e^{\frac{\pi}{3}}$ can be found with `exp(pi/3)`.

Function	Example	MATLAB
Exponential	$e^{5.2}$	`exp(5.2)`
Natural Log	$\ln(17)$	`log(17)`
Base b log	$\log_5 32$	`log(32)/log(5)`

2.4 Compound Expressions

Parentheses are our friends! When writing expressions with division, the most common mistake relates to order of operations. Take, for example, the expression:

$$\frac{6+4}{2}$$

We would write this as

```
>> (6+4) / 2
```

The common mistake involves writing expressions like the one above as `6 + 4 / 2` without parentheses, which results in an incorrect answer of 7 since the division would take precedence over addition. When in doubt, wrap expressions in parentheses to ensure proper order of operations.

Here are some examples of expressions translated into MATLAB:

Expression	MATLAB
$4-\frac{5}{7}$	`4 - 5/7`
$\frac{4-5}{7}$	`(4 - 5)/7`
$5-{}^{-}3$	`5-(-3)`
$\frac{5+3}{5-2}$	`(5+3) /(5-2)`
$5^{\frac{3}{2}}$	`5^(3/2)`
$\frac{17-4}{\frac{5}{3}}$	`(17-4)/(5/3)`
$4+\sqrt{\frac{2}{3+5}}$	`4 + sqrt(2/(3+5))`
$1-\sin^2(0.3)$	`1-sin(0.3)^2`
$\frac{5}{3}\cos\left(\frac{\pi}{2}\right)$	`(5/3)*cos(pi/2)`
$\frac{5}{3\cos\left(\frac{\pi}{2}\right)}$	`5/(3*cos(pi/2))`

2.5 Variables

A variable is a way of naming a value so that it can be referenced later. Consider the sequence of commands below:

```
>> a = -3
>> b = 15
>> a * b
     ans = -45

>> b = -6
>> a * b
     ans = 18
```

The first line creates a variable called `a` and assigns it the value -3. The second line creates a variable called `b` and assigns it the value 15. The expression `a*b` is then evaluated with those values of `a` and `b`. When we enter `b = -6` on the next line, we are overwriting the value of `b` to now be -6 instead of 15. Finally, the second execution of `a * b` results in 18 reflecting the updated value of `b`.

Variables are particularly useful when plugging values into formulas. Consider Newton's law of universal gravitation,

$$F = G\,\frac{m_1 m_2}{r^2}$$

Here, F is the force of gravity between two masses m_1 and m_2 separated by a distance r, and G is Newton's constant of gravitation. Anyone who has plugged values into this formula while taking physics knows how cumbersome this calculation can be, particularly when dealing with scientific notation on a calculator. Let's take a look at how we can calculate the force of gravity between a mass of 32000 kg and a mass of 250000 kg that are separated by 50 meters.

```
>> m1 = 32000
>> m2 = 250000
>> r = 50
>> G = 6.67 * 10^-11
>> F = G*m1*m2/r^2

     F = 0.00021344
```

If we wanted to recalculate the force for different masses then we would just update `m1` and `m2` and run the final calculation again.

In addition to numbers, we can store strings of characters.

```
>> greeting = "Hello!"
>> disp( greeting )
Hello!
```

Note that strings are defined between quotes. Common errors include neglecting quotes around strings or putting quotes around variables which are not strings.

It is frequently useful to *concatenate* (connect together) two or more values. For example, we can combine multiple variables into a single message with:

```
>> name = "Atticus"
>> num = 34
>> disp("Hello " + name + ", your number is: " + num)
Hello Atticus, your number is: 34
>>
```

The `+` symbol here is not acting like addition, rather it is concatenating the string `"Hello "` with the string variable `name` with the string `", your number is: "` with the value of `num`.

2.5.1 Assignment versus Equality

In mathematics, the equals sign, $=$, means something profound. It is a statement that the left-hand expression is equal to the right-hand expression.

In programming, the equals sign is used to assign a value to a variable. The expression

```
>> x = 22.7
```

assigns the value of 22.7 to the variable `x`. While this looks like an equation, it is an assignment. This is a subtle but important distinction, as it can cause a lot of confusion later on if you are not aware of it.

In section 7.1.1, we will see how to test `if` two expressions are equal.

2.5.2 Updating Variables

Variables can be overwritten to update their value. This is convenient for algorithms that *iterate*, repeating some process over and over.

Consider the 2000 year old algorithm for calculating the square root of a number. Given a number n that we want to find the square root of, we start with a first guess g_1. We can improve our guess to get g_2 with:

$$g_2 = \frac{g_1 + \frac{n}{g_1}}{2}$$

Then we can get an improved third guess with:

$$g_3 = \frac{g_2 + \frac{n}{g_2}}{2}$$

This calculation pattern continues so that we have:

$$g_{i+1} = \frac{g_i + \frac{n}{g_i}}{2}$$

Let's use this pattern to find the square root of 2800, using only one guess variable, `g`, and overwriting our guess as we go:

```
>> n = 2800;
>> g = 100
g =  100

>> g = (g+n/g)/2
g =  64

>> g = (g+n/g)/2
g =  53.875

>> g = (g+n/g)/2
g =  52.924

>> g = (g+n/g)/2
g =  52.915

>> g = (g+n/g)/2
g =  52.915
```

Notice that `g` converged to 52.915 (ignoring the decimal values that are not displayed). When we no longer see our value changing, our *stopping condition* is reached and we declare that we are close enough!

2.5.3 ans, the Built-in Variable

As we saw with our very first command in Section 1.3.3, each command results in an answer stored in a variable called `ans`. This is a variable of convenience, in case we want to use the previous answer in the next command. For example,

```
>> (3+7)/2
 ans = 5

>> ans + 3
 ans = 8
```

While `ans` is always available after a command, we tend to avoid using it. Instead, if we want to continue with a value then we would store that value in a variable.

2.5.4 Suppressing Output

The default behavior when we run a command is to print out the result. Especially when we start working with bigger data sets, we want to assign values to variables without those values being printed. To achieve this we can append a semicolon at the end of a command to suppress output.

```
>> h = (3+7)/2;

>>
```

Above, the expression is evaluated and assigned to variable `h`, then we are immediately prompted for the next command without seeing the output from the evaluation.

In addition to suppressing output, the `;` can separate commands to be written all on one line:

```
>> n1 = 13;   n2 = 4;   disp( n1 * n2 );
 52
>>
```

2.6 Plotting Points

The `plot` function is a versatile and fundamental function for visualizing data. We will see much more about data visualization later, but to introduce the most basic plotting, a single (x, y) coordinate can be plotted with:

```
>> plot(3,-1, 'rx')
```

where `'rx'` specifies the style of the point to be plotted, in this case as a red 'x'.

2.6.1 figure and hold on

Running `plot` again will overwrite the current figure with the new plot, which we may not want. Use

```
>> hold on
```

to continue using the same figure for subsequent plots. To create an entirely new figure, even after setting `hold on`, use

```
>> figure
```

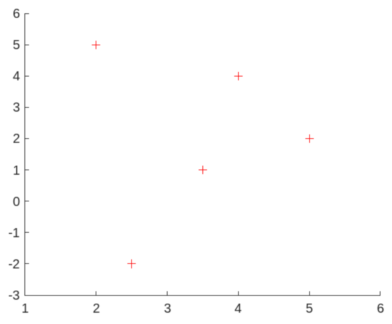
which will leave the previous figure open and create a figure in a new window.

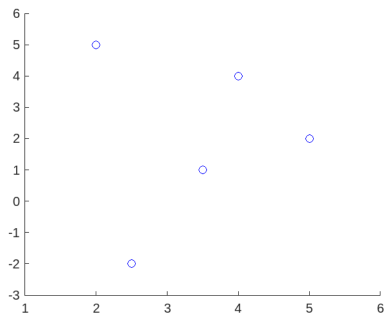
Consider the data in this table:

x	2	2.5	3.5	4	5
y	5	-2	1	4	2

which we can plot one point at a time. Figure 2.1 depicts this data plotted with a variety of markers. The code to create these plots is:

```
>> figure; hold on;
>> plot(  2,  5, 'r+')
>> plot(2.5, -2, 'r+')
>> plot(3.5,  1, 'r+')
>> plot(  4,  4, 'r+')
>> plot(  5,  2, 'r+')
>> figure; hold on;
>> plot(  2,  5, 'bo')
>> plot(2.5, -2, 'bo')
>> plot(3.5,  1, 'bo')
>> plot(  4,  4, 'bo')
>> plot(  5,  2, 'bo')
>> figure; hold on;
>> plot(  2,  5, 'gs', 'Linewidth', 3)
>> plot(2.5, -2, 'gs', 'Linewidth', 3)
>> plot(3.5,  1, 'gs', 'Linewidth', 3)
>> plot(  4,  4, 'gs', 'Linewidth', 3)
>> plot(  5,  2, 'gs', 'Linewidth', 3)
```


(a) Points plotted with `'r+'`


(b) Points plotted with `'bo'`

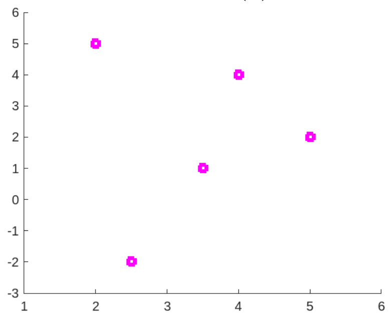

(c) Points plotted with `'gs', 'Linewidth', 3`

Figure 2.1: Sample scatter plots with various marker styles.

2.6.2 Title and Axes

The commands below create the plot in Figure 2.2. Note the labels on the x and y axes as well as the title above the plot. Try creating this plot and observe the effect of the last line of axis([0 4 3 7]).

```
>> figure
>> hold on
>> plot(1,5,'rx')
>> plot(2,4,'rx')
>> plot(3,6,'rx')
>> xlabel('X')
>> ylabel('Y')
>> title('My Plot')
>> axis([0 4 3 7])
```

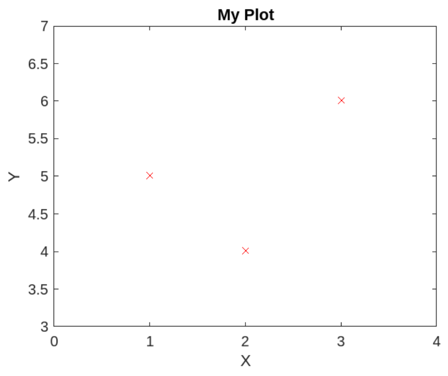


Figure 2.2

By default, the values of the axes extend only far enough to reach the plotted points. This can make plots difficult to read. The axis command sets specific ranges in the form:

```
axis([x_min x_max y_min y_max])
```

The scale of the axes can be set equal with

```
>> axis equal
```

Grid lines can be added to the current figure with

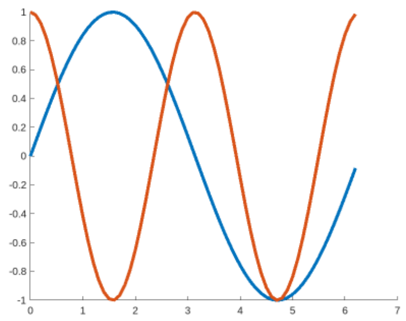
```
>> grid on
```

2.6.3 Plotting Functions

Consider the following plots of $\sin(x)$ and $\cos(2x)$:

```
>> figure
>> hold on
>> x = [0 : 0.1 : 2*pi];
>> plot( x, sin(x) )
>> plot( x, cos(2*x) )
```

The command `x = [0 : 0.1 : 2*pi]` creates a variable `x` and gives it an array of values from 0 to 2π, spaced apart by 0.1. When we plot `sin` or `cos`, we are not plotting a continuous function, rather all of the function outputs for each of the inputs in `x`. Try recreating the plot using `x = [0 : 0.5 : 2*pi]` and you'll notice how "chunky" the curve looks with fewer samples.


(a) Sinusoids plotted with `x = [0:0.1:2*pi]`

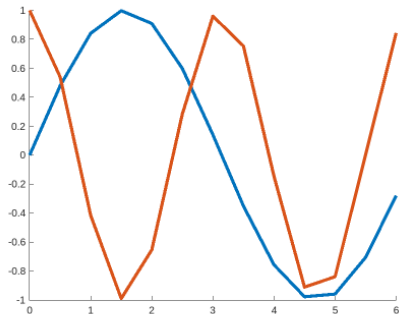

(b) Sinusoids plotted with `x = [0:0.5:2*pi]`

Figure 2.3

We can plot any function $f(x)$ given its definition. For example, to plot $f(x) = \frac{4}{7}x + 8$ from $x = -3$ to $x = 3$, we could write

```
>> x = [-3:0.1:3];
>> plot( x,  (4/7)*x+8)
```