MA2915
Linear and Numerical Methods

Matlab component, Autumn 2003
by M. K. Warby

Aims

The aims of the 20 credit module MA2915 as stated on the syllabus are as follows:

(a) To introduce numerical methods for ODE, and their implementation in Matlab, using direct solution techniques from numerical linear algebra.

(b) To develop a knowledge of, and facility with, methods of linear mathematics, like transforms, orthogonal series, and their applications, particularly to the solution of ordinary (ODEs) and partial (PDEs) differential equations.

In part (a) the aims of the Matlab component are as follows.

(i) To further develop the programming skills you acquired in the level one module MA1051S Numerical Methods. This involves improving your understanding of algorithms and in giving you more practice in implementing algorithms.

(ii) To enable you to implement numerical schemes for approximately solving systems of ordinary differential equations that you meet in the other part of MA2915.

(iii) To introduce the graphical capabilities of Matlab to help in your presentations.

Assessment

20% of the assessment of MA2915 is from the Matlab component of the module. You will be given the details of these during the lectures.

Organisation

All the Matlab sessions of MA2915 will be in room M124. You are expected to attend a Matlab session every two weeks during the autumn although as the seminar time and the lab times for MA2915 are different you can attend more sessions than this if there are working computers available.

The extent of these notes

The notes presented here are intended to supplement the notes you obtained in your level one module MA1051S Numerical Methods and as such are restricted to just a few things. We comment on what you have to do on the current computer set-up at Brunel in order to use Matlab and we describe a few of the commands that you will need to become familiar with to do your work in Matlab. The commands that are described are the control statements such as `if`, `for` and `while` statements and we also describe how to construct and use function files. All of these were covered in MA1051S. There is also information on solving linear equations and in particular on solving systems $T\underline{x} = \underline{b}$ involving a tridiagonal matrix T which will be new material and which is part of the syllabus for MA2915.

Other sources of reference

There are many books on Matlab although I am not going to recommend any core text. However you may wish to note the following books that I refer to quite frequently.

Hanselman, D. and B. Littlefield: Mastering Matlab 6, A Comprehensive Tutorial and Reference, Prentice Hall, New Jersey, 2001,
`QA297.H293`

Higham, D. J. and N.J. Higham: Matlab guide, SIAM (Society for Industrial and Applied Mathematics), Philadelphia, 2000,
`QA297.H5217`

Hahn, B: Essential MATLAB for Scientists and Engineers, Butterworth Heinemann, second edition, 2002.
This was mentioned in the module MA1051S.

Matlab itself is quite a good source of reference, see the comments on the help facilities given below.

You may also which to note that there has recently been released (that is in September 2003) a student edition of Matlab. The advert for this says that it has the ISBN 0-9672195-9-0, it costs £50 and there are versions for the MAC, Windows and Linux.

1 Your set-up and using Matlab at Brunel

At Brunel the computer centre has a site license for Matlab to be used simultaneously by up to 50 users on any platform. Currently the version implemented at Brunel is version 6.1 with the latest version of Matlab being about version 6.5 at this time. Version 6.1 is what you used at level 1 except that now you will be using it from a Windows 2000 platform instead of a Windows NT platform. This makes things slightly more complicated initially as you have file space on the Windows 2000 servers (this will be your `h:` drive) and you will also have the files that you used last academic year on another server which you can mount at the same time. In the description that follows this area which contains your files of last year will be set-up as your `u:` drive. It is this file space area that you have when you can connect to a UNIX computer such as `molnir` using the program `putty`. Before we describe how to start Matlab we first describe some aspects of this new set-up at Brunel. This includes information about setting up drives under windows to get and use your old files and with getting various utilties and files from us. We also comment on modifications to your unix environment which we would like you to do in order to make the electronic submission of coursework relatively easy.

1.1 Mounting a `r:` and a `u:` drive and your windows set-up

The following will help us in establishing a set-up for you which makes it easy for you to be able to obtain copies of various things that we will prepare throughout the Matlab part of the module. Once this has been done successfully you will not have to do most of this again.

(i) From the Start-Menu (click on the bottom left hand corner of the screen) select `MS-DOS Command Prompt`. This runs the program `cmd.exe`.

(Previously there was the option of selecting `Run` from the Start-Menu and then entering `cmd.exe` but currently this has been removed from the set-up of a student login.)

(ii) In the command window just created type the following

```
net use r: \\charon.brunel.ac.uk\$mastaff
```

You need to type this exactly as given (note the space after the : character). You will be prompted for a password. Assuming that you correctly enter your password a network drive labelled as `r:` will be mounted.

(iii) In the command window type the following

```
net use u: \\pinn.brunel.ac.uk\ma02xyz /user:ma02xyz
```

with `ma02xyz` replaced by your login id.. As in (ii) you will be prompted for a password. Assuming that this is all done correctly you will create a drive labelled as `u:` which contains the files you used last year. `pinn` is currently the server used for `ma02` logins on the unix network.

(iv) In the command window type the following

```
call r:\lab\mike\w2k\mkwsetup.bat
```

This will create a directory (folder) called `h:\mybin` and it will copy a few batch files to this directory.

On subsequent logins you will be prompted for your password to connect the `r:` and `u:` drive but you will not need to type the `net use` commands again unless you lose your settings. You will only need to create a command window as described in (i) and to type

```
h:\mybin\p
```

to set the path properly.

1.2 Your unix set-up

The following will be make it fairly easy for us to set things up for the electronic submission of coursework.

You can connect to unix by using the program `putty`. You can do this by using the menus or by simply creating a command window and typing

```
putty molnir
```

At the prompt on `molnir` you need to type, exactly as given, the two commands

```
source ~icstmkw/bin/ma2915startup
source $HOME/.mkwaaa
```

What this does is extend your PATH variable when you use unix so that more commands are immediately available and it gives you some 'short cuts' (technically aliases) to some long and cumbersome commands that we find useful. You should not need to type these again.

1.3 Creating a directory (folder) for your Matlab files

We recommend that you create a directory for storing you Matlab files. This can be done as follows. In a command window type the following.

```
mkdir h:\ma2915
```

1.4 Starting Matlab under windows

In any of the PC Labs you can do the following.

- If you have a command window set-up then you can type

  ```
  H:\mybin\ma
  ```

  or simply

  ```
  ma
  ```

  if you have set things up as described above. This runs the command

  ```
  l:\cc\matlab61\bin\win32\matlab.exe
  ```

In room `M123` you can also start Matlab by using the menus as follows.

```
Start -> Programs -> MatLab
```

1.5 Exiting Matlab

To leave Matlab you simply type

```
>> quit
```

That is `quit` is typed at the prompt in the Command Window of Matlab.

1.6 Using Matlab on subsequent logins

If you successfully modified your set-up as described above then on subsequent occasions you can start Matlab by doing the following.

1. Create a command prompt window.
2. Type

   ```
   cd ma2915
   ```

3. Type

   ```
   h:\mybin\ma
   ```

1.7 A few things that you can do from the command line

With a command prompt window you can list files, delete files, create directories, send files to a printer etc.. For example, if you have been creating graphs in Matlab which you want to print on certain printers then you can do the following. In Matlab you should save the graphics in postscript form with a command of the form

```
>> print -dps zyxw.ps
```

This will create the file called `zyxw.ps` which you can then send this to the printer in room `M123` by typing the following at a command prompt window

```
lpr -Pmalab zyzw.ps
```

You can similarly send it to a Computer Centre printer here at Uxbridge by typing

```
lpr -Ppublas.ccux zyzw.ps
```

1.8 Getting help within matlab

Within Matlab there is extensive help facilities which you can use interactively and which can be viewed in different ways. There are commands such as `help` and `lookfor` which display the information in your Matlab window. There are also commands such as `helpwin` and `doc` which create a help window session and which enable you to navigate around the help available more easily.

In the case of the Matlab command window you can simply type

```
>>help
```

to list the help topic names. You can then get help on a specific topic or command by typing

```
>>help  topic_name
>>help  command_name
```

For example you may wish to see what happens when you type

```
>>help graph2d
>>doc graph2d
>>help eig
>>doc eig
```

`graph2d` is a topic name whilst `eig` is the name of the built-in command for computing the eigenvalues and eigenvectors of a matrix. In some cases the amount of output created in the command window by the `help` command is more than will fit on the screen. In such cases you will need to type

```
more on
```

to start a paging mechanism. (Typing `more off` terminates the paging mechanism.)

If you wish to use something that you think may be available, but you do not know any specific details, you can search all the functions known to Matlab by typing a command of the form

```
>>lookfor  search_item
```

For example, to find all the function file commands associated with eigenvalues you can type

```
>>lookfor eigenvalues
```

What is searched is the first header line of all functions. The functions whose first header line contains the word eigenvalues are displayed.

2 Control statements in Matlab

When a script file or a function file is executed in Matlab the statements are executed in the order they appear in the file, i.e. the first statement, then the second statement, then the third statement etc. unless there are control statements to indicate otherwise. For example the statements

```
x=5
y=(x^2-5)/3
x=sin(3*y*pi/2)
```

gives the output

```
x =
     5
y =
    6.6667
x =
  -1.2246e-15
```

At the start `x` contains the value `5`, next `y` becomes 20/3 (to rounding error) and then `x` is reset to the outcome of `sin(3*y*pi/2)` which is close to 0. (We would get exactly 0 if all computations are done exactly.) Often however we only want to carry out a set of statements when a certain condition is true and we often want to repeat a set of statements without having to repeatedly enter the same set of statements. Statements which enable us to do this are known as control statements or flow control statements. In the level one module `MA1051S` Numerical Methods you would have met some of these. We review here the `if` and `if-elseif` constructions, the `for` loop and the `while` loop and we add the `break` statement which you may not have met before.

2.1 Relational operators

The *relational operators* are the operators such as `<` and `>` which can be used in tests of the form `x<y` or `a>b`. A list of these relational operators follows.

`==`	equal
`~=`	not equal
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

If we want to write something like `x==y` then we get the value 1 if the statement is true (i.e. `x` and `y` are identical) and we get the value 0 if it is false. (Note that we write `x==y` and not `x=y` which is what we have in an assignment statement. This is quite a common error and it is sometimes difficult to spot!)

2.2 Logical operators

You can combine a number of tests using the following *logical operators*

`&`	logical and
`|`	logical or
`~`	logical not

For example we can have the following

```
(0<x)&(x<2)
(x<=0)|(x>=2)
(a~=0)&(b*b-4*a*c>0)
```

In the first case the outcome is 1 (i.e. true) if `x` is between 0 and 2 and it is 0 (i.e. false) otherwise. The second case is the opposite of the first case and we could have written instead `~((0<x)&(x<2))`. The third case evaluates to true if both `a` is different from 0 and `b*b-4*a*c` is positive.

Logical test are usually used as part of an `if` or `elseif` statement or as the test in a `while` loop.

2.3 `if` and `elseif` statements

In its simplest form the `if` statement, or more specifically the `if` block of statements, has the form

```
if  logical expression
   ...
   ...  statements to be done if expression is true
   ...
end
```

As is indicated, if the logical expression evaluates to true then the block of statements are performed, otherwise the statements are ignored. If we want something to be done if the test is true and something else to be done if it is false then we can have instead

```
if  logical expression
   ...
   ...  statements to be done if expression is true
   ...
else
   ...
   ...  statements to be done if expression is false
   ...
end
```

A further form of the `if` construction involves the `elseif` statement which can be used to add further tests which are to be evaluated if the test in the `if` statement is false. For example we can have the following.

```
if  logical expression A
   ...
   ...   statements to be done if expression A is true
   ...
elseif  logical expression B
   ...
   ...   These statements are done if expression A is false
   ...   and expression B is true
   ...
else
   ...
   ...   statements to be done if expressions A and B are both false
   ...
end
```

In this form the *logical expression B* is only evaluated if the *logical expression A* is false. You can have any number of `elseif` statements in such a construction.

Note that in all these forms of the `if` statement there must be an `end` statement to terminate the block. The `end` statement also terminates the `for` loop and the `while` loop which we consider in later subsections.

As an example of using the `if` block consider the problem of solving the quadratic equation

$$ax^2 + bx + c = 0, \quad a, b, c \in \mathbb{R} \text{ and } a \neq 0.$$

The solution or solutions are given by the quadratic formulae

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

To create a reasonably robust Matlab program which implements this we need to make sure that $a \neq 0$ and when $a \neq 0$ we need to consider the 3 cases $b^2 - 4ac > 0$, $b^2 - 4ac = 0$ and $b^2 - 4ac < 0$. A set of instructions, written as a function file, is given below. Function files are described later in these notes. A function file is used here to ease the testing of the program.

```
function quadrat1(a,b,c)
%QUADRAT1  Solving quadratic equation function.
%          quadrat1(a,b,c)  This computes and displays the roots
%          of the quadratic a*x^2+b*x+c.
%          The function does not return any values.

% show the coefficients
fprintf('Solving a*x^2+b*x+c=0 with\n');
fprintf('a=%15.7e, b=%15.7e, c=%15.7e\n',a,b,c);

% compute b*b-4*a*c needed in the test
d=b*b-4*a*c;
v=sqrt(abs(d));

% now consider the different cases
if a==0
  fprintf('a should be nonzero!\n');
elseif d>0
  x1=(-b+v)/(2*a);
  x2=(-b-v)/(2*a);
  fprintf('2 real distinct roots: x1=%15.7e, x2=%15.7e\n',x1,x2);
elseif d==0
  x1=-b/(2*a);
  x2=x1;
  fprintf('a double root at x1=%15.7e\n',x1);
else
  realp=-b/(2*a);
  imagp=v/(2*a);
  fprintf('a complex conjugate pair of roots,\n');
  fprintf('real part=      %15.7e,\n',realp);
  fprintf('imaginary parts=%15.7e, %15.7e\n',imagp,-imagp);
end
fprintf('\n');
```

To test the function file the following script was used

```
echo on;
quadrat1(0,1,2);
quadrat1(1,2,1);
quadrat1(1,-4,1);
quadrat1(1,2,2);
echo off;
```

The output created, which was captured using the `diary` command, was as follows.

```
quadrat1(0,1,2);
Solving a*x^2+b*x+c=0 with
a=  0.0000000e+00, b=  1.0000000e+00, c=  2.0000000e+00
a should be nonzero!
```

```
quadrat1(1,2,1);
Solving a*x^2+b*x+c=0 with
a=  1.0000000e+00, b=  2.0000000e+00, c=  1.0000000e+00
a double root at x1= -1.0000000e+00

quadrat1(1,-4,1);
Solving a*x^2+b*x+c=0 with
a=  1.0000000e+00, b= -4.0000000e+00, c=  1.0000000e+00
2 real distinct roots: x1=  3.7320508e+00, x2=  2.6794919e-01

quadrat1(1,2,2);
Solving a*x^2+b*x+c=0 with
a=  1.0000000e+00, b=  2.0000000e+00, c=  2.0000000e+00
a complex conjugate pair of roots,
real part=       -1.0000000e+00,
imaginary parts=  1.0000000e+00,  -1.0000000e+00

echo off;
```

2.4 The `for` loop

The `for` loop is the Matlab way of repeating a set of instructions a certain number of times. The syntax of the loop is

```
for  variable=expression
...
...  statements
...
end
```

In this the *expression* is a vector of values and the *statements* are performed when the variable is the 1st value, then the 2nd value, then the 3rd value etc.. For example, consider the following set of statements.

```
s=0;
for k=1:1000
  s=s+1/k;
end
```

Here the *expression* is the vector `1:1000`, which is the same as `[1 2 ... 1000]`, and the variable `k` takes the value 1, then 2, then 3, $\cdots$, and finally 1000. The final value of `s` from these instructions is

$$\sum_{1}^{1000} \frac{1}{k}.$$

The value obtained is

```
s =
   7.48547086055034
```

In this short example the variable `k` is used in the *statements*. This need not be the case. For example, consider the following function file which performs 10 Newton iterations for attempting to solve $f(x) = 0$ in the specific case where $f(x) = x^3 - 2$. To keep everything in one place a function file is used and below it two sub-functions are used for respectively $f(x)$ and its derivative $f'(x)$.

```
function newton_ex1(x)
%NEWTON_EX1  10 iterations of the Newton method.
%            newton_ex1(x)  Given the starting point x, this performs
%            and outputs the results of 10 Newton iterations in the
%            case f(x)=x^3-2. The function itself returns no values.


fprintf('x=%24.16e\n',x);

% the loop
for it=1:10
  v=f(x)/df(x);
  x=x-v;
  fprintf('x=%24.16e\n',x);
end

function y=f(x)
y=x^3-2;

function y=df(x)
y=3*x^2;
```

By entering

```
>>newton_ex1(1)
```

the following output was obtained

```
x=  1.0000000000000000e+00
x=  1.3333333333333333e+00
x=  1.2638888888888888e+00
x=  1.2599334934499771e+00
x=  1.2599210500177698e+00
x=  1.2599210498948732e+00
x=  1.2599210498948732e+00
x=  1.2599210498948732e+00
x=  1.2599210498948732e+00
x=  1.2599210498948732e+00
x=  1.2599210498948732e+00
```

In this example the loop variable `it` does not appear in the statements in the loop but it is just part of the mechanism to enable 10 Newton steps to be performed.

Using the function file with starting point $x = 1$ leads to the required solution to machine precision but, as given, it is not too satisfactory as the iteration could have been terminated before all 10 iterations are performed. This can be achieved by using a `break` statement, which will be described later, or re-coded using a `while` loop.

You should note that the use of the two sub-functions here, `f` and `df`, is only valid within a function file and the sub-functions used are not known to any other function file that you might use. The sub-functions take precedence over any functions of the same name. Thus if you also have function files called `f.m` and `df.m` then these are not used. The sub-functions specified here are used instead.

As further examples of using `for` loops consider the following.

The Composite Simpson rule for approximating $\int_a^b f(x)\,dx$.

With $n > 0$, $h = (b-a)/(2n)$ and equally spaced points $x_k = a + kh$, $k = 0, 1, \cdots, 2n$ the composite simpson rule approximation is given by

$$I_n = \frac{h}{3}\left(f(a) + f(b) + 4\sum_{k=1}^{n} f(x_{2k-1}) + 2\sum_{k=1}^{n-1} f(x_{2k})\right).$$

To program this in Matlab for one value of n we can use `for` loops and to compute a finite sequence of approximations $I_2, I_4, I_8, \cdots, I_{256}$ we can use a construction of the following form.

```
for n=[2 4 8 16 32 64 128 256]
...
...   statements to compute the composite Simpson approximation
...   with the 2n+1 points a,a+h,a+2*h,...,a+2*n*h=b
...
end
```

A complete program, which uses sub-functions, which does this in the case $f(x) = x^5$ is shown below and it is tested in the case $a = 0$, $b = 1$. The exact value of the integral is of course known in this case as

$$\int_0^1 x^5\,dx = \left[\frac{x^6}{6}\right]_0^1 = \frac{1}{6}.$$

It is generally good practice to test programs on problems for which the solution is known, when this is possible, in order to gain confidence that everything appears to be working satisfactorily before trying the program on problems which cannot be solved exactly.

```
function simpson_ex1(a,b)
%SIMPSON_EX1  Simpson's rule example with uniform refinement.
%             simpson_ex1(a,b)  Function to compute a sequence of
%             Simpson's rule approximations to the integral of the
%             subfunction f from x=a to x=b.

% loop on the number of subdivisions
for n=[2 4 8 16 32 64 128 256]

  %compute the step length
  h=(b-a)/(2*n);

  %sum the odd indexed points
  odd=0;
  for k=1:n
    x=a+(2*k-1)*h;
    odd=odd+f(x);
  end

  %sum the even indexed points
  even=0;
  for k=1:n-1
    x=a+2*k*h;
    even=even+f(x);
  end

  %put everything together and show the approximation
  approx=h*(f(a)+f(b)+4*odd+2*even)/3;
  fprintf('%4d sub-divisions, approximation=%22.14e\n',2*n,approx);
end

function y=f(x)
y=x^5;
```

The instruction to test the function is

```
>>simpson_ex1(0,1)
```

and this created the output

```
   4 sub-divisions, approximation=  1.67968750000000e-01
   8 sub-divisions, approximation=  1.66748046875000e-01
  16 sub-divisions, approximation=  1.66671752929688e-01
  32 sub-divisions, approximation=  1.66666984558105e-01
  64 sub-divisions, approximation=  1.66666686534882e-01
 128 sub-divisions, approximation=  1.66666667908430e-01
 256 sub-divisions, approximation=  1.66666666744277e-01
 512 sub-divisions, approximation=  1.66666666671517e-01
```

The basic Euler method for solving $\dot{y} = f(t, y)$, $y(0) = y_0$.

The task of approximately determining the function $y = y(t)$ for $t \ge 0$ solving the initial value problem

$$\dot{y} = f(t, y), \quad y(0) = y_0$$

is one of the main topics of the module `MA2915`. The most basic of schemes for doing this is Euler's method which was covered in the level one module `MA1051S` Numerical Methods. If $T > 0$ is given and the number of steps $n > 0$ is given and we define $h = T/n$ and $t_k = kh$, $k = 0, 1, 2, \cdots, n$ then the scheme is as follows. Based on the difference approximation for the derivative

$$\frac{y(t_{k+1}) - y(t_k)}{h} \approx \dot{y}(t_k) = f(t_k, y(t_k))$$

we define $y_1, y_2, \cdots, y_n$ so that

$$\frac{y_{k+1} - y_k}{h} = f(t_k, y_k), \quad \text{i.e. } y_{k+1} = y_k + h\, f(t_k, y_k), \quad k = 0, 1, 2, \cdots, n-1.$$

In this we have the exact solution y_0 at the initial point $x_0 = 0$ and for $k = 1, 2, \cdots, n$ the value y_k is an approximation to $y(t_k)$. In particular y_n is the approximation to $y(T)$.

A Matlab function file which takes the final time T, the number of steps n and the initial value y_0 as input is shown below. As in the case of example programs given earlier a sub-function is included in order to test the program. The test problem is

$$\dot{y} = f(t, y), \quad \text{with } f(t, y) = -y - 5\exp(-t)\sin 5t \quad \text{and with } y(0) = 1.$$

This is a problem that can solve exactly and the exact solution is

$$y = \exp(-t)\, \cos 5t\ .$$

```
function [t,y]=euler_ex1(T,n,y0)
%EULER_EX1  The basic Euler method.
%           [t,y]=euler_ex1(T,n,y0) This approximately solves
%           dy/ds=f(s,y), y(0)=y0, 0<=s<=T,
%           by using the basic Euler method with n equal steps of size
%           h=T/n.  On output t is a row vector of the n+1 values
%           0,h,2*h,...,n*h=T and y is a row vector of length n+1
%           containing the approx solution, y(k) being the approximate
%           solution at t(k). f is specified as a sub-function.

% set the increment h, set t and reserve space for y
h=T/n;
t=linspace(0,T,n+1);
y=zeros(1,n+1);

% set y(1) to the given initial value
y(1)=y0;

% now do n steps of Euler's method
for k=1:n
  y(k+1)=y(k)+h*f(t(k),y(k));
end

function z=f(s,y)
z=-y-5*exp(-s)*sin(5*s);
```

To test the function we have the following script.

```
% get the approx solution when T=3 with 8 and 16 steps
T=3;
[t8,y8]=euler_ex1(T,8,1);
[t16,y16]=euler_ex1(T,16,1);

% evaluate the exact solution at 200 points
t=linspace(0,T,200);
exact=exp(-t).*cos(5*t);

% show the exact solution and two approx solutions on the same graph,
plot(t,exact,t8,y8,'s:',t16,y16,'x--');

% add some labels, a title, a legend and save in eps format
xlabel('t');
ylabel('y');
title('Basic Euler method for dy/dt=-y-5*exp(-t)*sin(5*t), y(0)=1');
legend('exact solution','8 steps','16 steps');
figure(1);
print -deps euler_ex1.eps
```

The graph created with these instructions is shown in Figure 1.

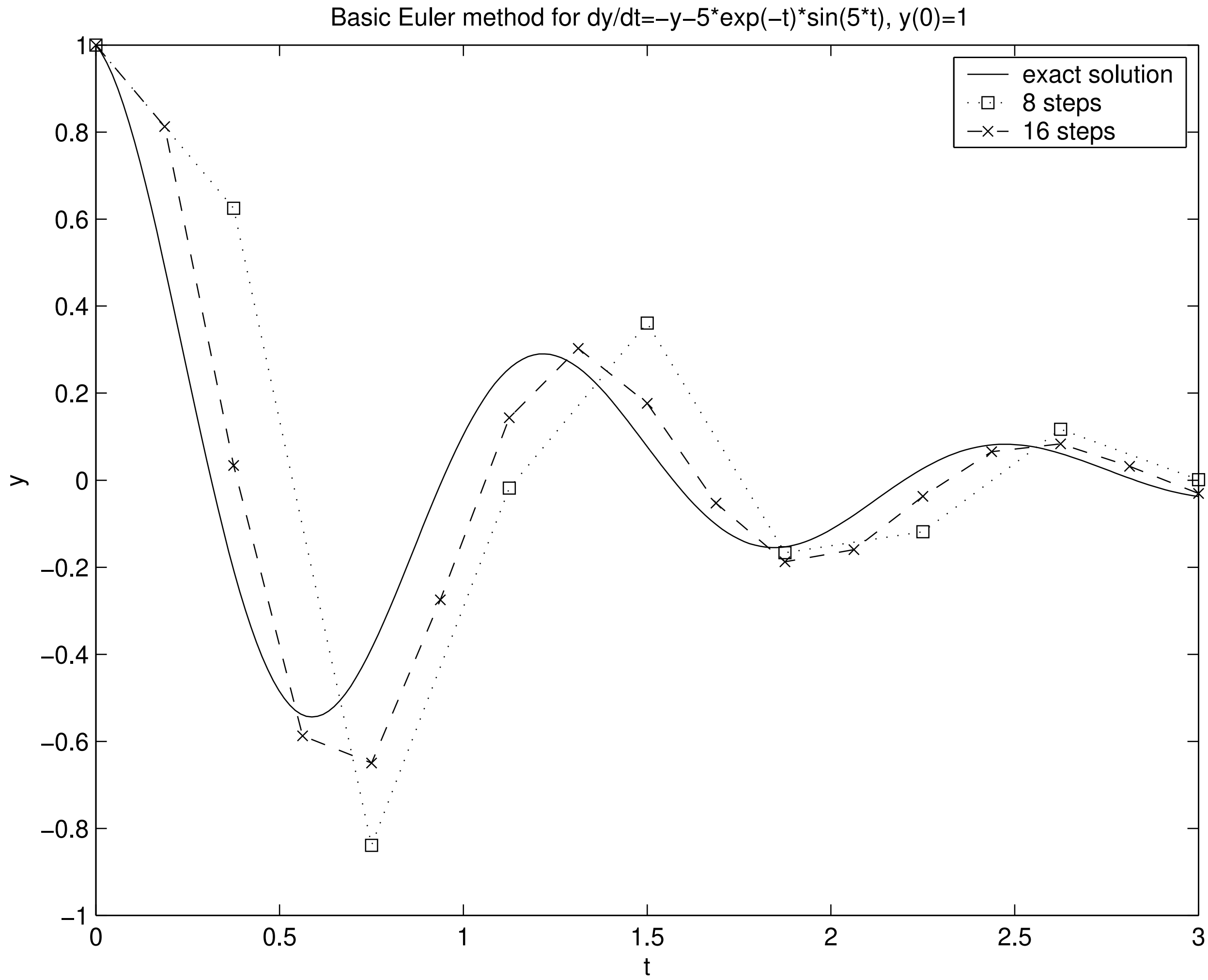

Figure 1: Graphical output from the basic Euler method test script

There are a few points to note in this example.

(a) The `for` loop being illustrated is just the statements

```
for k=1:n
  y(k+1)=y(k)+h*f(t(k),y(k));
end
```

(b) To increase efficiency the statement

```
y=zeros(1,n+1);
```

is included early on in the file. It would still work if this line is removed but it would take longer as much of the time taken in implementing

```
  y(k+1)=y(k)+h*f(t(k),y(k));
```

would be taken up in the allocation of more space for the vector `y` which would be growing in length at each stage.

(c) Note the syntax of the `plot` statement.

```
plot(t,exact,t8,y8,'s:',t16,y16,'x--');
```

There are 3 graphs here,

```
t,exact
```

gives the exact solution,

```
t8,y8,'s:'
```

generates the dotted line and the squares and

```
t16,y16,'x--'
```

generates the dashed line with the crosses. The `legend` statement, as specified, has 3 arguments with the strings describing respectively each of the 3 curves. See the output of the commands

```
more on
help plot
help legend
```

for further details. (The `more on` instruction starts a paging mechanism to ensure that you can read all the output of the `help` commands. The instruction `more off` turns off the paging.)

It can often be a matter of trial and error when producing graphical output in deciding what 'looks best' in a given situation.

2.5 The `while` loop

The syntax of the `while` loop is

```
while  logical expression
...
...  statements to be done while the logical expression is true
...
end
```

When this is executed the *logical expression* is tested. If it is false then the block of statements is skipped. If it is true then the set of statements are implemented and once they are complete the *logical expression* is tested again. If the *logical expression* is now false then the `while` block is finished but if it is true then the set of statements are implemented again. This process of evaluating the block of statements and then testing the logical expression is repeated while the logical expression evaluates to 1, i.e. when it is true.

A `while` loop is thus a way of repeating a set of instructions while a *logical expression* is true which differs from the `for` loop which repeats a set of instructions a fixed number of times. There is a danger when creating `while` loops that, due to a design error or merely a typing error, the *logical expression* always remains true so that the loop never terminates, i.e. we have an infinite loop. This danger can often be overcome by providing other ways of leaving the loop involving the `break` statement which is described in the next subsection.

Programming iterations such as the Newton method or the Secant method are situations in which you might want to use a `while` loop, instead of a `for` loop, as you do not know in advance how many iterations will be required to converge to a desired accuracy. Using a `while` loop instead of a `for` loop the Newton method example `newton_ex1.m` given earlier can now be replaced by the following.

```
function newton_ex2(x)
%NEWTON_EX2  The Newton iteration for the cube root of 2.
%            newton_ex2(x)  Given the starting point x, this performs
%            and outputs the results of Newton iterations in the specific
%            case that f(x)=x^3-2. The iteration is stopped if the
%            convergence criterion is met or when 20 iterations have
%            been completed.
%            The function itself returns no values.


fprintf('x=%24.16e\n',x);

% set the tolerance used in the stopping criterion, the maximum
% number of iterations and initialise the iteration counter

tol=1e-12;
itmax=20;
it=0;

% the loop
v=2*tol;
while (abs(v)>tol & it<=itmax)
  v=f(x)/df(x);
  x=x-v;
  fprintf('x=%24.16e\n',x);
  it=it+1;
end

% report whether or not the stopping criterion has been met
if abs(v)<tol
  fprintf('The iteration has terminated with tol=%e\n',tol);
else
  fprintf('***The convergence criterion has not been met ***\n');
  fprintf('*** with itmax,tol=%d, %e ***\n');
end

function y=f(x)
y=x^3-2;

function y=df(x)
y=3*x^2;
```

By entering

```
>>newton_ex2(1)
```

the following output was obtained

```
x=  1.0000000000000000e+00
```

```
x=  1.3333333333333333e+00
x=  1.2638888888888888e+00
x=  1.2599334934499771e+00
x=  1.2599210500177698e+00
x=  1.2599210498948732e+00
x=  1.2599210498948732e+00
The iteration has terminated with tol=1.000000e-12
```

In this example it is the use of the counter (the variable `it`) which ensures that the *logical expression* eventually becomes false whatever function or starting point is used. Note that `1e-12` means 10^{-12}.

2.6 The `break` statement

The `break` statement is a way of getting out of a loop, such as `for` loop or a `while` loop, when some condition is true. This is useful in many situations in which it is inconvenient to put all the criterion for performing the loop in one place. For example, a `for` loop and a `break` statement combination of doing what is done in `newton_ex2.m` is as follows.

```
function newton_ex3(x)
%NEWTON_EX3  The Newton iteration for the cube root of 2.
%            newton_ex3(x)  Given the starting point x, this performs
%            and outputs the results of Newton iterations in the specific
%            case that f(x)=x^3-2. The iteration is stopped if the
%            convergence criterion is met or when 20 iterations have
%            been completed.
%            The function itself returns no values.


fprintf('x=%24.16e\n',x);

% set the tolerance used in the stopping criterion and
% the maximum number of iterations

tol=1e-12;
itmax=20;

% the loop
for it=1:itmax
  v=f(x)/df(x);
  x=x-v;
  fprintf('x=%24.16e\n',x);
  if abs(v)<tol
    break;
  end
end

% report whether or not the stopping criterion has been met
if abs(v)<tol
  fprintf('The iteration has terminated with tol=%e\n',tol);
else
  fprintf('***The convergence criterion has not been met ***\n');
  fprintf('*** with itmax,tol=%d, %e ***\n');
end

function y=f(x)
y=x^3-2;

function y=df(x)
y=3*x^2;
```

Here the `for` loop deals with the counter mechanism with the `if` and `break` combination of statements enabling an early exit if the stopping criterion becomes satisfied.

3 Function files

A m-file, as it is known, is a set of Matlab statements contained in a file which has a name ending in the two characters `.m`. It is a called a function m-file or simply a function file if the first line of the file starts with the word `function`. Functions files are, roughly speaking, Matlab's equivalent of functions, subroutines and procedures used in other programming languages. They can take input arguments, they can produce output arguments and, unless you use a global statement, the internal variables are local to the function.

When you enter a valid command in Matlab you are either executing a built-in function or you are executing a function m-file that exists somewhere on the system. The function m-file may be part of the Matlab package itself or it may be something you or someone else has written. As examples, consider the following statements

```
y=sin(pi/3)
A=[2 1;1 2]
lambda=eig(A)
```

which produces the output

```
y =
   0.86602540378444
A =
     2     1
     1     2
lambda =
     1
     3
```

In the first statement the `sin` function is used and in the third statement the `eig` function is used. Both are built-in functions which you can check by typing

```
>>which sin
>>which eig
```

In the section on control statements most of the example programs given used function files created as part of these notes, the names of the files are `quadrat1.m`, `newton_ex1.m`, `simpson_ex1.m`, `euler_ex1.m`, `newton_ex2.m` and `newton_ex3.m` corresponding respectively to the function names `quadrat1`, `newton_ex1`, `simpson_ex1`, `euler_ex1`, `newton_ex2` and `newton_ex3`. In all these cases the functions have input arguments but only the function `euler_ex1` has output arguments. In this section we now consider the creation and use of function files in a little more detail.

The syntax of the function statement is as follows.

function [*output_arguments*]=*function_name*(*input_arguments*)

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

input_arguments	This is a list of variables separated by commas.
output_arguments	This is also a list of variables separated by commas. If there is only one output argument then the square brackets `[]` can be removed.

Example 1: Defining a function for a plot.

A function file can be as short as 2 lines. For example a function file for the function

$$f(x) = \sin(x) + \sin(3x)/3 + \sin(5x)/5 + \sin(7x)/7$$

can be simply the following.

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

Due to Matlab's ability in coping with complex numbers, vectors and matrices this function can take a real number, a complex number, a vector or a matrix as input and it will return a type of the same type and size. To take advantage of this consider the following commands for plotting this 2π-periodic function.

```
x=linspace(0,3*pi,300);
y=f(x);
plot(x,y)
```

The graphical output created is shown in Figure 2.

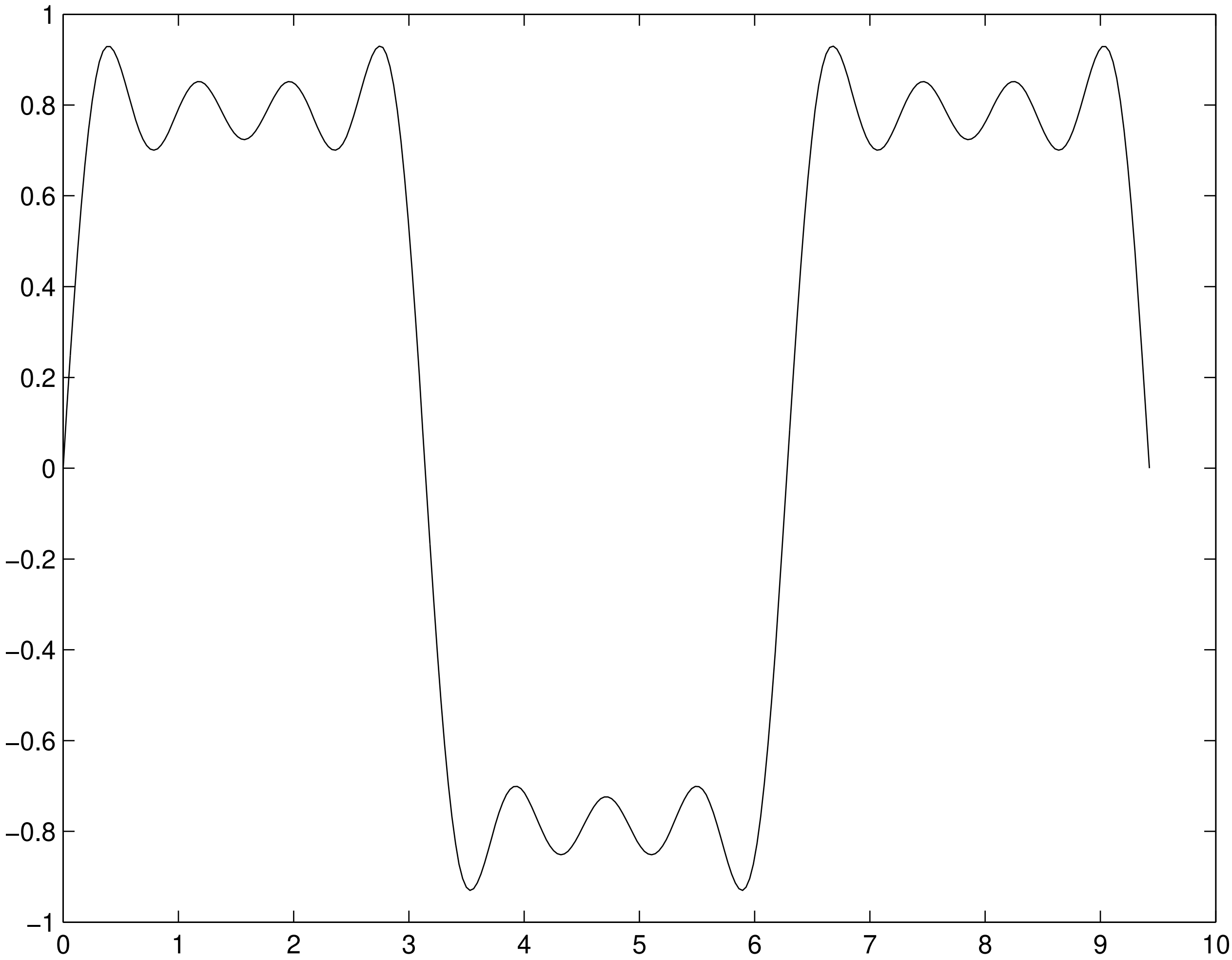

Figure 2: A graph of $f(x) = \sin(x) + \sin(3x)/3 + \sin(5x)/5 + \sin(7x)/7, \quad 0 \le x \le 3\pi$.

Example 2: Illustrating the effect of a linear transformation

In the module MA1002S Algebra II there were assignments which involved various aspects relating to determining the image of the unit square and the unit circle as as a result of the linear transformation represented by a 2×2 matrix A. We now consider how this can be done by a set of Matlab statements put together as a function file. Before giving the function file we consider what is involved mathematically.

Writing points in the plane $\mathbb{R}^2$ as 2×1 column vectors, the image $\underline{x}' = (x_1', x_2')^T$ of $\underline{x} = (x_1, x_2)^T$ is given by

$$\underline{x}' = A\underline{x}, \quad \text{with } \underline{x} = A^{-1}\underline{x}' \quad (\text{when } \det A \neq 0.)$$

As a linear transformation maps straight lines onto straight lines we just need to determine the images of the vertices of the unit square to determine how the unit square transforms. These are given by the columns of

$$A \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

The deal with the transformation of the circle we need a parametric equation for the circle and for this we can use

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos t \\ \sin t \end{pmatrix}, \quad 0 \le t < 2\pi \ .$$

A parametric equation of the image, which is an ellipse provided $\det A \neq 0$, is thus given by

$$\underline{x}' = A \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}(t) = A \begin{pmatrix} \cos t \\ \sin t \end{pmatrix}, \quad 0 \le t < 2\pi \ .$$

To obtain a cartesian equation for this image we can do the following.

$$1 = x_1^2 + x_2^2 = (x_1, x_2) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \underline{x}^T \underline{x} = (\underline{x}')^T (A^{-1})^T A^{-1} \underline{x}'.$$

Now let $B = A\,A^T$, which is a symmetric matrix with non-negative eigenvalues and orthogonal eigenvectors, and note that

$$B^{-1} = (A\,A^T)^{-1} = (A^T)^{-1} A^{-1} = (A^{-1})^T A^{-1} \ .$$

If $\underline{v}_1$, $\underline{v}_2$ are normalised (orthogonal) eigenvectors of B with corresponding eigenvalues $\lambda_1 \ge 0$ and $\lambda_2 \ge 0$ then $Q = (\underline{v}_1, \underline{v}_2)$ is an orthogonal matrix and

$$BQ = B(\underline{v}_1, \underline{v}_2) = (B\underline{v}_1, B\underline{v}_2) = (\lambda_1 \underline{v}_1, \lambda_2 \underline{v}_2) = (\underline{v}_1, \underline{v}_2)D = QD \quad \text{where } D = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

and thus

$$B = QDQ^T \quad \text{and} \quad B^{-1} = QD^{-1}Q^T$$

so that the ellipse is

$$1 = (\underline{x}')^T QD^{-1}Q^T \underline{x}' = (\underline{y}')^T D^{-1} \underline{y}' = \frac{y_1^2}{\lambda_1} + \frac{y_2^2}{\lambda_2}, \quad \text{where } \underline{y}' = Q^T \underline{x}' \quad \text{and} \quad \underline{x}' = Q\underline{y}' \ .$$

Points $\underline{e}_1 = (1,0)^T$ and $\underline{e}_2 = (0,1)^T$ in the y_1, y_2 plane correspond to points $Q\underline{e}_1$ and $Q\underline{e}_2$ in the x_1', x_2' plane and hence the 2 columns of Q give the directions of the axis of the ellipse. The lengths of the axis are given by $\sqrt{\lambda_1}$ and $\sqrt{\lambda_2}$. The two columns of $A^{-1}Q$ give the directions which map to these axis. Observe that

$$BQ = AA^TQ = QD \quad \text{rearranges to } A^TQ = A^{-1}QD$$

so that

$$(A^TA)(A^{-1}Q) = A^TQ = (A^{-1}Q)D\ .$$

This shows that the columns of $A^{-1}Q$ are eigenvectors of A^TA, which are thus orthogonal as A^TA is symmetric, and that A^TA and AA^T have the same eigenvalues.

The function file which displays the various regions and the various orthogonal directions is shown below.

```
function lintran(A)
%LINTRAN  represent graphically the effect of the 2x2 matrix A.
%         The effect of the linear transformation on the unit square
%         and the unit circle is shown.

% set sqv: the columns of sqv are the vertices of the unit square,
%          the 1st vertex is repeated to get a closed curve.
sqv=[0 1 1 0 0;
     0 0 1 1 0];

% set cirv: the columns of cirv are points on the unit circle
t=linspace(0,2*pi,200);
cirv=[cos(t); sin(t)];

% get the deformed points as a result of multiplying by A
def_sqv =A*sqv;
def_cirv=A*cirv;

% get the eigenvalues and eigenvectors of the symmetric matrix A*A'
B=A*A';
[Q D]=eig(B);
lam1=sqrt(D(1,1));
lam2=sqrt(D(2,2));

% the axis of the ellipse are the cols of Q scaled by lam1,lam2
% the corresponding directions wrt the circle are the cols of inv(A)*Q
ellpt1=lam1*Q(:,1); ellpt2=lam2*Q(:,2);
tem=inv(A)*Q;
pt1=lam1*tem(:,1); pt2=lam2*tem(:,2);
s=sprintf('ellipse, parameters=%7.4f, %7.4f',lam1,lam2);

% show the unit square, its image, the unit circle and its image
% as a 2x2 grid of plots and also add the axis of the ellipse.

subplot(2,2,1),
  plot(sqv(1,:),sqv(2,:)), axis equal, title('unit square') ,
subplot(2,2,2),
  plot(def_sqv(1,:),def_sqv(2,:)), axis equal, title('deformed square'),
subplot(2,2,3),
  plot(cirv(1,:),cirv(2,:),...
  [0,pt1(1)],[0,pt1(2)],':',[0,pt2(1)],[0,pt2(2)],'-.'),
  axis equal, title('unit circle'),
subplot(2,2,4),
  plot(def_cirv(1,:),def_cirv(2,:),...
  [0,ellpt1(1)],[0,ellpt1(2)],':',[0,ellpt2(1)],[0,ellpt2(2)],'-.'),
  axis equal, title(s),
figure(1);
```

The function is tested with the following instruction

```
>> lintran([2 1;-1 1])
```

which created the graphical output shown in Figure 3.

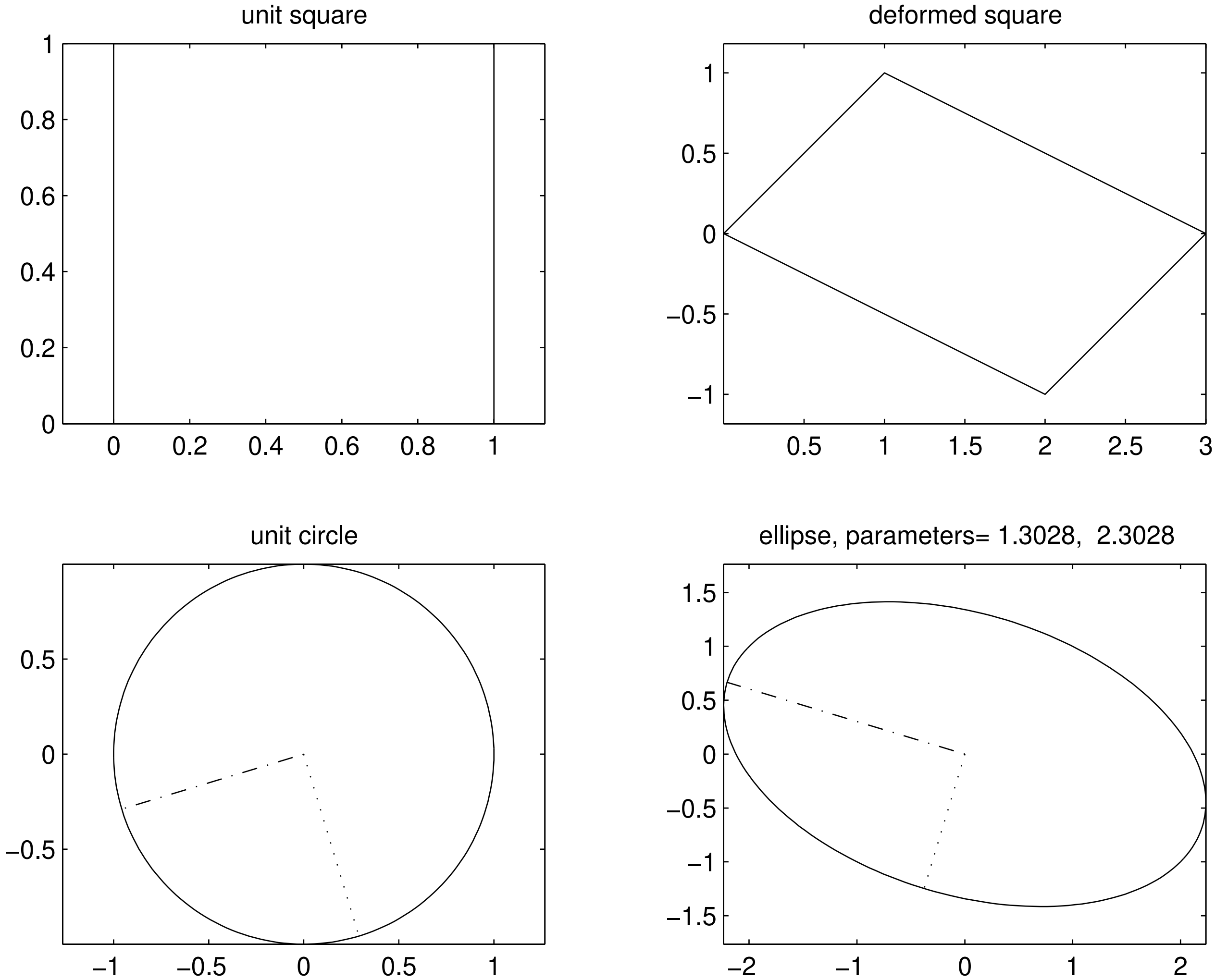

Figure 3: Plots illustrating the effect of a linear transformation.

There are a few points to note here.

(i) The function `lintran` returns no values which is why it can be used in the form

```
>> lintran([2 1;-1 1])
```

(ii) The eigenvalues and eigenvectors are computed using the built-in function `eig`. Type

```
>>help eig
```

for more details.

(iii) The 4 plots are put in a 2×2 grid using the `subplot` command and the use of the `axis` command as given ensures that a circle looks like a circle. Type

```
>>help subplot
>>help axis
```

for more details.

(iv) The use of ... as the last 3 characters on some of the lines is the way in which a long statement can be represented over more than one line.

(v) The use of `sprintf` is a way of creating a string that involves numbers created by the Matlab program that can used in a title to a plot. `sprintf` is similar to `fprintf` used previously. Type

```
>>help sprintf
>>help fprintf
```

for further details.

(vi) If we type

```
>> help lintran
```

then the set of comments at the top of the file are shown and we create

```
help lintran

 LINTRAN  represent graphically the effect of the 2x2 matrix A.
          The effect of the linear transformation on the unit square
          and the unit circle is shown.
```

If we type

```
>> lookfor graphical
```

then the first comment line of all functions within Matlabs search path that contain the word `graphical` are shown. The output obtained is as follows.

```
lookfor graphical
LINTRAN  represent graphically the effect of the 2x2 matrix A.
openvar Open a workspace variable for graphical editing.
GINPUT Graphical input from mouse.
PROPEDIT  Graphical property editor
CELLPLOT Display graphical depiction of cell array.
EIGSHOW Graphical demonstration of eigenvalues and singular values.
```

Example 3: Some statistical processing of an array of numbers

Example 1 involved a 2 line function file `f.m` with the function `f` having only one input argument and producing only one output argument. In example 2 the function file `lintran.m` for the function `lintran` has an even simpler specification as it only has one input argument and it has no output arguments. We now consider an example with a function file which has several output arguments and consider how to use the built-in function `nargout` within the function to determine the number of output arguments that are expected in the reference to the function. There is also a built-in function `nargin` which gives the number of input arguments in the reference although this is not used in this example. `nargin` and `nargout` are used in many of the functions which are part of the Matlab package. For example, the built-in function `plot` can take any number of input arguments depending on the number of graphs to be plotted and the amount of control over the plots that is required. Similarly the built-in function `eig` will compute both the eigenvalues and eigenvectors of a matrix or will just compute the eigenvalues of the matrix depending on how it is used. For example the statement

```
s=eig(A);
```

just creates a column vector `s` containing the eigenvalues of the matrix `A`. If instead we have

```
[V,D]=eig(A);
```

then matrices `V` and `D` are created. The columns of `V` are eigenvectors of `A` and `D` is a diagonal matrix with the diagonal entries being the eigenvalues of `A`. It computes `V` and `D` such that, to rounding error, `A*V=V*D`. These different references to the function `eig` are possible as the built-in function `eig` is written in such a way that it can detect how much output is expected and it performs the necessary computation based on this. We consider now how we can create a function file which has this flexible usage.

Given a set of marks to an examination is often useful to compute the mean, the standard deviation, to sort the marks, to compute the medium and to determine the number of marks in various grade bands. A function file which does this follows.

```
function [x_mean,x_std,x_sort,x_med,x_grades]=p_marks(x)
%P_MARKS  Basic statistical processing of a set of marks.
%         [x_mean,x_std,x_sort,x_med,x_grades]=p_marks(x)
%         On input:  x is an array of numbers,
%         On output: x_mean is the mean, x_std is the standard dev,
%                    x_sort is the array in ascending order,
%                    x_med is the medium,
%                    x_grades is the number in each grade bands
%                    <35, 35--39, 40--49, 50--59, 60--69, >=70

% determine the number of output arguments expected
no=nargout;

% determine the number of points
n=length(x);

% process depending on the value of no

if no==0 return; end

% basic computations using the built-in functions
x_mean=mean(x);

if no>=2 x_std=std(x); end

if no>=3 x_sort=sort(x); end

if no>=4 x_med=median(x); end

% now get the number in each grade band
if no>= 5
  x_grades=zeros(1,6);
  for k=1:n
    if x(k)<35
      x_grades(1)=x_grades(1)+1;
    elseif x(k)<40
      x_grades(2)=x_grades(2)+1;
    elseif x(k)<50
      x_grades(3)=x_grades(3)+1;
    elseif x(k)<60
      x_grades(4)=x_grades(4)+1;
    elseif x(k)<70
      x_grades(5)=x_grades(5)+1;
    else
      x_grades(6)=x_grades(6)+1;
    end
  end
end
```

To test the script the following instructions were used.

```
echo on
x=55+15*randn(1,38);
x=floor(x)
x_mean=p_marks(x)
[x_mean,x_std,x_sort,x_med,x_grades]=p_marks(x)
echo off
```

The output created is shown below.

```
x=55+15*randn(1,38);
x=floor(x)
x =
  Columns 1 through 14
    51    60    51    30    39    58    36    49    40    37    39    48    57    53
  Columns 15 through 28
    70    51    32    55    56    59    62    74    46    58    54    46    87    51
  Columns 29 through 38
    33    81    59    38    64    74    41    57    52    37
x_mean=p_marks(x)
x_mean =
   52.2368
[x_mean,x_std,x_sort,x_med,x_grades]=p_marks(x)
x_mean =
   52.2368
x_std =
   13.5216
x_sort =
  Columns 1 through 14
    30    32    33    36    37    37    38    39    39    40    41    46    46    48
  Columns 15 through 28
    49    51    51    51    51    52    53    54    55    56    57    57    58    58
  Columns 29 through 38
    59    59    60    62    64    70    74    74    81    87
x_med =
   51.5000
x_grades =
     3     6     6    15     3     5
echo off
```

To understand the output you should note that the statement

```
x_mean=p_marks(x)
```

is expecting only 1 piece of output and thus only the mean is computed. However the statement

```
[x_mean,x_std,x_sort,x_med,x_grades]=p_marks(x)
```

is expecting 5 pieces of output and that is what is returned. 5 pieces of output is the maximum this function can return and an error is generated if you put a statement of the form

```
[x1,x2,x3,x4,x5,x6]=p_marks(x)
```

Function names in the argument list and `feval`

In all the examples considered so far the input arguments have all been of the matrix type `double array`, that is they have been scalars, vectors or matrices in which each entry is of type `double`. Numbers of type `double` each require 8 bytes of storage which is 64 bits and which corresponds to about 16 significant decimal digits of precision. You can also pass strings as arguments where a string is almost anything that is enclosed between the single quote character '. For example `'football'`, `'cricket'` `'athletics'` and `'z_7_D9'` are all valid string variables. In many situations it is useful to make a function name a string variable and to put this as an input argument to another function. For example, consider again the Newton iteration described by

$$x_0 \text{ given}, \quad x_{k+1} = x_k - f(x_k)/f'(x_k), \quad k = 0, 1, 2, \cdots$$

In the section on control statements a number of function files implementing this are given and to test each version sub-functions were added called respectively `f` and `df`. This is fine up to a point but it means that these sub-functions have to be changed each time a different function `f` is considered. To avoid this we need a way of passing the names of the function files, which give f and f', as arguments to a Matlab function which implements the Newton iteration. A way of doing this in Matlab is to write the Newton solver in such a way that two of the input arguments are string variables corresponding to the name of the function and to the name of the derivative. Within the Newton solver the built-in function `feval` is used to perform the function evaluations. That is the function file should have statements of the form

```
function root=newton_it(x,f,df)
...
...
v=feval(f,x)/feval(df,x);
x=x-v;
...
```

To attempt to find a root of a function defined in `mkwf9.m` with the derivative function defined in `mkwdf9.m` the reference to `newton_it` needs to be of the form

```
theroot=newton_it(x0,'mkwf9','mkwdf9')
```

A possible form of `newton_it.m` based on the function `newton_ex3.m` described previously in these notes is given below.

```
function r=newton_it(x,f,df)
%NEWTON_IT  The Newton iteration for finding a root of a function.
%           r=newton_it(x,f,df)  Given the starting point x and strings
%           f and df corresponding to the name of a function of one
%           variable and its derivative this function performs up to
%           20 Newton iterations to attempt to locate a root of the
%           function.
%           On exit: r contains the last iterate computed which is the
%           approximate root if the iteration has converged.


fprintf('x=%24.16e\n',x);

% set the tolerance used in the stopping criterion
% and the maximum number of iterations

tol=1e-12;
itmax=20;

% the loop
for it=1:itmax
  v=feval(f,x)/feval(df,x);
  x=x-v;
  fprintf('x=%24.16e\n',x);
  if abs(v)<tol
    break;
  end
end
r=x;

% report whether or not the stopping criterion has been met
if abs(v)<tol
  fprintf('The iteration has terminated with tol=%e\n',tol);
else
  fprintf('***The convergence criterion has not been met ***\n');
  fprintf('*** with itmax,tol=%d, %e ***\n');
end
```

Consider the following function files `mkwf1.m`, `mkwdf1.m`, `mkwf2.m` and `mkwdf2.m`.

```
function y=mkwf1(x)
y=x-exp(-x);
```

```
function y=mkwdf1(x)
y=1+exp(-x);
```

```
function y=mkwf2(x)
y=exp(x)-tan(0.5*x);
```

```
function y=mkwdf2(x)
y=exp(x)-0.5*sec(x).^2;
```

To test newton_it.m the following statements were used.

```
root1=newton_it(1,'mkwf1','mkwdf1')
root2=newton_it(-1,'mkwf2','mkwdf2')
```

This created the output

```
x=  1.0000000000000000e+00
x=  5.3788284273999021e-01
x=  5.6698699140541331e-01
x=  5.6714328598912300e-01
x=  5.6714329040978384e-01
x=  5.6714329040978395e-01
The iteration has terminated with tol=1.000000e-12
root1 =
   0.56714329040978
x= -1.0000000000000000e+00
x= -3.2025016958606023e-01
x= -5.5111995225633041e+00
x= -5.9260494699099775e+00
x= -6.2396971292137930e+00
x= -6.2793709064834875e+00
x= -6.2794363973689586e+00
x= -6.2794363980823107e+00
x= -6.2794363980823178e+00
The iteration has terminated with tol=1.000000e-12
root2 =
  -6.27943639808232
```

The function file euler_ex1.m described previously can similarly be modified so that a function name is passed as a string. The new version is as follows.

```
function [t,y]=euler_mth(f,T,n,y0)
%EULER_MTH  The basic Euler method.
%           [t,y]=euler_mth(f,T,n,y0) This approximately solves
%           dy/ds=f(s,y), y(0)=y0, 0<=s<=T,
%           by using the basic Euler method with n equal steps of size
%           h=T/n.  On output t is a row vector of the n+1 values
%           0,h,2*h,...,n*h=T and y is a row vector of length n+1
%           containing the approx solution, y(k) being the approximate
%           solution at t(k). f is the name (as a string) of
%           the function file defining f(s,y).

% set the increment h, set t and reserve space for y
h=T/n;
t=linspace(0,T,n+1);
y=zeros(1,n+1);

% set y(1) to the given initial value
y(1)=y0;

% now do n steps of Euler's method
for k=1:n
  y(k+1)=y(k)+h*feval(f,t(k),y(k));
end
```

To test the function we use the function file `euler_test_fn1.m` given below.

```
function z=euler_test_fn1(t,y)
z=-y-5*exp(-t)*sin(5*t);
```

We can generate the approximate solution by a statement of the form

```
[t16,y16]=euler_mth('euler_test_fn1',3,16,1);
plot(t16,y16,'x--');
figure(1);
print -deps euler_test_fn1.eps
```

The graphical output generated is shown in Figure 4 and it should be compared with that shown in Figure 1.

Exercise

By editing the function file `simpson_ex1.m` given earlier in the notes create a new function file called `simpson_mth.m` whose first line is

```
function simpson_mth(f,a,b)
```

where `f` is the name of the function to be integrated from `a` to `b`. Note that you will need to remove the subfunction and you need to modify every statement given previously which involves the function `f`. Test `simpson_mth` using the function

$$f(x) = \sin x$$

and with $a = 0$ and $b = \pi$.

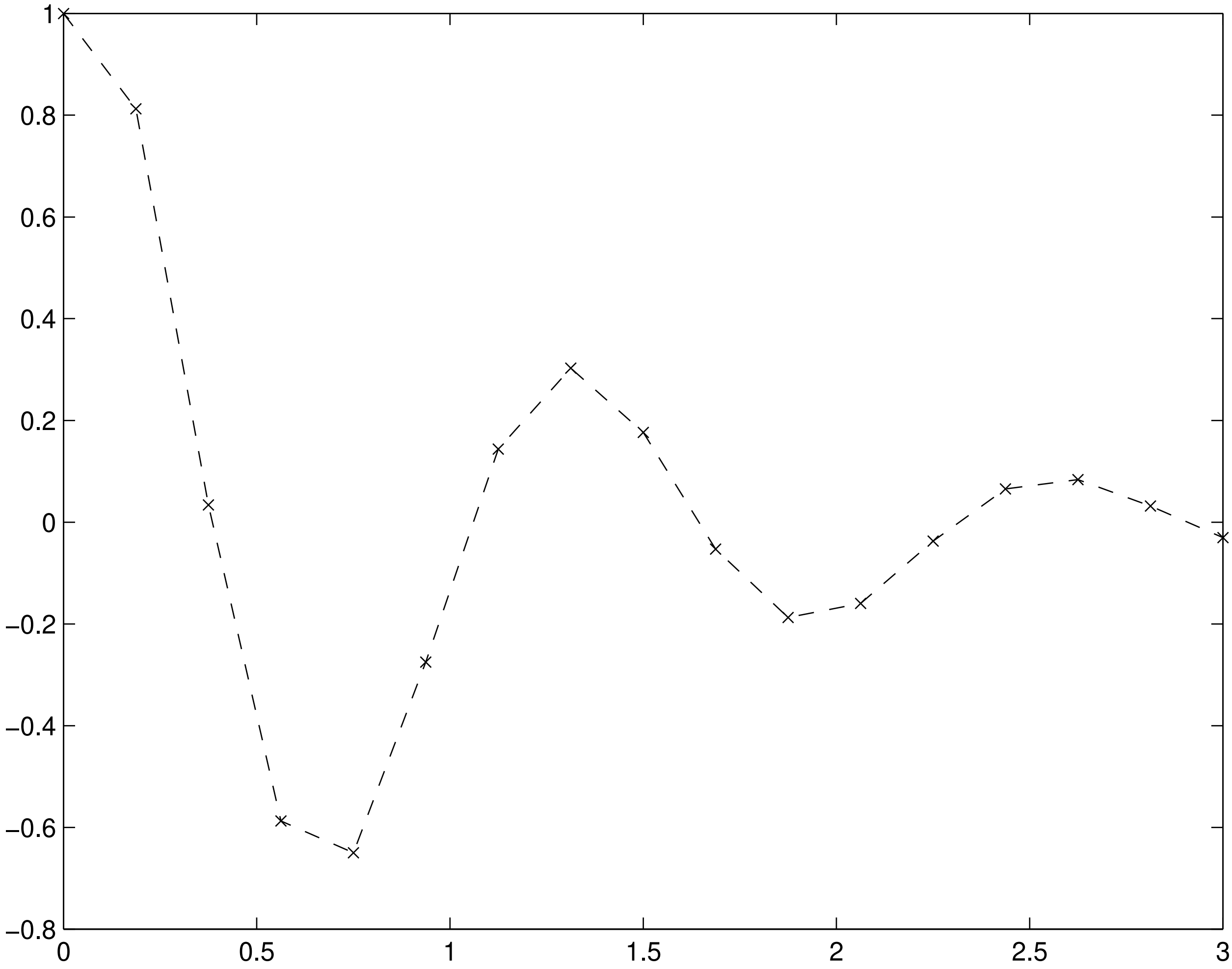

Figure 4: Graphical output from `euler_test_fn1`

4 Solving linear equations in Matlab

Matlab stands for matrix laboratory and one of the main purposes for its creation was to make it easy to create and manipulate matrices and vectors. For example, given a 10×10 matrix `A` and a 10×1 vector `x` within Matlab you can write

```
b=A*x;
```

to create a 10×1 vector `b` corresponding to the matrix multiplication of `A` and `x`. Further, you can write in Matlab

```
y=A\b;
```

which involves solving (numerically) the linear system $A\underline{y} = \underline{b}$. If `A` and `b` are given and if `A` is not singular or nearly singular then the statements

```
b=A*x;
y=A\b;
```

will generate a vector `y` which agrees with `x` to about machine precision accuracy which is about 15 significant decimal digits. The backslash operation \ is Matlab operation for solving linear equations. When Matlab implements this command it will first do a number of tests on the matrix `A` to determine which method it should use to solve the system inorder to accurately and reliably do the calculations.

In the context of boundary value problems (bvp's) which are part of `MA2915` the linear systems that you need to solve generally involve a banded matrix and in particular for bvp's for second order ordinary differential equations it involves a tri-diagonal matrix, i.e. a matrix of the form

$$\begin{pmatrix} d_1 & b_2 & \cdots & & & 0 \\ a_2 & d_2 & b_3 & & & \vdots \\ \vdots & a_3 & d_3 & b_4 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & b_n \\ 0 & \cdots & & & a_n & d_n \end{pmatrix}$$

As the matrix only has $3n-2$ non-zero entries (the n diagonal entries and the $n-1$ entries on the sub- and super- diagonals) it is very wasteful in storage to reserve space for n^2 entries when n is large. Also, to solve the system $T\underline{x} = \underline{b}$ you can perform Gauss elimination in such a way that you basically only need these $3n-2$ entries and you can determine $\underline{x}$ using $\mathcal{O}(n)$ computations (actually about $5n$ operations to be precise). A procedure which does this is part of Matlab and this is what is done when you type

```
x=T\b;
```

provided `T` has already been set-up as a sparse tridiagonal matrix using some of the sparse commands in Matlab. In the case of tridiagonal systems the command `spdiags` is all that is needed. As a small example you can have the following.

```
n=5;
o=ones(n,1);
T=spdiags([o,-2*o,o],-1:1,n,n);
b=zeros(n,1); b(1)=1;
x=T\b;
disp('T='); disp(full(T)) %to display T in full form
disp('b^T=');disp(b')      %to display b as a row
disp('x^T=');disp(x')      %to display x as a row
disp('T*x=');disp((T*x)')  %to check the answer, this should give b
```

The output obtained is shown below.

```
T=
    -2     1     0     0     0
     1    -2     1     0     0
     0     1    -2     1     0
     0     0     1    -2     1
     0     0     0     1    -2
b^T=
     1     0     0     0     0
x^T=
   -0.8333   -0.6667   -0.5000   -0.3333   -0.1667
T*x=
    1.0000   -0.0000   -0.0000    0.0000         0
```

In the above, `[o -2*o o]` creates an array of size $n \times 3$ and `-1:1` is equivalent to the row vector `[-1,0,1]`. Each of the columns of the $n \times 3$ array is 'put' onto a diagonal band of the matrix corresponding to the integer in `[-1,0,1]`. The value `-1` means that the first column is put onto the subdiagonal band (the `T(k+1,k)` values), `0` means that the second column is put onto the diagonal (the `T(k,k)` entries) and `1` means that the third column is put onto the superdiagonal (the `T(k,k+1)` entries). The output above shows what is created.

As a technical point, the special structure of the matrix `T` in the above example with in particular a common entry on each of the diagonal bands means that it is not clear in this case, and indeed it does not matter in this case, which `n-1` of the `n` values are used on the sub- and super- diagonals. You can determine this by reading the help information for `spdiags` or possibly better by noting the output of the following statements.

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

The output obtained is shown below.

```
ans =
    10    -2     0     0     0
     1    20    -3     0     0
     0     2    30    -4     0
     0     0     3    40    -5
     0     0     0     4    50
```

This shows that the first `n-1` values are used on the subdiagonal and the last `n-1` values are used on the superdiagonal.

As a final remark about the above examples you should note that the command `full` converts the sparse form to the full storage form. You should only use this when the matrix involved is small as in these examples. By using the command `whos` you can check whether or not your matrices are being stored in sparse form or full form. For example the statements

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

creates

```
  Name       Size                   Bytes  Class

  FT         5x5                      200  double array
  T          5x5                      180  double array (sparse)

Grand total is 38 elements using 380 bytes
```

If we replace `n=5` by `n=100`, i.e. we have

```
n=100; a=(1:n)';
T=spdiags([a,10*a,-a],-1:1,n,n);
FT=full(T);
whos T FT
```

then this creates

```
  Name       Size                   Bytes  Class

  FT       100x100                  80000  double array
  T        100x100                   3980  double array (sparse)

Grand total is 10298 elements using 83980 bytes
```

As this shows there is virtually no saving in storage with `n=5` but the saving is large when `n=100`.

5 Remarks about debugging commands in Matlab

If the function files that you create are not working as you intend, and assuming that you have removed all syntax errors, then you may wish to make use of some of the debugging commands that are available. If the function file that you wish to debug is called `func1.m` then you can type

```
>>dbstop in func1
```

This sets a break point at the first executable line in `func1.m` so that when you run the program it will pause at this line in the file. It will also change your command prompt to `k>>` and it will start the matlab editor, if it is not already active, with an arrow pointing to the next line to be executed. You are then able to step through the function (i.e. execute one command at a time) and determine the current value of any of the variables by just typing the name of the variable at the `k>>` prompt as you would normally do in Matlab. Typing `dbstep` repeatedly at the `k>>` prompt or repeatedly clicking on the appropriate point on the top bar enables you to do this. If you are in debugging mode (as indicated by the prompt `k>>`) then you return to the usual mode (with the usual `>>` prompt) by typing

```
k>>dbquit
```

For more information on the debugging facilities type

```
>>doc general
```

and scroll down to the debugging section.