Short Version of Matlab Manual

This is an extract from the manual written by Ivan Graham for MA10126 in the first year. This short version and the full manual are available at: `http://www.maths.bath.ac.uk/~jz203/teaching/matlab/`; the `.m` files can also be found there.

1 Starting MATLAB

This is explained in a separate handout, see `http://www.maths.bath.ac.uk/~jz203/teaching/matlab/matlab-starting.pdf`.

2 Do-it-Yourself Tutorial on Loops and Logical Branching

In many situations you will need to repeat the same command several times on different data, or perhaps under different conditions. Statements that tell the computer which command is to be executed next are fundamental building blocks for computer programs, and are called *control-flow statements*. In this tutorial you will revise the use of three control-flow statements for `MATLAB` programs:

- the `for` loop, for repeating a statement or group of statements a fixed number of times,
- the `while` loop, for repeating a statement or group of statements an indefinite number of times while a condition remains true, and
- the `if-elseif-else` statement, which tells the computer to perform different calculations in different situations.

You will revise the writing of these loops by doing the following four exercises. In the exercises you should write *script* programs for each of the tasks, i.e. you should create files (using the `MATLAB` editor/debugger) which are saved (by clicking on `Files --> Save As`) in M-files (with the `.m` extension). These should contain the sequence of `MATLAB` commands necessary to do the tasks required. I suggest you start the exercises immediately. If you have forgotten something about the necessary `MATLAB` commands, please refer to the manual entries in the sections below.

1. Write and test a script program which reads in a positive integer n, and computes $n!$, using a `for` loop. (A convenient way to read in the data is via the `input` command – type `help input` if you have forgotten it.)

2. Using a simple `while` loop, write a script to sum the series 1 + 2 + 3 + ... such that the sum is as large as possible without exceeding 100. The program should display the sum and also how many terms are used in the sum.

3. Write a script that takes as input an integer n and creates the $n \times n$ matrix A with (i, j)th component given by $A(i, j) = \sin(1/(i + j - 1))$.

4. Write a script that takes as input three numbers a, b and c and prints out either the solutions of the quadratic equation $ax^2 + bx + c = 0$, when these solutions are real, or a message indicating that the solutions are not real.

3 The manual pages

3.1 The `for` loop

A simple form of such a loop is

```
for index = 1:n
```

```
statements
end
```

The statements are executed n times with `index` $= 1, 2, 3, \ldots, n$. Here `n` is a number that must be fixed before we enter the loop. More generally, the loop may start with the line `for index = j:m:k`. Usually `j,m` and `k` are integers and `k-j` is divisible by `m`. The statements are executed repeatedly starting from `index = j` with `index` incremented by `m` each time and continuing until `index = k` (which is the last time round the loop). If `m = 1`, we can replace `j:m:k` in the first line of the loop by `j:k`. Even more generally, we can begin with the line

```
for index = v
```

where `v` is a vector. The statements inside the loop are repeatedly executed with `index` set to each of the elements of `v` in turn. Type `help for` and read the manual page for the `for` loop.

As an example, consider the computation of the series

$$1 - 1/2 + 1/3 - 1/4 + 1/5 - \ldots \tag{3.1}$$

This is implemented in the script file `tutorial3a.m`:

```
% script tutorial3a.m
% computes the sum of the series
%
% 1 - 1/2 + 1/3 - 1/4 + ...
%
N = input('type the number of terms to be added up: ')
               % asks the user to provide a value for N

sign = 1;      % initialise the sign for the term
sum_series = 0;    % initialise the sum of the series

for n = 1:N
sum_series = sum_series + sign/n;
sign = -sign;    % changes sign for alternating series
end

sum_series     % prints out the sum of the series to N terms
```

When you have worked with `MATLAB` for a while you will find that program speeds can be improved by using (if possible) vector or matrix operations instead of loops. For example, to sum the series (3.1) to an even number of terms `N`, we can use the following very short script.

```
% script tutorial3b.m
% computes the sum of the series
%
% 1 - 1/2 + 1/3 - 1/4 + ...
%
% to an even number of terms
% using vector operations

N = input('type the number of terms N (with N even):  ')
```

```
sum_series = sum(1./(1:2:N-1) - 1./(2:2:N))
```

Exercise: Make sure you understand how the program above works. Use `help sum`.

We have seen how to use a `for` loop to create a vector whose entries are given by a formula. If the entries of a matrix are given by a formula then we can use nested `for` loops to create it. For example the *Hilbert matrix* H is an $n \times n$ matrix whose entry in the ith row and jth column is $1/(i+j-1)$. If a value has been assigned to n, we can write

```
for i = 1:n
 for j = 1:n
  H(i,j) = 1/(i+j-1);
 end % for j
end % for i
```

I have added the comment after the `end` statements to show which loop is ended. This is useful when there are several nested loops.

Note that this may not be the most efficient way of assembling the Hilbert matrix - see the full version of this manual for more detail. But our main purpose here is correctness rather than efficiency.

3.2 The `while` loop

The general form of the `while` statement is

```
while (condition)
statements
end
```

The `condition` is a logical relation and the `statements` are executed repeatedly while the `condition` remains true. The condition is tested each time before the `statements` are repeated. It must eventually become false after a finite number of steps, or the program will never terminate.

Example. Suppose we have invested some money in a fund which pays 5% (compound) interest per year, and we would like to know how long it takes for the value of the investment to double. Indeed we would like to obtain a statement of the account for each year until the balance is doubled. We cannot use a `for` loop in this case, because we do not know beforehand how long this will take, so we cannot assign a value for the number of iterations on entering the loop. Instead, we must use a `while` loop.

```
%script tutorial3c.m

format bank                                % output with 2 dec places
invest = input('type initial investment:  ')

r = 0.05;                                   % rate of interest
bal = invest;                              % initial balance
year = 0;                                  % initial year
disp('           Year     Balance')       % header for output
```

```
                                        % (You can experiment with
this)
while (bal < 2*invest)   % repeat while balance is
                         % less than twice the investment,
                         % and stop when balance exceeds this
  bal = bal + r*bal;                    % update bal
  year = year + 1;                      % update year
  disp([year,bal])

end
```

3.3 The `if-elseif-else` statement

A simple form of the `if` statement is

```
if (condition)
statements
end
```

Here `condition` and `statements` are the same as in the `while` loop, but in this case the `statements` are executed only once if `condition` is true and are not executed at all if `condition` is false. For example the following script divides 1 by i, provided i is non-zero; otherwise, j is not assigned a value.

```
if (i ~= 0)
j=1/i;
end
```

The symbol `~=` is a *relational operator* and stands for *is not equal to*. Other relational operators include `==`, `<=`, `>=`, etc. Type `help ops` to find out about these. Note the difference between the relational operator `==` and the usual use of the symbol `=`, which assigns a value to a variable.

The `if-else` statement allows us to choose between two courses of action. For example the following script reads in a number and prints out a message to say if it is negative or non-negative.

```
x = input(' Type x :   ')

if (x<0)
  disp('x is negative')
else
  disp('x is non-negative')
end
```

Note that indenting of statements inside loops and `if` statements helps make your program more readable.

Going further, adding `elseif` allows us to choose between a number of possible courses of action.

```
x = input(' Type x :   ')

if (x<0)
  disp('x is negative')
elseif (x>0)
  disp('x is positive')
else
  disp('x is zero')
end
```

A more general form is

```
if (condition1)
  statementsA
elseif (condition2)
  statementsB
elseif (condition3)
  statementsC
...
else
  statementsE
end
```

This is sometimes called an `elseif` *ladder*. Its effect is the following.

- First `condition1` is tested. If it is true then `statementsA` are executed and execution then skips to the next statement after `end`.
- If `condition1` is false, then `condition2` is tested. If it is true, then `statementsB` are executed and execution skips to the next statement after `end`.
- Continuing in this way, all the conditions appearing in `elseif` lines are tested until one is true. If none is true, then `statementsE` are executed.

There can be any number of `elseif`s but only one `else`.

Example. Suppose a bank offers annual interest of 3% on balances of less than £5,000, 3.25% on balances of £5,000 or more but less than £10,000, and 3.5% for balances of £10,000 or more. The following program calculates an investor's new balance after one year.

```
% script tutorial3d.m

bal = input('type balance: ')

if (bal < 5000)
 rate = 0.03;
elseif (bal < 10000)
 rate = 0.0325;
else
 rate = 0.035;
end

disp(['new balance is :  ',num2str((1+rate)*bal)])
```

The logical relations that make up the `condition` in a `while` or `if` statement can quite complicated. Simple relations can be converted into more complex ones using the three logical operators `&` (and), `|` (or) and `~` (not). For example the quadratic equation $ax^2 + bx + c = 0$ has two equal roots, $-b/(2a)$, provided that $b^2 - 4ac = 0$ and $a \neq 0$. This can be programmed as:

```
if((b^2 - 4*a*c == 0)&(a~=0))
  x = - b/(2*a);
end
```

4 Appendix: Some useful MATLAB commands

On-line help

`help`	lists topics on which help is available
`helpwin`	opens the interactive help window
`helpdesk`	opens the web-browser-based help facility
`help topic`	provides help on `topic`
`lookfor string`	lists help topics containing `string`
`demo`	runs the demo program

Workspace information and control

`who`	lists variables currently in the workspace
`whos`	as above, giving their size
`what`	lists M-files on the disk
`clear`	clears the workspace, removing all variables
`clear all`	clears all variables and functions from the workspace

Command window control

`clc`	clears command window, command history is lost
`home`	same as `clc`
↑	recall previous command

Graphics

`plot`	plots a graph
`xlabel('x')`	labels x axis `x`
`ylabel('y')`	labels y axis `y`
`title('title')`	gives a figure a title `title`
`axis`	fixes figure axes
`clf`	clears figure from figure window
`cla`	clears figure from figure window, leaving axes

Controlling program execution

`break`	terminates execution of a `for` or `while` loop
`error('message')`	aborts execution, displays `message` on screen
`return`	exit from function, return to invoking program

Input from and output to terminal

`x=input('type x:')`	asks user to give a value to be assigned to `x`
`disp('string')`	outputs `string` to terminal

String-number conversion

`num2str`	converts a number to a string (so it can be output e.g. as part of a message)

Logical functions

`isempty`	true (=1) if a matrix is empty
`find`	finds indices of non-zero elements of a matrix

Arithmetic functions

`sum(x)`	calculates the sum of the elements of the vector `x`
`prod(x)`	calculates the product of the elements of the vector `x`

Termination

`^c` (Control-c)	local abort, kills the current command execution
`quit`	quits `MATLAB`
`exit`	same as `quit`