Matlab notes for MA2715/MA2895 in 2019/0

The scope of these notes

These notes are intended as mostly a brief revision of material that you are likely to have already met in level 1 and will hopefully also be useful if you are also using Matlab at the moment as part of other modules. The notes should also help you in your preparation for the class test which is planned for week 22. The intention here is that you try out the statements given here and you attempt the exercises in the supervised lab sessions. Later when the assignment tasks are given out you should mostly work on the assignment tasks in the supervised lab sessions and this is likely to be the case for at least 6 weeks.

Being brief the material covered here is quite selective in what is included and if other things are needed later on then information about these will be given when required. Matlab itself is a large package and any given user only learns the parts which are needed to solve certain problems and thus the selection of material given here is limited to the revision needed for the numerical problems considered in `MA2715` which include solving linear systems $A\underline{x} = \underline{b}$ and with solving ordinary differential equations. Some exercises may be close to things you have already met in level 1. In many situations it is useful to represent the output graphically and hence the notes also contain a few details about creating two-dimensional plots.

Contents

1 The Matlab set-up at Brunel

1.1 Your windows set-up at Brunel

To help me in the lab sessions please do the following.

1. After you have logged in start a web brower such as Microsoft edge or internet explorer and open the URL

   `http://people.brunel.ac.uk/~icstmkw/ma2715/lab_setup.html`

2. Keep the web browser open and get a command prompt window by clicking on the search box symbol (next to the bottom left hand corner of the screen) and typing `cmd.exe` in the box shown.

3. Highlight the 6 lines given below in the web browser and press together the `Ctrl` and `C` keys to save these to the clipboard. Next make the `cmd.exe` window your active window, select `Edit` from the top bar and from the sub-menu select `Paste`. If the copying to the clipboard did not save the final end of line character then you will need to press the return key. This should execute the batch of lines that you have highlighted. The 6 lines are as follows.

```
net use s: /d
net use s: "\\v-csfs01\Math Utilities"
s:
cd s:\mkw\mybin20\
s:\mkw\mybin20\mkw_mat20.bat
h:
```

If you have done this correctly then you should get, among other things, an icon with a link to material for this module. Some of the other things created may make it easier for me to help you in the lab sessions in certain situations and it will also make it easier to make available files that you might need. For information, a folder `h:\mybin20` is created and this is where most of the files are put and in particular it includes a few files with names ending in `.m` which may be useful in some Matlab tasks.

1.2 Which version of Matlab do we have at Brunel?

You can start Matlab by clicking on the search box symbol (next to the bottom left hand corner), entering `matlab` in the search box and checking if any versions of Matlab are listed. In the labs it should indicate a R2019a CEDPS teaching version which is what you should select.

If you are elsewhere on campus and Matlab is not shown after you do the above then for a while it likely still to be possible to use a R2010 version by doing the following which assumes that you have previously run the set-up instructions described in section 1.1. Start a command window by clicking on an icon which starts with `cmd_win` and in the command window type `mat_l.bat` which should run the commands in the file `h:\mybin20\mat_l.bat`. This first command in the file gives you an `L:` drive, which is where the software is stored, and the other instructions starts the software. As long as this version still exists you can also use this in the labs where the later version is available.

1.3 Comments about the file `startup.m`

If everything ran smoothly when you implemented the set-up instructions then you may wish to skip this subsection. However, for some explanation, the folder `h:\Matlab_Level2` should have been created and the file called `startup.m` was put in the folders

`h:\My Documents\MATLAB` and `h:\Documents\MATLAB`

(only the first is needed with the version in the labs but if Windows and/or newer versions of Matlab are introduced then the second location may become the one that is needed). When Matlab starts it searches for the existence of a file with this name and if it does exist it runs the commands in the file. The version of `startup.m` created by the set-up just changes the current working directory of Matlab to the folder `h:\Matlab_Level2` .

If you prefer to organise this differently, but are unsure as to what to adjust, then please ask.

1.4 `prefdir` and changing preferences

In the top bar part of the Matlab screen there is an icon called preferences which you can use to change a number of things such as the size of the fonts used in the different windows. Unfortunately, at the time of writing these notes the set-up at Brunel with Windows 10 and version R2019a is such that there is no roaming profile and thus any changes that you make to your preferences in one session will not be available in your next session on a different PC. You can determine whether or not this is the case by typing the following at the Matlab command prompt.

```
prefdir
```

If the folder shown starts with `C:\` then this is on the local hard disk and you preferences are not saved. If you make an effort to adjust things and you do not want to repeat this every time then you can make use of function files called `sav_pref.m` and `recover_pref.m` which will be in the folder `h:\mybin20` if you have run the set-up instructions. You need to copy these to the folder you usually use (or you add `h:\mybin20` to the Matlab path) to be able to use them. When you type the following at the Matlab command prompt the contents of the preferences folder is saved as a zip file on your `h:\` drive.

```
sav_pref
```

In a later Matlab session you would then type

```
recover_pref
```

For this to take affect you need to quit Matlab and start it again so that it uses the adjusted version of the folder.

I frequently have to change the font size depending on whether it is for my own use or I want a larger font as things are being projected. Please note that the default size is 10 although you can change this to any number from 8, 9, 10, 12, 14, 18, 24, 36 and 48.

2 Using a script file — the create, edit, run cycle

You can execute a command in Matlab by typing the command at the Matlab prompt `>>`, e.g. typing

```
pi
```

creates the output

```
ans =
    3.1416
```

When several commands are needed to solve a given task then it is usually more productive to type all the commands in an editor, save the file with a name ending in `.m` (e.g. `test1.m`) and then type the name of the file (without the `.m`) at the Matlab prompt. As an example, suppose that `test1.m` contains the lines

```
format compact
A=[4, 1, 1; 1, 4, 1; 1, 1, 4]
y=[1; 2; 3]
b=A*y
x=A\b
```

You can run all the instructions by just typing `test1` at the `>>` prompt. Alternatively, if the Matlab editor is being used and it still has this file open then you can just click on the green right arrow symbol on one of the top bars to do this. All the output of doing this is displayed in the Matlab command window and in the case of the above you should get the following.

```
A =
     4     1     1
     1     4     1
     1     1     4
y =
     1
     2
     3
b =
     9
    12
    15
x =
    1.0000
    2.0000
    3.0000
```

In the above the instruction `A\b` determines the solution of the linear system which in `MA2715` I write mathematically as $A\underline{x} = \underline{b}$. Matlab does some checks on the matrix first

and then does the computation using what it considers to be the best method to use. As part of MA2715 we will consider the main technique that is used for general linear systems when A is a $n \times n$ matrix.

The advantage of having all the instructions in a file is that you can easily add more lines, you can edit the file in other ways to change the problem and you can edit the file to correct mistakes. The point being that you do not have to type everything again. A typical mode of working is to create a few lines, to then check if they work, to edit to correct any mistakes, to check again and to repeat until the intended task is done correctly.

If you wish to implement things quickly then there is a plain text file containing many of the Matlab listings generated from this document available via the web page with the URL

`http://people.brunel.ac.uk/~icstmkw/ma2715/index.html`

If you cut and paste things from this file then please attempt to understand things in order to best prepare for the class test and the assignment.

3 Scalars, vectors and matrices of type double

The basic type in matlab is a matrix with each entry being of type double which means that on our system each entry is a number stored in 8 bytes which gives close to 16 decimal digits of accuracy. To illustrate this suppose that you type the following.

```
clear;
p=2*pi;
r=[1, 2];
x=r';
A=[1, 2, 3; 4, 5, 6; 7, 8, 9];
whos
```

This gives the output

```
  Name      Size             Bytes  Class     Attributes

  A         3x3                 72  double
  p         1x1                  8  double
  r         1x2                 16  double
  x         2x1                 16  double
```

Please note that as the statements start with `clear` these are the only things in the local workspace.

In the case of the scalar `p` you can actually write `p(1)` or `p(1,1)` to refer to `p` but it is simpler to just type `p`.

The quantities `r` and `x` are both vectors as one of the dimensions is just 1 and you can refer to the entries with just one index, e.g. `r(1)` and `x(2)`.

The quantity A is a matrix and you would usually refer to the entries using two index values, e.g. `A(1, 2)` or `A(2, 2)`. Please note however what happens when you put the following.

```
A
A(:)'
A(6)
```

This creates the output

```
A =
     1     2     3
     4     5     6
     7     8     9
ans =
     1     4     7     2     5     8     3     6     9
ans =
     8
```

This illustrates that the 9 entries of `A` are actually stored column-by-column and if only one index is used then we get the entry in the position in which it is stored.

Please note that in Matlab index values start at 1 (in some other programming languages index values start at 0).

With some functions and with some expressions you need to take care as to whether you have a row vector or a column vector. If you want to write some statements which cope with either but it is convenient to just have column vectors then you can achieve this with statements such as

```
r=r(:);
x=x(:);
```

With such statements a row vector is converted to a column vector and a column vector is unchanged.

As a final point for this section, if you need to know the number of rows and the number of columns in a matrix with these being saved for later use then you can use the `size()` function and type a statement of the form

```
[m, n]=size(A)
```

If you just need to know the number of entries then you can type

```
[nA, ~]=size(A(:));
```

Here `size()` returns two outputs but you only need one of them and the character `~` is the recommended way of doing this in Matlab which avoids having another variable that you do not need.

To avoid using `~` in the above example there are other ways of determining the number of entries. We can put

```
nA=size(A(:), 1);
```

and we can put

```
nA=length(A(:));
```

The function `length` gives the largest dimension and when we have a vector this is the number of entries.

4 Making decisions and loops

4.1 `if` and `if--else` statements

In programs the actions that we take often depends on whether or not some condition is true or false. If we only need to do something if the condition is true then the Matlab syntax is as follows.

```
if  testcondition
   statements to do if testcondition is true
end
```

A situation such as this arises if we have an angle t and we wish to change it to be in the range $[0, 2\pi)$ if it is outside this range. One way of doing this is to put the following in the case of a variable `t` which already exists.

```
if t<0 || t>=2*pi
    t=mod(t, 2*pi);
end
```

If the test is false then no action is done as `t` does need changing.

If you want something to be done if a condition is true and something else to be done if the condition is false then the Matlab syntax is

```
if  testcondition
   statements to do if testcondition is true
else
   statements to do if testcondition is false
end
```

A situation such as this arises if wish to solve a quadratic equation

$$ax^2 + bx + c = 0, \quad a \neq 0, \quad a, b, c \text{ are real.}$$

If we want to know when the roots are complex and we only want to compute them when they are real then we can have the following in the case when variables `a`, `b` and `c` already exist.

```
d=b^2-4*a*c;
sd=sqrt(abs(d));
if d>=0
    x1=(-b-sd)/(2*a);
    x2=(-b+sd)/(2*a);
else
    disp('quadratic has complex roots')
end
```

By first adding statements to create variables `a`, `b` and `c` try these statements.

4.2 The `elseif` statement

If you wish to successively do tests until something is true then you might want to make use of the `elseif` statement in the `if` blocks described in the previous subsection. (Please take care to distinguish "`elseif`" with "`else if`". The latter case starts a new `if`-block.) The syntax in this case with `elseif` is to have the following.

```
if  testcondition1
   statements if true
elseif  testcondition2
   statements if true
elseif  testcondition3
   statements if true
   ...
else
   statements if every test is false
end
```

The 2nd test is only considered if `testcondition1` gives false and the 3rd test is only considered if both the previous tests give false. The statements in the final `else` block are only done if all the previous tests evaluate to `false`.

4.3 `for`–loops

In many situations we need to repeat a set of instructions and the Matlab way of doing this which will be used most in this module is the for–loop. Examples of doing this are as follows.

```
for k=1:8
  fprintf('%d, %2d, %3d\n', k, k^2, k^3);
end
```

Here the first part of `fprintf` is the formatting instruction with `%d`, `%2d` and `%3d` being the format to use for `k`, `k^2` and `k^3` respectively. Create a script file with these 3 lines and check that it creates the following output.

```
1,  1,   1
2,  4,   8
3,  9,  27
4, 16,  64
5, 25, 125
6, 36, 216
7, 49, 343
8, 64, 512
```

Here `1:8` is the row vector `[1, 2, 3, 4, 5, 6, 7, 8]` and `k` successively takes each value in the list. In a set-up such as this with equally spaced values the row vector is not actually created as Matlab knows that `k` starts with `1` and increases to `8` in steps of `1`.

If we want steps other than `1` then we can have the following which you may wish to try.

```
for k=1:2:8
   fprintf('%d, %2d, %3d\n', k, k^2, k^3);
end
fprintf('\n')
for k=8:-3:1
   fprintf('%d, %2d, %3d\n', k, k^2, k^3);
end
```

You should get the following output.

```
1,  1,   1
3,  9,  27
5, 25, 125
7, 49, 343

8, 64, 512
5, 25, 125
2,  4,   8
```

In both cases it is the middle value which is the step that is used.

We can also explicitly use a given list as in the following example to sum the squares of all prime numbers which are less than 20.

```
s=0;
for p=[2, 3, 5, 7, 11, 13, 17, 19];
  s=s+p^2;
end
```

Exercise

In Matlab there is a built-in function called `primes` such that when we have the statements

```
p=primes(n);
np=length(p);
```

we obtain a row vector called `p` which contains all the prime numbers which are less than or equal to `n` with the second statement giving the length of `p`. By using these functions and an appropriate `for` loop determine how many prime numbers lie in each of the 10 equal intervals $(0, 1000]$, $(1000, 2000]$, ..., $(9000, 10000]$.

With brief output the numbers that you should get are as follows.

```
168 135 127 120 119 114 117 107 110 112
```

4.4 `break` and `continue` statements within loops

Often we want to leave a loop early or we want to immediately jump to the next case usually as a consequence of same outcome. In Matlab the `break` statement is the way to leave a loop and the `continue` statement is the way to skip statements in a loop and go to the next case. We consider next two examples where these statements are useful to use.

4.4.1 Illustrating the use of the `break` statement with Newton's method and with the Secant method

As an example of using the `break` statement consider Newton's method for attempting to solve $f(x) = 0$ for a given function f. The method involves choosing a starting point x_0 in some way and computing

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \ldots$$

When you implement this on a computer you should consider a maximum number of steps to take (whether it has converged or not), it is efficient to finish as soon as sufficient accuracy is obtained and a robust program will also abort if the iteration appears to be diverging. If we restrict to the cases of not doing too many iterations and stopping early if possible when the function is $f(x) = x^2 - 2$ then in Matlab we can have the following.

```
format long
x=1;
for n=1:20
  d=(x^2-2)/(2*x);
  x=x-d;
  fprintf('n=%2d, x=%21.18f, d=%10.2e\n', n, x, d);
  if abs(d)<1e-12
      break
  end
end
```

Try this and check that it creates the following output.

```
n= 1, x= 1.500000000000000000, d= -5.00e-01
n= 2, x= 1.416666666666666741, d=  8.33e-02
n= 3, x= 1.414215686274509887, d=  2.45e-03
n= 4, x= 1.414213562374689870, d=  2.12e-06
n= 5, x= 1.414213562373095145, d=  1.59e-12
n= 6, x= 1.414213562373094923, d=  1.57e-16
```

With a set-up such as this we either do the instructions in the loop 20 times or we leave the loop when the change in the values is sufficiently small which is 6 in this example.

The secant method for attempting to solve $f(x) = 0$ for this given function f involves instead the iteration

$$x_{n+1} = x_n - \left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right) f(x_n), \quad n = 1, 2, \ldots.$$

Change the previous program to implement this method taking $x_0 = 1$ and $x_1 = x_0 + \sqrt{\epsilon}$ where ϵ is the machine precision which in Matlab is given by the variable `eps`. The values that I obtained are as follows.

```
n= 0, x= 1.000000000000000000, f(x)= -1.00e+00
n= 1, x= 1.000000014901161194, f(x)= -1.00e+00
n= 2, x= 1.499999996274709702, f(x)=  2.50e-01
n= 3, x= 1.400000001192092780, f(x)= -4.00e-02
n= 4, x= 1.413793103501431059, f(x)= -1.19e-03
n= 5, x= 1.414215686274062245, f(x)=  6.01e-06
n= 6, x= 1.414213562057320628, f(x)= -8.93e-10
n= 7, x= 1.414213562373094701, f(x)= -8.88e-16
```

4.4.2 Illustraing the `continue` statement: Examples of matrices with real positive eigenvalues

As an example of using a `continue` statement consider the following case of attempting to generate 3 random matrices with integer entries in the interval $[-5, 5]$ and with only displaying the matrices which have real eigenvalues which are positive.

```
rand('seed', 1);
count=0;
for k=1:500
  % generate the random matrix and get the eigenvalues
  A=randi([-5, 5], 2, 2);
  mu=eig(A);

  % do not consider further if the required conditions
  % are not met
  if abs(imag(mu(1)))>0 || abs(imag(mu(2)))>0 ...
     || mu(1)<0 || mu(2)<0
      continue;
  end

  % display what has been found and stop when we have
  % 3 matrices
  fprintf('The next matrix has positive eigenvalues.\n');
  A
  fprintf('The eigenvalues are %e and %e\n', mu(1), mu(2));
  count=count+1;
  if count==3
      break;
  end
end
fprintf('count=%d after %d attempts\n', count, k);
```

The above could also been done with an if–block but the use of the `continue` statement avoids having too many statements in such a block. The choice of 500 here was arbitrary. To make the program more robust we should also test when the loop has finished if `count` is equal to 3 or less than 3 and warn if the later is the case although the version given does show the final value of `count`.

Exercise

Let

$$f(x) = \tan(x) - x, \qquad \text{and note that } f'(x) = \sec^2(x) - 1.$$

Modify the previous program involving Newton's method to attempt to solve $f(x) = 0$ in the interval $(m\pi, m\pi + \pi/2)$ for $m = 1, 2, 3, 4, 5$. To have a good starting point we note that when x is close to $\alpha = m\pi + \pi/2$ the Taylor expansions of sine and cosine give

$$\begin{aligned} \sin(x) &= \sin(\alpha) + (x-\alpha)\cos(\alpha) + \cdots \approx \cos(m\pi) = (-1)^m, \\ \cos(x) &= \cos(\alpha) - (x-\alpha)\sin(\alpha) + \cdots \approx -(x-\alpha)\cos(m\pi) = (-1)^m(\alpha - x) \end{aligned}$$

and thus

$$\tan(x) = \frac{\sin(x)}{\cos(x)} \approx \frac{1}{\alpha - x}.$$

Now

$$\frac{1}{\alpha - x} = x \quad \text{gives } x^2 - \alpha x + 1 = 0.$$

For the starting point x_0 in the Newton iteration we take the solution of this which is close to α, i.e. we take

$$x_0 = \frac{1}{2}\left(\alpha + \sqrt{\alpha^2 - 4}\right).$$

My output in the case when $m = 5$ is as follows.

```
Attempting to solve tan(x)-x=0 in (5*pi, 5*pi+pi/2)
n= 0, x=17.220689911989524
n= 1, x=17.220755345744934, d= -6.54e-05
n= 2, x=17.220755271930862, d=  7.38e-08
n= 3, x=17.220755271930770, d=  9.36e-14
x/pi= 5.481537
```

5 Matrix operations, entry-wise operations and sub-matrices

5.1 Matrix operations

When you first study matrices and vectors you learn about matrix multiplication and this is what Matlab does when we put `A*x` or `A*B` or `A^2` when `A`, `B` and `x` have compatible shapes. Thus the following instructions are valid.

```
A=[1 2 3
   3 2 1
   4 0 2];
x=[1; 1; 1];
b=A*x
r=(x'*b)/(x'*x)
C=A^2
```

As there is no semi-colon at the end of the last 3 statements the following output is created.

```
b =
     6
     6
     6
r =
     6
C =
    19     6    11
    13    10    13
    12     8    16
```

Note that in the expression for `r` the part `x'` is the transpose of `x` and the product of a row vector and a column vector gives a scalar.

There are many functions in Matlab which take matrices as arguments and there has already been examples using `eig(A)` to get the eigenvalues of a matrix in the previous section. To illustrate further things about matrix operations and also to check things that you may have done in earlier modules using hand calculations consider the following statements.

```
A=[4, 1, 1;
   1, 4, 1;
   1, 1, 4];
p=poly(A)
E=p(1)*A^3+p(2)*A^2+p(3)*A+p(4)*eye(3)
```

The output created is as follows.

```
p =
     1   -12    45   -54
E =
     0     0     0
     0     0     0
     0     0     0
```

The statement `poly(A)` gives the coefficients of the characteristic polynomial with the coefficient of the highest power given first, i.e. the polynomial is

$$\det(tI - A) = t^3 - 12t^2 + 45t - 54.$$

The statement for `E` evaluates the matrix polynomial

$$A^3 - 12A^2 + 45A - 54I.$$

There is actually a function in Matlab which does this and thus the last line can be replaced by

```
E=polyvalm(p, A)
```

The result that `E` is the zero matrix is as a consequence of what is known as the Cayley Hamilton theorem which states that every matrix satisfies its own characteristic equation. That is, in the absence of rounding errors, we would observe the same outcome for E if A is replaced by any other square matrix.

One point to note in connection with the previous matrix is illustrated by the following statements.

```
format compact
A=[4, 1, 1;
   1, 4, 1;
   1, 1, 4];
B=A-6
C=A-6*eye(3)
```

This generates the following output.

```
B =
    -2    -5    -5
    -5    -2    -5
    -5    -5    -2
C =
    -2     1     1
     1    -2     1
     1     1    -2
```

In the first case `6` is subtracted from every entry whilst to just subtract `6` from each diagonal entry we need to create the matrix `6*eye(3)`.

Exercises

1. The matrix A given above with 4 on the diagonal and all off-diagonal entries being equal to 1 is on an exercise sheet of `MA2715`. Let $G = A/6$. Compute the matrices $3G^2$, $3G^4$, $3G^8$, $3G^{16}$, $3G^{32}$ and $3G^{64}$. Can you explain the answers?

2. Consider now the matrix G given by

$$G = \frac{1}{6}\begin{pmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{pmatrix},$$

   i.e. it is the same as in the previous question except that the $1,3$ and $3,1$ entries are 0. Compute the matrices G^2, G^4, G^8, G^{16}, G^{32}, G^{64}, G^{128} and G^{256}. Can you explain the answers?

3. In other modules you meet the geometric series which is such that when $|z| < 1$ we have
$$\frac{1}{1-z} = (1-z)^{-1} = 1 + z + z^2 + \cdots + z^n + \cdots .$$
A similar result holds for square matrices in that if the spectral radius of a matrix E is such that $\rho(E) < 1$ then
$$(I-E)^{-1} = I + E + E^2 + \cdots + E^n + \cdots .$$
(Recall that $\rho(E) = \max\{|\lambda_i| : \lambda_i \text{ is an eigenvalue of } E\}$.) To check this in a particular case let G be as in question 2 and let $E = I - G$ so that $G = I - E$ and let
$$S_m = \sum_{n=0}^{m} E^n.$$
Compute G^{-1} using the Matlab function `inv` and compare with entries of the sequence of matrices $S_1, S_2, \ldots$.

5.2 Entry-wise operations

Matlab also has what are known as entry-wise operations which you get by using `.*`, `./` and `.^`. These are useful in evaluating a function at many different points using one statement and to illustrate this consider the following statements.

```
t=linspace(0, 2*pi, 500);
y=sin(t)+0.3*exp(-0.1*t.^2).*sin(10*t);
figure(2)
plot(t, y)
```

The first statement creates 500 equally spaced points starting at 0 and finishing with 2π. The function
$$f(t) = \sin(t) + 0.3\exp(-0.1t^2)\sin(10t)$$
is evaluated at each of the `500` values in forming `y`. To do this instead using for–loops involves the following 4 statements with in this case `z` being created.

```
z=zeros(1, 500);
for k=1:500
  z(k)=sin(t(k))+0.3*exp(-0.1*t(k)^2)*sin(10*t(k));
end
```

5.3 Using the colon notation

You can extract parts of a vector or matrix by using the colon notation to indicate the list of rows and/or columns that you want and the following statements illustrate some

of the possibilities.

```
A=[  1  4  8  9  2;
     0  1  9  8  7;
     3  1  4  5  2;
     1  9  7  5  2;
     0  1  8  6  4];
r3=A(3, :)
c2=A(:, 2)
A3=A(2:4, 2:4)
E=A(4:end, 4:end)
d=diag(A)
D=diag(d)
```

The output generated is as follows.

```
r3 =
     3     1     4     5     2
c2 =
     4
     1
     1
     9
     1
A3 =
     1     9     8
     1     4     5
     9     7     5
E =
     5     2
     6     4


d =
     1
     1
     4
     5
     4
D =
     1     0     0     0     0
     0     1     0     0     0
     0     0     4     0     0
     0     0     0     5     0
     0     0     0     0     4
```

We get the 3rd row, the 2nd column, an interior sub-matrix and a submatrix in the bottom right hand corner to form `E`. In the expression for `E` the part `4:end` corresponds

to `4:5` when the matrix is 5×5.

6 Function files and anonymous (1-line) functions

In section 2 we discussed the benefit of creating a script m-file which is just a file containing Matlab statements. A function m-file or simply a function file is similarly a file containing Matlab statements with the first line of the file starting with the word `function`. Functions can take input arguments, they can produce output arguments and, unless you use a global statement, the internal variables are local to the function.

The syntax of the function statement is as follows.

`function [`*`output_arguments`*`]=fun_name(`*`input_arguments)`*

This should be the first line of a file called `fun_name.m`.

As a short example a file with the name `f7.m` could be as follows.

```
function y=f7(x)
y=sin(x)+sin(3*x)/3+sin(5*x)/5+sin(7*x)/7;
```

When this exists and it is in the Matlab path any other m-file can use the function. For example, we can plot the function over a $[0, 3\pi]$ interval with statements such as the following.

```
x=linspace(0, 3*pi, 300);
figure(3)
plot(x, f7(x));
```

If this function is not actually needed in other applications then we can avoid the creation of another file by using instead a one-line or anonymous function in the file that is using the function. Hence if we only want to plot `f7` then we can avoid creating `f7.m` and for the entire script file we can have the following.

```
f7 =@(x) sin(x)+sin(3*x)/3+sin(5*x)/5+sin(7*x)/7;
x=linspace(0, 3*pi, 300);
figure(4)
plot(x, f7(x));
```

In the case of functions which need more than 1 statement we should create function files and as an example involving solving a quadratic equation we can have the following in a file called `solve_quad.m` .

```
function [x1, x2]=solve_quad(a, b, c)
% [x1, x2]=solve_quad(a, b, c)
% Output arguments x1 and x2 solve a*x^2+b*x+c=0

d=b^2-4*a*c;
sd=sqrt(abs(d));
if d>=0
    x1=(-b-sd)/(2*a);
    x2=(-b+sd)/(2*a);
else
    x1=(-b-1i*sd)/(2*a);
    x2=(-b+1i*sd)/(2*a);
end
```

This has 3 input arguments and 2 output arguments. To use the function we can put the following statements.

```
[x1a, x2a]=solve_quad(1, 3, 2)
[x1b, x2b]=solve_quad(1, 1, 1)
[x1c, x2c]=solve_quad(1, 4, 1)
```

The output created is as follows.

```
x1a =
    -2
x2a =
    -1
x1b =
  -0.5000 - 0.8660i
x2b =
  -0.5000 + 0.8660i
x1c =
   -3.7321
x2c =
   -0.2679
```

The assignment tasks for MA2895 are likely to involve the creation of function files.

7 Basic two-dimensional plots

7.1 The `plot` and the `figure` commands

There have been several examples already involving two-dimensional plots. In its most basic form a two-dimensional plot is obtained by the statement

```
plot(x, y)
```

provided `x` and `y` are both vectors of the same length. If no figure exists then 'Figure 1' is created and the graph is shown in the figure. If a figure already exists then that figure is replaced by the latest plot. If you want the plotting to be done in a specific figure and you would like several figure windows still to be available then you should use the command `figure`. As an example to create 3 figures involving the graphs of $y = \sin(2x)$, $y = \sin(4x)$ and both together we can have the following statements.

```
x=linspace(0, 2*pi, 200);
figure(20)
plot(x, sin(2*x));

figure(21)
plot(x, sin(4*x));

figure(22)
plot(x, sin(2*x), '--', x, sin(4*x))
```

7.2 Using `clf`, `hold on` and `hold off`

Sometimes we want to have things on the same plot but it is awkward to create everything first before any plotting instructions are given. To cope with this we can use the `hold on` and `hold off` mechanism as illustrated by the following statements.

```
% create uniformly spaced points on the unit circle
n=12;
t=1:n;
x=cos(2*pi*t/n);
y=sin(2*pi*t/n);

% join every point with every other point
figure(10)
clf
hold on
for i=1:n
  for j=1:i-1
    plot([x(i), x(j)], [y(i), y(j)]);
  end
end
axis equal
axis off
hold off
```

The graph generated is shown in figure 7.1 on page 28. The command `clf` in the statements above clears any graphics in the figure window, e.g. from a previous run of the statements.

There are many things that you can do to control the appearance of a plot and you usually need to look these up if they involve commands that you do not use regularly. As a final example of what can be done consider the following statements in which the line thickness is increased from the default and the size of all numbers and labels are increased.

```
% create the vectors for exp(x)
x1=linspace(-2, 2);
y1=exp(x1);

% create the vectors for log(x)
x2=linspace(exp(-2), exp(2));
y2=log(x2);

% create the data for the y=x line
x3=[-2, exp(2)];

% plot all the curves together with an increased line width
figure(12)
plot(x1, y1, x2, y2, x3, x3, '--', 'LineWidth', 3)

% increase the size of the numbers on the axis
set(gca, 'FontSize', 14);

% add other things at 16pt
text(2.2, exp(2), 'y=exp(x)', 'FontSize', 16)
text(exp(1.6), 1.2, 'y=log(x)', 'FontSize', 16)
xlabel('x', 'FontSize', 16)
ylabel('y', 'FontSize', 16)
title('Plots of exp(x) and log(x)', 'FontSize', 16)
```

The graph generated is shown in figure 7.2 on page 28. Increasing the thickness and the size of fonts is often needed in presentations and when a graph is to be shrunk when it is included in the paper version.

Exercise

Try and add additional lines to the script which generated the graph shown in figure 7.1 on page 28 to create the graph shown in figure 7.3 on page 29.

To help you with this task you may wish to create and use the following function file.

```
function p=line_intersect(r1, r2, r3, r4)
%%function p=line_intersect(r1, r2, r3, r4)
% p is the point of interection of the line r1 to r2 with r3 to r4

A=[r2(:)-r1(:), r3(:)-r4(:)];
b=r3(:)-r1(:);
c=A\b;
p=r1(:)+c(1)*(r2(:)-r1(:));
```

This function determines the point of intersection of the line that passes through $\underline{r}_1$ and $\underline{r}_2$ with the line that passes through $\underline{r}_3$ and $\underline{r}_4$ in the case that the lines are not parallel. The point which is on both lines is such that

$$\underline{r}_1 + s(\underline{r}_2 - \underline{r}_1) = \underline{r}_3 + t(\underline{r}_4 - \underline{r}_3)$$

for some s and t. As a linear system with a column-by-column representation of the matrix we have

$$\begin{pmatrix} \underline{r}_2 - \underline{r}_1, & \underline{r}_3 - \underline{r}_4 \end{pmatrix} \begin{pmatrix} s \\ t \end{pmatrix} = \underline{r}_3 - \underline{r}_1.$$

To do the task you need to first decide on which 12 pairs of lines should be used when getting the intersection of the lines to generate the points on the "inner part". Once the points are generated you plot the polygon which goes through these points.

7.3 Saving the graphics to get a high quality PDF version

To get a high quality paper copy of the graphics as a stand-alone item or to save the graphics in some way so that it can be included in a document can be achieved by using the `print` command. My preferred way of doing things, which is the method that I used to create this document, is to use the `print` command to create what is known as an encapsulated postscript file which is vector format which is scalable. A PDF version was then created using a free program known as `epstopdf`. The program `epstopdf` is usually part of TeX distributions. To get the plot involving the points of unity on the unit circle you just need to add the following two lines to the script file.

```
print('circle_12_points.eps', '-depsc')
system('epstopdf circle_12_points.eps');
```

Together these create the files `circle_12_points.eps` and `circle_12_points.pdf` .

If you do not have `epstopdf` then another way to get a suitable `pdf` file is to use the function file `print2pdf.m` which is in `h:\mybin20`. To use this you first need to copy the file to the current directory or add `h:\mybin20` to the Matlab path. Once the command is available you just put

```
print2pdf('circle_12_points.pdf');
```

The only difference between this way of doing things and the previous statements is in the amount of white space surrounding the graphics.

7.4 Shading in a region, the `fill` function

Sometimes you want to shade in a region with a given colour. To do this in Matlab you need to create a closed polygon and use the `fill` function. As an example, to create the unit disk and to fill-in the region with the colour blue you can do the following.

```
t=linspace(0, 2*pi, 201);
x=cos(t);
y=sin(t);
figure(91)
fill(x, y, 'b')
axis equal
```

In this example the polygon has 201 points with `(x(1), y(1))` being the same as `(x(201), y(201))` to within rounding error when these statements are run.

As a longer example try the following which is a crude attempt at showing traffic lights. The graphical output of running this script is shown in figure 7.4 on page 29.

```
% create the points for any of the circles
t=linspace(0, 2*pi, 201);
r=1.5;
xc=r*cos(t);
yc=r*sin(t);

% create the points for the outer rectangle
xouter=[-2, 2, 2, -2, -2];
youter=[-2, -2, 12, 12, -2];

% create the points for the positions with
% xp() for 4 possibilites and yp() for the red, amber, green positions
xp=[0, 6, 12, 18];
yp=[0, 5, 10];

% define the colours in rgb form if necessary
amber=[255, 151, 0]/255;
gray=[17, 17, 17]/255;
col={'g', amber, 'r'};

% start the figure with equal scales and with no axis shown
figure(101)
clf
axis equal
axis off
hold on

% just red case
fill(xp(1)+xouter, youter, gray);
fill(xp(1)+xc, yp(3)+yc, col{3});

% just red and amber together
fill(xp(2)+xouter, youter, gray);
```

```
fill(xp(2)+xc, yp(2)+yc, col{2});
fill(xp(2)+xc, yp(3)+yc, col{3});

% just green case
fill(xp(3)+xouter, youter, gray);
fill(xp(3)+xc, yp(1)+yc, col{1});

% just amber case
fill(xp(4)+xouter, youter, gray);
fill(xp(4)+xc, yp(2)+yc, col{2});
hold off
```

7.5 The size of the figure window

If you find the default size of the figure window to be too small then the following can be done to get a larger size for an individual figure.

```
fsz=get(0, 'ScreenSize')
f10=figure(10);
set(f10, 'Position', 0.8*fsz);
% .. add plotting instructions
```

In this case the figure window is 0.8 the size of the entire screen.

7.6 Closing all the `figure` windows

If you wish to close all the `figure` windows without exiting Matlab then you type

```
close all
```

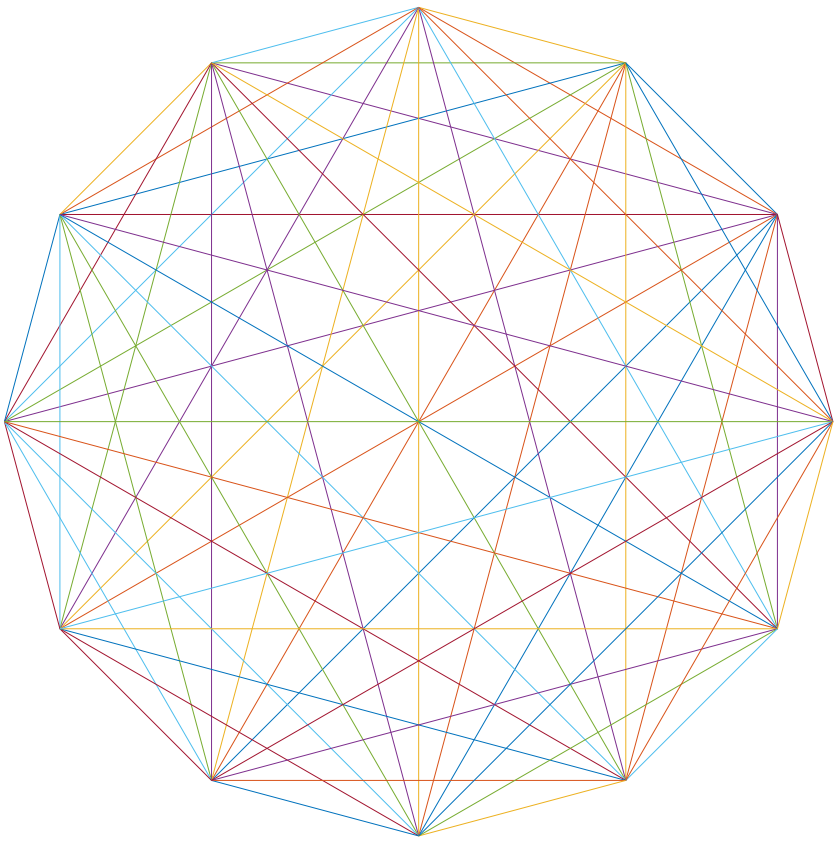

Figure 7.1: Line segments joining all the roots of unity in the complex plane when $n = 12$.

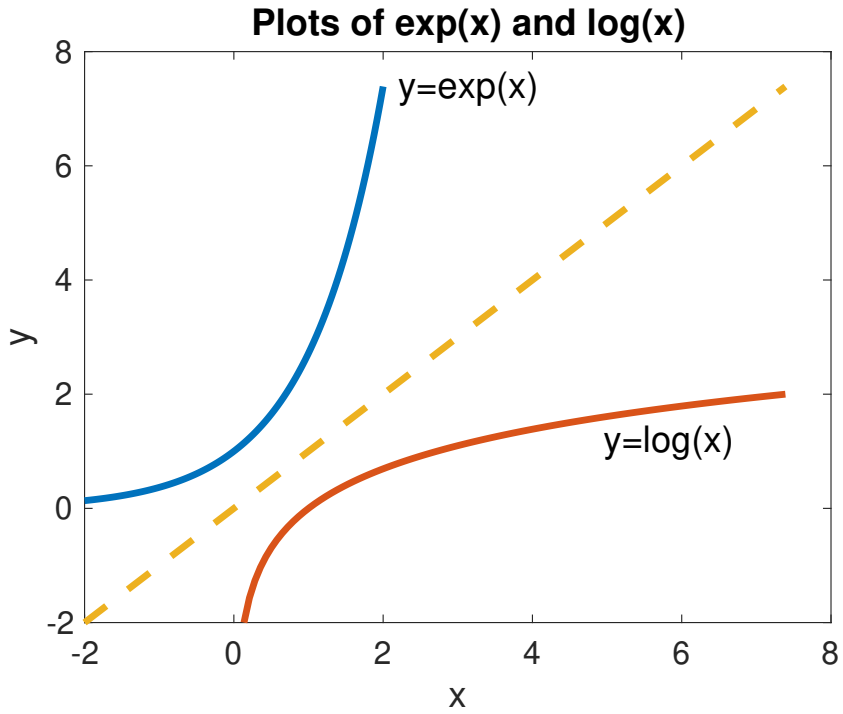

Figure 7.2: An example with thicker lines and larger labels with the graph showing that the plot of $y = \mathrm{e}^x$ is the mirror image of the plot of $y = \log(x)$ in the line $y = x$.

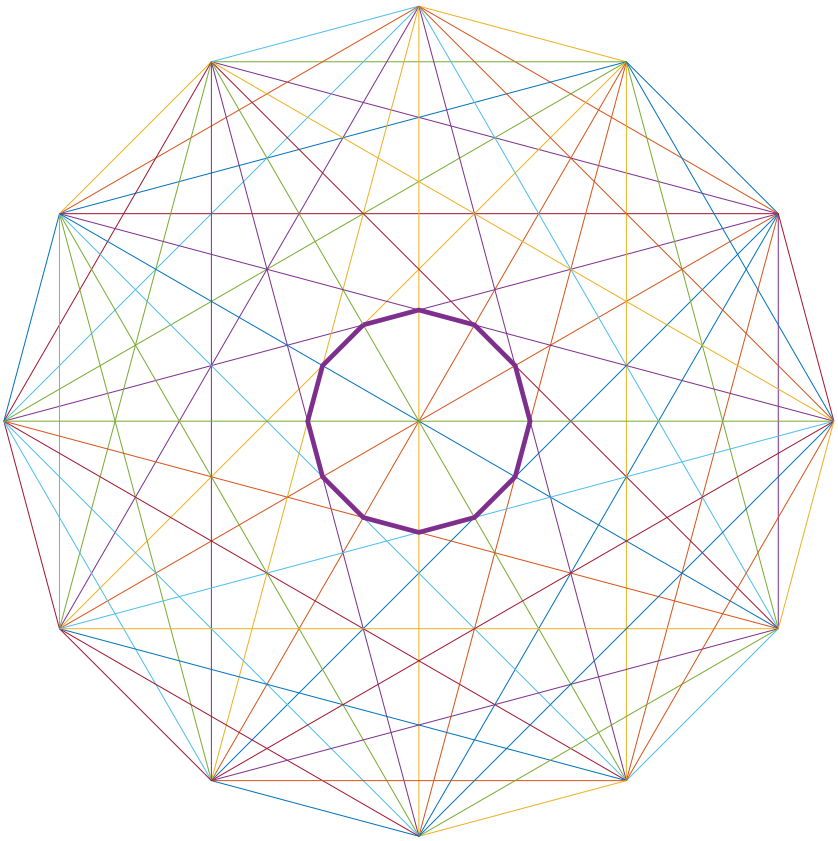

Figure 7.3: Line segments joining all the roots of unity in the complex plane when $n = 12$ with an inner polygon with 12 sides added.

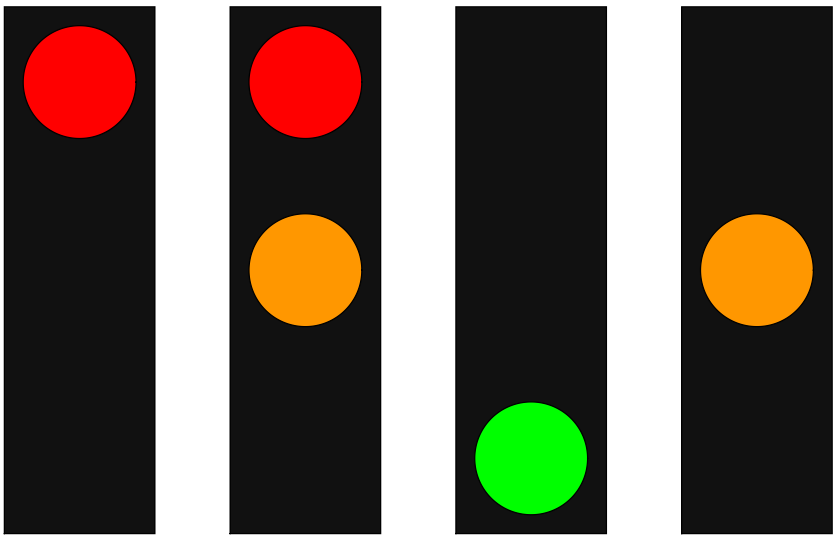

Figure 7.4: The 4 possibilities for traffic lights.

8 Remarks about doing things efficiently in Matlab

Remember to allocate the space for a vector first

When you are creating a vector or matrix in Matlab it is recommended that you allocate the space first if this is known when you start. To illustrate what can happen when you do not do this create a script file which contains the following and run it and note the times taken.

```
clear
n=1e6;
% slow or very slow
tic
for k=1:n
  a(k)=1/k;
end
toc

% quicker
clear a;
tic
a=zeros(n, 1);
for k=1:n
  a(k)=1/k;
end
toc

% another quick version
clear a;
tic
a=1./(1:n);
toc
```

Tri-diagonal matrices — using `spdiags`

Create and run the following script file which is about using `spdiags` to create banded matrices. The first part considers a small matrix so that it is easy to compare with the full storage version. The second part then compares the solution time for a larger matrix when sparse storage is used instead of full storage.

```
% small version for display
n=7;
o=ones(n,1);
T=spdiags([o 4*o o], -1:1, n, n);
F=full(T)
whos T F

% larger sparse and full version
n=5000;
o=ones(n, 1);
T=spdiags([o 4*o o], -1:1, n, n);
F=full(T);
whos T F
b=T*o;

% solve using sparse and full version
tic
x=T\b;
toc

tic
x=F\b;
toc
```

In the above script the matrix created by `[o 4*o o]` has 3 columns and the positions of the columns is specified by the part `-1:1` which is `-1, 0, 1`. The value `-1` means the sub-diagonal, the value `0` means the diagonal and the value `1` means the super-diagonal. The sub-diagonal and super-diagonal have one less entry than the diagonal and some further investigation is needed to work out which entry is not used. Create and run the following script file.

```
n=5;
a=(1:n)';
T=spdiags([a, 10*a, -a], -1:1, n, n)
full(T)
```

Can you determine which entry is not used for the sub-diagonal and which entry is not used for the super-diagonal?

Tri-diagonal matrices will appear in `MA2715` when the finite difference method is used to approximately solve the two-point boundary value problem.

9 Some more advanced features

9.1 Accessing fields from a struct in the case of the `whos` statement

Type

```
whos
```

at the Matlab command prompt. This shows all the quantities in the base workspace. Also type

```
t=whos
```

In the R2019a version of Matlab on Windows, and with other recent versions, this shows the following.

```
t =
  29x1 struct array with fields:
    name
    size
    bytes
    class
    global
    sparse
    complex
    nesting
    persistent
```

We have an array with each entry being type `struct`. The number `29` is because I had that number of quantities in the workspace when the command was run. To illustrate how to extract the fields create the following function file.

```
function s=size_ws(t)
% s=size_ws(whos) returns the number of bytes used
% in the workspace with the details in t

n=size(t, 1);
s=0;
for k=1:n
  s=s+sum( t(k).bytes(:) );
end
```

Here as `bytes` is one of the fields the value is obtained by putting `.bytes`, i.e. with a dot before the field name.

When the function is complete type

```
size_ws(whos)
```

at the Matlab command prompt to use it.

Do you know why the `whos` command was not put in the function file?

When the function is used in this way it determines the total amount of space being used in the workspace from which it is called.

9.2 An example of creating and using a cell array

In a vector or a matrix every entry is the same type. With cell arrays the entries can have different types and to set these up and to access the entries the syntax involves using the brackets `{` and `}`. As a short example create and run a script file which contains the following.

```
f1 =@(x) sin(x/6)-0.5;
f2 =@(x) tan(x/4)-1;
f3 =@(x) cos(x/3)-0.5;

allf={f1; f2; f3};
m=size(allf, 1);
for k=1:m
  c=fzero(allf{k}, 3.0);
  fprintf('root of function number=%d at %18.15f\n', k, c);
end
```

Here `fzero` is a Matlab function which attempts to find a zero of the function given as the first argument using the second argument as the guess. The quantity `allf` is a cell array with the entries being function handles. As these statements show, we are able to test all the functions in a loop and we have avoided the need to repeat the statement to test each of `f1`, `f2` and `f3`. I use a mechanism similar to this to do tests on a collection of function files when I mark assignments.

10 Getting help and some text books

As stated at the start, these notes only describe a few things that you can do with Matlab and almost certainly you will need at some stage to use the Matlab help facilities, refer to text books or search online. In Matlab the commands `help` and `doc` can be used to get information about a given function and the command `lookfor` can be used to search the comments in all the files in the Matlab path for your search word.

There is no core Matlab book that you have to buy for this module although I do myself own several books and among those that I consult sometimes are the following.

1. Timothy A. Davis. *MATLAB primer*. CRC Press, eighth edition, 2011.
`QA297.D38`.

2. Brian D. Hahn and D. T. D. T. Valentine. *Essential MATLAB for engineers and scientists*. Academic Press, 5th edition, 2013.
`QA297.H345`. Older editions are likely to also exist in the Brunel library.

3. D. J. Higham and Nicholas J. Higham. *MATLAB guide*. Society for Industrial and Applied Mathematics, 2nd edition, 2005.
`QA297.H525`

   (I also have the 3rd edition published in Feb 2017 but this is not yet in the Brunel library.)

11 A few other points

1. To learn most things you need to practice and this is particularly the case with programming in any language. Often when you practice things do not work properly on the first attempt, which can be frustrating, but you do learn from these situations when you are able to correct the mistakes.

2. It is best to attempt to layout your statements in a reasonably uniform way most of the time although different people have their preferred styles. Having a good layout will not affect how the program runs but it may make it easier to follow what is going on and it may make you more productive in the long run. For example, I suggest that you usually just have 1 statement per line, if the statement is long then you spread it over several lines and you uniformly indent statements in blocks such as `if`–`else` constructions and `for`–loops. There are function files called `smart_file.m` and `smart_dir.m` in the folder `h:\mybin20` which when run use the Matlab editor to uniformly indent a single file or all m-files in a directory which may help to automate this process.

   As an example of spreading a long statement over two lines we can have the following.

```
y=sin(x)+sin(3*x)/3+sin(5*x)/5+sin(7*x)/7...
  +sin(9*x)/9+sin(11*x)/11+sin(13*x)/13+sin(15*x)/15;
```

   The 3 dots at the end of the line indicates that the statement continues on the next line.

3. Use comments to help explain what a group of statements should do. Personally I prefer comments which appear before the statements that they describe as I have used in these notes. As in item 2 I also recommend that you avoid having single comment lines which are very long. If there is a lot to put in a comment then spread it over several comment lines.

4. If you wish to save part of what is shown in the command window then you can use the `diary` command in a set-up of the following form.

```
diary my_output.txt
 ....matlab statements
diary off
```

Everything which is displayed in the command window after the first `diary` statement is also sent to the file with the name given. If the file does not exist then it is created but otherwise it is appended to the file.

5. Matlab has the command `checkcode` which you can use to analyse your files with suggestions given as to how it may be improved. In the case of a m-file called `test123.m` you should type the following at the Matlab command line.

   ```
   checkcode('test123.m')
   ```

   If you have an older version of Matlab then `checkcode` may not be available but `mlint` can be used instead. For example, if you use the older R2010b version then you only have `mlint`.

12 Answers to the exercises

These will not be available at the start of the term.