3
Loops and Conditional Statements

3.1 Introduction

We now consider how MATLAB can be used to repeat an operation many times and how decisions are taken. We shall conclude with a description of a conditional loop. The examples we shall use for demonstrating the loop structures are by necessity simplistic and, as we shall see, many of the commands can be reduced to a single line. The true power of computers comes into play when we need to repeat calculations over and over again.

In order to help you to understand the commands in this chapter, it is suggested that you work through the codes on paper. You should play the rôle of the computer and make sure that you only use values which are assigned at that time. Remember that computers usually operate in a serial fashion and that they can only use a variable once it has been defined and given a value. This kind of thought process is very helpful when designing your own codes.

3.2 Loops Structures

The basic MATLAB loop command is `for` and it uses the idea of repeating an operation for all the elements of a vector. A simple example helps to illustrate this:

```
%
% looping.m
%
N = 5;
for ii = 1:N
    disp([int2str(ii) ' squared equals ' int2str(ii^2)])
end
```

This gives the output

```
1 squared equals 1
2 squared equals 4
3 squared equals 9
4 squared equals 16
5 squared equals 25
```

The first three lines start with % indicating that these are merely comments and are ignored by MATLAB. They are included purely for clarity, and here they just tell us the name of the code. The fourth line sets the variable `N` equal to 5 (the answer is suppressed by using the semicolon). The `for` loop will run over the vector `1:N`, which in this case gives `[1 2 3 4 5]`, setting the variable `ii` to be each of these values in turn. The body of the loop is a single line which displays the answer. Note the use of `int2str` to convert the integers to strings so they can be combined with the message " squared equals ". Finally we have the `end` statement which indicates the end of the body of the loop.

We pause here to clarify the syntax associated with the `for` command:

```
for ii = 1:N
    commands
end
```

This repeats the `commands` for each of the values in the vector with `ii`$= 1, 2, \cdots, N$. If instead we had `for ii = 1:2:5` then the commands would be repeated with `ii` equal to 1, 3 and 5. Notice in the code above we have indented the `disp` command. This is to help the reading of the code and is also useful when you are debugging. The spaces are not required by MATLAB. If you use MATLAB built-in editor (using the command `edit`) then this indentation is done automatically.

Example 3.1 *The following code writes out the seven times table up to ten seven's.*

```
str = ' times seven is ';

for j = 1:10
    x = 7 * j ;
    disp([int2str(j) str int2str(x)])
end
```

The first line sets the variable `str` *to be the string " times seven is " and this phrase will be used in printing out the answer. In the code this character string is contained within single quotes. It also has a space at the start and end (inside the quotes); this ensures the answer is padded out.*

The start of the `for` *loop on the third line tells us the variable* `j` *is to run from 1 to 10 (in steps of the default value of 1), and the commands in the* `for` *loop are to be repeated for these values. The command on the fourth line sets the variable* `x` *to be equal to seven times the current value of* `j`*. The fifth line constructs a vector consisting of the value of* `j` *then the string* `str` *and finally the answer* `x`*. Again we have used the command* `int2str` *to change the variables* `j` *and* `x` *into character strings, which are then combined with the message* `str`*.*

Example 3.2 *The following code prints out the value of the integers from 1 to 20 (inclusive) and their prime factors.*

To calculate the prime factors of an integer we use the MATLAB command `factor`

```
for i = 1:20
  disp([i factor(i)])
end
```

This loop runs from `i` *equals 1 to 20 (in unit steps) and displays the integer and its prime factors. There is no need to use* `int2str` *(or* `num2str`*) here since all of the elements of the vector are integers.*

The values for which the `for` loop is evaluated do not need to be specified inline, instead they could be set before the actual `for` statement. For example

```
r = 1:3:19;
for ii = r
    disp(ii)
end
```

displays the elements of the vector `r` one at a time, that is 1, 4, 7, 10, 13, 16 and 19; of course we could have used more complicated expressions in the loop.

The following simple example shows how loops can be used not only to repeat instructions but also to operate on the same quantity.

Example 3.3 *Suppose we want to calculate the quantity six factorial (*$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$*) using MATLAB.*

One possible way is

```
fact = 1;
for i = 2:6
    fact = fact * i;
end
```

To understand this example it is helpful to unwind the loop and see what code has actually executed (we shall put two commands on the same line for ease)

```
fact = 1;
i=2; fact = fact * i; At this point fact is equal to 2
i=3; fact = fact * i; At this point fact is equal to 6
i=4; fact = fact * i; At this point fact is equal to 24
i=5; fact = fact * i; At this point fact is equal to 120
i=6; fact = fact * i; At this point fact is equal to 720
```

The same calculation could be done using the MATLAB command `factorial(6)`.

Example 3.4 *Calculate the expression* ${}^nC_m$ *for a variety of values of* n *and* m*. This is read as 'n choose m' and is the number of ways of choosing* m *objects from* n*. The mathematical expression for it is*

$$ {}^nC_m = \frac{n!}{m!(n-m)!}. $$

We could rush in and work out the three factorials in the expression, or we could try to be a little more elegant. Let's consider $n!/(n-m)!$ *is equal to* $n \times (n-1) \times (n-2) \times \cdots \times (n-m+1)$*. We can therefore use the loop structure. Note also that there are* m *terms in this product, as there are in* $m!$*, so we can do both calculations within one loop*

```
prod = 1;
mfact = 1;
for i = 0:(m-1)
    mfact = mfact * (i+1);
    prod = prod * (n-i);
end
soln = prod/mfact;
```

Breaking up the calculation like this can lead to problems for large values of m and so it is often best to work out the answer directly:

```
soln = 1;
for i = 0:(m-1)
    soln = soln * (n-i) / (i+1);
end
```

This product could also be written as:

```
soln = 1;
for i = 0:(m-1)
    soln = soln * (n-i) / (m-i);
end
```

This version may have a slight computational advantage since the terms appearing in the fractions are closer in magnitude. That these two versions are identical can be seen by rewriting the product as

$$t = \left(\frac{n}{1}\right)\left(\frac{n-1}{2}\right)\cdots\left(\frac{n-m+2}{m-1}\right)\left(\frac{n-m+1}{m}\right)$$

or as

$$s = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-m+2}{2}\right)\left(\frac{n-m+1}{1}\right).$$

This example could also be done using the MATLAB command `prod`, which calculates the product of the elements of a vector; in fact MATLAB has a command `nchoosek` designed for just this calculation. In order to use `prod` we could use:

```
M = 1:m;
t = (n-1+M)./M; prod(t)
s = (n-1+M)./(m-M+1); prod(s)
```

where `t` and `s` correspond to the two expressions above.

So far we have dealt with products: in the next example we shall consider a simple summation.

Example 3.5 *Determine the sum of the geometric progression*

$$\sum_{n=1}^{6} 2^n.$$

This is accomplished using the code:

```
total = 0
for n = 1:6
    total = total + 2^n;
end
```

which gives the answer 126, as you might expect from the formula for the sum of a geometric progression

$$S = a\frac{1-r^n}{1-r}$$

where a is the first term, r is the ratio of the terms and n is the number of terms. In our case $a = 2$, $r = 2$ and $n = 6$, which gives $S = 126$.

We now go through the ideas of summing series more thoroughly since it provides important information on how loop structures work.

3.3 Summing Series

In Example 3.5 we have summed a series: we now describe this topic in more detail. We start by constructing a code to evaluate

$$\sum_{i=1}^{N} i^2.$$

Firstly, we note that we do not necessarily know N so this will need to be entered by the user or on a command line. We will first do one case by "hand". Let us consider $N = 4$, so we wish to determine

$$\sum_{i=1}^{4} i^2.$$

If we were to do this on paper we would probably write down all the terms in the summation and then add them up: evaluating the terms in the series

$$\sum_{i=1}^{4} i^2 = 1 + 4 + 9 + 16$$

and adding them up we obtain

30.

At this stage we can add the terms up in our heads: however, what would happen if N was larger, for instance 10. In this case note the answer is given by

$$\sum_{i=1}^{10} i^2 = 1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100.$$

But in fact we may actually do it like this:

- The first term corresponds to $i = 1$, for which $i^2 = 1$.
- The second term, that is $i = 2$, has $i^2 = 4$ and adding to the previous answer gives $1 + 4 = 5$.
- The third term, that is $i = 3$, has $i^2 = 9$ and adding to our previous answer gives $5 + 9 = 14$, etc.

We can automate this process by using the MATLAB code:

```
N = input('Enter the number of terms required: ');
s = 0;

for i = 1:N
    s = s + i^2;
end

disp(['Sum of the first ' int2str(N) ...
     ' squares is ' int2str(s)])
```

As we mentioned earlier the command `int2str` converts an integer to a string. This means we can concatenate these strings into a sentence. If we tried

```
apple = 8;disp(['I have ' apple ' apples'])
```

MATLAB misses out the number and gives "I have apples", whereas the command

```
disp(['I have ' int2str(apple) ' apples'])
```

gives "I have 8 apples", as required: for more information on this command see page 185.

We can break down our summation code as follows:

- The first line simply asks the user to enter the value of `N` when prompted with the string contained in the `input` statement.
- The second line sets a variable `s` to be zero: this will be used to store the cumulative sum. The blank line is included to make our code more readable.
- The next three lines form a loop. The loop variable is `i` which runs from 1 to `N`. This means the command in the loop will be repeated for each of these values. The value of `i^2` will be added to the previous value of `s`.
- After the loop there is another blank line (again purely for readability).
- Finally, we display our results using the `disp` command. We have constructed a row vector with four elements: the first and third elements are merely the strings `'Sum of the first '` and `' squares is '` whereas the second and fourth are the values `N` and `s` converted from integers to strings using the MATLAB command `int2str`.

We can now modify the above code to determine the value of the summation

$$\sum_{i=3}^{7} i^3.$$

Previously we were asked to calculate the sum of square terms and now we need the sum of cubes so the line `s = s + i^2;` needs to be modified to `s = s + i^3;`. We also need to change the range which the loop runs over from `1:N` to `3:7`, which is accomplished by changing the argument of the `for` loop to `i = 3:7`. The modified code will now work, however it will ask for a value of `N` which is no longer relevant. We can comment out this line by prepending it with a percentage sign. This instructs MATLAB to ignore everything else on this line. This can be very useful when you want to change a code but do not want to just delete lines. It can also be used to include comments so you can understand the code. Finally, we need to change the line which outputs the results, the amended code looks like this:

```
% N = input('Enter the number of terms required: ');
s = 0;
for i = 3:7
    s = s + i^3;
end
disp(['Required value of the summation is ' int2str(s)])
```

Again we have used the command `int2str` to change the variable `s` into a string. This only works correctly if `s` is an integer. If `s` is not an integer we need to use the command `num2str` which changes a general number to a string.

In all the cases so far we have only been interested in the final answer. Let's consider instead the problem of determining the values of

$$I_N = \sum_{i=1}^{N} f(i),$$

where the `f(i)` are defined by the problem at hand. At the moment we will stick to `f(i) = i^2`. Again we shall start by doing an example by hand, let us determine the values of `I_N` for `N` equals one to four. Firstly `N = 1`

$$I_1 = \sum_{i=1}^{1} i^2 = 1,$$

and now `N = 2`, which is

$$I_2 = \sum_{i=1}^{2} i^2 = 1 + 4 = 5,$$

and `N = 3`,

$$I_3 = \sum_{i=1}^{3} i^2 = 1 + 4 + 9 = 14,$$

and finally `N = 4`, which gives

$$I_4 = \sum_{i=1}^{4} i^2 = 1 + 4 + 9 + 16 = 30.$$

You should have noticed that `I_2 = I_1+2^2`, `I_3 = I_2+3^2` and `I_4 = I_3+4^2`: These are just like the elements of an array (or vector). In general

$$I_N = I_{N-1} + N^2. \tag{3.1}$$

It is exactly this kind of recursion relation which a computer can take advantage of. In order to calculate these values we can use the code:

```
maxN = input('Enter the maximum value of N required: ');
I(1) = 1^2;

for N = 2:maxN
    I(N) = I(N-1) + N^2;
end

disp(['Values of I_N'])
disp([1:N; I])
```

This code uses a vector `I` to store the results of the calculation. It first prompts the user to input the value of `N`, which it stores in the variable `maxN`. The first value of the vector is then set as 1^2. Equation (3.1) now tells us the relation between the first and second elements of `I`, which we exploit in the loop structure. The final two lines display the string `Values of I_N` and the actual values. The answers are given as a matrix, the first row of which contains the numbers 1 to N and the second row gives the corresponding values of `I_N`.

This produces the results:

```
Enter the maximum value of N required: 10
Values of I_N
     1     2     3     4     5     6     7     8     9    10
     1     5    14    30    55    91   140   204   285   385
```

(for $N = 10$). This can be compared with the analytical solution, see equation (3.4). Up until now we have considered very simple summations. Let's now see how we can sum the series where the terms are defined by some function `f(i)`. We use a separate MATLAB program to construct the series. Let's consider

$$I_N = \sum_{i=1}^{N} i \sin \frac{i\pi}{4}.$$

We start by using a code `f.m` which will return the coefficients we are going to sum. You should enter this code and save it as `f.m`.

```
function [value] = f(inp)
value = inp * sin(inp*pi/4);
```

– The first line tells the computer this program is a function, which should take as input a value `inp` and return the variable `value` (if `value` is undefined MATLAB will complain).

– The second line simply works out the required value. Notice that MATLAB already has a variable `pi`.

The above code is now modified to:

```
maxN = input('Enter the maximum value of N required: ');
I(1) = f(1);

for N=2:maxN
    I(N) = I(N-1) + f(N);
end

disp(['Values of I_N'])
disp([1:N; I])
```

This produces the result

```
Enter the maximum value of N required: 5
Values of I_N
    1.0000    2.0000    3.0000    4.0000    5.0000
    0.7071    2.7071    4.8284    4.8284    1.2929
```

(for $N = 5$). We note that here we have a good example of the differing meanings of the command `y(n)`. In the case of `I(N)`, this refers to the N^{th} entry of the array `I`, whereas for `f(N)`, it refers to the function `f` evaluated at the point `N`. This subtle difference is critical!

3.3.1 Sums of Series of the Form $\sum_{j=1}^{N} j^p$, $p \in \mathbb{N}$

Let's consider the simple code which works out the sum of the first N integers raised to the power p:

```
% Summing series

N = input('Please enter the number of terms required ');
p = input('Please enter the power ');

sums = 0;
for j = 1:N
    sums = sums + j^p;
end

disp(['Sum of the first ' int2str(N) ...
       ' integers raised to the power ' ...
       int2str(p) ' is ' int2str(sums)])
```

so this produces

$$S_N = \sum_{j=1}^{N} j^p.$$

We note the formula when $p = 1$ is given by $N(N + 1)/2$. We pause here to think how we could work this out. If we substitute in the values $N = 1$, $N = 2$ and $N = 3$ (all for $p = 1$) we would obtain three points on the 'curve' and these will uniquely determine its coefficients (assuming it is a quadratic). We assume that $S_N = aN^2 + bN + c$ and use the three values above to give three simultaneous equations from which we can determine the coefficients a, b and c, these equations are:

$$N = 1 \qquad a + b + c = S_1 = 1, \tag{3.2a}$$

$$N = 2 \qquad 4a + 2b + c = S_2 = 1 + 2 = 3, \tag{3.2b}$$

$$N = 3 \qquad 9a + 3b + c = S_3 = 1 + 2 + 3 = 6. \tag{3.2c}$$

From (3.2a) we see that $c = 1 - a - b$ and this can be substituted into the other two equations to yield

$$3a + b = 2, \tag{3.3a}$$

$$8a + 2b = 5. \tag{3.3b}$$

Now using (3.3a) in (3.3b) we find $a = 1/2$ and then in (3.3a), $b = 1/2$ and finally using (3.2a) we have $c = 0$. Hence

$$S_N = \frac{1}{2}N^2 + \frac{1}{2}N, \quad \text{for} \quad N = 1, 2, 3.$$

In order to prove that this is true for all N we can use proof by induction.

It seems reasonable to expect that the sum for a certain power of p will be of degree $p+1$. In order to determine the coefficients of a polynomial of degree $p+1$ we require $p+2$ points. Consider this example:

```
clear all
format rat
p = input('Please enter the power you require ');

points = p+2;
n = 1:points;
for N=n
    sums(N)=0;
    for j = 1:N
        sums(N) = sums(N) + j^p;
    end
end

[coe] = polyfit(n,sums,p+1)

format
```

This code is worth dissecting, since it is our first example of nested loops.

- The first line ensures all values of variables have been cleared. This is good programming practice since it means undeclared variables will be noticed rather than used incorrectly.
- The second line sets the format for the duration of this run to be `rat`, forcing MATLAB to try to print the answers as rationals (notice this is coupled with the final line which merely resets the formatting to the default).
- The next line asks the user for the value of p. As noted above we need to determine $p+2$ coefficients and as such we need $p+2$ equations. A vector `n` is set up running over these values.
- We now have a nested `for` loop structure which sets up the points (from 1 to $p+2$).
- The next line sets the initial value of the element of the array `sums` to be zero, ready to be set to the required cumulative sum. This sum is calculated within the inner loop. We note again that each `for` statement needs to be balanced with an `end`.
- Now we use the MATLAB intrinsic command `polyfit` which gives the coefficients of the required polynomial. The syntax of this command is

`polyfit(x,y,n)` which returns the coefficients of order n through the points in (x, y).

For the examples $p = 2$ and $p = 3$ we have

```
>> sumser2
Please enter the power you require 2

coe =

     1/3          1/2          1/6          *

>> sumser2
Please enter the power you require 3

coe =

     1/4          1/2          1/4          *          *
```

Notice here we have asked the code to try to give us rational answers. The asterisks are merely where MATLAB cannot express zero (or something close to it) as a rational. In the first case we have

$$\sum_{j=1}^{N} j^2 = \frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6} = \frac{N}{6}(2N+1)(N+1), \tag{3.4}$$

(where we have factorised our answer) and in the second case

$$\sum_{j=1}^{N} j^3 = \frac{N^4}{4} + \frac{N^3}{2} + \frac{N^2}{4} = \frac{N^2}{4}(N+1)^2.$$

We could use this procedure for any integer value of p: however a rigorous proof requires mathematical induction (or another similar process).

3.3.2 Summing Infinite Series

We now consider examples where we need to truncate the series.

Example 3.6 *The Taylor series for a function $f(x)$ about a point $x = a$ is given by*

$$f(x) = f(a) + \sum_{n=0}^{\infty} \frac{(x-a)^n}{n!} f^{(n)}(a).$$

We can use this to approximate $\sin x$ *by an infinite series in* x *as*

$$\sin x = \lim_{N\to\infty} \sum_{n=0}^{N} (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

Of course in using a computer we cannot actually take N *to be infinity, but we shall take it to be large in the hope that the error will be small. We can write the above series (taking* $N = 10$*) and evaluate the series at the points* 0*,* 0.1*,* 0.2*,* 0.3*,* 0.4 *and* 0.5 *using*

```
v = 0.0:0.1:0.5;
sinx = zeros(size(v));
N = 10; range = 0:N;
ints = 2*range+1;
for n = range
     sinx = sinx+(-1)^n*v.^ints(n+1)...
        /(factorial(ints(n+1)));
end
```

This is perhaps the most complicated code we have produced so far, so we shall pause and break it down.

`v = 0.0:0.1:0.5;` *sets up the vector* `v` *running from zero to a half in steps of a tenth.*

`sinx = zeros(size(v));` *In order to understand this command it is best to start with what's inside the brackets. The command* `size(v)` *gives the size of the vector* `v` *and then the command* `zero` *sets up* `sinx` *as an array of zeros of the same size.*

`N = 10; range = 0:N;` *sets a variable* `N` *to be equal to 10 and then sets up a vector* `range` *which runs from zero to* `N`*.*

`ints = 2*range+1;` *This gives the mathematical expression* $2n + 1$ *for all the elements of the vector* `range` *and puts them in* `ints`*.*

`for n = range ..... end` *This loop structure repeats its arguments for* `n` *equal to all the elements of range.*

`sinx = sinx+(-1)^n*v.^ints(n+1)/(factorial(ints(n+1)));` *This mathematical expression evaluates the subject of the summation as a function of* n*. (Notice the necessary offset when referring to the elements of* `ints`*, for instance* `ints(1)` *refers to the value when* $n = 0$*). Also note the use of* `.^` *when operating on* `v` *as this is a vector.*

Running this gives

```
>> sinx

sinx =

         0    0.0998    0.1987    0.2955    0.3894    0.4794
```

These values can be compared to those calculated directly by MATLAB to give

```
>> sin(v)

ans =

         0    0.0998    0.1987    0.2955    0.3894    0.4794

>> sinx-sin(v)

ans =

   1.0e-16 *

         0    0.1388    0.2776         0   -0.5551         0
```

Since all the elements of the vector `sinx-sin(v)` *share a factor of* `1.0e-16` *this has been extracted. This results shows that our approximation has worked very well (for relatively few terms). For larger values of x more terms might be needed. For a value of $x = \pi$ we find the same number of terms as used above produces a value of $\sim 10^{-11}$, which is still a good approximation (since the actual value of $\sin\pi$ is zero), but not quite as good as those shown above. The infinite series given above actually converges for all values of x.*

Example 3.7 *Calculate the sum*

$$\sum_{n=0}^{\infty} \mathrm{e}^{-n}.$$

This poses a problem for us since we can obviously not add up an infinite number of terms, we need to truncate the calculation. In this case we do not need very many terms since e^{-x} gets very small very quickly. We can use the code:

```
N = 10;
total = 0;
for n = 0:N
    total = total + exp(-n);
end
```

To test for convergence of our results we should compare our answer for different values of `N`*. Fortunately, in this case, we are able to work out the value of the truncated series, since it is merely a geometric progression (the formula for the sum of a geometric progression is given on page 68). The first term is* $\mathrm{e}^0 = 1$ *and the ratio between successive terms is* $1/e$*, hence the sum is*

$$S_N = \frac{1 - \mathrm{e}^{-(N+1)}}{1 - \mathrm{e}^{-1}}.$$

In the limit as $N \to \infty$ *this tends to*

$$S_\infty = \frac{1}{1 - \mathrm{e}^{-1}} = \frac{e}{e-1} \approx 1.581977.$$

If we compare the results for $N = 10$ *the error is already* $\sim 10^{-5}$*. To find a formula for the error we need to calculate* $|S_N - S_\infty|$*, which is*

$$|S_N - S_\infty| = \left| \frac{1 - \mathrm{e}^{-(N+1)}}{1 - \mathrm{e}^{-1}} - \frac{1}{1 - \mathrm{e}^{-1}} \right| = \left| \frac{\mathrm{e}^{-N}}{e-1} \right|,$$

which decreases (exponentially fast) as N increases.

3.3.3 Summing Series Using MATLAB Specific Commands

So far we have used commands which are common to many programming languages (or at least similar to) and have not exploited the power of MATLAB. In the first example, given above, we mentioned the idea of adding things up in our head. MATLAB is very good at this type of exercise. Consider the previous example

$$\sum_{i=1}^{10} i^2.$$

Firstly we set up a vector running from one to ten:

```
i = 1:10;
```

and now a vector which contains the values in `i` squared:

```
i_squared = i.^2;
```

Now we use the MATLAB command `sum` to evaluate this:

```
value = sum(i_squared)
```

(*notice here we have left off the semicolon so the result is displayed automatically*). The full code for this example is

```
i = 1:10;
i_squared = i.^2;
value = sum(i_squared)
```

This can all be contracted on to one line `sum((1:10).^2)`: *you should make sure you know how this works!*

Using the command `sum` allows us to simplify our codes: however it is essential we understand exactly what it is doing. Consider a problem where we have a set of values y_1, y_2 up to y_N, where for the sake of argument N is taken to be odd. The problem is to work out the sum of the series

$$\sum_{i=1}^{N} y_i f(i),$$

where $f(i) = 1$ when i is odd and $f(i) = 2$ when i is even. We can set up these values of f using the commands:

```
N = 11;
iodd = 1:2:N;
ieven = 2:2:(N-1);
f(iodd) = 1;
f(ieven) = 2;
```

which gives

```
f =

     1     2     1     2     1     2     1     2     1     2     1
```

This can be achieved in one command, namely `2-mod(1:11,2)`. We can now use a similar code to those we wrote earlier to sum the series. For example if $y = x^2$, we have

```
x = 1:11;
y = x.^2;
sum(y.*f)
```

This kind of expression will prove to be very useful when we come to consider numerically evaluating integrals in a later chapter.

Example 3.8 *Evaluate the expression*

$$\prod_{n=1}^{N}\left(1+\frac{2}{n}\right)$$

for $N = 10$ *(the symbol* $\prod$ *means the product of the terms, much in the same way* $\sum$ *means summation). This can be done using the code:*

```
n = 1:10;
f = 1+(2)./n;
pr = prod(f)
```

The first line sets up a vector with elements running from 1 to 10; the second line sets up a vector `f` *whose elements have the values* $1 + 2/n$*; notice* `f` *is the same shape as* `n`*. We have divided 2 by a vector and so we have used "./" rather than just "/". If you want to see what the code does, just leave off the semicolons; this will show you the vectors which are generated. Finally, the last line gives the product of all the elements of the vector* `f`*.*

This gives the answer 66. This can be seen by a slight rearrangement of the expression

$$\prod_{n=1}^{N}\left(1+\frac{2}{n}\right)=\prod_{n=1}^{N}\left(\frac{n+2}{n}\right)=\frac{\prod_{n=1}^{N} n+2}{\prod_{n=1}^{N} n},$$

where we have used the fact $\prod a_n b_n = (\prod a_n)(\prod b_n)$*. Now actually evaluating the products:*

$$\begin{aligned}&=\frac{(N+2)(N+1)\cdots 3}{N(N-1)\cdots 1}\\&=\frac{(N+2)!/(2.1)}{N!}=\frac{(N+2)(N+1)}{2}.\end{aligned}$$

When $N = 10$ *this gives a value of 66.*

3.3.4 Loops Within Loops (Nested)

Many algorithms require us to use nested loops (loops within loops), as in the example of summing series on page 75. We illustrate this using a simple example of constructing an array of numbers:

```
for ii = 1:3
    for jj = 1:3
        a(ii,jj) = ii+jj;
    end
end
```

Notice that the inner loop (that is, the one in terms of the variable `jj`) is executed three times with `ii` equal to 1, 2 and then 3. These structures can be extended to have further levels. Notice each `for` command must be paired within an `end`, and for the sake of readability these have been included at the same level of indentation as their corresponding `for` statement.

Example 3.9 *Calculate the summations*

$$\sum_{j=1}^{N} j^p$$

for p equal to one, two and three for $N = 6$.

We could perform each of these calculations separately but since they are so similar it is better to perform them within a loop structure:

```
N = 6;
for p = 1:3
    sums(p) = 0.0;
    for j = 1:N
        sums(p) = sums(p)+j^p;
    end
end
disp(sums)
```

The order in which the loops occur should be obvious for each problem but in many examples the outer loop and inner loop could be reversed.

3.4 Conditional Statements

MATLAB has a very rich vocabulary when it comes to conditional operations but we shall start with the one which is common to many programming languages (even though the syntax may vary slightly). This is the `if` command which takes the form:

```
if (expression)
        commands
        ...
end
```

As you might expect, if the `expression` is true then the commands are executed, otherwise the programme continues with the next command immediately beyond the `end` statement. There are more involved forms of the command, but before proceeding with these let's first discuss the construction of the `expression`. Firstly there are the simple mathematical comparisons

`a < b`	True if a is less than b
`a <= b`	True if a is less than or equal to b
`a > b`	True if a is greater than or equal to b
`a >= b`	True if a is greater than or equal to b
`a == b`	True if a is equal to b
`a ~= b`	True if a is not equal to b

More often than not we will need to form compound statements, comprising more than one condition. This is done by using logical expressions, these are:

`and(a,b)`	`a & b`	Logical AND
`or(a,b)`	`a | b`	Logical OR
`not(a)`	`~a`	NOT
`xor(a,b)`		Logical exclusive OR

The effect of each of these commands is perhaps best illustrated by using tables

AND	false	true
false	false	false
true	false	true

OR	false	true
false	false	true
true	true	true

XOR	false	true
false	false	true
true	true	false

NOT (∼) This simply changes the state so ∼(true)=false and ∼(false)=true.

We pause and just run through these logical operators:

a AND b This is true if both a and b are true

a OR b This is true if one of a and b is true (or both).

a XOR b This is true if one of a and b is true, but not both.

In many languages you can define Boolean (named after George Boole) variables and these will have actual logical meaning. For instance in Excel you can set values to be `True` or `False`. In MATLAB zero represents False and any other value implies True. We shall adopt the convention that `1` (one) is true and `0` (zero) is false. In fact sometimes it is useful to define variables such that `false=0; true= (∼false)`. This statement is read as `set true to be equal to not false`; we have added the brackets for clarification.

Example 3.10 *In the following examples it may be helpful to draw pictures, but we should at least teach you to read these statements. We recall that in this context a square bracket means a closed set (that is, including the end point) whereas a round bracket indicates an open set (that is, not including the end point).*

Determine the sets for which these statements are true:

1. `x>1 & x<2`
   We can read this command as "x is greater than one and less than two". This is obviously true only for values between one and two (not inclusive). This is the open set $(1, 2)$.

2. `x<0 | x>=1`
   This command can be read as "x less than zero or greater than (or equal to) one". This set has two parts: x strictly negative or x greater than (or equal to one). Hence the set is $(-\infty, 0) \cup [1, \infty)$.

3. `x>1 | x<2`
   This command can be read as "x greater than 1 or x less than 2". In fact all values of x are greater than one or less than 2, hence the answer is $(-\infty, \infty)$.

4. `x<=1 | x>=1`
   This one says "x is less than (or equal to) 1 or x is greater than (or equal to 1)". This again is true for all values of x, hence the answer is $(-\infty, \infty)$.

5. `x<=1 & x>=1`
   This one is similar to the previous one with one word changed so it now reads:"x is less than (or equal to) 1 and x is greater than (or equal to 1)".

The only value which is less than (or equal to) one and greater than (or equal to) one is one itself. The answer is the single value one, written as $\{1\}$.

6. `~(x>2)`
   This is our first example of negation, which is read as "x is not greater than 2", which is true for values of x less than (or equal to) 2, that is the set $(-\infty, 2]$.

7. `(x>1) & (~(x<2))`
   Here we have "x is greater than 1 and x is not less than 2". The second part of this expression means that x is greater than (or equal to 2) (and the first part is always true for this range), so the solution is $[2, \infty)$.

8. `abs(x-1) < 2`
   Here we have an expression involving a mathematical function: this one is read as: "the modulus of x minus 1 is less than two". This means points which are within 2 units of the point 1, that is $(-1, 3)$.

9. `rem(n,4) == 1`
   This command calculates the remainder when divided by 4 and checks to see if this is equal to 1. The values for which this is true are $\{4n+1 : \ n \in \mathbb{Z}\}$.

3.4.1 Constructing Logical Statements

We shall now actually construct logical arguments which can be used in `if` statements.

Example 3.11 *Let us consider a command which is only executed if a value x lies between 1 and 2 or it is greater than or equal to 4. We shall try to describe the thought processes involved:*

– *In order that a value lies between one and two, it has to be greater than one* **and** *less than two, so this component is written as:*

```
(x>1) & (x<2)
```

– *If x is greater than or equal to 4, which is written simply as* `x>=4`.

– *Finally, we need to combine these conditions and this is done using the logical operation* `or`, *since the value of x could lie in one or the other of the regions. Hence we have*

```
((x>1) & (x<2)) | (x>=4)
```

We could use the commands in a different form

```
a = and(x>1,x<2);
b = (x>=4);
c = or(a,b)
```

by making use of the `and` *and* `or` *commands. Notice here we have actually set "Boolean" variables* `a`*,* `b` *and* `c` *(in fact they are only normal variables which take the values zero or one).*

We can also use these commands for vectors and matrices (comparing them element-wise). Consider the following

```
>> x = [1 2 0 3];
>> and(x>1,x<4)

ans =

     0     1     0     1
>> A = eye(2);
>> and(A==1,A>0)

ans =

     1     0
     0     1
```

If two sets do not intersect then by definition $A \cap B = \emptyset$ (the null, or empty, set). In terms of logical statements we say that the conditions are mutually exclusive. For instance, the condition `(x>1)&(x<0)` is never true, since x cannot be greater than one **and** less than zero.

Notice that we have used brackets to group the terms in the previous example. This is not always necessary but it is very good practice, especially since we cannot guarantee that the precedence in other languages will be the same. In fact although the expression `(x>=4) | ((x>1) & (x<2))` is the same as that given above, removing the brackets from both of them yields different results. In the former case the expression still works, whereas in the latter case

MATLAB complains about precedence and invites the user to find out more by using `help precedence`.

It is convenient at this stage to introduce some of the other commands which are available to us when constructing conditional statements, namely `else` and `elseif`. The general form of these is given by:

```
if (expression)
         commands ...
elseif (expression)
         commands ...
else
         commands ...
end
```

Notice that each `if` statement must be paired with an `end` statement, but we can use as many `elseif` statements as we like, but only one `else` (within a certain level). We are also able to nest `if` statements: again it is recommended that the new levels are indented so as to aid readability.

Example 3.12 *Consider the following piece of code which determines which numbers between 2 and 9 go into a specified integer exactly:*

```
str = 'Divisible by ';
x = input('Number to test: ');

for j = 2:9
    if rem(x,j) == 0
      disp([str int2str(j)])
    end
end
```

The results of this code can be checked using the `factor` *command. We have used the command* `rem` *to obtain the remainder.*

Example 3.13 *Here we construct a conditional statement which evaluates the function:*

$$f(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leqslant x \leqslant 1 \\ 2 - x & 1 < x \leqslant 2 \\ 0 & x > 2 \end{cases}$$

One of the possible solutions to this problem is:

```
if x >= 0 & x <= 1
   f = x;
elseif x > 1 & x <= 2
   f = 2-x;
else
   f = 0;
end
```

Notice that it has not been necessary to treat the end conditions separately; they are both included in the final `else` *clause.*

Example 3.14 (Nested if statements) *The ideas behind nested if statements is made clear by the following example*

```
if raining
    if money_available > 20
       party
    elseif money_available > 10
       cinema
    else
       comedy_night_on_telly
    end
else
    if temperature > 70 & money_available> 40
       beach_bbq
    elseif temperature>70
       beach
    else
       you_must_be_in_the_UK
    end
end
```

3.4.2 The MATLAB Command `switch`

MATLAB has a command called `switch` which is similar to the BASIC command `select` (the syntax is virtually identical). The command takes the form:

```
switch switch_expr
        case case_expr1
          commands ...
        case {case_expr2,case_expr3}
          commands ...
        otherwise
          commands ...
end
```

We shall work through the example given in the manual documentation for this command:

```
switch lower(METHOD)
        case {'linear','bilinear'}
               disp('Method is linear')
        case 'cubic'
               disp('Method is cubic')
        case 'nearest'
               disp('Method is nearest')
        otherwise
               disp('Unknown method.')
end
```

This code assumes that `METHOD` has been set as a string. The first command `lower(METHOD)` changes the string to be lower case (there is also the corresponding command `upper`). This value is then compared to each case in turn and if no matches are found the `otherwise` code is executed.

Example 3.15 *We refer to the old rhyme that allows us to remember the number of days in a particular month.*

Thirty days hath September
April, June and November
All the rest have thirty-one
Except February alone
which has twenty-eight days clear
and twenty-nine on leap year

This is exploited in the code:

```
msg = 'Enter first three letters of the month: ';
month = input(msg,'s');
month = month(1:3); % Just use the first three letters
if lower(month)=='feb'
   leap = input('Is it a leap year (y/n): ','s');
end
switch lower(month)
     case {'sep','apr','jun','nov'}
          days = 30;
     case 'feb'
          switch lower(leap)
              case 'y'
                  days = 29;
              otherwise
                  days = 28;
          end
     otherwise
          days = 31;
end
```

Before we proceed it is worth discussing a couple of the commands used above. The MATLAB command `lower` *forces the characters in the string to be lower case: for instance if the user typed 'Feb' the command* `lower` *would return 'feb'. We have also included a command which makes sure the code only considers the first three characters of what the user inputs. The command* `input` *is used here with a second argument* `'s'` *which tells MATLAB that a string is expected as an input; without this the user would need to add single quotes at the start and end of the string.*

You might want to consider what you would need to do to check the user has input a valid month. The next command is very useful for this kind of problem.

3.5 Conditional loops

Suppose we now want to repeat a loop until a certain condition is satisfied. This is achieved by making use of the MATLAB command `while`, which has the syntax

```
while (condition)
     commands...
end
```

This translates into English as: while `condition` holds continue executing the `commands...`. The true power of this method will be found when tackling real examples, but here we give a couple of simple examples.

Example 3.16 *Write out the values of x^2 for all positive integer values x such that $x^3 < 2000$. To do this we will use the code*

```
x = 1;
while x^3 < 2000
     disp(x^2)
     x = x+1;
end
value = floor((2000)^(1/3))^2;
```

This first sets the variable `x` *to be the smallest positive integer (that is one) and then checks whether $x^3 < 2000$ (which it is). Since the condition is satisfied it executes the command within the loop, that is it displays the values of x^2 and then increases the value of x by one.*

The final line uses some mathematics to check the answer, with the MATLAB command `floor` (which gives the integer obtained by rounding down; recall the corresponding command to find the integer obtained by rounding up is `ceil`).

Example 3.17 *Consider the one-dimensional map:*

$$x_{n+1} = \frac{x_n}{2} + \frac{3}{2x_n},$$

subject to the initial condition $x_n = 1$. Let's determine what happens as n increases. We note that the fixed points of this map, that is the points where $x_{n+1} = x_n$, are given by the solutions of the equation:

$$x_n = \frac{x_n}{2} + \frac{3}{2x_n},$$

which are $x_n = \pm\sqrt{3}$. We can use the code

```
xold = 2; xnew = 1;

while abs(xnew-xold) > 1e-5
     xold = xnew;
     xnew = xnew/2+3/(2*xnew);
end
```

This checks to see if $x_{n+1} = x_n$ to within a certain tolerance. This procedure gives a reasonable approximation to $\sqrt{3}$ and would improve if the tolerance (`1e-5`) was reduced.

3.5.1 The `break` Command

A command which is of great importance within the context of loops and conditional statements is the break command. This allows loops to stop when certain conditions are met. For instance, consider the loop structure

```
x = 1;
while 1 == 1
    x = x+1;
    if x > 10
       break
    end
end
```

The loop structure `while 1==1 ... end` is very dangerous since this can give rise to an infinite loop, which will continue *ad infinitum*; however the break command allows control to jump out of the loop.

3.6 MATLAB Specific Commands

We could have coded the Example 3.16 using the MATLAB command `find`, which returns an array of locations at which a certain condition is satisfied. The condition is formed as above using logical expressions and combinations of them.

Example 3.18 *Find all integers between 1 and 20 for which their sine is negative.*

Firstly we set up the array of integers, then calculate their sines and subsequently test whether these values are negative.

```
ii = 1:20;
f = sin(ii);
il = find(f<0);
disp(ii(il))
```

We briefly dissect this code.

– In the first line set up a vector which runs from one to twenty;

– the second line calculates the sine of the elements in vector `ii` and stores the result in the vector `f`;

– in the third line we use `find` to determine the locations in the vector `f` where the sine is negative. The command returns a list of positions in the vector `f` (and equivalently `ii`) where the condition is true.

– The final command prints out a list of these integers.

Other similar commands are `any` and `all`. Examples of their use are:

```
x = 0.0:0.1:1.0;
v = sin(x);
if any(v<0)
    disp('Found a negative value')
else
    disp('All values zero or positive')
end

if all(v>0)
    disp('All values positive')
else
    disp('One value is zero or negative')
end
```

Some care is needed with the logic at this point and these commands seem to be, in some sense, complements of each other.

There are many other commands which test the properties of a variable, for instance `isempty`, `isreal` etc.

Example 3.19 *Now we evaluate the expression*

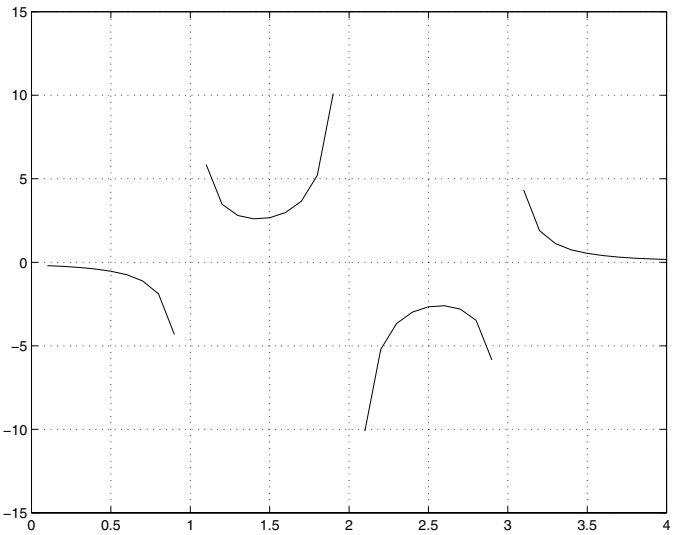
$$f(x) = \frac{1}{(x-1)(x-2)(x-3)}$$

on a grid of points $n/10$ *for* $n = 1$ *to* $n = 40$ *for which the expression is finite, else the function is to be returned as 'NaN'.*

This is accomplished using the code:

```
x = (1:40)/10;
g = (x-1).*(x-2).*(x-3);
izero = find(g==0);
ii = find(g~=0);
f(izero) = NaN;
f(ii) = 1./g(ii);
```

This allows us to plot the function whilst missing out the infinite parts. This would be accomplished using the command `plot(x,f)`



3.7 Error Checking

To this point we have assumed the data made available to a code is suitable; this is generally a very dangerous assumption. We have found MATLAB will object to mathematically impossible tasks: however it will happily perform operations provided they are viable (which unfortunately they are more often than not).

Now that we have seen conditional statements we can make sure our codes are more robust. The ultimate aim would be to make them totally "idiot proof", but this is virtually impossible. No matter how many different scenarios a programmer comes up with, they can never predict everything a user will do.

MATLAB gives us three very useful commands in this context: `break`, `warning` and `error`. The first of these we have already met. The latter two

allow us to either warn the user of a problem or actually stop the code because of an irretrievable problem, respectively. Both the commands `warning` and `error` are used with an argument, which is displayed when the command is encountered. The typical structure might be:

```
if code_fails
   error(' Irretrievable error ')
elseif code_problem
   warning(' Results may be suspect ')
end
```

The actual use of the commands is perhaps best demonstrated by a couple of examples:

Example 3.20 *Let's now write a code which asks the user for an integer and returns the prime factors of that integer.*

```
msg = 'Please enter a strictly positive integer: ';
msg0 = 'You entered zero';
msg1 = 'You failed to enter an integer';
msg2 = 'You entered a negative integer';
x = input(msg);
if x==0
   error(msg0)
end
if round(x)~= x
   error(msg1)
end
if sign(x)==-1
   warning(msg2)
   x = -x;
end
disp(factor(x))
```

Fortunately the command ***input*** *has its own error checking routine and will only let you input numbers (if you try inputting a string it will complain). We now check to see whether zero is entered, in which case the code stops after having informed the user that they entered a zero. Similarly, if the user enters a non integer the code will stop. If the user enters a negative integer the code warns the user they have done so but simply makes it equal to the corresponding positive value.*

The command **input** *allows us to input a matrix, which would cause our program to fail (in fact the routine* **factor** *issues an error message). We could incorporate a check to ensure that what is entered is a scalar. This could be accomplished by using the condition that the* **length** *of a scalar is unity. To this end we could add (below the* **input** *statement):*

```
msg3 = 'Code needs a scalar integer';
if length(x) ~= 1 | x ~= floor(x)
   error(msg3)
end
```

When MATLAB encounters these commands it produces a message which indicates on which line they occurred and in which code. This is very helpful when debugging codes.

Example 3.21 *Now consider a code which compels the user to enter a four character string and until they have the code will not proceed.*

```
msg = ['Please enter a ' ...
    'four character string '];
msg0 = ' is not valid, please re-enter';

str = 'X'; % Clearly not valid
first = 1;
while length(str) ~= 4
   if ~(first)
      warning([str msg0])
   end
   first = 0;
   str = input(msg,'s');
end
```

In order to avoid the warning message during the first run, we have used the flag **first** *which basically says if this is the first time through don't issue the warning. This uses a* **while** *loop to repeat until the user enters a string of length 4.*

As we can see this additional code can make the program cumbersome, but sometimes it is necessary, especially if other users are going to read and use your codes. Again, it also helps when it comes to debugging.

Another very useful command in this context is `exist`. This can be used to check whether the requisite variables exist. This is particularly useful if input is required from another code.

Example 3.22 *Let us consider this header for a code:*

```
msg = ['The variable z does not exist ' ...
       'and will be required for this code'];
if ~(exist('z'))
   error(msg)
end
```

We note this code will not work at the prompt, since the command `error` *will only run during the execution of an m-file.*

3.8 Tasks

Task 3.1 *Modify the code on page 71 to calculate the value of the summation*

$$\sum_{i=1}^{100} \frac{1}{i^2}.$$

(Notice the answer will probably not be an integer, so you will have to modify the display line to use `num2str`*).*

Task 3.2 *Modify the code from Task 3.1 to only sum the values corresponding to odd values of i. (Hint: This only involves a very minor change to the* `for` *loop. You should check the answer you get is smaller than the one from the previous task.)*

Task 3.3 *Modify the function* `f.m` *(on page 72) to calculate the values of:*

$$I_N = \sum_{i=1}^{N} \frac{\sin \frac{i\pi}{2}}{i^2 + 1},$$

for N up to `N=20`*.*

Task 3.4 *Construct a program to display the values of the function* $f(x) = x^2 + 1$ *for* $x = 0$ *to* $x = \pi$ *in steps of* $\pi/4$. *The key here is to set up an array of points from* a *to* b *in steps of* h *(you can either use the colon command or* `linspace`*). Make sure you remember to use the dot operator.*

Task 3.5 *Change the code given in Example 3.6, used to evaluate* $\sin x$, *to one that calculates* $\cos x$ *whose series expansion is given by*

$$\cos x = \lim_{N\to\infty} \sum_{n=0}^{N} (-1)^n \frac{x^{2n}}{(2n)!}.$$

Compare the approximations to the values calculated directly from MATLAB for $x = 0$, $1/4$, $1/2$ *and* $3/4$. *As for Example 3.6 you will need to truncate the summation, choosing* N *to be suitably large.*

Task 3.6 *Calculate the sum of the series* S_N, *where*

$$S_N = \sum_{n=1}^{N} \frac{1}{n^2}$$

for different values of N. *Given that as* $N \to \infty$, $S_N \to \pi^2/c$ *where* c *is a constant. Determine the value of* c. *(Here you should modify one of the codes for summation: the best one is probably the solution to Task 3.1.)*

Task 3.7 *Calculate the summations*

$$\sum_{j=1}^{p+1} j^p$$

for p *equal to one, two, three and four (using a nested loop structure).*

Task 3.8 *Show that, to within the accuracy permitted by MATLAB,*

$$\lim_{N\to\infty} \sum_{n=1}^{N} \frac{(-1)^n}{n} = -\ln 2 \text{ and } \lim_{N\to\infty} \sum_{n=1}^{N} \frac{1}{n(n+1)} = 2.$$

Task 3.9 *Write down (on paper) a logic expression which is true for values of* x *such that* $2 < x < 4$. *Make sure your expression works for* $x = 1$, 3 *and* 5. *Repeat the exercise for the union of the sets* $x > 3$ *and* $x < -1$. *(You need to write the answers in the form of the combination of* x *is greater than something and/or* x *is less than something else).*

Task 3.10 *Write down a logical expression which is true for even values of n. Extend this to only be true for even values of n greater than 20. (Hint: Try using the MATLAB command* `mod`*: for information on the command type* `help mod`*.)*

Task 3.11 *Try to work out what value of x the following code returns (initially without running it).*

```
x = 1;
if tan(73*pi*x/4) >= 0
   x = 2;
else
   x = pi;
end
if floor(x) == x
   x = 10;
else
   x = 7;
end
if isprime(x)
   x = 'True';
else
   x = 'False';
end
```

Is this result true whichever value of `x` *we start with? (You can find out about the command* `isprime` *by typing* `help isprime`*.)*

Task 3.12 (*) *Write a loop structure which iterates* $x_{n+1} = 5x_n|1$ *for* $x_n = 1/7$ *until it returns to the same value (use* `while`*).*

Task 3.13 (*) *Determine all integers between 1 and 50 for which* $n^3 - n^2 + 40$ *is greater than 1000 and n is not divisible by 3. Are* `any` *integers between 1 and 50 perfect (that is, are they equal to the sum of their factors)?*

Task 3.14 *Write a code which only allows the user to input an integer n between 1 and 10 (inclusive) and then prints out a string of the first n letters of the alphabet. (Hint: You could start with a string of the form "abcdefghij".)*

Task 3.15 (*) *Write a code which allows the user to input a two character string, the first being a letter and the second being a digit. (Note that to check if a character is a letter we can use inequalities on strings.)*

Task 3.16 (*) *By modifying the code on page 88 use the* ***find*** *command to plot the function*

$$f(x) = \begin{cases} 0 & x < -1 \\ x^2 & -1 \leqslant x \leqslant 1 \\ 1 & 1 < x < 4 \\ 0 & x \geqslant 4 \end{cases}$$

for the range $x \in [-3, 5]$ *using one hundred points (try using the command* ***linspace*** *to set up the array of points).*

Task 3.17 (*) *Using the command* ***find****, and modifying the code on page 93, plot the function*

$$f(x) = \frac{1}{\cos \pi x}$$

for $x \in [-3, 3]$*. You will need to change the code to determine when the denominator becomes small, rather than when it is identically zero.*

Task 3.18 (D) *The following code is supposed to evaluate the function*

$$f(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leqslant x \leqslant 1 \\ 2 - x & 1 \leqslant x \leqslant 2 \\ 0 & x > 2 \end{cases}$$

```
x=lnspace(-4,4);
N = length x

for j = 1;N

if x(j)>=0 and x(i)<=1
   f(j) = x(j);
elseif x(j)>1 or x(i)<2
   f(j) = 2 - x
else
   f(j) = zero;
end
```

Correct the code so that it accomplishes this. This can be checked by using the command `plot(x,f)` *and comparing the figure to what you expect the function to look like.*