Conditional statements and loops

1. Introduction

Whenever a program is designed to perform a practical function, there will almost certainly be conditions to be met and repeated actions to be performed. In the next section we will examine how conditional statements may be constructed, and what reserved words are needed for this. In fact, every computer language will necessarily employ conditional statements: here we concentrate on the syntax that is used in Matlab to achieve this; fortunately, in most cases only slight adaptation is needed to adapt from other languages to Matlab.

2. Conditional statements

The simplest of all conditional statements starts with if and a condition, and follows this with a required action:

```matlab
if < condition >
    < action >
end
```

Note that an if statement continues with the action on the next line—thereby avoiding the need to write then—and ends with the reserved word end: this should be regarded as a close bracket on the if clause.

Matlab also has an else function and an elseif function: these allow more complex conditional clauses to be spelt out:

```matlab
if < condition >
    < action >
else
    < action >
end
```

or even:

```matlab
if < condition >
    < action >
elseif < condition >
    < action >
elseif < condition >
    < action >
else
    < action >
end
```

In each of the above cases, else action is optional, and in the second case, any number of elseif conditions and actions may be employed. Notice that each complete
*if* clause is terminated by an *end* bracket; however, the *action* in each part of the *if* clause must be terminated by one of the reserved words *elseif*, *else* or *end*.

We can also include a totally new *if* clause within or instead of any of the actions:

```matlab
if < condition >
    < action >
elseif < condition >
    < action >
    if < condition >
        < action >
    elseif < condition >
        < action >
    else
        < action >
    end
end
```

With a multiplicity of *if* clauses, there is considerable possibility of confusion and of erroneous logic. To avoid this problem, each *if* clause (*if*, *elseif*, *else*, *end*) should be regarded as a **block** of code, which should be placed in its entirety within an action statement, as indicated by the boxes around sections of the above code. Similarly, further nesting of *if* clauses and blocks is permissible, but significant rigour is required to ensure that the above rule is obeyed. Note particularly that it is never permissible to have blocks that intersect each other.

Another structure may be used to clarify the situation when a number of conditions are to be checked and acted upon. That is the **switch** statement. First, an expression is evaluated to determine which of a set of actions is to be followed: this is indicated by the parameter *option*. We then write down all the options and the actions that they indicate:

```matlab
switch option
    case 1
        action1
    case 2
        action2
    case 3
        action3
    otherwise
        action4
end
```
The above code may be regarded as entirely equivalent to, but a neater form of, the following piece of code

```matlab
if option == 1
  action1
elseif option == 2
  action2
elseif option == 3
  action3
else
  action4
end
```

In a switch statement, the option parameter is not restricted to numbers: characters or strings can be used instead.

### 3. Loop statements

There are two main types of loop to be learnt—the `for` loop and the `while` loop. We shall consider the `for` loop first, as it is the more versatile of the two. Basically, the `for` loop involves trying a sequence of numerical values and taking appropriate action for each of them.

A typical use of the `for` loop is to add a set of numbers together:

```matlab
number = [ 2 6 4 9 7 3 5 ];
total = 0;
for i = 1: 7
  total = total + number(i);
end
```

Here, the actual `for` loop starts with `for` and ends with `end`. The `for` loop may be generalised by specifying an increment in the range of allowed range values: using the value 2 instead the default value 1, we get the following form of the `for` loop:

```matlab
for i = 1: 2: 7
  total = total + number(i);
end
```

Another variant which is often useful is the following one (in this case a vector is used to specify a sequence of prime numbers):

```matlab
for i = [ 1 2 3 5 7 11 13 ]
  total = total + i;
end
```

We can also have nested `for` loops, as would happen when scanning over a 2-D image. In that case the second `for` loop would be a block that appeared inside the
existing for loop. Again, blocks are not allowed to intersect: one must be entirely within the other: this is usually indicated by indentation, but in the case below it is also indicated by a box around the inner loop.

```matlab
number = ones(7);
total = 0;
for i = 1: 7
  for j = 1: 7
    total = total + number(i, j);
  end
end
```

The while loop can provide a very neat solution to certain problems, such as finding the median of a distribution. An algorithm for achieving this is shown below: it is run by applying the command `medianscriptA` after the Matlab prompt: the output produced by the algorithm then follows.

```
% This program finds the median of a distribution
distribution = [ 3 6 8 2 4 9 5 7 4 ];
area = sum(distribution);
halfway = area/2;
i = 1;
tot = 0;
while tot < halfway;
  tot = tot + distribution(i);
  fprintf('%4d%4d%4d\n', i, halfway, tot)
  i = i + 1;
end
median = i - 1
```

```
>> medianscriptA
 1 24 3
 2 24 9
 3 24 17
 4 24 19
 5 24 23
 6 24 32
median =
 6
```

We have already seen another use of the while loop—to terminate input from a file when the end of a file has been reached: the basic idea is to take an input while not at the end of the file, i.e., while not eof, or, using the proper file-handling functions:
finfo = fopen('filename', 'r');
while ~feof(finfo)
    fgetl(finfo, nextline);
    % nextline needs to be dissected using strtok
end
closeinfo = fclose(finfo);

There is actually no need to use while loops: everything that can be done using them can be carried out using for loops in their place, and noting when a particular condition has been reached (such as passing the median position when the summed area is half the total area of the distribution). Clearly, this wastes a lot of computation: one solution is to use the condition to escape from the loop, e.g., using the break function. This is illustrated in the following alternative code for the median algorithm given above:

```matlab
% This program finds medians using a for loop
distribution = [ 3 6 8 2 4 9 5 7 4 ];
area = sum(distribution);
halfway = area/2;
tot = 0;
for i = 1: 9
    tot = tot + distribution(i);
    median = i;
    fprintf('%4d%4d%4d%4d
', i, halfway, tot, median)
    if tot >= halfway
        break;
    end
end
```

```
>> medianscriptB
1  24  3  1
2  24  9  2
3  24  17  3
4  24  19  4
5  24  23  5
6  24  32  6
```

In fact, resorting to use of break is commonly regarded as bad programming practice—and should only be used when an error condition is detected and it is desired to exit the whole program or a whole function or procedure call. Note that issuing a break command only results in an exit from the current loop: if the latter is nested within another loop, further action would be needed to exit the whole program. To achieve this, a breakflag parameter should be set to true before applying the break function. Overall, these considerations mean that the while loop often provides a very neat alternative to the for loop.
4. Code vectorisation

The idea of vectorising code is to eliminate as many loops as possible and to rely on the intrinsic properties of vectors and matrices, which permit certain well-defined actions to be taken simultaneously on all their elements. The concept was aired early on in the *Introduction to Matlab*, where we learnt that operations such as the following could be used to change the values of all the elements of a matrix in a single operation:

```
>> C = A + B;
>> C = A .* B;
```

This is clearly aesthetically advantageous, and also has the advantage of helping to reduce coding errors. It also has the advantage of reducing execution time: indeed, in Matlab it can often reduce execution time by one or two orders of magnitude. To achieve all this more widely than might be implied by the above two examples, note that many of the operations described earlier are already vectorised in the sense that no overt loops are needed to express them—for example:

```
>> vec6n = vec6(vec6 < 0);
>> vec6(vec6 < 0) = [];
```

In addition, many Matlab functions are available which allow loop-free computation of vectors and matrices and a number of derived quantities. The list includes the functions `cumsum`, `size`, `max`, `min`, `sum`, the colon operator, the transpose operator, `sign`, `isequal`, `any` and `all`—as we can see from the following examples:

```
>> vec = [ 1 2 3 5 7 11 ];
>> vcum = cumsum(vec)
vcum =
   1  3  6 11 18 29
>> mat = [ 1 2 3 4; 5 6 7 8; 9 10 11 12 ];
mat =
   1  2  3  4
   5  6  7  8
   9 10 11 12
>> vsize = size(mat)
vsize =
   3  4
>> vmax = max(mat)
vmax =
   9 10 11 12
>> vsum = sum(mat)
vsum =
  15 18 21 24
>> sumsum1 = sum(vsum)
sumsum1 =
   78
```
In the above examples, all the new entities starting with 'v' are vectors' and, apart from three new matrices, all the other variables are scalars. Thus, they provide examples of conversion of vectors into vectors, matrices into matrices, matrices into vectors, and matrices into scalars—in all cases without the use of overt loops. Of course, there are clearly loops in the inner workings of most of the relevant functions,
but where this is the case they will have been coded carefully, with due regard to
(a) saving computation, and (b) maintenance of the underlying data in a way such that
the minimum of housekeeping is needed, e.g., data will be left in the right place for
subsequent operations. If needed, we could probably emulate the necessary coding,
especially recreating the necessary functions as required, but in reality it is far better
to have all the debugging of function code done properly once and for all than have to
redo it repeatedly on the fly. This means that it is always better to use standard Matlab
functions where possible.

It is worth noting that further vectorisation is available when mathematical and
computational functions are called. In particular, standard mathematical functions
such as \( \sin \), \( \cos \), \( \log \), \( \exp \), \( \abs \), can take matrices as arguments; similarly for a random
matrix variable generated by \( \text{rand} \) or \( \text{randi} \). Finally, we must not forget that
polynomial expressions can be regarded as matrix or vector expressions, once
properly defined. For example:

\[
\text{pianonote} = t \cdot \exp(-b \cdot t) \cdot \sin(w \cdot t);
\]
\[
\text{if } t < 0
\]
\[
\text{pianonote} = 0;
\]
\[
\text{end}
\]

Here, the aim is to create a matrix \( M(i, j) \) which depends on another matrix \( S(i, j) \)
which is defined by the values of the sine function, which in turn depends on the
values of the parameter matrix \( t(i, j) \), for all required values of the indices \( i, j \).

Note that equations such as the above are special in that they produce a matrix or
vector of the same size as the input on the right. However, certain operations such as
\( \text{cumsum} \), \( \text{sum} \) and \( \text{max} \) are designed to reduce dimensionality, so care may be needed
to ensure that the desired result is obtained in such cases.

5. Index

\begin{align*}
\text{all}, & 6, 7 & \text{isequal}, & 6, 7 \\
\text{any}, & 6, 7 & \text{max}, & 6 \\
\text{block}, & 2 & \text{median}, & 4, 5 \\
\text{break}, & 5 & \text{min}, & 6 \\
\text{colon operator}, & 6 & \text{nesting}, & 2 \\
\text{cumsum}, & 6 & \text{otherwise}, & 2 \\
\text{else}, & 1, 2 & \text{rem}, & 7 \\
\text{elseif}, & 1, 2 & \text{sign}, & 6, 7 \\
\text{end}, & 1, 2, 3 & \text{sin}, & 8 \\
\text{eof}, & 4 & \text{size}, & 6 \\
\text{exp}, & 8 & \text{strtok}, & 5 \\
\text{fclose}, & 5 & \text{sum}, & 6 \\
\text{feof}, & 5 & \text{switch}, & 2 \\
\text{fgetl}, & 5 & \text{transpose operator}, & 6 \\
\text{for}, & 3, 4, 5 & \text{vectorisation}, & 6, 8 \\
\text{fprintf}, & 4, 5 & \text{while}, & 3, 4, 5 \\
\text{if}, & 1, 2 &
\end{align*}