1
Simple Calculations with MATLAB

1.1 Introduction and a Word of Warning

MATLAB is an incredibly powerful tool, but in order to use it safely you need to be able to understand how it works and to be very precise when you enter commands. Changing the way you enter a command, even subtly can completely change its meaning.

The main aim of this text is to teach you to converse with MATLAB and understand its responses. It is possible to interact with MATLAB using a "phrase book" approach, which is fine if the answer is what you expect. However it is far better to learn the language so that you can understand the response. As well as learning the language it is essential that you learn the grammar or syntax; this is perhaps even more important with computer languages than conventional languages! MATLAB uses an interpreter to try to understand what you type and this can come back with suggestions as to where you might have gone wrong: sometimes what you have written makes sense to MATLAB but does not mean what you expect! So you need to be careful. It is crucial that you formulate ideas clearly in your head (or on paper) before trying to translate them into MATLAB (or any other language).

We begin by discussing mathematical operations performed on scalars[1]. It is crucial that the material in this chapter is understood before proceeding, as it forms the basis of all that is to follow[2].

[1] That is numbers.

[2] MATLAB has a wealth of introductory material available to the user that can

We shall start by introducing MATLAB commands which can be typed at the MATLAB prompt; these will ultimately form part of our vocabulary of MATLAB commands. MATLAB already has an extensive vocabulary: however we will learn that we can expand this set. As the name MATLAB (**MAT**rix **LAB**oratory) suggests, most of the commands work with matrices and these will be discussed in due course. We shall start with scalar operations, for which MATLAB acts like a very powerful calculator.

1.2 Scalar Quantities and Variables

We will begin with the basic ideas of equations and variables. Try entering the commands as they are given. Consider the following two commands:

```
>> a = 3

a =

       3

>> b = 4;
```

[3] These two commands are entered on separate lines; the MATLAB prompt is denoted by `>>` (which does not need to be typed), as distinguished from the standard *greater than* sign `>`. The command on the first line sets the variable `a` to be equal to three (3) and that on the second line sets the variable `b` to be equal to four (4). The two commands also differ because the second one ends with a semicolon. This instructs MATLAB to execute the command but suppress any output; whereas above we can see that the value of `a` has been set to 3. These commands can be read as

```
set a equal to 3
set b equal to 4 (and suppress output)
```

Reading the commands in this way it should be clear that it is not possible to have a command of the form `7 = x` (`set 7 equal to x`), whereas we could have `x = 7 (set x equal to 7)`. These variables can now be used again, for instance

be accessed using the commands `demo` or `tour`. There is also a good help facility which, unsurprisingly, can be accessed by typing `help` followed by the command in question. There is also a facility to use a web browser (`helpdesk` or `helpbrowser`).

[3] Here, you would type `a = 3`, and then press RETURN, and then type `b = 4;` and press RETURN again. The spaces are included purely for clarity.

```
>> a = 3;
>> b = a+1;
>> x = a+b;
```

The first line sets the variable `a` to be equal to 3, the semicolon instructing MATLAB to execute the command but to suppress the output. The second line sets `b` to be equal to `a` plus one, namely 4: again the semicolon suppresses output. The third line sets `x` to be `a+b` which is 7 (again output is suppressed).

MATLAB can be used as a very powerful calculator and its operations fall into two basic groups: *unary* and *binary*, the former operating on one quantity and the latter on two. We shall begin by considering simple arithmetic operations, which are *binary*. For instance typing `3*4` generates

```
>> 3*4

ans =

    12
```

Notice here that we have multiplied the two integers `3` and `4`, and the answer has been returned correctly as `12`. MATLAB uses the variable `ans` to store the result of our calculation, in this case the value `12`, so that it can be used in the subsequent commands. For instance the command `ans*3` will generate the result `36` (and now the variable `ans` will have the value 36). We could also have used the commands `a = 3; b = 4; x = a*b` which can be typed on one line and read as

```
set a equal to 3 (don't output anything),
set b equal to 4 (don't output anything)
and set x equal to a times b
```

Division works in exactly the same way as in the multiplication example above. If we try the command `3/4`, MATLAB returns the value `0.75`.

It is a good idea to use meaningful variable names and we shall shortly discuss valid forms for these.

Example 1.1 *Try entering the following commands into MATLAB, but before you do so try to work out what output you would expect.*

```
>> 3*5*6
>> z1 = 34;
>> z2 = 17;
>> z3 = -8;
>> z1/z2
```

```
>> z1-z3
>> z2+z3-z1
```

Hopefully you should get the answers, 90, 2, 42 *and* −25.

Example 1.2 *Here we give an example of the simple use of brackets:*

```
>> format rat
>> a = 2; b = 3; c = 4;
>> a*(b+c)
>> a*b+c
>> a/b+c
>> a/(b+c)
>> format
```

In this example you should get the answers, 14, 10, 14/3 and 2/7. Hopefully this gives you some idea that brackets make MATLAB perform those calculations first. (The command `format rat` *has been used to force the results to be shown as rationals, the final command* `format` *reverts to the default, which happens to be* `format short`*.)*

1.2.1 Rules for Naming of Variables

In the examples we have seen so far we have simply used variable names which seemed to suit the task at hand with no mention of restrictions on allowable variable names in MATLAB. The rules for naming variables in MATLAB can be summarised as follows:

1. Variable names in MATLAB must start with a letter and can be up to 31 characters long. The trailing characters can be numbers, letters or underscores (some other characters are also available but in this text we shall stick to these). There are many choices which are forbidden as variable names, some for very obvious reasons (such as `a*b` which signifies a multiplication of the variables `a` and `b`) and others for more subtle reasons (a good example is[4] `a.b`).

   The rules for naming variables also hold for naming MATLAB files. However, in this case a single dot is allowed within the name of the file; everything after the dot is used to tell MATLAB what type of file it is dealing

[4] The reason this is not a valid variable name lays in the fact that MATLAB supports object orientated programming. Because of this `a.b` refers to the value of the "`b`" component of the object `a`.

with (whether it be a file containing MATLAB code, or data etc). We will see more on this later in the section on *script* files.

2. Variable names in MATLAB **are** case sensitive, so that `a` and `A` are two different objects.

3. It is good programming practise to employ meaningful variable names. In our initial examples we have only used very simple (but appropriate) names: however as the examples become more complex our variable names will be more informative.

4. Variables names should not coincide with a predefined MATLAB command or with any user-defined subroutines. To see whether a variable name is already in use we can use the command `type variable_name`, but it may be better to use the command `which variable_name` (this will tell you whether the name `variable_name` corresponds to an existing code or intrinsic function.

1.2.2 Precedence: The Order in Which Calculations Are Performed

This represents one of the most common sources of errors and it is often the most difficult to detect. Before proceeding we briefly comment on the question of **precedence**, or the order in which commands are executed. Consider the mathematical expression $a(b+c)$ which you might read as "a times b plus c" which would appear to translate to the MATLAB command `a*b+c`. Hopefully you can see that this actually is equal to $ab+c$. The correct MATLAB command for $a(b+c)$ is `a*(b+c)`. The brackets have been used to force MATLAB to first evaluate the expression `(b+c)` and then to multiply the result by `a`. We should avoid falling into the trap of assuming that commands are performed from left-to-right, for instance `c+a*b` is equal to $c+ab$ (not $(c+a)b$ as if the addition was performed first).

At this point we should pause briefly and make sure the ideas of brackets are firmly in place. Brackets should always appear in pairs and the mathematics contained within brackets (or equivalently MATLAB) will be evaluated first. Hopefully this concept is familiar to you: however it is worth reiterating, since one of the most common problems in using MATLAB occurs due to either unbalanced or incorrectly placed brackets. For example the commands `(3+4/5)` and `(3+4)/5` are obviously different, the former being $3\frac{4}{5}$ and the latter being $\frac{3+4}{5}$.

The most critical use of brackets, which circumvents another popular source

of error, is in terms of division. We should note that in the syntax of MATLAB `a/b*c` is **not** equal to $\frac{a}{bc}$ but $\frac{a}{b}c$. In order to ensure that the denominator of the fraction is calculated first we would need to use `a/(b*c)`, which is equal to $\frac{a}{bc}$. Similarly for examples like `a/b+c` versus `a/(b+c)`.

Example 1.3 *Determine the value of the expression* $a(b + c(c + d))a$*, where* $a = 2$*,* $b = 3$*,* $c = -4$ *and* $d = -3$*.*

Although this is a relatively simple example it is worth constructing the MATLAB statement to evaluate the expression:

```
>> a = 2; b = 3; c = -4; d = -3;
>> a*(b+c*(c+d))*a
```

This gives the answer `124`*. It is worth pausing here to consider the syntax of these commands. In the first line of this code we initialize the four variables* `a`*,* `b`*,* `c` *and* `d` *to have the values* 2*,* 3*,* -4 *and* -3 *respectively. The commands each end with semicolons; we have chosen to place all four commands on one line: however they could just as easily be placed on separate lines. With the variables assigned values we can now use them to perform calculations, such as in the second line where we form the mathematical expression* $a(b + c(c + d))a$*. Note all multiplications must be denoted by an asterisk and brackets have been used to force precedence of the operation; of course the brackets must balance (for each left bracket there is a corresponding right bracket) for the expression to make sense.*

Example 1.4 *Evaluate the MATLAB expressions*

```
1+2/3*4-5
1/2/3/4
1/2+3/4*5
5-2*3*(2+7)
(1+3)*(2-3)/3*4
(2-3*(4-3))*4/5
```

by hand and then check answers with MATLAB.

Recall that the operations of division and multiplication take precedence over addition and subtraction (type `help precedence` *at the MATLAB prompt for more details).*

The expressions are given by

$$\texttt{1+2/3*4-5} = 1 + \frac{2}{3}4 - 5 = -\frac{4}{3},$$
$$\texttt{1/2/3/4} = (((1/2)/3)/4) = \frac{1}{24},$$
$$\texttt{1/2+3/4*5} = \frac{1}{2} + \frac{3}{4}5 = \frac{17}{4},$$
$$\texttt{5-2*3*(2+7)} = 5 - 6(9) = -49,$$
$$\texttt{(1+3)*(2-3)/3*4} = \frac{4 \times (-1)}{3}4 = -\frac{16}{3},$$
$$\texttt{(2-3*(4-3))*4/5} = (2 - 3 \times 1)\frac{4}{5} = -\frac{4}{5};$$

which can be verified in MATLAB; we can use the command `format rat` *to force MATLAB to output the results as rational numbers (that is, fractions).*

We mention here MATLAB has a number of intrinsic constants which the programmer can use, for instance `pi` and `eps`. The former is merely $\pi = 3.14159265\cdots$ and the latter is the distance from unity to the next real number in MATLAB[5]. It is also possible to enter numbers using the exponent-mantissa form. This uses the fact that numbers can be written as "mantissa $\times 10^{\text{exponent}}$", for example

Number	mantissa - exponent	MATLAB form
789.34	7.8934×10^2	7.8934e2
0.0001	1×10^{-4}	1e-4
4	4×10^0	4
400000000000	4×10^{11}	4e11

Example 1.5 *Write 3432.6 in exponent-mantissa form and write* 100×10^{10} *in normal form.*

We have

$$3432.6 \equiv 3.4326 \times 10^3$$

and

$$100 \times 10^{10} \equiv 1,000,000,000,000.$$

[5] The smallest positive number that MATLAB can store which is different from zero is `realmin` which is approximately 10^{-308}, whilst the largest number is `realmax` which approximately 10^{308}. These intrinsic constants may be dependent upon your version of MATLAB and/or your computer's operating system.

Example 1.6 *Use MATLAB to calculate the expression*

$$b - \frac{a}{b + \frac{b+a}{ca}}$$

where $a = 3$, $b = 5$ *and* $c = -3$.

The code for this purpose is:

```
a = 3;
b = 5;
c = -3;
x = b-a/(b+(b+a)/(c*a));
```

with the solution being contained in the variable `x`.

Example 1.7 *Enter the numbers* $x = 45 \times 10^9$ *and* $y = 0.0000003123$ *using the exponent-mantissa syntax described above. Calculate the quantity* xy *using MATLAB and by hand.*

This is accomplished using the code

```
x = 45e9;
y = 3.123e-7;
xy = x*y;
```

Notice that here we have used a variable name `xy` *which should not be confused with the mathematical expression* xy *(that is* $x \times y$*).*

We can now set the values of variables and perform basic arithmetic operations. We now proceed to discuss other mathematical operations.

1.2.3 Mathematical Functions

Before we proceed let us try some more of the "calculator" functions (that is, those which are familiar from any scientific calculator).

Arithmetic functions `+`, `-`, `/` and `*`.

Trigonometric functions `sin` (sine), `cos` (cosine) and `tan` (tangent) (with their inverses being obtained by appending an `a` as in `asin`, `acos` or `atan`). These functions take an argument in radians, and the result of the inverse functions is returned in radians. It should be noted these are functions and as such should operate on an input; the syntax of the commands is `sin(x)` rather than `sin x`.

Exponential functions `exp`, `log`, `log10` and `^`. These are largely self explanatory, but notice the default in MATLAB for a logarithm is the natural logarithm $\ln x$. The final command takes two arguments (and hence is a binary operation) so that `a^b` gives a^b.

Other functions There are a variety of other functions available in MATLAB that are not so commonly used, but which will definitely be useful:

`round(x)`	Rounds a number to the nearest integer
`ceil(x)`	Rounds a number up to the nearest integer
`floor(x)`	Rounds a number down to the nearest integer
`fix(x)`	Rounds a number to the nearest integer towards zero
`rem(x,y)`	The remainder left after division
`mod(x,y)`	The signed remainder left after division
`abs(x)`	The absolute value of `x`
`sign(x)`	The sign of `x`
`factor(x)`	The prime factors of `x`

There are many others which we will meet throughout this book. We note that the final command `factor` gives multiple outputs.

We now construct some more involved examples to illustrate how these functions work.

Example 1.8 *Calculate the expressions:* $\sin 60°$ *(and the same quantity squared),* $\exp(\ln(4))$, $\cos 45° - \sin 45°$, $\ln \exp(2 + \cos \pi)$ *and* $\tan 30°/(\tan \pi/4 + \tan \pi/3)$.

We shall give the MATLAB code used for the calculation together with the results:

```
>> x = sin(60/180*pi)

x =

    0.8660

>> y = x^2

y =

    0.7500

>> exp(log(4))
```

```
ans =

     4

>> z = 45/180*pi; cos(z)-sin(z)

ans =

   1.1102e-16

>> log(exp(2+cos(pi)))

ans =

     1

>> tan(30/180*pi)/(tan(pi/4)+tan(pi/3))

ans =

    0.2113
```

The values of these expressions should be $\sqrt{3}/2$, $3/4$, 4, 0, 1 *and* $1/(3+\sqrt{3})$. *Notice that zero has been approximated by* `1.1102e-16` *which is smaller than the MATLAB variable* `eps`*, which reflects the accuracy to which this calculation is performed.*

It is worth going through the previous example in order to practise the command syntax. Getting this right is crucial since it is only through mastering the correct syntax (that is, the MATLAB language) that you will be able to communicate with MATLAB. When you first start programming it is common to get the command syntax confused. To emphasise this let's consider some of the commands above in a little more detail. Let us start with $f(x) = x \sin x$: the MATLAB command to return a value of this expression is `x*sin(x)` and not `x*sinx` or `xsin(x)`. The command `x*sinx` would try to multiply the variable `x` by the variable `sinx`; unless the variable `sinx` is defined (it isn't) MATLAB would return an error message

```
??? Undefined function or variable 'sinx'.
```

Similarly the command `xsin(x)` tries to evaluate the MATLAB function `xsin`, which isn't defined, at the point `x`. Again MATLAB would return an error

message, in this case

```
??? Undefined function or variable 'xsin'.
```

In cases such as these MATLAB provides useful information as to where we have gone wrong; information we can use to remedy the syntax error in our piece of code. This simple example emphasises the need to read your code very carefully to ensure such syntax errors are avoided.

Important Point

> It is essential that arguments for functions are contained within round brackets, for instance `cos(x)` and that where functions are multiplied together an asterisk is used, for instance $f(x) = (x+2)\cos x$ should be written `(x+2)*cos(x)`.

Example 1.9 *The functions we used in the previous example all took a single argument as input, for example* ***`sin(x)`****. Mathematically we can define functions of two or more variables. MATLAB has a number of intrinsic functions of this type (such as the remainder function* ***`rem`****). To see how these are employed in MATLAB we consider two examples of such functions, one of which takes multiple inputs and returns a single output and the other which takes a single input and returns multiple outputs.*

Our first example is the MATLAB function ***`rem`****. The command* ***`rem(x,y)`*** *calculates the remainder when* ***`x`*** *is divided by* ***`y`****. For example* $12345 = 9 \times 1371 + 6$*, so the remainder when 12345 is divided by 9 is equal to* 6*. We can determine this with MATLAB by simply using* ***`rem(12345,9)`****.*

An example of a command which takes a single input and returns multiple outputs is ***`factor`*** *which provides the prime decomposition of an integer. For example*

```
>> factor(24)

ans =

    [2 2 2 3]
```

Here the solution is returned as an array of numbers as the answer is not a scalar quantity. We could just as easily used the command ***`x = factor(24)`*** *to set* ***`x`*** *equal to the array* ***`[2 2 2 3]`****. We can now check MATLAB has correctly*

determined the prime decomposition of the number 24 *by multiplying the elements of the array* `x` *together; this is most readily achieved by using another intrinsic function* `prod(x)`.

1.3 Format: The Way in Which Numbers Appear

Before we proceed we stop to discuss this important topic. This can be simply illustrated by the following example:

Example 1.10 *Consider the following code*

```
s = [1/2 1/3 pi sqrt(2)];
format short; s
format long; s
format rat; s
format ; s
```

this generates the output

```
>> format short; s

s =

    0.5000    0.3333    3.1416    1.4142

>> format long; s

s =

   0.50000000000000   0.33333333333333   3.14159265358979   1.41421356237310

>> format rat; s

s =

     1/2          1/3        355/113     1393/985

>> format ; s

s =

    0.5000    0.3333    3.1416    1.4142
```

There are other options for `format` which you can see by typing `help format`. The default option is `format short` (which can be reverted back to by simply typing `format`). The above options are

`short` – 5 digits

`long` – 15 digits

`rat` – try to represent the answer as a rational.

You should note that whilst `format rat` is very useful, it can lead to misleading answers (in the above example clearly π is not equal to 355/113). At the start of a calculation it is a good idea to ensure that the data is being displayed in the appropriate format. In this example we have performed an operation on four numbers at once using the vector construction in MATLAB. We now proceed to discuss this further.

1.4 Vectors in MATLAB

One of the most powerful aspects of MATLAB is its use of vectors (and ultimately matrices) as objects. In this section we shall introduce the idea of initiating vectors and how they can be manipulated as "MATLAB objects".

1.4.1 Initialising Vector Objects

We shall start with simple objects and construct these using the colon symbol:

```
r = 1:5;
```

This sets the variable `r` to be equal to the vector `[1 2 3 4 5]` (and the semicolon suppresses output, as normal). This is a row vector, which we can see by typing `size(r)` (which returns `[1 5]`, indicating that `r` has one row and five columns). This simple way of constructing a vector `r = a:b` creates a vector `r` which runs from `a` to `b` in steps of one. We can change the step by using the slightly more involved syntax `r = a:h:b`, which creates the vector `r` running from `a` to `b` in steps of `h`, for instance

```
r = 1:2:5;
s = 1:0.5:3.5;
```

gives `r = [1 3 5]` and `s = [1 1.5 2 2.5 3 3.5]`. We note that if the interval `b-a` is not exactly divisible by `h`, then the loop will run up until it exceeds `b`, for instance `t = 1:2:6` gives `t = [1 3 5]`. We can also initiate vectors by typing the individual entries; this is especially useful if the data is irregular, for instance `t = [14 20 27 10];`. There are many other ways of setting up vectors and for the moment we shall only mention one more. This is the command `linspace`: this has two syntaxes

```
s = linspace(0,1);
t = linspace(0,1,10);
```

Here `s` is set up as a row vector which runs from zero to one and has one hundred elements and `t` again runs from zero to one but now has ten elements. Note here that to set up a vector which runs from zero to one in steps of $1/N$, we can use `w = 0:1/N:1` or `W = linspace(0,1,N+1)`. (For example trying typing `s=0:0.1:1.0; length(s)`. You will find that `s` has eleven elements!). The command `linspace` is especially useful when setting up mathematical functions as we shall discover in the next section.

1.4.2 Manipulating Vectors and Dot Arithmetic

We shall now talk about the idea of calculations involving vectors and for this purpose we shall discuss dot arithmetic. This allows us to manipulate vectors in an element-wise fashion rather than treating them as mathematical objects (in fact for addition and subtraction this is the same thing).

To see how dot arithmetic works let's consider a simple example:

```
>> a = [1 2 3];
>> 2*a;

ans =

     2     4     6
```

Suppose now we try to multiply a vector by a vector, as in

```
>> a = [1 2 3];
>> b = [4 5 6];
>> a*b
??? Error using ==> *
Inner matrix dimensions must agree.
```

An error message appears because both `a` and `b` are row vectors and therefore cannot be multiplied together. Suppose however that what we really want to achieve is to multiply the elements of vector `a` by the elements of vector `b` in an element by element sense. We can achieve this in MATLAB by using **dot arithmetic** as follows

```
>> a = [1 2 3];
>> b = [4 5 6];
>> a.*b

ans =

     4    10    18
```

A glance at the answer shows that MATLAB has returned a vector containing the elements

$$[a_1b_1, a_2b_2, a_3b_3].$$

The `.` indicates to MATLAB to perform the operation term by term and the `*` indicates we require a multiplication. We can also do a term by term division with

```
>> a = [1 2 3];
>> b = [4 5 6];
>> a./b

ans =

    0.2500    0.4000    0.5000
```

The result is, as we would expect,

$$\left[\frac{a_1}{b_1}, \frac{a_2}{b_2}, \frac{a_3}{b_3}\right].$$

Example 1.11 *We shall create two vectors running from one to six and from six to one and then demonstrate the use of the dot arithmetical operations:*

```
s = 1:6;
t = 6:-1:1;
s+t
s-t
s.*t
s./t
s.^2
1./s
s/2
s+1
```

This produces the output

```
>> s+t

ans =

     7     7     7     7     7     7

>> s-t

ans =

    -5    -3    -1     1     3     5

>> s.*t

ans =

     6    10    12    12    10     6

>> s./t

ans =

    0.1667    0.4000    0.7500    1.3333    2.5000    6.0000

>> s.^2

ans =

     1     4     9    16    25    36
```

```
>> 1./s

ans =

    1.0000    0.5000    0.3333    0.2500    0.2000    0.1667

>> s/2

ans =

    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000

>> s+1

ans =

     2     3     4     5     6     7
```

These represent most of the simple operations which we may want to use.

We note that in order for these operations to be viable the vectors need to be of the same size (unless one of them is a scalar – as in the last three examples).

1.5 Setting Up Mathematical Functions

Following on from the previous section we discuss how one might evaluate a function. It is crucial that you understand this section before you proceed.

We revisit the topics introduced in the previous section and discuss the ways in which you can set up the input to the function

Example 1.12 *Set up a vector* $\boldsymbol{x}$ *which contains the values from zero to one in steps of one tenth.*

This can be done in a variety of ways:

```
% Firstly just list all the values:
x = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0];

% Use the colon construction
x = 0:0.1:1.0;

% Or use the command linspace
x = linspace(0,1,11);
```

As noted previously we note that there are eleven values between zero and one (inclusive) for a step length of one tenth. You may want to try `linspace(0,1,10)` *and see what values you get.*

Each of these methods are equally valid (and more importantly will produce the same answer) but the latter two are probably preferable, since they are easily extended to more elements.

We now wish to set up a simple mathematical function, say for instance $y = x^2$. Initially you may want to type `x^2` but this will generate the error message

```
??? Error using ==> ^
Matrix must be square.
```

This is because this operation is trying to perform the mathematical operation $\mathbf{x} \times \mathbf{x}$ and this operation is not possible. Instead we need to use `y=x.^2` which gives

```
>> y = x.^2

y =

  Columns 1 through 7

         0    0.0100    0.0400    0.0900    0.1600    0.2500    0.3600

  Columns 8 through 11

    0.4900    0.6400    0.8100    1.0000
```

Here we see that each element of `x` has been squared and stored in the array `y`. Equivalently we could use `y = x.*x;`.

Example 1.13 *Construct the polynomial* $y = (x+2)^2(x^3+1)$ *for values of* x *from minus one to one in steps of 0.1.*

Here it would be laborious to type out all the elements of the vector so instead we use the colon construction. We shall also define $f = (x+2)$ *and* $g = x^3+1$*, so that we have the code:*

```
x = -1:0.1:1;
f = x+2;
g = x.^3+1;
y = (f.^2).*(g);
```

In the construction of `g` *we have used the dot arithmetic to cube each element and then add one to it. When constructing* `y` *we firstly square each element of* `f` *(with* `f.^2`*) and then multiply each element of this by the corresponding element of* `g`*.*

You should make sure that you are able to understand this example.

Example 1.14 *Construct the function* $y = \dfrac{x^2}{x^3+1}$ *for values of* x *from one to two in steps of* 0.01*.*

Here we merely give the solution:

```
x = 1:0.01:2;
f = x.^2;
g = x.^3+1;
y = f./g;
```

(We could have combined the last three lines into the single expression `y = x.^2./(x.^3+1);`*).*

For the moment it may be a good idea to use intermediate functions when constructing complicated functions.

Example 1.15 *Construct the function*

$$y(x) = \sin\left(\frac{x\cos x}{x^2+3x+1}\right),$$

for values of x *from one to three in steps of 0.02.*

Here, again, we use the idea of intermediate functions

```
x = 1:0.02:3;
f = x.*cos(x);
g = x.^2+3*x+1;
y = sin(f./g);
```

NB MATLAB will actually calculate `f/g` and in this case it will return a scalar value of -0.1081. Unfortunately this will not generate an error but it will mean that the answer is not a vector as we should be expecting.

1.6 Some MATLAB Specific Commands

We shall now introduce a couple of commands which can be used to make calculations where the input can take a variety of forms. The first command is `polyval`. This command takes two inputs, namely the coefficients of a polynomial and the values at which you want to evaluate it. In the following example we shall use a cubic but hopefully you will be able to see how this generalises to polynomials of other orders.

Example 1.16 *Evaluate the cubic* $y = x^3 + 3x^2 - x - 1$ *at the points* $x = (1, 2, 3, 4, 5, 6)$. *We provide the solution to this example as a commented code:*

```
% Firstly set up the points at which the polynomial
% is to be evaluated
x = 1:6;

% Enter the coefficients of the cubic (note that
% these are entered starting with the
% coefficient of the highest power first
c = [1 3 -1 -1];

% Now perform the evaluation using polyval

y = polyval(c,x)
```

Note that in this short piece of code everything after the `%` is treated by MATLAB as a comment and so is ignored. It is good practice to provide brief, but meaningful, comments at important points within your code.

Important Point

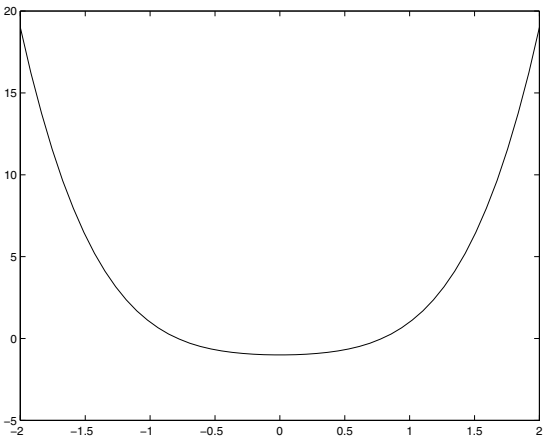
It is important that you remember to enter the coefficients of the polynomial starting with the one associated with the highest power and that zeros are included in the sequence.

We might want to plot the results of this calculation and this can be simply accomplished using the `plot` command. Consider the following example:

Example 1.17 *Plot the polynomial* $y = x^4 + x^2 - 1$ *between* $x = -2$ *and* $x = 2$ *(using fifty points).*

```
x = linspace(-2,2,50);
c = [1 0 1 0 -1];
y = polyval(c,x);
plot(x,y)
```

This produces the output



In the next chapter we shall discuss plotting in more detail and show how plots can be customised.

There are many other commands which allow us to manipulate polynomials: perhaps one of the most useful ones is the `roots`. The polynomial is defined in the same way as in the previous examples. The input to the routine is simply these coefficients and the output is the roots of the polynomial.

Example 1.18 *Find the roots of the polynomial* $y = x^3 - 3x^2 + 2x$ *using the*

command `roots`.

```
c = [1 -3 2 0];
r = roots(c)
```

This returns the answers as zero, two and one.

In fact the converse command also exists, which is `poly`. This takes the roots and generates the coefficients of the polynomial having those roots (which is monic, that is the coefficient of the highest term is unity).

1.6.1 Looking at Variables and Their Sizes

Before we proceed we mention a couple of useful commands for seeing which variables are defined. To list the variables which are currently defined we can use the command `whos`. This will give a list of the variables which are currently defined (a shorter output can be obtained by using the command `who`). This command can be used to list certain variables only, for instance `whos re*` lists only the variables whose names start with `re`.

Example 1.19 *The following code*

```
clear all
a = linspace(0,1,20);
b = 0:0.3:5;
c = 1.;
whos
```

gives the output

```
  Name      Size         Bytes  Class

  a         1x20           160  double array
  b         1x17           136  double array
  c         1x1              8  double array

Grand total is 38 elements using 304 bytes
```

Here we have used the `clear all` *command to remove all previously defined variables. To look at the size of one variable we can use the command* `length`*, for instance with the previous example* `length(a)` *will give the answer* `20`*. We note that the command* `size(a)` *will give two dimensions of the array, that is*

in this case `[1 20]`; this will be particularly useful when we consider matrices in due course.

1.7 Accessing Elements of Arrays

This is one of the most important ideas in MATLAB and other programming languages which is often misunderstood. Let us start by considering a simple array `x = 0:0.1:1.;`. The elements of this array can be recalled by using the format `x(1)` through to `x(11)`. The number in the bracket is the index and refers to which value of `x` we require. A convenient mathematical notation for this would be x_j where $j = 1, \cdots, 11$. This programming notation should not be confused with $x(j)$; that is x is a function of j. Let us consider the following illustrative example:

Example 1.20 *Construct the function $f(x) = x^2+2$ on the set of points $x = 0$ to 2 in steps of 0.1 and give the value of $f(x)$ at $x = 0$, $x = 1$ and $x = 2$. The code to construct the function is:*

```
x = 0:0.1:2;
f = x.^2+2;

% Function at x=0
f(1)
% Function at x=1
f(11)
% Function at x=2
f(21)
```

Note that the three points are **not** *`f(0)`, `f(1)` and `f(2)`!*

In this example we have noted that $x_j = (j-1)/10$ and hence $x_1 = 0$, $x_{11} = 1$ and $x_{21} = 2$. These three indices are the ones we have used to find the value of the function.

Important Point

> In MATLAB `f(j)` the value of `j` refers to the index within the array rather than the function `f(.)` evaluated at the value `j`!

The expression `end` is very useful at this point, since it can be used to refer to the final element within an array. In the previous example `f(end)` gives the value of `f(21)` since the length of `f` is 21.

Example 1.21 *We now show how to extract various parts of the array* `x`*.*

```
x = linspace(0,1,10);
y = x(1:end);       % Whole of x
y = x(1:end/2);     % First half
y = x(2:2:end);     % Even indices only
y = x(2:end-1);     % All but the last one
```

1.8 Tasks

In this introductory chapter we shall give quite a few details (at least initially) concerning these suggested tasks. However, as the reader's grasp of the MATLAB syntax develops the tasks will be presented more like standard questions (the solutions are given at the back of the book in Appendix C).

Task 1.1 *Calculate the values of the following expressions (to find the MATLAB commands for each function you can use the Glossary, see for instance the entry for* `tan` *on page 386 or the* `help` *command,* `help tan`*).*

$$p(x) = x^2 + 3x + 1 \text{ at } x = 1.3,$$
$$y(x) = \sin(x) \text{ at } x = 30^\circ,$$
$$f(x) = \tan^{-1}(x) \text{ at } x = 1,$$
$$g(x) = \sin\left(\cos^{-1}(x)\right) \text{ at } x = \frac{\sqrt{3}}{2}.$$

Task 1.2 *Calculate the value of the function* $y(x) = |x| \sin x^2$ *for values of* $x = \pi/3$ *and* $\pi/6$ *(use the MATLAB command* `abs(x)` *to calculate* $|x|$*).*

Task 1.3 *Calculate the quantities* $\sin(\pi/2)$, $\cos(\pi/3)$, $\tan 60^\circ$ *and* $\ln(x + \sqrt{x^2+1})$ *where* $x = 1/2$ *and* $x = 1$*. Calculate the expression* $x/((x^2+1)\sin x)$ *where* $x = \pi/4$ *and* $x = \pi/2$*. (If you are getting strange answers in the form*

of rationals you may well have left the format as `rat`*, so go back to the default by typing* `format`*).*

Task 1.4 *Explore the use of the functions* `round`*,* `ceil`*,* `floor` *and* `fix` *for the values* $x = 0.3$*,* $x = 1/3$*,* $x = 0.5$*,* $x = 1/2$*,* $x = 1.65$ *and* $x = -1.34$*.*

Task 1.5 *Compare the MATLAB functions* `rem(x,y)` *and* `mod(x,y)` *for a variety of values of* x *and* y *(try* $x = 3, 4, 5$ *and* $y = 3, 4, -4, 6$*). (Details of the commands can be found using the* `help` *feature).*

Task 1.6 *Evaluate the functions*

1. $y = x^3 + 3x^2 + 1$
2. $y = \sin x^2$
3. $y = (\sin x)^2$
4. $y = \sin 2x + x \cos 4x$
5. $y = x/(x^2 + 1)$
6. $y = \frac{\cos x}{1+\sin x}$
7. $y = 1/x + x^3/(x^4 + 5x \sin x)$

for x *from 1 to 2 in steps of 0.1*

Task 1.7 *Evaluate the function*

$$y = \frac{x}{x + \frac{1}{x^2}},$$

for $x = 3$ *to* $x = 5$ *in steps of* 0.01*.*

Task 1.8 *Evaluate the function*

$$y = \frac{1}{x^3} + \frac{1}{x^2} + \frac{3}{x},$$

for $x = -2$ *to* $x = -1$ *in steps of* 0.1*.*

Task 1.9 (D) *The following code is supposed to evaluate the function*

$$f(x) = \frac{x^2 \cos \pi x}{(x^3 + 1)(x + 2)},$$

for $x \in [0,1]$ (using 200 steps). Correct the code and check this by evaluating the function at $x = 1$ using `f(200)` *which should be $-1/6$.*

```
x = linspace(0,1);
clear all
g = x^3+1;
H = x+2;
z = x.^2;
y = cos xpi;
f = y*z/g*h
```

Task 1.10 (D) *Debug the code which is supposed to plot the polynomial $x^4 - 1$ between $x = -2$ and $x = 2$ using 20 points.*

```
x = -2:0.1:2;
c = [1 0 0 -1];
y = polyval(c,x);
plot(y,x)
```

Task 1.11 (D) *Debug the code which is supposed to set up the function $f(x) = x^3 \cos(x+1)$ on the grid $x = 0$ to 3 in steps of 0.1 and give the value of the function at $x = 2$ and $x = 3$.*

```
x = linspace(0,3);
f = x^3.*cos x+1;
% x = 2
f(2)
% x = 3
f(End)
```