Course Notes for MS4024 Numerical Computation Part 1 — Matlab

J. Kinsella

February 6, 2012

Contents

0 About the Course **1**

0.1 Lecture Notes 7

0.2 Module Description/Syllabus 8

0.3 Learning Outcomes 9

0.4 Recommended Texts 10

I Matlab **12**

1 File Management **12**

1.1 Procedure For Uploading Your Work To The Server At The End Of A Class 13

1.2 Procedure For Retrieving Your Work From The Server At The Start Of A Class 17

2 Introduction to Matlab **20**

2.1 Preliminaries 20

2.2 A Quick Tour of **Matlab** 24

3 Matlab Basics **51**

3.1 Interacting with **Matlab** 52

3.1.1 Command Entry 52

3.1.2 Command Syntax and Variables 54

3.1.3 Variable Behaviour 56

3.1.4 Script Files 59

3.2 More Basics 61

3.2.1 Help 64
3.2.2 Some Tips for Command Entry 68
3.2.3 Some Points on Arithmetic 70
3.2.4 Managing Variables 73
3.2.5 Saving Variables to a File 75
3.2.6 Logging a **Matlab** Session 80
3.2.7 The **disp** Command 81
3.2.8 Automatic Storage Allocation 82
3.2.9 How **Matlab** Does Arithmetic 86
3.3 An Example 93

4 Matrices **103**
4.1 Generating Matrices 104

4.1.1 Block Matrix Techniques 110
4.2 Subscripts and Colon Notation 111
4.3 Matrix and Array Operations 120
4.3.1 Scalar Expansion 130
4.4 Matrix Manipulation 133
4.5 Data Analysis 138

5 Loops, If, etc. 146

5.1 Relational and Logical Operators 147
5.1.1 Testing Variables for **complex** in **Matlab** 153
5.1.2 Logical Operators in **Matlab** 154
5.1.3 "Short-Circuiting" 157
5.1.4 Finding Elements in Matrices 161

5.2 Branching Commands 166
5.2.1 If/Then/Else 167
5.2.2 For Loops 170
5.2.3 While Loops, Break and Continue 175
5.2.4 The Switch Statement 179

6 M-Files **182**
6.1 Scripts and Functions 182
6.2 Function M-Files 187
6.2.1 Passing Function Names as Parameters 204
6.3 Naming and Editing M-Files 209
6.4 The **Matlab** Path 210
6.5 Command/Function Duality 215

7 Plotting **219**
7.1 Two-Dimensional Graphics 220
7.1.1 Basic Plots 220
7.1.2 Axes and Annotation 240
7.1.3 Multiple Plots in a Single Figure 256
7.2 Three-Dimensional Graphics 264
7.2.1 Contour Plots 267
7.2.2 Surface Plots 276
7.3 Saving and Printing Figures 280

8 Linear Algebra **287**
8.1 Norms and Condition Numbers 289
8.2 Linear Equations 295

8.2.1 Square Systems 296
8.2.2 Overdetermined Systems 300
8.2.3 Underdetermined Systems 303
8.3 Eigenvalue problems 305
8.3.1 Eigenvalues 306

9 Input and Output **311**
9.1 User Input 312
9.2 Output to the Screen 315
9.3 File Input and Output 322

10 Using Matlab Efficiently **325**
10.1 Vectorisation 328
10.1.1 J.I.T. 334

10.2 Preallocating Arrays 342
10.3 Miscellaneous Optimisations 345

X Supplementary Material **347**

A First Matlab Project for 10% **348**

B Second Matlab Project for 15% **352**

C Third Matlab Project for 25% **359**

D More Examples **381**

0 About the Course

- This course is split into two main parts; Part 1 (on the **Matlab** mathematical programming package) and Part 2 (on the **R** statistical programming package) which will be taught by a Statistics lecturer.
- There will be a short introduction to LaTeX (a mathematical document preparation package) in Week 2 or 3.
- The notes for the LaTeXintroduction may be found at `http://jkcray.maths.ul.ie/ms4024/LaTeX-Files/LaTeX.pdf`
- Part 1 will run in Weeks 1–7 (six weeks on **Matlab** , one on LaTeX),
- Part 2 will run for six weeks — in weeks 8–13 inclusive.
- All classes will be held in C2-062.

- We meet for three hours per week.
- All classes are designated labs, rather than lectures or tutorials.
- Our labs will run from 13:00–16:00 on Mondays.
- Extra slots are available in C2-062 09:00–12:00 Wednesdays.
- These are the scheduled times from Week 8 on but are available from Week 1)
- We'll use them when necessary once work starts on the projects.
- **Extra class time (outside normal lecture hours — i.e. after 18:00) will be provided if necessary to complete projects.**

- In Week 1, I will introduce you to **Matlab** using these Notes — which you will work through on your PC (like the MS4101 Mathematics Laboratory module in Year 1).
- From Week 3 on you will work on your projects in class — with help from me when needed.
- I'll continue to introduce new material from time to time as needed for the projects.
- Once projects begin we will work on them in all three labs each week.
- **A record of attendance will be kept!**
- In class you will learn how to use **Matlab** and be shown how to use it to solve increasingly challenging problems.
- Approximately every two weeks starting in Week 3 you will be assigned a new task/project.

- **To ensure that students get credit for their own work all project work will be done and submitted in class**.
- **You will upload your work to a server at the end of every class and retrieve it at the beginning of the next.**
- The PC's in the lab are configured by ITD to have the student folders wiped daily so you **must** upload your work to the server at the end of each class.

- You will work on the local **C:** drive, in the **WorkArea** folder.
- You will **not** save your work to a usb stick.
- You will use a folder named as your Student ID, in the **WorkArea** folder.
- At the end of a class, navigate to the **C:\WorkArea** folder.
- You will right click on the desktop (blank area) your personal folder (say **123456789**) and choose the option **Send to a zipfile** which creates a compressed (or zipped) file called (say)**123456789.zip** in the **C:\WorkArea** folder.
- **You will use the two links at the foot of the module web page:** `http://jkcray.maths.ul.ie/ms4024.html`
  - To upload your zipfile at the end of a class.
  - To retrieve your zipfile at the start of a class.
- Details in Ch. 1 below.

- **You may — and should — work on your projects between tutorial classes but may not bring your work into class.**
- **The module will be assessed by three assessments during the period of Part 1 (weighted as 10%, 15% and 25%).**
- **There will be no end-of-semester assessment.**

0.1 Lecture Notes

- These notes (for Part 1) are available in printed form from the U.L. Print Room — Ref 5808, price €6.00.
- And may be downloaded from `http://jkcray.maths.ul.ie/ms4024/M-Slides.pdf`.
- You may also download other material including example **Matlab** files and material for LaTeX and for Part 2 from `http://jkcray.maths.ul.ie/ms4024.html`.

0.2 Module Description/Syllabus

- The Matlab language:
  - Introduce Matlab command syntax; Matlab workspace, arithmetic, number formats, variables, built-in functions.
  - Using vectors in Matlab; colon notation.
  - Arrays; array indexing, array manipulation.
  - Two-dimensional graphics; basic plots, axes, multiple plots in a single figure, saving and printing figures.
  - Matlab commands in “batch” mode; script M-files, saving variables to a file, the diary function.

- Relational and logical operations; testing for equality/inequality, and/or/not.
- Control flow: for, while, if/else, case, try/catch.
- Function M-files: parameter passing mechanisms, global and local variables.

- Applications of Matlab; topics to be taken from:
  - Numerical Linear Algebra; norms and condition numbers, solution of linear equations, inverse, pseudo-inverse and determinant, LU and Cholesky factorisations, QR factorisation, Singular Value Decomposition, eigenvalue problems.
  - Polynomials and data fitting.
  - Nonlinear equations and optimisation.
  - Numerical solution of ordinary differential equations.

0.3 Learning Outcomes

Learning Outcome	Assessment Mode
Use Matlab in command mode to perform simple numerical and matrix computations and to generate graphical output.	Lab sessions with submitted report.
Construct Matlab script M-files to perform vector, matrix and general numerical computations.	Lab sessions with submitted script file and submitted report.
Design and code a set of Matlab function M-files to solve an Applied Mathematics problem (see Syllabus).	Lab sessions with submitted function M-files and submitted report.

0.4 Recommended Texts

In addition to these notes, the following books are useful as references.

1. Mastering Matlab 7, D. Hanselman and B. Littlefield, Pearson Education N.J. 2005, ISBN 0131857142, U.L. Library Link.
2. Matlab Guide D.J. Higham & N.J. Higham, SIAM Philadelphia, 2005, ISBN 0898715784, U.L. Library Link.
3. Numerical Computing with MATLAB, Cleve B. Moler, Cambridge University Press, 2004. ISBN 0898715601.
4. Matlab Primer, T. A. Davis, K. Sigmon, CRC Press, 2005. ISBN 1584885238.
5. See also the link `http://jkcray.maths.ul.ie/ms4327.html` for a list of on-line introductions to **Matlab**.

Part I

Matlab

1 File Management

As mentioned in the Introduction, you will upload your work to a server at the end of every class and retrieve it at the beginning of the next.

We will start the first lab by going through this procedure with a practice folder with a couple of dummy files.

The lines in **blue** are the **steps that you will perform** using the Windows File Manager (Windows Explorer) and your Internet Browser (probably Internet Explorer — though Firefox is better).

1.1 Procedure For Uploading Your Work To The Server At The End Of A Class

- Start by loading the Windows File Manager.
  - **Double-click on the Windows File Manager icon.**
- You will work on the local **C:** drive, in the **WorkArea** folder.
  - **Navigate to the folder C:\WorkArea.**
- You will use a folder named as your Student ID, in the **WorkArea** folder.
  - **Create a folder (say 123456789) (use your Student ID)**
  - **Navigate to this folder by double-clicking on the icon.**

- Create three (empty) text files in your new folder called **one.txt**, **two.txt** and **three.txt** so that we will have some files to work with.
  - **Right-click on the pane and choose the Create Text File option.**
  - **Create files called one.txt, two.txt and three.txt.**
- Open each file in NotePad, type in a line of text then save the file.
  - **Double-click on the icon for each file in turn to open it in NotePad.**
  - **Type in a line of text (e.g. "This is file number one" for the first file and so on) for each file in turn.**
  - **Use the File/Save procedure followed by File/Exit to save each file in turn and exit NotePad.**

- Now navigate up one level to the **C:\WorkArea** folder.
  - **Click on the UpOneLevel icon on the top of the File Manager pane.**
- You will create a compressed (or zipped) file called (say)**123456789.zip** containing your personal Student ID folder. The zipfile will be created in the **C:\WorkArea** folder
  - **Right click on your personal folder (say 123456789) and choose the option SendToZipfile**
  - **N.B. On some PC's a different option appears: Create New Archive — if you are presented with this option then select it**. The `WinRAR` program will run. **Important: choose the `ZIP` archive format, not the `RAR` archive format.**
  - An icon for the new file **123456789.zip** will appear in the File Manager pane.

- You will use the secondlast link at the foot of the module web page: `http://jkcray.maths.ul.ie/ms4024.html` to upload your zipfile at the end of a class and to retrieve your zipfile at the start of a class.
  - **Click on the secondlast link in the webpage:** **Click here to upload (project or exam) files.**
  - **Use the browser in the window to select your zipfile.**
  - **Enter your Student ID in the lower box then Click the Upload File button.**
- The lecturer will confirm that your file has been uploaded.
- If you want, you can repeat the process as often as you want.
- Don't do this unnecessarily as it makes work for the lecturer!

1.2 Procedure For Retrieving Your Work From The Server At The Start Of A Class

This is a straightforward procedure! The lecturer will demonstrate it with some sample files.

- The lecturer will tell you your personal code (hash code) to be used to retrieve your zip file so that you can carry on working with the contents.
- The code will be a long string like: `b2e2bff12ca2f7d6dc206431de2257c0f56debd3`.
- Each student will have a different hash code at the start of each class — generated automatically by the system.

- You will use the last link at the foot of the module web page: `http://jkcray.maths.ul.ie/ms4024.html` to retrieve your zipfile at the end of a class and to retrieve your zipfile at the start of a class.
  - **Click on the last link in the webpage:**
    Location of your saved (Matlab) zip files.
- When you click on the link you will see a list of directory/folder links whose names are the hash codes.
- Click on the link corresponding to the hash code you were given.
  - **Right-Click on the zipfile in the folder (it should be called 123456789.zip) and select the SaveFileAs option.**

- Download the zipfile to the **C:\WorkArea** folder.
- Unzip the zipfile, re-creating your personal folder.
  - **Right-Click on the zipfile in the File Manager pane and select ExtractHere.**
- Now let's do some real work — time to start learning about **Matlab**!

2 Introduction to Matlab

2.1 Preliminaries

The best way to learn **Matlab** is by trying it yourself!

- You have access to **Matlab** (Version 7.10.0.499 (R2010a)) on MS Windows XP in the M.S. Dept. classroom C2-062.
- **Matlab** also runs on Apple Macs and on Linux (which is the O.S. upon which these Notes were prepared).
- A much older version — Version 6.5.0.180913a Release 13— is also installed on student PC's throughout U.L.

The first thing that you will realise is that **Matlab** has some similarities with Maple — but that it has many differences!

Feature	Maple	**Matlab**
Semicolon “;” at end of line	terminates line	suppresses output
Colon “:” at end of line	terminates line and suppresses output	to reference array elements
Comma“,”	used to separate items in a list	to separate commands on same line
Case sensitive	Yes	Yes
Does symbolic algebra	Yes	Only if Symbolic Toolbox available

Table 1: Comparing Maple and **Matlab**.

Feature	Maple	**Matlab**
Define array or list	$A :=< 1, 2, 3 >$ $A := [1, 2, 3]$	$A = [1, 2, 3]$ (or just $A = [1\,2\,3]$)
Elements of array	$A[i]$, $A[i, j]$	A(i), A(i,j) A(:,j) is j^{th} column
Variable behaviour	Automatic recalculation of expressions when variables change. $a := 1 : b := a :$ $a := 2;\ b;$ b changes.	No recalculation. Similar to Java. $a = 1;\ b = a;$ $a = 2;\ b$ b does not change!

Table 2: Comparing Maple and **Matlab** , continued.

It is probably easiest to temporarily “forget” Maple when learning **Matlab**!

2.2 A Quick Tour of Matlab

This short Section gives an overview of what **Matlab** can do. We will not fill in all the details at this stage.

The up arrow and down arrow keys can be used to scroll through your previous commands. Also, an old command can be recalled by typing the first few characters followed by up arrow. You can type **help topic** to access online help on the command, function or symbol topic.

You can quit **Matlab** by typing exit or quit.

Having entered **Matlab** , you should work through this tutorial by typing in the boxed text after the **Matlab** prompt, >> , in the Command Window.

We begin with:

```
a=[1 2 3]
```

This means that you are to type "a = [1 2 3]" and press **Enter** , after which you will see **Matlab** 's output "a =" and "1 2 3" on separate lines separated by a blank line.

(In these notes, to save space, we generally will not display the output of **Matlab** commands. When results are displayed, blank lines will often be omitted.)

This example sets up a 1×3 array a (a row vector).

In Matlab , row vectors are the default — in mathematics, vectors (as distinct from the coordinates of a point) are usually represented as a column. This never causes a problem as Matlab makes it easy to "flip" (transpose) a row vector into a column vector or vice versa.

In the next example, semicolons separate the entries:

```
c = [4; 5; 6]
```

A semicolon tells **Matlab** to start a new row, so c is 3×1 (a column vector).

You can also simply press Enter at the end of each row — this is useful when typing in matrices.

Now you can multiply the arrays a and c:

```
a*c
```

Here, you performed an inner product: a 1×3 array multiplied into a 3×1 array.

Matlab automatically assigns the result of a command to the variable **ans**, which is short for answer.

You may also form the outer product:

```
A = c*a
```

```
A =

     4     8    12
     5    10    15
     6    12    18
```

Here, the answer is a 3×3 matrix — we've saved the answer as `A`.

The product **a*a** is not defined, since the dimensions are incompatible for matrix multiplication— try it:

```
a*a
```

You'll see an error message:

```
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Arithmetic operations on matrices and vectors come in two distinct so-called **Senses**:

1. "Matrix Sense" operations are based on the normal rules of linear algebra and are obtained with the usual symbols `+`, `-`, `*`, `/` and `^`.
2. "Array Sense" operations are defined to act elementwise and are obtained by **preceding the symbol with a dot**.

So if you want to square **each element** of `a` you can write

```
b = a.^ 2
```

Since the new vector `b` is 1×3, like `a`, you can form the array product of it with `a`:

```
a.*b
```

Matlab has many mathematical functions that operate in the array sense when given a vector or matrix argument. For example,

```
exp(a)
log(ans)
sqrt(a)
```

Matlab displays floating point numbers to 5 decimal digits, by default, but always stores numbers and computes to the equivalent of 16 decimal digits. The output format can be changed using the **format** command:

```
format long
sqrt(a)
format
```

The last command reinstates the default output format of 5 digits.

Very large and very small numbers are displayed in **exponential notation**, with a power of 10 scale factor preceded by e:

```
2^(-24)
```

```
ans =
     5.960464477539062e-08
```

```
2^(240)
```

```
ans =
     1.766847064778384e+72
```

Various data analysis functions are also available:

```
sum(b), mean(c)
```

As this example shows, you may include more than one command on the same line by separating them with commas.

If a command is followed by a semicolon then **Matlab** suppresses the output:

```
pi
y = tan(pi/6);
```

The variable **pi** is a permanent variable with value $\pi \approx 3 \cdot 14159265358979$.

The variable **ans** always contains the result of the most recent "unassigned" expression, so after the assignment to **y**, **ans** still holds the value π.

You can set up a matrix (**Matlab** calls them **two-dimensional arrays**) by using spaces to separate entries within a row and semicolons to separate rows:

```
B = [−301 25; −7 −148]
```

```
B =
 -301      25
   -7    -148
```

Or you can type

```
B = [−301 25 <ENTER>
−7 −148]
```

where `<ENTER>` means "press the Enter key".

This is usually the easiest way to type in a matrix.

At the heart of **Matlab** is a vast range of Linear Algebra functions.

For example, if we re-define **c** to be a 2×1 vector, we can solve the linear system $\mathbf{B} * \mathbf{x} = \mathbf{c}$.

This can be done with the backslash operator:

```
c=[1;2]; x = B \c
```

```
x =


   -0.004427252196856
   -0.013304116450149
```

(Think of this as "c over B", i.e. $\mathtt{B}^{-1} * \mathtt{c}$.)

The order is important — we can use the "forwards slash" operator to form:

```
x = c/B
```

though still "c over B" this gives an error

```
??? Error using ==> mrdivide
Matrix dimensions must agree.
```

as it means $\mathtt{c} * \mathrm{B}^{-1}$ which does not make sense as $\mathbf{c}$ is a column vector.

We will discuss this topic in detail later.

Note that the error generated by the last command does not change the value of **x**.

You can check the result by computing the residual:

```
r=B*x-c
```

```
r =
   1.0e-15 *


   0.444089209850063
                   0
```

Although the answer "should be" a 2×1 vector of zeroes , it is as close to zero as we could expect — given that **Matlab** works to about 16-digit (decimal) accuracy.

Notice how Matlab displays the "common factor" of `1.0e-15`.

The eigenvalues of B can be found using `eig`:

```
e = eig(B)
```

```
e =

   1.0e+02 *

 -2.998475281611812
 -1.491524718388188
```

You may also specify two output arguments for the function eig:

```
[V,D] = eig(B)
```

```
V =
  -0.998939137463588  -0.162451849591476
  -0.046049968984827  -0.986716472227107


D =
   1.0e+02 *
  -2.998475281611812                   0
                   0  -1.491524718388188
```

In this case the columns of V are eigenvectors of B and the diagonal elements of D are the corresponding eigenvalues.

We can check this by calculating BV − VD (**not** BV − DV — you should know why from Linear Algebra 1!).

```
B*V-V*D
```

```
ans =

   1.0e-14 *
                  0   0.710542735760100
   0.177635683940025                  0
```

Again the answer “should” be exactly zero but is computed to be very close to zero.

The colon notation is useful for constructing vectors of equally spaced values.

For example,

```
v = 1:6
```

generates a vector of the numbers from 1 to 6.

In general, **m:n** generates the vector with entries $\mathbf{m}, \mathbf{m+1}, \ldots, \mathbf{n}$. Nonunit spacing can be specified with **m:s:n**, which generates entries that start at m and increase (or decrease) in steps of s as far as n:

```
w = 2:3:10
y = 1:-0.25:0
```

You can construct bigger matrices out of smaller ones by following the conventions that (a) square brackets enclose an array, (b) spaces or commas separate entries in a row and (c) semicolons separate rows (blank lines omitted in output):

```
C = [A, [8; 9; 10] ], d=[4 5], D = [B; d]
```

```
C =
     4     8    12     8
     5    10    15     9
     6    12    18    10
d =
     4     5
D =
  -301    25
    -7  -148
     4     5
```

The element in row i and column j of the matrix C (**i and j always start at 1**) can be accessed as **C(i, j)**:

```
C(2,3)
```

```
ans =

    15
```

More generally, **C(i1 : i2, j1 : j2)** picks out the submatrix formed by the intersection of rows i_1 to i_2 and columns j_1 to j_2.

```
C(2:3,1:2)
```

```
ans =

     5    10
     6    12
```

You can build certain types of matrix automatically.

For example, identities and matrices of 0s and 1s can be constructed with **eye**, **zeros** and **ones**:

```
I3 = eye(3,3), Z34 = zeros(3,5), O2 = ones(2)
```

```
I3 =

     1     0     0

     0     1     0

     0     0     1

Z35 =

     0     0     0     0     0

     0     0     0     0     0

     0     0     0     0     0

O2 =

     1     1

     1     1
```

Note that for these functions the first argument specifies the number of rows and the second the number of columns.

If both arguments are the same then only one need be given.

```
I3=eye(3),Z4=zeros(4)
```

```
I3 =
     1     0     0
     0     1     0
     0     0     1
Z4 =
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

N.B. the variable names `I3`, `Z4` etc used above are chosen for clarity — Matlab doesn't care what names are used!

The functions **rand** and **randn** work in a similar way, generating random entries from the uniform distribution over $[0, 1]$ and the normal $(0, 1)$ distribution, respectively.

If you want the numbers to be the same, you should set the state of the two random number generators.

Here, they are set to 20 (short format so numbers will fit on screen):

```
format short
rand('state',20), randn('state',20)
F = rand(3), G = randn(1,5)
```

```
F =
    0.5093    0.2817    0.0052
    0.9339    0.5119    0.1787
    0.0179    0.1258    0.0567
G =
   -1.2668   -1.8050   -0.7018   -1.1612   -0.7991
```

Single (right) quotes act as string delimiters, so **'state'** is a string.

Many **Matlab** functions take string arguments.

At this stage you have created quite a few variables in the workspace. You can obtain a list with the **who** command:

```
who
```

Your variables are: A F Y b w B G Z c x C I3 a e y D V ans v

(The output you get may be different — especially if you skipped some of the commands above!)

Alternatively, type **whos** for a more detailed list showing the size and class of each variable, too.

Loops Like most programming languages, **Matlab** has loop constructs. The following example uses a **for** loop to evaluate a "continued fraction"

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\dots}}}}$$

that approximates the golden ratio, $(1+\sqrt{5})/2$.

```
g = 2; % Try changing this --- does it change the final value of g?
for k=1:20, g = 1 + 1/g; end % And try changing 20 to (say) 200
g-(1+sqrt(5))/2
```

```
ans =

      1.425702222945802e-09
```

(So the final value of g is very close to $(1+\sqrt{5})/2$.)

We will study **for** and **while** loops later in the course.

Plots The **plot** function produces two-dimensional (2D) pictures.

Suppose that we want to plot $y(x) = e^{10x(x-1)} \sin(12\pi x)$.

- In Maple we would just type in the formula for $y(x)$.
- In **Matlab** we must create a vector **x** of “x-values” (of course any letter or variable name can be used) and a corresponding vector **y** of “y-values” each element **y(i)** of which is the function y evaluated at **x(i)**.
- A subtle point is that to apply the function y element-by-element to the vector **x** we must use **.*** rather than * when forming the products of factors that make up $y(x)$.

So we enter:

```
x = 0:0.005:1;
y = exp(10*x.*(x-1)).*sin(12*pi*x);
plot(x,y)
```

Here, **plot(x,y)** joins the points **x(i),y(i)** using the default solid linetype.

Matlab opens a figure window in which the picture is displayed.

See Fig. 1 on the next Slide.

Try changing the “function” to be plotted and the domain (range of x–values).

Much more later on plotting in **Matlab** in Section 7.1.1.

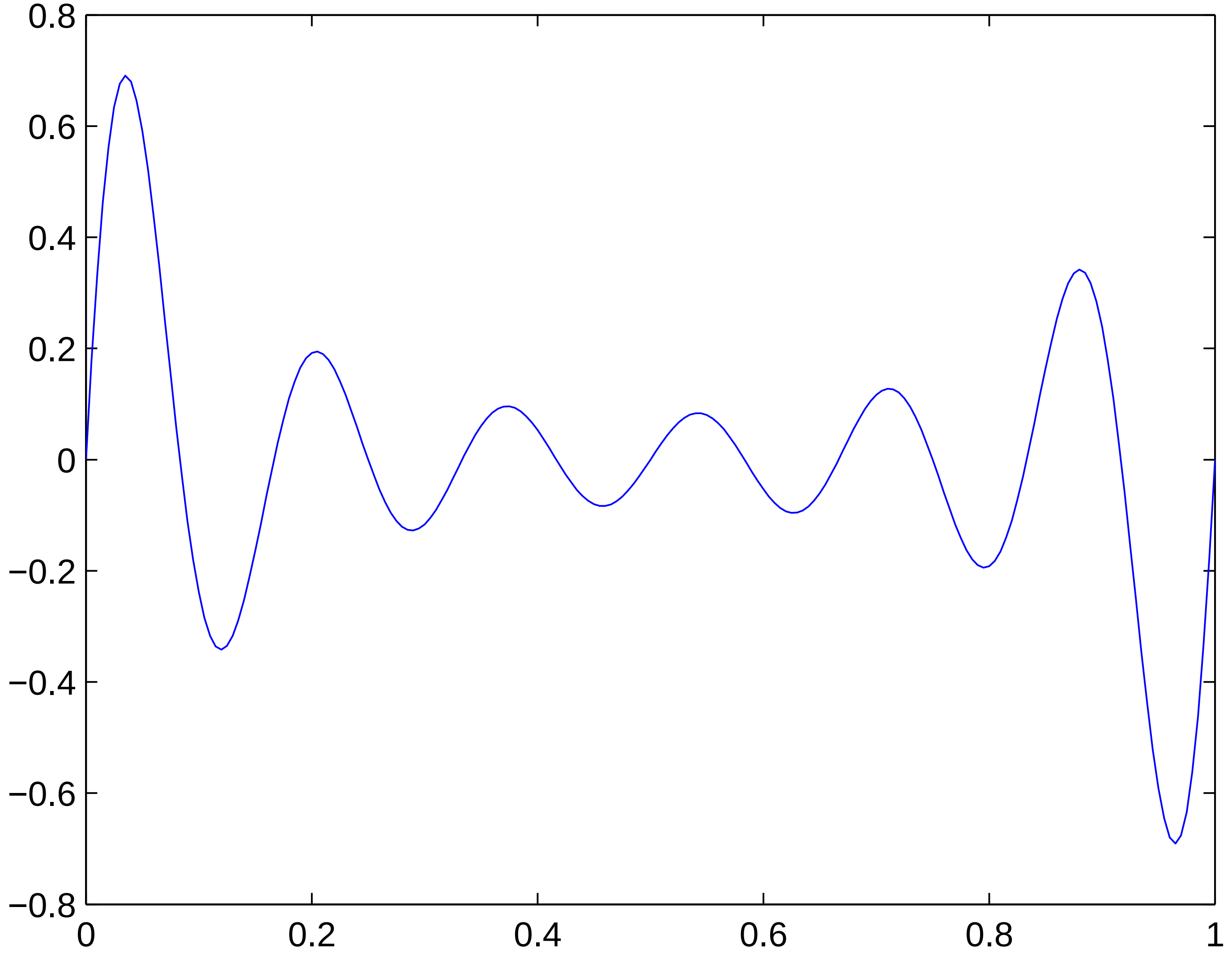


Figure 1: Plot of y(x)=$e^{10x(x-1)}\sin(12\pi x)$

3 Matlab Basics

- Just as in the previous Chapter, when working though this and later Chapters you should enter (copy/paste) and execute each command below at the **Matlab** command prompt.
- Again, as in the first Chapter, the results displayed by **Matlab** are not usually shown here to save space.
- We will be more thorough from now on in defining and explaining new **Matlab** ideas.
- We will **not** go through the following material line-by-line in class.
- Instead we will move rapidly through the Chapters, taking topics as needed for the three Projects.
- The majority of the material is intended as a Reference.

3.1 Interacting with Matlab

3.1.1 Command Entry

- **Matlab** is an interactive system.
- You type commands at the prompt

  >>

  in the Command Window and computations are performed when you press the enter or return key.
- At its simplest level, **Matlab** can be used like a pocket calculator:

```
(1+sqrt(5))/2
2 ^(-53)
```

- The first example computes $(1+\sqrt{5})/2$ and the second 2^{-53}.
- Note that the second result is displayed in exponential notation: it represents 1.1102×10^{-16}.
- The variable **ans** is created (or overwritten, if it already exists) when an expression is not assigned to a variable.
- It can be referenced later, just like any other variable.

3.1.2 Command Syntax and Variables

Matlab is case sensitive. This means, for example, that x and X are distinct variables.

Unlike most programming languages, variables are not declared prior to use but are created by **Matlab** when they are assigned:

```
x = sin(22)
```

Here we have assigned to x the sine of 22 radians. The printing of output can be suppressed by appending a semicolon. The next example assigns a value to y without displaying the result:

```
y = 2*x + exp(-3)/(1+cos(.1));
```

Commas or semicolons are used to separate statements that appear on the same line:

```
x = 2, y = cos(.3), z = 3*x*y
x = 5; y = cos(.5); z = x*y ^ 2
```

Note again that the semicolon causes output to be suppressed.

3.1.3 Variable Behaviour

This very short sub-section is also very important!

- As noted briefly in the Table (Slide 22) of similarities and differences between Maple & **Matlab** , Maple automatically recalculates expressions (formulas) when the variables mentioned in the formula change.
- **Matlab** does not.
- In this respect **Matlab** is much more like Java in its behaviour than Maple is.

For example (as in the Table) if we execute the following **Matlab** commands

```
a=1;b=a
a=2;b
```

the value of b does not change when a does — even though b is defined in terms of a.

- In other words, in **Matlab** , variables are just labels or names for data.
- A variable may be defined in terms of one or more other variables (by an expression or formula).
- But the definition just uses the values of the component variables at the time of the definition.
- Subsequent changes in the value of the component variables do not affect the value of any variable defined in terms of them.

One last example:

```
1 a=1;b=2;c=3;
2 my_var=a^2+b^2+c^2
3                              % my_var is a variable
4                              % defined using an expression or formula
5 a=2;b=4;c=6;
6 my_var
7                              % Has my_var changed in value? (No.)
```

Check that the value of `my_var` does not change. from Line 2 to Line 4.

3.1.4 Script Files

To perform a sequence of related commands, you can write them into a script M-file, which is a text file with a .m filename extension. For example, suppose you wish to process a set of exam marks using the **Matlab** functions **sort, mean, median** and **std**, which, respectively, sort into increasing order and compute the arithmetic mean, the median and the standard deviation. You can create a file, say **marks.m**, of the form

Listing 1: marks.m

```
% MARKS
exmark = [12 0 5 28 87 3 56];
exsort = sort(exmark)
exmean = mean(exmark)
exmed = median(exmark)
exstd = std(exmark)
```

- The % denotes a comment line.
- You can load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/marks.m`.
- Use the **Matlab** editor to edit the file **marks.m** and replace the marks by a new list of 10 numbers between 0 and 100.
- Run it ("call it") by typing **marks** at the command prompt.
- Calling **marks** is entirely equivalent to typing each of the individual commands in sequence at the command line.
- More about Script (and Function) M-files in Section 6.1.
- To quit **Matlab** type **exit** or **quit**.

3.2 More Basics

Matlab has many useful functions in addition to the usual ones found on a pocket calculator. For example, you can set up a random 3×3 matrix by typing

```
A = rand(3)
```

Here each entry of A is chosen independently from the uniform distribution on the interval $[0; 1]$. The **inv** command inverts A:

```
inv(A)
```

The inverse has the property that its product with the matrix is the identity matrix. We can check this property for our example by typing

```
ans*A
```

- The product has 1's on the diagonal, as expected.
- The off-diagonal elements, displayed as plus or minus 0.0000, are, in fact, not exactly zero. **Matlab** stores numbers and computes to a relative precision of about 16 decimal digits.
- By default it displays numbers in a 5-digit fixed point format.
- While concise, this is not always the most useful format.
- The format command can be used to set a 5-digit floating point format (also known as scientific or exponential notation):

```
format short e
```

```
ans
```

- Now we see that the off-diagonal elements of the product are nonzero but tiny — the result of rounding errors.

- The default format can be reinstated by typing **format short**, or simply **format**.
- The **format** command has many options, which can be seen by typing **help format**.

3.2.1 Help

- Generally, **help TOPIC** displays information on the command or function named **TOPIC**. For example try:

```
help sqrt
```

- For some fun try

```
help spy
```

and then

```
spy
```

- Then try the same process with the **why** command!

- The nicest "Easter Egg" in **Matlab** is (sound card/speakers needed)

```
% Handel's Messiah!!!
load handel
sound(y,Fs)
```

And that's enough fun!

- Note that it is a convention that function names are capitalized within help lines, in order to make them easy to identify.
- The names of all functions that are part of **Matlab** or one of its toolboxes should be typed in lower case, however.
- The most comprehensive documentation is available in the Help Browser, which provides help for all **Matlab** functions, release and upgrade notes, and online versions of the complete **Matlab** documentation in html and pdf format.
- The Help Browser includes a Help Navigator pane containing tabs for a Contents listing, an Index listing, a Search facility, and Favorites.
- The attached display pane displays html documentation containing links to related subjects and allows you to move back or forward a page or to search the current page.

- The Help Browser is accessed by clicking the "?" icon on the toolbar of the **Matlab** desktop, by selecting Help from the Help menu, or by typing **helpbrowser** at the command line prompt.
- You can type **doc TOPIC** to call up help on function `TOPIC` directly in the Help Browser.
- Typing **helpwin** calls up the Help Browser with the same list of directories produced by help; clicking on a directory takes you to a list of M-files in that directory and you can click on an M-file name to obtain help on that M-file.
- A useful search facility is provided by the **lookfor** command. Type **lookfor keyword** to search for functions relating to the keyword. Example:

```
lookfor rand
```

3.2.2 Some Tips for Command Entry

- If you make an error when typing at the prompt you can correct it using the arrow keys and the backspace or delete keys.
- Previous command lines can be recalled using the up arrow key, and the down arrow key takes you forward through the command list.
- If you type a few characters before hitting up arrow then the most recent command line beginning with those characters is recalled.
- "Tab completion": If you type the first few characters of a variable or function and then press the `<TAB>` key, **Matlab** will try to complete the name and will offer you a choice of matching names if there is more than one. Try

  ```
  >> pl<TAB>
  ```

Some Tips for Command Entry continued

- You can enter multiple lines at the command prompt and run them all at once by pressing `<SHIFT-ENTER>` at the end of each line then `<ENTER>` at the end of the last line to execute them all — like in Maple.
- A **Matlab** computation can be aborted by pressing `<Ctrl-C>` (holding down the **control** key and pressing the **c** key). If **Matlab** is executing a built-in function it may take some time to respond to this keypress.
- A line can be terminated with three full stops "periods" (...), which causes the next line to be a continuation line:

```
x = 1 + 1/2 + 1/3 + 1/4 + 1/5 + ...
1/6 + 1/7 + 1/8 + 1/9 + 1/10
```

3.2.3 Some Points on Arithmetic

The value of **x** above illustrates the fact that, unlike in some other programming languages, arithmetic with integers is done in floating point and can be written in the natural way.

- As we have already seen **pi** is $\pi = 3.14159\ldots$ to about 16 places.
- The square root of minus one **i** is $\sqrt{-1}$, as is **j** — electrical engineers use j instead of i which explains the following:

```
i-j
```

```
ans =
     0
```

- Complex numbers are entered as, for example, **2-3i**, **2-3*i**, **2-3*sqrt(-1)**, or **complex(2,-3)**.
- Note that the form **2-3*i** may not produce the intended results if i is being used as a variable (say the counter in a **for** loop), so the other forms are generally preferred.
- For example:

```
i =3;
z=2-3i;
z^2
w=2-3*i;
w^2
```

```
ans =
   -5.0000 -12.0000i
ans =
     49
```

More Points on (Complex) Arithmetic

- **Matlab** fully supports complex arithmetic.
- For example

```
clear i % resets i to sqrt(1)
w = (-1)^ 0.25;
wbar=conj(w);
w *wbar % should be 1
real(w)
imag(w)
r=abs(w) % should also be 1
phase=angle(w) % angle in radians that w makes with positive x axis
                % should be pi/4
r*exp(i*phase)-w % should be zero!
```

3.2.4 Managing Variables

- It is possible to override "built-in" variables and functions by creating new ones with the same names.
- This is **not** a good idea as it can lead to confusion.
- However, i and j are often used as counting variables — this will occasionally cause you problems when you forget that you have changed their values from the deafult of the square root of minus one.
- For example,

```
i=7, pi=2
sin(pi), i^2
```

Managing Variables continued

- Variable names are **case sensitive** and can be up to 31 characters long, consisting of a letter followed by any combination of letters, digits and underscores.

```
X =1; x=2; X−x
```

- A list of variables in the workspace can be obtained by typing **who**, while **whos** shows the size and class of each variable as well.
- An existing variable **var** can be removed from the workspace by typing **clear var**, while **clear** clears all existing variables.
- For example,

```
clear pi i
```

restores the default values of **pi** and **i**.

3.2.5 Saving Variables to a File

- To save variables (**not** commands) for recall in a future **Matlab** session type

```
save filename
```

- All variables in the workspace are saved to a binary (not viewable with a text editor) file **filename.mat**.
- Alternatively, **save filename A B** saves just the variables **A** and **B**.
- The command **load filename** loads in the variables from **filename.mat**, and individual variables can be loaded using the same syntax as for **save**.

For example, the following listing

- clears the workspace,
- creates some variables,
- saves the workspace,
- clears the workspace,
- reloads the variables into the workspace.

```
clear
x=1, y=2, z=3
save myfile
clear
whos % check no variables are defined
load myfile
whos % check variables have been recovered
x,y,z % display them
```

Saving Variables to a File continued

- The default is to save and load variables in binary (not readable in a text editor) form but you can save your work as a text file (**mytextfile.txt**, say) by entering

```
x=1,y=2,z=3
save mytextfile.txt -ascii
clear
whos % check no variables are defined
load mytextfile.txt
whos % check variables have been recovered - or have they?
```

- But

```
x,y,z % display them
```

gives an error!

Saving Variables to a File continued There are three reasons **not** to save variables in ascii/text files.

1. • The first limitation (illustrated on the previous Slide) is that the variable names (in this case **x,y,z**) are **not** stored when we save some or all of our variables in a file with the **ascii** option.

   • The values are loaded into a **new** variable called `mytextfile` — it is up to us to decide create new variables corresponding to the elements of the data loaded.

2. • Another problem is that **Matlab** appends each variable's value to the text-file row-by-row.

   • As a result, unless the variables are all the same size (number of rows and columns) it will not be possible to re-load the saved data.

- Another simple example to illustrate the problem.

```
a=rand(3,3);b=rand(2,2);c=5;
save -ascii myfile.txt % save all the variables
clear % clear the workspace
load -ascii myfile.txt % load all the variables
```

  — gives an error.

- Use the **Matlab** editor to look at the file **myfile.txt** to see why.
- This is a second reason why it is better to use **Matlab** 's native **mat**-file format — despite the disadvantage of not being able to view the contents with a text editor.

3. The third reason is that native **mat**-file format files are readable by **Matlab** on different operating systems (MS Windows/Linux/Apple Mac).

3.2.6 Logging a Matlab Session

- Often you need to capture **Matlab** output for incorporation into a report.
- This is most conveniently done with the **diary** command.
- If you type **diary filename** then all subsequent input and (most) text output is copied to the specified file; **diary off** turns off the **diary** facility.
- After typing **diary off** you can later type **diary on** to cause subsequent output to be appended to the same **diary** file.
- You may be asked to log a **Matlab** session using this technique as part of a project.

3.2.7 The disp Command

To print the value of a variable or expression without the name of the variable or ans being displayed, you can use disp:

```
A = eye(2); disp(A)
```

This provides a quick way to output matrices with column headings, e.g. (from **doc disp**)

```
  disp('  Corn  Oats  Hay')
  disp(rand(5,3))
```

3.2.8 Automatic Storage Allocation

- **Matlab** has features that distinguish it from most other modern programming languages and problem solving environments.
- As we saw earlier, variables need not be declared prior to being assigned. This property is referred to as **automatic storage allocation**. it applies to arrays as well as scalars.
- **Matlab** automatically expands the dimensions of arrays in order for assignments to make sense.

- Starting with an empty workspace, we can set up a 1-by-3 vector x of zeros with

```
x(3) =0
```

and then expand it to length 6 with

```
x(6) = 0
```

More about Automatic Storage Allocation However...

- There is a price to be paid for using this technique when solving large problems.
- **Matlab** will slow down if Automatic Storage Allocation is used with large matrices.
- Instead of:

```
for i =1:10000, x(i)=i/(i+1); end
```

- It is much better to “pre-allocate” memory by first executing:

```
x=zeros(10000);
```

- How great a difference pre-allocation makes depends on the version of **Matlab** in use.

- Experiment: first type `clear` then run this command twice:

```
tic; for i =1:10000 x(i)=i/(i+1); end, toc
```

- You should find that the "elapsed time" is much less the second time as **Matlab** has allocated space for the array `x`.
- More on this topic later in Section 10.2.

3.2.9 How Matlab Does Arithmetic

- **Matlab** carries out all its arithmetic computations in double precision floating point arithmetic. The function **computer** returns the type of computer on which **Matlab** is running:

```
computer
```

- On your P.C. running M.S. Windows you should see **ans = PCWIN**.

- In **Matlab** 's **double** data type each number occupies a 64-bit word. Nonzero numbers range in magnitude between approximately 10^{-308} and 10^{+308} and the unit roundoff is $2^{-53} \approx 1.11 \times 10^{-16}$.

How Matlab Does Arithmetic continued

- The significance of the unit roundoff is that it is a lower bound for the relative error in converting a real number to floating point form and also for the relative error in (say) adding, subtracting, multiplying or dividing two floating point numbers.
- In simple terms, **Matlab** stores floating point numbers and carries out elementary operations to an accuracy of about 16 significant decimal digits.
- The function **eps** returns the distance from 1.0 to the next larger floating point number:

```
eps
```

How Matlab Does Arithmetic continued

- Try the following:

```
% Estimates eps
ep=1; while (1+ep>1) ep=ep/2; end;ep=ep*2
```

- Compare the value of **ep** with that of **eps**.
- The unit roundoff is just **eps/2**.
- Compute **(1+eps/2)-1** to confirm this.

How Matlab Does Arithmetic (Technical Details)

- If the result of a computation is larger than the value returned by the function **realmax** then overflow occurs and the result is **Inf** (also written **inf**), representing infinity.
- Similarly, a result more negative than **-realmax** produces `-inf`. Example:

```
realmax
```

```
−2*realmax
```

```
1.1*realmax
```

- A computation whose result is not mathematically defined produces a **NaN**, standing for **Not a Number**.

How Matlab Does Arithmetic (Technical Details) Continued

- A **NaN** (also written **nan**) is generated by expressions such as **0/0**, **inf/inf** and **0 *inf**.

```
0/0
inf/inf
inf-inf
```

- Once generated, a NaN "contaminates" all subsequent computations:

```
nan-nan
0*nan
```

- The function **realmin** returns the smallest positive normalized floating point number.

How Matlab Does Arithmetic (Technical Details) Continued

- Any computation whose result is smaller than **realmin** either underflows to zero if it is smaller than `eps * realmin` or produces a "subnormal" number—one with leading zero bits in its mantissa.
- To illustrate:

```
realmin
```

```
realmin*eps
```

```
realmin*eps/2
```

How **Matlab** Does Arithmetic — Built-In Functions

- Finally, **Matlab** contains a large set of mathematical functions.
- Typing **help elfun** and **help specfun** calls up full lists of elementary and special functions.

3.3 An Example

We will finish this introduction to **Matlab** basics with an example that shows how differently — and how cleverly — **Matlab** works as compared with Maple.

Suppose that we want to plot a function of two variables $f(x, y)$. This is easily done in Maple using plot3d.

In **Matlab** there are many ways of generating a "surface plot" $z = f(x, y)$.

The following is one way to do it. Given a function $f(x, y)$ and a region of the x–y plane we need to

1. generate a "grid" of points (x, y) in this region
2. then evaluate the function $f(x, y)$ at each point.
3. We can then use the **Matlab** **surf** command to create a smooth surface plot.

1. - Suppose that the x-range is 0 to 5 and the y-range is -2 to 3 and lets take the grid points equally spaced at unit intervals.
   - So

```
x=0:5 % Six equally spaced points
y=-2:2 % Five equally spaced points
```

```
x =
       0       1       2       3       4       5
y =
      -2      -1       0       1       2
```

   - What we want is a rectangular "grid" of points, the top left would be $(0, -2)$, the top right $(5, -2)$, the lower left $(0, 3)$ and the lower right $(5, 3)$ (the convention is that y "increases down").

- Here's the key observation: the `x`–coordinates of all the points in a given **column** are the same.
- And the `y`–coordinates of all the points in a given **row** are the same.
- So what we need to do is to generate two matrices/arrays:
  - a matrix/array `X` the same shape/size as our required grid of points; `X` will just consist of the vector `x` (treated as a row vector) copied as many times as `y` has elements.
  - a matrix/array `Y` also the same shape/size as our required grid of points; `Y` will just consist of the vector `y` (treated as a column vector) copied as many times as `x` has elements.
- Is it obvious that `X` and `Y` will be the same shape/size as our required grid of points?

- How to create `X` and `Y`?
- Fortunately **Matlab** has the built-in command **meshgrid** to do the job.
- Let's use the vectors `x` and `y` from above.

- We just enter:

```
[X,Y] = meshgrid(x,y)
```

```
X =

     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5

Y =

    -2    -2    -2    -2    -2    -2
    -1    -1    -1    -1    -1    -1
     0     0     0     0     0     0
     1     1     1     1     1     1
     2     2     2     2     2     2
```

- It does not matter whether x and y are row or column vectors, `meshgrid` treats x as a row vector and y as a column vector.
- Notice that `X` just consists of the vector `x` (treated as a row vector) copied as many times as y has elements.
- And `Y` just consists of the vector `y` (treated as a column vector) copied as many times as x has elements.
- So (for example) the (x, y) coordinates of the point in the second row, third column of our grid is just $(X(2, 3), Y(2, 3))$.
- So instead of creating a grid of ordered pairs (x, y), we have created two matrices,
  - one containing the x–coordinate of each point in the grid,
  - the second containing the y–coordinate of each point in the grid.

2. Now suppose that we want to create a "surface plot" $z = f(x, y)$ for some function $f(x, y)$ based on the grid points represented by `X` and `Y`.

   - Taking the function $x^2 + y^2$ we just define:

   ```
   Z=X.^2+Y.^2
   ```

   ```
   Z =

        4     5     8    13    20    29
        1     2     5    10    17    26
        0     1     4     9    16    25
        1     2     5    10    17    26
        4     5     8    13    20    29
   ```

   - Notice that we used "array sense" multiplication `.*` as we are squaring the entries of the two arrays `X` and `Y` before adding them.

- The entries of the matrix `Z` are just the z–values at each grid point.

3. Now, finally, how to create a surface plot?

   - Easy, now that we have `X`, `Y` and `Z`.

```
surf(X,Y,Z)
```

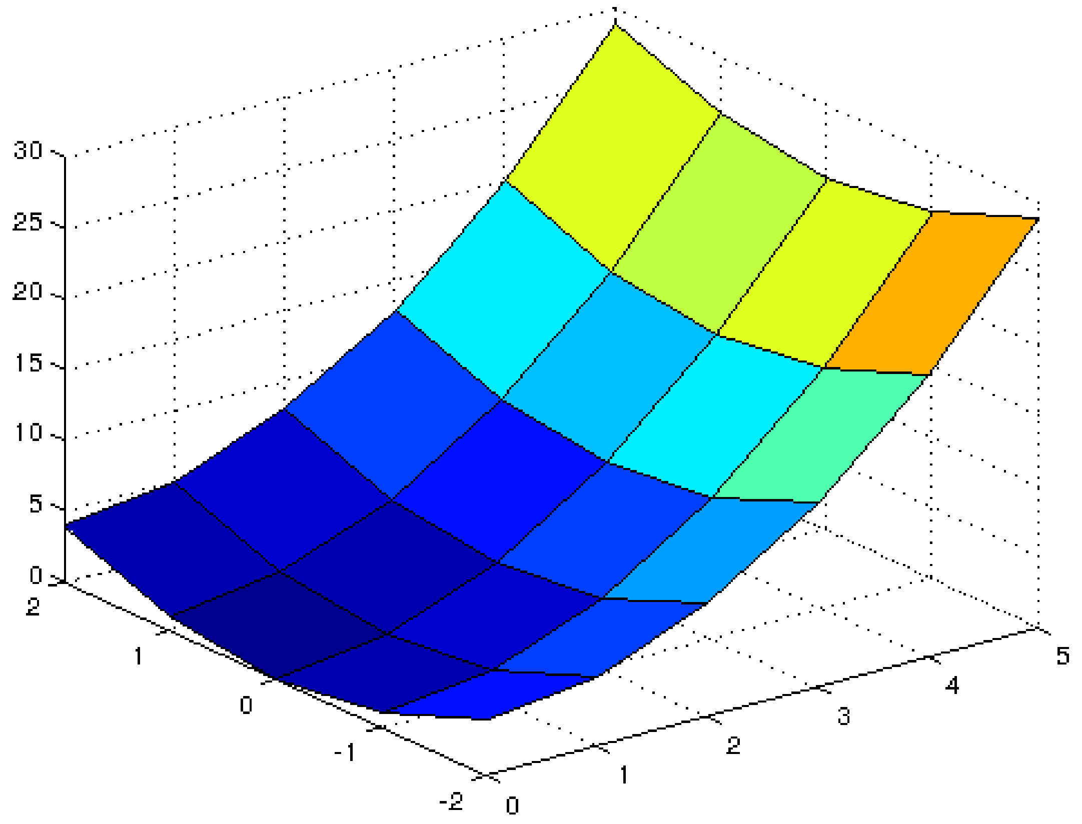


Figure 2: Simple Surface Plot

A few final points on this Example.

- The mesh spacing is very coarse.
- The function to be plotted is not very interesting.
- So try the following: create a surface plot of the function $f(x, y) = \sin(x^2 + y^2)e^{-(x^2+y^2)}$ on the grid $[-2, 2] \times [-2, 2]$ with a horizontal & vertical grid spacing of 0.1.

See Appendix D for two more advanced examples, one of them using `meshgrid`.

More about surface & 3D plots in Section 7.2.1.

4 Matrices

- An $m \times n$ matrix is a two-dimensional array of numbers consisting of m rows and n columns.
- Special cases are a column vector ($n = 1$) and a row vector ($m = 1$).
- Matrices are fundamental to **Matlab** and even if you are not intending to use **Matlab** for linear algebra computations you need to become familiar with matrix generation and manipulation

4.1 Generating Matrices

- Matrices can be generated in several ways. Many elementary matrices can be constructed directly with a **Matlab** function.
- The matrix of zeros, the matrix of ones and the identity matrix (which has ones on the diagonal and zeros elsewhere) are returned by the functions **zeros**, **ones** and **eye**, respectively.
- All have the same syntax.

- For example, **zeros(m,n)** or **zeros([m,n])** produces an $m \times n$ matrix of zeros, while **zeros(n)** produces an $n \times n$ zero matrix. Examples:

```
zeros(2)
ones(2,3)
eye(3,2)
```

- We will often need to set up an identity matrix whose dimensions match those of a given matrix A.
- This can be done with `eye(size(A))`, where **size** returns the number of rows and columns of a matrix.
- Another useful built-in **Matlab** function is `length` where **length A** is the larger of the two dimensions of A so if **x** is an $n \times 1$ or a $1 \times n$ vector, `length(x)` returns **n**.

- Two other very important matrix generation functions are **rand** and **randn**, which generate matrices of (pseudo-)random numbers using the same syntax as **eye**.
- The function **rand** produces a matrix of numbers from the uniform distribution over the interval $[0, 1]$.
- The function **randn** produces a matrix of numbers from the standard normal (0,1) distribution.
- Called without any arguments, both functions produce a single random number.

```
rand
rand(3)
```

- In carrying out experiments with random numbers it is often important to be able to regenerate the same numbers on a subsequent occasion.
- The numbers produced by a call to rand depend on the state of the generator.
- The state can be set using the command `rand('state',j)`.
- For j=0 the **rand** generator is set to its initial state (the state it has when **Matlab** starts).
- For a nonzero integer j, the generator is set to its jth state.
- The state of `randn` is set in the same way. The periods of **rand** and `randn`, that is, the number of terms generated before the sequences start to repeat, exceed $2^{1492} \approx 10^{449\cdot}$.

(Exercise: can you confirm this approximate equality?)

- Matrices can be built explicitly using the square bracket notation.
- For example, a 3-by-3 matrix comprising the first 9 primes can be set up with the command

```
A = [2 3 5
7 11 13
17 19 23]
```

- The end of a row can be specified by a semicolon instead of a carriage return, so a more compact command with the same effect is “**S**emi-colons **s**eparate **R**ows”(not easy to remember).

```
A = [2 3 5; 7 11 13; 17 19 23]
```

- Within a row, elements can be separated by spaces or by commas “**C**ommas **s**eparate **C**olumns” (very easy to remember).

- If you decide not to use commas to separate your columns, if numbers are specified with a plus or minus sign be careful not to leave a space after the sign, else **Matlab** will interpret the sign as an addition or subtraction operator.
- To illustrate with vectors:

```
v = [-1 2 -3 4]
w = [-1, 2, -3, 4]
x = [-1 2 - 3 4] % check this one carefully!
```

4.1.1 Block Matrix Techniques

- Matrices can be constructed in block form.
- With B defined by

  B = [1 2; 3 4], we may create

```
C = [B zeros(2)
ones(2) eye(2)]
```

- Block diagonal matrices can be defined using the function **blkdiag**, which is easier than using the square bracket notation.
- Example:

```
A = blkdiag(2*eye(2),ones(2))
```

- One of the most useful functions in **Matlab** is **repmat**.
- We can use it to construct "tiled" block matrices — **repmat(A,m,n)** creates a block $m \times n$ matrix in which each block is a copy of A. If m is omitted, it defaults to n. Example:

```
A = repmat(eye(2),2)
```

  Matlab provides a number of special matrices such as the famous Hilbert matrix

```
hilb(4)
```

- The function **gallery** provides access to a large collection of test matrices

```
help gallery
```

- Another way to generate a matrix is to load it from a file using the **load** command as we saw on Slide 75.

4.2 Subscripts and Colon Notation

- To enable access and assignment to submatrices **Matlab** has a clever notation based on the colon character.
- The colon is used to define vectors that can act as subscripts.
- For integers i and j, **i:j** denotes the row vector of integers from i to j (in steps of 1). A nonunit step s is specified as **i:s:j**.
- This notation is valid even for non-integer i, j and s.
- Examples:

```
1:5
4:−1:−2
0:.75:3
```

- Single elements of a matrix are accessed as **A(i,j)**, where $i \geq 1$ and $j \geq 1$ (**no zero or negative subscripts in Matlab**).

Subscripts and Colon Notation Continued

- The submatrix comprising the intersection of rows p to q and columns r to s is denoted by **A(p:q,r:s)**.
- **As an important special case, a lone colon as the row or column specifier covers all entries in that row or column; so A(:,j) is the jth column of A and `A(i,:)` the ith row.**
- The keyword **end** used in this context denotes the last index in the specified dimension; so **A(end,:)** picks out the last row of A.
- Finally, an arbitrary submatrix can be selected by specifying the individual row and column indices. For example, **A([i j k],[p q])** produces the submatrix given by the intersection of rows i, j and k and columns p and q.

Subscripts and Colon Notation Continued

- Here are some examples, using the matrix of primes set up above:

```
A= [2 3 5; 7 11 13; 17 19 23]
A(2,1)
A(2:3,2:3)
A(:,1)
A(2,:)
A(end:-1:1,end)
A([1 3],[2 3])
```

- For any matrix of vector, A(:) is a vector comprising all the elements of A column by column:

```
B = A(:)
```

This allows a handy way to generate a column vector from **any** vector without having to check whether it is a row or column vector in advance:

```
r=1:5
c=r(:) % gives a column vector
v=[1;2;3]
v(:) % no change, still a column!
```

This can be useful when writing an M-file (a **Matlab** “program”) that needs its input to be a column vector....

A Clever Trick

- When placed on the left side of an assignment statement `A(:)` **fills** `A` column by column, **preserving its shape**.
- Using this notation, a much neater way to define our 3-by-3 matrix of primes is

```
A = zeros(3); A(:) = primes(23); A=A'
```

- The function **primes** returns a vector of the prime numbers less than or equal to its argument.
- The **matrix transpose** **A = A'** (see the next Section) is necessary to reorder the primes across the rows rather than down the columns.

Another Neat Trick

- There is one situation where the number of elements on each side of a "subscripted assignment" need not be equal — when the rhs is a single element.
- In this case the scalar on the rhs is "expanded" to match the number of elements on the lhs:

```
A=ones(3)
A(2:3,2:3)=0
```

A=

1 1 1

1 0 0

1 0 0

The linspace Function

- Related to the colon notation for generating vectors of equally spaced numbers is the function **linspace**, which accepts the number of points rather than the increment: **linspace(a,b,n)** generates n equally spaced points between a and b.
- If n is omitted it defaults to 100.
- Example:

```
linspace(−1,1,9)
```

The Empty Matrix

- The notation [] denotes an empty, 0×0 matrix.
- Setting a row or column to [] is one way to delete that row or column from a matrix.

```
A(2,:) = []
```

- In this example the same effect is achieved by

```
A = A([1 3],:)
```

- The empty matrix is also useful as a placeholder in lists of function arguments, as we will see later in this Chapter.

4.3 Matrix and Array Operations

For scalars (real and complex numbers) a and b, the operators +, -, *, / and ^ produce the obvious results. As well as the usual right division operator, /, **Matlab** has a left division operator, \

Matlab notation	Mathematical equivalent
Right division: **a/b**	$\frac{a}{b}$
Right division: **a\b**	$\frac{b}{a}$

Table 3: Left and Right Division

Try it!

```
2/3, 3\2 % are they the same? (Yes)
```

- It is hard to imagine why you would want to use \ to calculate simple ratios of scalars.
- On the other hand, the two versions of “division” come into their own when working with matrices as we will see shortly.
- For matrices, all the above operations can be carried out in a matrix sense (according to the rules of matrix algebra) or an array sense (elementwise) as summarised in the Table.

Operation	Matrix sense	Array Sense
Addition	+	+
Subtraction	-	-
Multiplication	*	.*
Left Division	\	.\
Right Division	/	./
Exponentiation	^	.^

Table 4: Elementary matrix and array operations

- Addition and subtraction, which are identical operations in the matrix and array senses, are defined for matrices of the same dimension.
- The product A*B is the result of matrix multiplication, defined only when the number of columns of A and the number of rows of B are the same.
- The backslash and the forward slash define solutions of linear systems
  - `X= A\B` is equivalent to saying X is a solution to `A*X = B` ($\mathtt{X} = \mathtt{A}^{-1}\mathtt{B}$ if A has an inverse).
  - `X= A/B` is equivalent to saying X is a solution to `X*B = A` ($\mathtt{X} = \mathtt{AB}^{-1}$ if B has an inverse).

Some examples:

```
A = [1 2; 3 4]
B = ones(2)
A+B
A*B
A \ B
A / B
det B
```

(The output from the last command should explain the result of the previous one!)

- Multiplication and division in the array, or elementwise, sense are specified by preceding the operator with a full stop (period) ".".
- If A and B are matrices of the same dimensions then
  - **C = A.*B** sets **C(i,j) = A(i,j)*B(i,j)**.
  - **C = A./B** sets **C(i,j) = A(i,j)/B(i,j)** for each value of **i** and **j**.
- The assignment **C = A.\B** is equivalent to **C = B./A**.

With the same A and B as in the previous example:

```
A.*B
A.\B
B./A % Is this the same as the previous line? (Yes)
```

- Exponentiation with ^ means taking powers of a matrix.
- The dot form .^ exponentiates element-by-element.
- If A is a square matrix then A^2 is the matrix product A*A, but A.^2 is A with each element squared:

```
A^2, A.^2
2.^ x
```

- The dot form of exponentiation allows the power to be an array when the dimensions of the base and the power agree, or when the base is a scalar:

```
x = [1 2 3]; y = [2 3 4]; Z = [1 2; 3 4];
x.^ y
2.^ Z
```

- Matrix exponentiation is defined for all powers, not just for positive integers.
- If $n < 0$ is an integer **A ^ n** is defined as **inv**(**A**) ^ **n**.
- For noninteger p, **A ^ p** is evaluated using the eigensystem of A; results can be incorrect or inaccurate when A is not diagonalizable or when A has an ill-conditioned eigensystem.
- The conjugate transpose of the matrix A is obtained with **A'**.
- If A is real, this is simply the transpose.
- The transpose without conjugation is obtained with **A.'** .
- The functional alternatives **ctranspose(A)** and **transpose(A)** are sometimes more convenient.
- For the special case of column vectors x and y, **x'*y** is the inner or dot product, which can also be obtained using the **dot** function as **dot(x,y)**.

- The vector or cross product of two 3-by-1 or 1-by-3 vectors (as used in mechanics) is produced by **cross**.
- Examples:

```
x = [−1; 0; 1]; y = [3; 4; 5];
x = [−1 0 1]'; y = [3 4 5]'; % same reult but less typing
x'*y
dot(x,y)
n=cross(x,y)
dot(n,x) , dot(n,y) % n is orthogonal to x and y
```

The **kron** function evaluates the Kronecker (or outer) product of two matrices. The Kronecker product of an $m \times n$ matrix A and $p \times q$ matrix B has dimensions $mp \times nq$ and can be expressed as a block $m \times n$ matrix with (i, j) block $a_{ij}B$. Example:

```
A = [1 10; −10 100];
B = [1 2 3; 4 5 6; 7 8 9];
kron(A,B)
```

4.3.1 Scalar Expansion

- If a scalar is added to a matrix **Matlab** will **expand the scalar** into a matrix with all elements equal to that scalar.
- For example:

```
B= [4 3; 2 1] + 4
A= [1 −1] −6
```

- However, if an assignment makes sense without expansion then **Matlab** will interpret it in that (non-expansion) sense.
- Thus if the previous command is followed by $\mathbf{A = 1}$ then A becomes the scalar 1, not **ones(1,2)**.

- The “clever trick” mentioned earlier allows us to enter

```
A =[1 −1] −6
A(:)=1
```

  if we really want **A** to be **ones(1,2)**.

- Much easier to just enter **A=ones(1,2)**.

- If a matrix is multiplied or divided by a scalar, the operation is performed elementwise. For example:

```
[3 4 5;4 5 6]/12
```

- Most of the elementary and special functions included in **Matlab** in can be given a matrix argument, in which case the functions are computed elementwise.
- Functions of a matrix in the linear algebra sense are signified by names ending in m **expm, funm, logm, sqrtm**.
- For example, for **A** = [**2 2; 0 2**],

```
sqrt(A)
ans*ans %=A? Nope...
sqrtm(A)
ans*ans %=A? Yes, almost...
```

4.4 Matrix Manipulation

- Several commands are available for manipulating matrices (commands more specifically associated with linear algebra are discussed in Chapter 8).
- The **reshape** function changes the dimensions of a matrix: **reshape(A,m,n)** produces an $m \times n$ matrix whose elements are taken **columnwise** from A.
- For example:

```
A = [1 4 9;16 25 36]
B = reshape(A,3,2)
```

- The function **diag** deals with the diagonals of a matrix and can take a vector or a matrix as argument.
- For a vector x, **diag(x)** is the diagonal **square** matrix with main diagonal x:

```
diag([1 2 3])
```

- More generally, **diag(x,k)** puts x on the kth diagonal, where $k > 0$ specifies diagonals above the main diagonal and $k < 0$ diagonals below the main diagonal ($k = 0$ gives the main diagonal):

```
diag([1 2], 1)
diag([3 4], -2)
```

- On the other hand, for a matrix A, **diag(A)** is the column vector comprising the main diagonal of A.
- So to produce a diagonal matrix with diagonal the same as that of A you must enter **diag(diag(A))**!
- Analogously to the vector case, **diag(A,k)** produces a column vector made up from the kth diagonal of A.

- Thus if

  A =[2 3 5; 7 11 13; 17 19 23]

  then

```
diag(A)
```

outputs **ans** =

$$\begin{matrix}2\\11\\23\end{matrix}$$

and

```
diag(A,-1)
```

outputs **ans** =

$$\begin{matrix}7\\19\end{matrix}$$

- Triangular parts of a matrix can be extracted using **tril** and **triu.**
- The lower triangular part of A (the elements on and below the main diagonal) is specified by **tril(A)** and the upper triangular part of A (the elements on and above the main diagonal) is specified by **triu(A)**.
- More generally, **tril(A,k)** gives the elements on and below the kth diagonal of A, while **triu(A,k)** gives the elements on and above the kth diagonal of A. With A as above:

```
tril(A)
triu(A,1)
triu(A,-1)
```

4.5 Data Analysis

- You will learn how to use **R** for statistical Computing in Part 2 — so just a brief mention of **Matlab** 's data processing facilities.
- Some of **Matlab** 's data processing functions are given in the Table:

max	Largest Component	**sum**	Sum of elements
min	Smallest Component	**prod**	Product of elements
mean	Average/mean value	**cumsum**	Cumulative sum
median	Median value	**cumprod**	Cumulative product
std	Standard Deviation	**diff**	Difference of els.
var	Variance		
sort	Sort, ascending order		

Table 5: Basic Data Processing Functions

- The simplest usage is to apply these functions to vectors. For example:

```
x = [4 -8 -2 1 0], [min(x) max(x)], sort(x), sum(x)
```

- The **sort** function sorts into ascending order.
- For a real vector x, descending order is obtained with **-sort(-x)**.
- From **Matlab** 7 on, you can sort in descending order with **sort(x,'descend')**.
- For complex vectors, **sort** sorts by absolute value:

```
x = [1+i, -3-4i, 2i, 1];
y = sort(x)
```

- For matrices the functions are defined columnwise.

- Thus **max** and **min** return a vector containing the maximum and minimum element, respectively, in each column, **sum** returns a vector containing the column sums and **sort** sorts the elements in each column of the matrix into ascending order.
- The functions **min** and **max** can return a second argument that specifies in which components the minimum and maximum elements are located.
- For example, if

```
A =[0 −1 2 ;1 2 −4; 5 −3 −4]
```

then

```
max(A)
```

ans =
5 2 2

```
[m,i] = min(A)
```

m =

0 -3 -4

i =

1 3 2

- As this example shows, if there are two or more minimal elements in a column then the index of the first is returned. The smallest element in the matrix can be found by applying min twice in succession:

```
min(min(A))
```

- The functions **max** and **min** can be made to act row-wise via a third argument:

```
max(A,[],2)
```

- The 2 in **max(A,[],2)** specifies the maximum over the second dimension, that is, over the column index.

- The empty second argument, [], is needed because with just two arguments **max** and **min** return the elementwise maxima and minima of the two arguments:

```
max(A,0) , min(A,0)
```

- The functions **sort** and **sum** can also be made to act row-wise, via a second argument. (Try **help sort**.)
- The **diff** function forms differences.
- Applied to a vector x of length n it produces the vector [**x(2)-x(1), x(3)-x(2), . . . , x(n)-x(n-1)**] of length **n-1**.
- Example:

```
x = (1:8).^ 2
y = diff(x)
z = diff(y)
```

- This example suggests a general property of lists generated by a polynomial (quadratic here), namely that each "differencing operation" reduces the degree by one.
- Try

```
x = (−10:10).^ 4
y = diff(x)
z = diff(y)
```

- What should the degree of the polynomial that generates **z** be?
- Try plotting **z** against the vector **[-10:8]**.

5 Loops, If, etc.

- In this Chapter we will examine **Matlab** looping (e.g. **for** and **while**) and branching commands (e.g. **if/then/else**).
- These are similar in operation though slightly different in syntax to those encountered in Java and Maple.
- We will also briefly consider the issue of making **Matlab** scripts more efficient by avoiding the use of these very operators!
- First we need to understand how **Matlab** handles logical constants, variables and expressions.

5.1 Relational and Logical Operators

Matlab has a **logical** data type with values **true** and **false**.

```
a=true
b=false
c=1
whos
```

```
Name   Size   Bytes   Class
a      1x1    1       logical
b      1x1    1       logical
c      1x1    8       double
```

- Notice that although we set `a` and `b` to `true` and `false` above their values are displayed as 1 and 0 respectively.
- In fact we almost always use 1 and 0 instead of `true` and `false` — **Matlab** interprets 1 as `true` and 0 as `false`.
- The fact that 0 and 1 are double precision reals (data type `double`) unless they result from a logical expression does not cause difficulties.
- As the output from **whos** shows, logicals occupy one byte rather than the eight bytes needed for a **double** precision real variable.

- **Matlab** 's relational (comparison) operators are the largely familiar

==	equal
~=	equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

Table 6: **Matlab** relational operators

- Note that a single = denotes assignment and **never** a test for equality in **Matlab** .

- Comparisons between scalars produce (logical) 1 if the relation is true and (logical) 0 if it is false.
- Comparisons are also defined between matrices of the same dimension and between a matrix and a scalar, the result being a matrix of 0s and 1s in both cases.
- For matrix-matrix comparisons corresponding pairs of elements are compared, while for matrix-scalar comparisons the scalar is compared with each matrix element.

For example:

```
A = [1 2;3 4];
B = 2*ones(2);
A == B
A > 2
```

- To test whether matrices A and B are identical, the expression **isequal(A,B)** can be used:

```
isequal(A,B)
```

- The function **isequal** is one of many useful logical functions whose names begin with **is**, a selection of which is listed in Table 7; for a full list type **doc is**.
- For example, the function **isinf(A)** returns a **logical** array of the same size as A containing **true** (displayed as `1`) where the elements of A are plus or minus **Inf** and **false** (displayed as `0`) otherwise.
- The function **isnan** is necessary because the test `x == NaN` always produces the result 0 (false), even if x is a NaN!
- A NaN is defined to compare as unequal and unordered with everything — including itself.

ischar	True for char array (string)
isempty	True for empty array
isequal	True if arrays are identical
isfinite	True for finite array elements
isinf	True for infinite array elements
islogical	True for logical array
isnan	True for NaN (Not a Number)
isnumeric	True for numeric array
isreal	True for real array

Table 7: Some logical **is*** functions

5.1.1 Testing Variables for complex in Matlab

- An array can be real in the mathematical sense (zero imaginary part) but not real as reported by **isreal**!
- How a (mathematically) real **Matlab** variable is formed can determine whether **Matlab** treats it as real or complex.
- The example should make the point clear:

```
a=1;
b=complex(1,0);
c=1+0i;
[a,b,c]
whos a b c
[isreal(a) , isreal(b) , isreal(c)]
```

See **help complex**.

5.1.2 Logical Operators in Matlab

- **Matlab** 's logical operators are listed in Table 8.
- Like the relational operators, the **and**, | and ˜ operators produce matrices of logical 0's and 1's when one of the components of the expression is a matrix.
- When applied to a vector, the **all** function returns 1 if all the elements of the vector are nonzero and 0 otherwise.
- The **any** function is defined in the same way, with "any" replacing "all".

- Examples:

```
x = [-1 1 1];
y = [1 2 -3];
x>0 & y>0
x>0 | y>0
xor(x>0,y>0)
any(x>0)
all(x>0)
```

&	logical and
&&	logical and (for scalars) with "short-circuiting"
|	logical or
||	logical or (for scalars) with "short-circuiting"
~	logical not
xor	logical exclusive or
all	true if all elements of vector are nonzero
any	true if any element of vector is nonzero

Table 8: **Matlab** logical operators

- Note that **xor** must be called as a function: **xor(a,b).**
- The **and**, **or** and **not** and relational operators can also be called in functional form as **and(a,b)** , **eq(a,b)**, etc.

5.1.3 "Short-Circuiting"

The doubled operators **&&** and || are worth mentioning.

- They **only work with scalar expressions** and should always be used instead of the conventional & and | (with scalars) for reasons of efficiency.
- Continuing the previous example; compare the outputs of the two commands:

```
any(x>0) && any(y>0)
```

ans =
1

```
(x>0) && (y>0)
```

??? Operands to the || and && operators must be convertible to logical scalar values.

- The second feature of these "doubled" operators is that — whenever possible — they "short-circuit" the evaluation of the logical expressions in which they are used.
- In the compound expression **exp1 && exp2**, if **exp1** is **false** then **Matlab** will not evaluate **exp2**.
- Similarly , in **exp1** || **exp2**, if **exp1** is **true** then **Matlab** will not evaluate **exp2**.
- Short-circuiting saves on computation and also can allow warnings and errors to be avoided.
- For example the command (we will examine the **if** command shortly) beginning

```
if x >0 && sin(1/x) <0.5
```

avoids a divide-by-zero error! — can you see how?

Rules of Precedence

- The precedence of arithmetic, relational and logical operators is as is familiar from other programming languages — see **help precedence**.
- For operators of equal precedence **Matlab** evaluates from left to right.
- Precedence can be overridden by using parentheses (brackets).
- In fact, you **should** use parentheses to make your intention clear to the reader of your code.

Testing Matrices for Equality

- For matrices, **all** returns a row vector containing the result of **all** applied to each column.
- Therefore **all(all(A==B))** is another way of testing equality of the matrices A and B.
- The **any** function works in the corresponding way.
- Thus, for example, **any(any(A==B))** has the value 1 if A and B have any equal elements and 0 otherwise.

5.1.4 Finding Elements in Matrices

- The **find** command returns the indices corresponding to the nonzero elements of a vector. For example

```
x = [-3 1 0 -inf 0];
f = find(x)
```

f =

1 2 4

- The result of find can then be used to extract just those elements of the vector:

```
x(f)
```

- With x as above, we can use find to obtain the finite elements of x,

```
x(find(isfinite(x)))
```

ans =

-3 1 0 0

and to replace negative components of x by zero:

```
x(find(x < 0)) = 0
```

x =

0 1 0 0 0

- When **find** is applied to a matrix A, the index vector corresponds to A regarded as a vector of the columns stacked one on top of the other (that is, **A(:)**) and this vector can be used to index into A.

- In the following example we use **find** to set to zero those elements of A that are less than the corresponding elements of B:

```
A =[4 2 16;12 4 3], B = [12 3 1;10 -1 7]
f = find(A<B)
A(f) = 0
```

- An alternative usage of find for matrices is **[i,j] = find(A)**, which returns vectors i and j containing the row and column indices of the nonzero elements.
- The results of **Matlab** 's logical operators and logical functions are arrays of 0's and 1's that are examples of logical arrays.
- Logical arrays can also be created by applying the function **logical** to a numeric array.
- Logical arrays can be used for subscripting.

- Consider the following examples:

```
clear
y = [1 2 0 −3 0];
i1= (y~=0);
i2= [1 1 0 1 0] ;
y(i1);
```

```
ans =
1 2 -3
```

```
y(i2)
```

??? Subscript indices must either be real positive integers or logicals.

```
whos i1 i2
```

```
isequal(i1,i2) % perhaps surprisingly!!
```

```
i3 =[1 2 4];
y(i3)
```

- Even though the **numeric** array i2 has the same elements as the **logical** array i1 (and compares as equal with it), only the logical array `i1` can be used for subscripting as `i1` and `i2` contain zero elements.
- A call to **find** can sometimes be avoided when its argument is a logical array.
- In our earlier example, **x(find(isfinite(x)))** can be replaced by **x(isfinite(x)).**
- Using **find** can improve clarity.

5.2 Branching Commands

To allow **Matlab** commands, scripts and function m-files to execute differently depending on circumstances (make choices), the language has four flow control structures:

- the **if** statement,
- the **for** loop,
- the **while** loop and
- the **switch** statement.

5.2.1 If/Then/Else

- The simplest form of the **if** statement is

```
if expression
statements
end
```

where the *statements* are executed if the (real parts of) the elements of *expression* are all nonzero.

- For example, the following swaps x and y if x is greater than y:

```
if x>y
temp = y;
y = x;
x = temp;
end
```

- When an **if** statement is followed on its line by further statements, a comma is useful for readability (but not required) to separate the **if** *expression* from the next statement:

```
if x > 0, x = sqrt(x); end
```

- Statements to be executed only if *expression* is false can be placed after else, as in the example

```
e = exp(1);
if 2^ e > e^ 2
disp('2^ e is bigger')
else
disp('e^ 2 is bigger')
end
```

- Finally, one or more further tests can be added with **elseif** (note that there must be no space between **else** and **if**):

```
if isnan(x) disp('Not a Number')
elseif isinf(x) disp('Plus or minus infinity')
else
disp('A ''regular'' floating point number')
end
```

- In the third **disp,** '' prints as a single quote '.

5.2.2 For Loops

- The **for** loop is one of the most useful **Matlab** constructs although, as we will see later, experienced programmers who are concerned with producing compact and fast code try to avoid **for** loops wherever possible.
- The syntax is

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

- Usually, *expression* is a vector of the form **i:s:j**.
- The *statements* are executed with *variable* equal to each element of expression in turn.

- For example, the sum of the first 25 terms of the harmonic series $\sum_{i=1}^{25} 1/i$ is computed by

```
s = 0;
for k = 1:25
s = s + 1/k;
end
s
```

A nice result from calculus is that $\lim_{N\to\infty}\left(\sum_{k=1}^{N}\frac{1}{k}\right) - \log(N) = \gamma$, where γ is called Euler's constant.

In other words, even though both $\sum_{k=1}^{N} 1/k$ and $\log N$ go to infinity, their difference has a limit, γ.

Can you show that $\gamma \approx 0.5772156649$?

- Another way to define *expression* is using the square bracket notation:

```
for x = [pi/6 pi/4 pi/3]
disp([x, sin(x)]),
end
```

- Multiple *for* loops can of course be nested, in which case indentation helps to improve the readability.

- The following code forms the 5-by-5 symmetric matrix A with (i, j) element i/j for $j \geq i$: you can load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/nestfor.m`.

Listing 2: nestfor.m

```
% NESTED FOR LOOP
n = 5; A = eye(n);
for j=2:n
   for i = 1:j-1
      A(i,j) = i/j;
      A(j,i) = i/j;
   end
end
```

```
A =
         1.0000    0.5000    0.3333    0.2500    0.2000
         0.5000    1.0000    0.6667    0.5000    0.4000
         0.3333    0.6667    1.0000    0.7500    0.6000
         0.2500    0.5000    0.7500    1.0000    0.8000
         0.2000    0.4000    0.6000    0.8000    1.0000
```

- The expression in the **for** loop can be a matrix, in which case *variable* is assigned the columns of *expression* from first to last.
- For example, to set x to each of the unit vectors in turn, we can write

```
for x=eye(n)
 %statements that do stuff with x
end
```

5.2.3 While Loops, Break and Continue

- The **while** loop has the form

```
while expression
statements
end
```

- The *statements* are executed as long as *expression* is true. The following example approximates the smallest nonzero floating point number (**not** `eps`):

```
x = 1;
while x>0,
xmin = x;x = x/2;
end
xmin
```

- A **while** loop can be terminated with the **break** statement, which passes control to the first statement after the corresponding end.
- An infinite loop can be constructed using **while 1, ..., end**, which is useful when it is not convenient to put the exit test at the top of the loop.
- Note that, unlike some other languages, **Matlab** does not have a "**repeat-until**" loop.

- We can rewrite the previous example less concisely as:

```
x = 1;
while 1 xmin = x; x = x/2;
 if x == 0
    break
 end
end
xmin
```

- The **break** statement can also be used to exit a **for** loop. In a nested loop a **break** exits to the loop at the next higher level.

- The **continue** statement causes execution of a **for** or **while** loop to pass immediately to the next iteration of the loop, skipping the remaining statements in the loop.
- As a trivial example,

```
for i=1:10
   if i < 5, continue, end
   disp(i)
end
```

displays the integers 5 to 10.

- In more complicated loops the continue statement can be useful to avoid long-bodied if statements.

5.2.4 The Switch Statement

- The final control structure is the **switch** statement.
- It consists of "*switch* **expression**" followed by a list of "**case** *expression statements*", optionally ending with "**otherwise** *statements*" and followed by **end.**
- The **switch** *expression* is evaluated and the statements following the first matching **case** *expression* are executed.
- If none of the cases produces a match then the statements following **otherwise** are executed.
- The next example evaluates the p-norm of a column vector x (i.e., **norm(x,p)**) for just three values of p: You can load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/switchex.m`.

Listing 3: switchex.m

```
% EVALUATE 1, 2 & INFINITY NORM
x=x(:); % Make sure x is a column vector
switch p
   case 1, y = sum(abs(x));
   case 2, y = sqrt(x'*x);
   case inf, y = max(abs(x));
   otherwise error('p must be 1, 2 or inf.')
end
```

```
x=[1;2;3];
p=1; switchex;y
p=2;switchex;y
p=inf;switchex;y
```

- See **help error** for an explanation of the **error** function.

- The expression following **case** can be a list of values enclosed in parentheses “curly brackets”.
- In this case the **switch** *expression* can match any value in the case list.
- The following example, Listing 4 illustrates the idea. You can load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/caselist.m`.

Listing 4: caselist.m

```
  % Check number type
x = input('Enter a real number: ');
switch x
  case {-inf,inf}
    disp('Plus or minus infinity')
  case 0
    disp('Zero')
  otherwise
    disp('Nonzero and finite')
end
```

6 M-Files

6.1 Scripts and Functions

Although you can do many useful computations working entirely at the **Matlab** command line, sooner or later you will need to write M-files. These are the equivalents of programs, functions, subroutines and procedures in other programming languages.

Collecting together a sequence of commands into an M-file opens up many possibilities, including

- experimenting with an algorithm by editing a file, rather than retyping a long list of commands,
- making a permanent record of a numerical experiment,
- building up utilities that can be used again,

An M-file is a text file that has a **.m** filename extension and contains **Matlab** commands. There are two types:

Script M-files (or command files) have no input or output arguments and operate on variables in the workspace.

Function M-files contain a **function** definition line, can accept input arguments and return output arguments and their internal variables are local to the function (unless declared **global**).

We have seen many examples of script M-files already.

A script enables you to store a sequence of commands that are to be used repeatedly or will be needed at some future time. A simple example of a script M-file, marks.m, was given in Listing 1.

Here is a script M-file that calculates the number of days in a given month. Load the file from
`http://jkcray.maths.ul.ie/ms4024/M-Files/days.m`.

Listing 5: days.m

```
% DAYS
% Prompts user to input the month number
% Outputs the number of days in the selected month.
% Performs simple checks on user input
clear
ok=false;
while ~ok
    month = input('Give month number (1-12): ' );
    if isnumeric(month)&& (round(month)==month)&&(1 <=month) &&(month <=12)
        ok=true;
    end
end
if month==1 | month==3 | month ==5 | month==7 ...
        | month==10 | month==12
    'Your month has 31 days'
else
    if month==2
        'Your month has 28 days'
    else
        'Your month has 30 days'
    end
end
```

Some comments on `days.m`:

- Note the simple data validation in Line 9.
- Note the use of **&&** ("short-circuiting") in Line 9 to ensure that the test condition computes as `false` once **any** of the component conditions is found to be false and no further checking is done or needs to be done.
- Also note the order in which the tests are done (from left to right).

6.2 Function M-Files

Function M-files enable you to extend the **Matlab** language by writing your own functions that accept and return arguments.

They can be used in exactly the same way as existing **Matlab** functions such as **sin, eye, size,** etc.

Listing 6 shows a simple function that evaluates the largest element in absolute value of a matrix.

Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/maxentry.m`.

Listing 6: maxentry.m

```
function y = maxentry(A)
% MAXENTRY Largest absolute value of matrix entries.
% MAXENTRY(A) is the maximum of the absolute values
% of the entries of A.

y = max(max(abs(A)));
```

This example illustrates a number of features.

- The first line begins with the keyword **function** followed by the output argument, y, and the = symbol.
- On the right of = comes the function name, **maxentry,** followed by the input argument, A, within parentheses. (In general there can be any number of input and output arguments.)
- The function name must be the same as the name of the .m file in which the function is stored — in this case the file must be named **maxentry.m**.
- The second line of a function file is called the H1 (help 1) line.
- It should be a comment line of a special form: a line beginning with a % character, followed without any space by the function name in capital letters, followed by one or more spaces and then a brief description.

- The description should begin with a capital letter, end with a period.
- All the comment lines from the first comment line up to the first noncomment line (usually a blank line, for readability of the source code) are displayed when **help function_name** is typed.
- Therefore these lines should describe the function and its arguments.
- It is conventional to capitalize function and argument names in these comment lines.

For the **maxentry.m** example, we have

```
help maxentry
```

```
 MAXENTRY   Largest absolute value of matrix entries.
            MAXENTRY(A) is the maximum of the absolute values
            of the entries of A.
```

- You should document all your function files in this way, however short they may be.
- It is often useful to record in comment lines the date when the function was first written and to note any subsequent changes that have been made. The help command works in a similar manner on script files, displaying the initial sequence of comment lines.

The function **maxentry** is called just like any other **Matlab** function:

```
maxentry(1:10)
```

```
maxentry(magic(4))
```

- The function **flogist** shown in Listing 7 illustrates the use of multiple input and output arguments.
- This function evaluates the scalar logistic function $x(1 - ax)$ and its derivative with respect to x.
- The two output arguments **f** and **fprime** are enclosed in square brackets.
- When calling a function with multiple input or output arguments it is not necessary to request all the output arguments — the omitted arguments start at the end of the list.
- If more than one output argument is requested the arguments must be listed within square brackets.

Listing 7: flogist.m

```
function [f,fprime] = flogist(x,a)
%FLOGIST Logistic function and its derivative.
% [F,FPRIME] = FLOGIST(X,A) evaluates the logistic
% function F(X) = X.*(1-A*X) and its derivative FPRIME
% at the matrix argument X, where A is a scalar parameter.

f = x.*(1-a*x);
if nargout>1
    fprime = 1-2*a*x;
end
```

Load the file from
`http://jkcray.maths.ul.ie/ms4024/M-Files/flogist.m`.

- Examples of usage are

```
f = flogist(2,.1), [f,fprime] = flogist(2,.1)
```

- The function **nargout** returns the number of output arguments requested so avoiding calculating the derivative unless the user asks for it.
- In function **logist,** array multiplication (.*) is used in the statement f = x.*(1-a*x).
- So, if a vector or matrix is supplied for x, the function is evaluated at each element simultaneously:

```
flogist(1:4,2)
```

Another function using array multiplication is **cheby** in Listing 8, which we will plot with **Matlab** 's **fplot** in the next Chapter.

- The kth Chebyshev polynomial, $T_k(x)$, can be defined by the recurrence relation

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x),$$

$$\text{for } \mathbf{k \geq 2} \text{ with } T_0(x) = 1 \text{ and } T_1(x) = x.$$

- The function **cheby** accepts a vector x and an integer p and returns a matrix T whose ith row gives the values of $T_0(x), T_1(x), \ldots T_{p-1}(x)$ at x = x(i).
- This is the form of output argument that the **Matlab** plotting function **fplot** requires — as we will see in the next Chapter.

Listing 8: cheby.m

```
function T = cheby(x,p)
%CHEBT Chebyshev polynomials.
% T = CHEBT(X,P) evaluates the first P Chebyshev polynomials
% at the vector X. The K'th column of T contains the
% Chebyshev polynomial of degree K-1 evaluated at X.

T = ones(length(x),p);
x = x(:); % Ensure x is a column vector.
if p == 1, return, end
T(:,2) = x;
for k = 3:p
    T(:,k) = 2*x.*T(:,k-1) - T(:,k-2);
end
```

Load the file from
`http://jkcray.maths.ul.ie/ms4024/M-Files/cheby.m`.

Note that **cheby** uses the **return** command, which causes an immediate return from the M-file. It is not necessary to put a **return** statement at the end of a function or script, unlike in some other programming languages.

A more complicated function example is **sqrtn**, shown in Listing 9. Given $a > 0$, it implements the Newton iteration for $\sqrt{a}$;

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right), \quad x_1 = a,$$

printing the progress of the iteration. Output is controlled by the **fprintf** command — see **help fprintf**.

Listing 9: sqrtn.m

```
function [x,iter] = sqrtn(a,tol)
%SQRTN Square root of a scalar by Newton's method.
% X = SQRTN(A,TOL) computes the square root of the scalar
% A by Newton's method
% A is assumed to be >= 0.
% TOL is a convergence tolerance (default EPS).
% [X,ITER] = SQRTN(A,TOL) returns also the number of
% iterations ITER for convergence.

if nargin < 2, tol = eps; end
x = a;
iter = 0;
xdiff = inf;
fprintf(' k x_k rel. change\n')
while xdiff > tol
    iter = iter + 1;
    xold = x;
    x = (x + a/x)/2;
    xdiff = abs(x-xold)/abs(x);
    fprintf('%2.0f: %20.16e %9.2e\n', iter, x, xdiff)
    if iter > 50
        error('Not converged after 50 iterations.')
    end
end
```

Load the file from

http://jkcray.maths.ul.ie/ms4024/M-Files/sqrtn.m.

- Examples of usage are:

```
[x,iter] = sqrtn(2)
```

```
 x = sqrtn(2,1e-4);
```

- This M-file illustrates the use of optional input arguments.
- The function **nargin** returns the number of input arguments supplied when the function was called and enables default values to be assigned to arguments that have not been specified.
- In this case, if the call to **sqrtn** does not specify a value for **tol,** then **tol** is set to **eps**.

- In this example there is no need to check **nargout,** because **iter** is computed by the function whether or not it is requested as an output argument.
- Some functions like `flogist.m` (Listing 7 above) gain efficiency by inspecting **nargout** and computing only those output arguments that are requested.

Listing 10: marks2.m

```
function [x_sort,x_mean,x_med,x_std] = marks2(x)
% MARKS2 Statistical analysis of marks vector.
% Given a vector of
%  marks X, [X_SORT,X_MEAN,X_MED,X_STD] = MARKS2(X)
% computes a sorted marks list and the mean, median,
%  and standard deviation of the marks.

x_sort = sort(x);
if nargout >1, x_mean = mean(x); end
if nargout >2, x_med = median(x); end
if nargout >3, x_std = std(x); end
```

Load the file from
`http://jkcray.maths.ul.ie/ms4024/M-Files/marks2.m`.

To illustrate, Listing 10 shows how the **marks** M-file on Slide 60 can be rewritten as a function. Its usage is illustrated by

```
exmark = [12 0 5 28 87 3 56]
x_sort = marks2(exmark)
[x_sort,x_mean,x_med] = marks2(exmark)
[x_sort,x_mean,x_med,x_std] = marks2(exmark)
```

6.2.1 Passing Function Names as Parameters

The function `fdderiv` in Listing 11 below evaluates the finite difference approximation

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

to the function passed as its first argument.

Listing 11: fd_deriv.m

```
function [y,yc] = fd_deriv(f,x,h)
% FD_DERIV Finite difference approximation to derivative.
% FD_DERIV(F,X,H) is a finite difference approximation
% to the derivative of function F at X with difference
% parameter H. H defaults to SQRT(EPS).
% The second output parameter is the
% "central difference" --- usually more accurate.

if nargin < 3, h = sqrt(eps); end
y = (f(x+h) - f(x))/h;
if nargout>1
    yc=(f(x+h)-f(x-h))/2/h;
end
```

You can load the file **fd_deriv.m** from
`http://jkcray.maths.ul.ie/ms4024/M-Files/fd_deriv.m`.

To estimate the derivative of the built-in **Matlab** function **sqrt** we type

```
fd_deriv(@sin,pi/2)
```

We can use any built-in function or a user-written function m-file as the first argument in **fd_deriv**.

The (usually more accurate) central difference is returned if the optional second output parameter is requested:

```
[y,yc]=fd_deriv(@sin,pi/4,1.0e-4)
```

```
y =
   0.707071424669303
yc =
   0.707106780007960
```

Which answer is better (notice h–value is not particularly small)?

On the next slide another example is given. You can load the file **binsearch.m** from
`http://jkcray.maths.ul.ie/ms4024/M-Files/binsearch.m`

(The usual comments are omitted so that it will fit on a single slide!)

`binsearch` is called with (for example)

```
binsearch(@myfun, L,R,mytol)
```

where **myfun** is the name of the **Matlab** function m-file containing the function whose root we want to find. **L** and **R** bracket the interval containing a root — to ensure that they do, **binsearch** checks that **myfun(L)** and **myfun(R)** have opposite signs. If not **binsearch** displays a warning message and stops (returning control to the keyboard). Finally; **mytol** is optional, **binsearch** will set **tol** to a default value if this input parameter is not set.

Listing 12: binsearch.m

```
function [r,iter]=binsearch(fun,a,b,tol)
fa=fun(a);fb=fun(b);
if sign(fa)==sign(fb)
    my_warning=['Your function must have opposite signs at ' ...
        num2str(a) ' and ' num2str(b)];
    warning(my_warning);r=[];iter=[]; return
end
converged=0; %not converged yet!
maxiter=100;
iter=0;
if nargin<4
    tol=sqrt(eps);
end
while ~converged && iter<maxiter
    m=(a+b)/2; fm=fun(m);
    if sign(fa)~=sign(fm)
        b=m; % fb=fun(b); % not necessary!!
    else
        a=m; fa=fun(a);
    end
    dx=abs(b-a);
    converged= (dx<tol )|| abs(fm)<tol;
    iter=iter+1;
end
if converged
    r=m;
else
    r=[];
end;
```

6.3 Naming and Editing M-Files

To create and edit M-files you have two choices. You can use a generic text editor . Or you can use the built-in **Matlab** Editor. This is invoked by typing edit at the command prompt or from the <Alt> File-New or <Alt> File-Open menu options. The **Matlab** editor has various features to aid in editing M-files, including automatic indentation of loops and if structures, color syntax highlighting, and bracket and quote matching.

A really nice feature is "smart indentation" — where **Matlab** inserts tabs to show the structure of your file. Just enter <Alt> Edit Select All followed by <Alt> Text Smart Indent.

6.4 The Matlab Path

- Many **Matlab** functions are M-files residing on the disk, while others are built into the **Matlab** software (interpreter) when it is installed.
- The **Matlab** search path is a list of directories that specifies where **Matlab** looks for M-files.
- An M-file is available only if it is on the search path.
- Type **path** to see the current search path.
- The path can be set and added to with the **path** and **addpath** commands, or from the Path Browser that is invoked by the File-Set Path menu option or by typing **pathtool**.

- Several commands can be used to search the path.
- The **what** command lists the **Matlab** files in the current directory;**what** *dirname* lists the **Matlab** files in the directory *dirname* on the path.
- The command **lookfor** *keyword* (mentioned on Slide 68) searches the path for Mfiles containing *keyword* in their H1 line (the first line of help text).
- All the comment lines displayed by the **help** command can be searched using **lookfor** *keyword* **-all**.
- Some **Matlab** functions use comment lines after the initial block of comment lines to provide further information, such as bibliographic references (an example is **fminsearch)**).
- This information can be accessed using **type** but is not displayed by **help**.

- Typing **which myfile** displays the pathname of the function `myfile` or declares it to be built in or not found.
- This is useful if you want to know in which directory on the path an M-file is located.
- If you suspect there may be more than one M-file with a given name on the path you can use **which myfile -all** to display all of them.
- A script (but not a function) not on the search path can be invoked by typing **run** followed by a statement in which the full pathname to the M-file is given — e.g. **run 'C:\TMP\myfile.m'**

- You can list any M-file, *TOPIC.m* say, to the screen with **type TOPIC** or **type TOPIC.m**.
- (If there is an ASCII file called **TOPIC** then the former command will list TOPIC rather than TOPIC.m.)
- Preceding a **type** command with **more on** will cause the listing to be displayed a page at a time (**more off** turns off paging).
- Before writing an M-file it is important to check whether the name you are planning to give it is the name of an existing M-file or built-in function.
- This can be done in several ways: using which as just described, using **type** (e.g., **type lu** produces the response that `lu` is a built-in function), using **help**, or using the function **exist**.

- The command **exist('myname')** tests whether **myname** is a variable in the workspace, a file (with various possible extensions, including .m) on the path, or a directory.
- A result of 0 means no matches were found, while the numbers 1-7 indicate a match; see **help exist** for the precise meaning of these numbers.
- When a function that is on the search path is invoked for the first time it is compiled into memory.
- **Matlab** can usually detect when a function M-file has changed and then automatically recompiles it when it is invoked.
- To clear function fun from memory, type **clear fun**. To clear all functions type **clear functions**.

6.5 Command/Function Duality

- User-written functions are usually called by giving the function name followed by a list of arguments in parentheses.
- Yet some built-in **Matlab** functions, such as **type** and **what** described in the previous section, are normally called with arguments separated from the function name by spaces. This is not an inconsistency but an illustration of "command/function duality"!

Consider the toy function

```
function comfun(x,y,z)
% COMFUN Illustrative function with three string arguments.
disp(x), disp(y), disp(z)
```

We can call it with string arguments in parentheses (functional form), or with the string arguments separated by spaces after the function name (command form):

```
comfun('ab','cd','ef') % functional form
```

```
comfun ab cd ef % command form
```

The two invocations are equivalent — the output is:

```
ab
cd
ef
```

Other examples of command/function duality are (with the first in each pair being the most commonly used)

format long,	**format('long')**
disp('Hello'),	**disp Hello**
diary mydiary,	**diary('mydiary')**
warning off,	**warning('off')**

Note, however, that the command form should be used only for functions that take string arguments. In the example

```
mean 2
```

Matlab interprets 2 as a string and **mean** is applied to the ASCII code for the **character** 2, namely 50!!!

```
char(50)
```

```
double('2')
```

Finally, the command form can only be used when no output argument is required. So

```
x= mean 2
```

gives an error — which is probably just as well.

7 Plotting

Matlab has powerful graphics capabilities. Figures of many types can be easily generated and their ”look and feel” is highly customizable. In this chapter we cover the basic use of **Matlab** ’s most popular tools for graphing two- and three-dimensional data.

Our emphasis in this chapter is on generating graphics at the command line or in M-files, existing figures can also be modified and annotated interactively using the Plot Editor. To use the Plot Editor see **help plotedit** and the Tools menu and toolbar of the figure window.

7.1 Two-Dimensional Graphics

7.1.1 Basic Plots

- **Matlab** 's **plot** function can be used for simple "join-the-dots" x-y plots. Typing

```
x =[1.5 2.2 3.1 4.6 5.7 6.3 9.4];
y = [2.3 3.9 4.3 7.2 4.5 3.8 1.1];
plot(x,y)
```

  produces the picture in Figure 3, where the points x(i), y(i) are joined in sequence.

- **Matlab** opens a figure window (unless one has already been opened as a result of a previous command) in which to draw the picture.

- In this example, default values are used for a number of features, including the ranges for the x- and y-axes, the spacing of the axis tick marks, and the color and type of the line used for the plot.

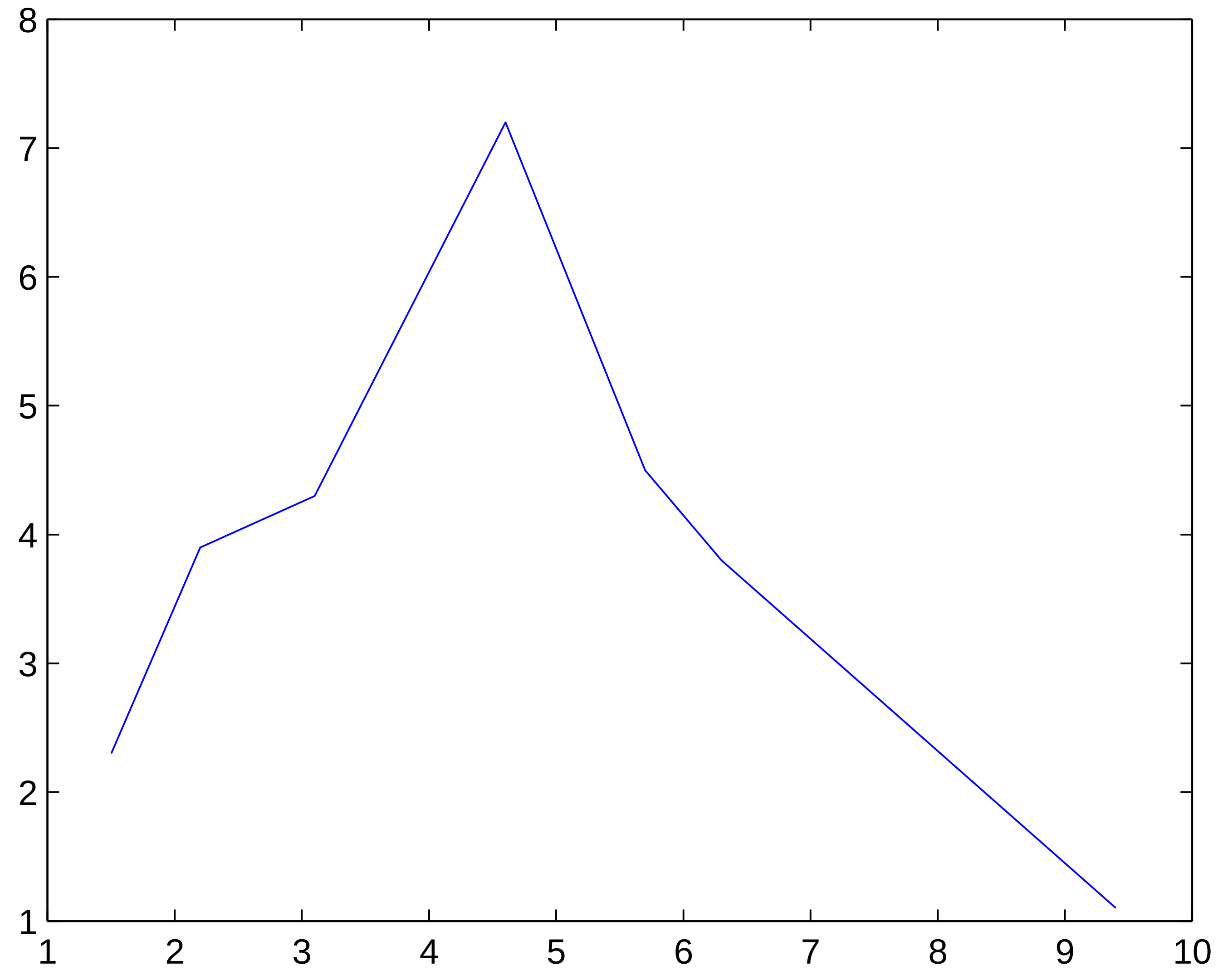


Figure 3: Simple x-y Plot

- More generally, we could replace **plot(x,y)** with **plot(x,y,string)**, where *string* combines up to three elements that control the color, marker and line style.
- For example, **plot(x,y,'r*–')** specifies that a red asterisk is to be placed at each point x(i), y(i) and that the points are to be joined by a red dashed line, whereas **plot(x,y,'y+')** specifies a yellow cross marker with no line joining the points.
- The picture in Figure 4 was produced with **plot(x,y,'kd:')**, which gives a black dotted line with diamond marker.

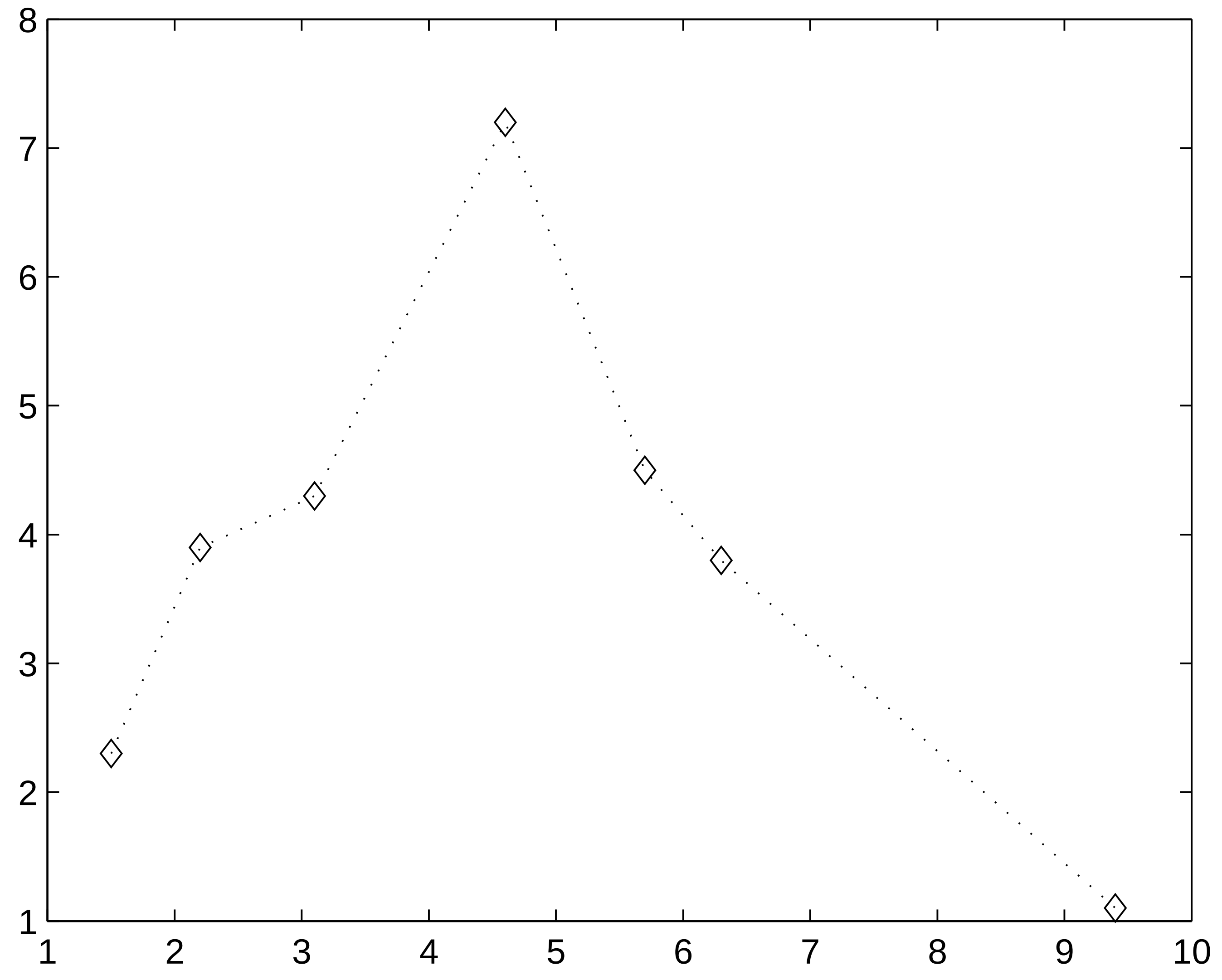


Figure 4: Variation on Simple x-y Plot

- Tables 9, 10 and 11 list the options available.
- The three elements in string may appear in any order, so, for example, **plot(x,y,'ms–')** and **plot(x,y,'s–m')** are equivalent.
- Note that more than one set of data can be passed to `plot`.
- For example, **plot(x,y,'g-',x, y.*sin(y),'r–')** superimposes plots of (x, y) and (x, y*sin(y)) with solid green and dashed red line styles, respectively.
- Experiment!

Symbol	Colour
r	Red
g	Green
b	Blue
c	Cyan
m	Magenta
y	Yellow
k	Black
w	White

Table 9: **Matlab** Colour Symbols

Symbol	Marker
o	Circle
*	Asterisk
.	Point
+	Plus
x	Cross
s	Square
d	Diamond

Symbol	Marker
^	Upward triangle
v	Downward triangle
>	Right triangle
<	Left triangle
p	Five-point star
h	Six-point star

Table 10: **Matlab** Marker Symbols

Symbol	Line style
-	Solid line (default)
–	Dashed line
:	Dotted line
-.	Dash-dot line

Table 11: **Matlab** Line Style Symbols

- The **plot** command also accepts matrix arguments.
- If x is an m-vector and Y is an $m \times n$ matrix, **plot(x,Y)** superimposes the plots created by x and each column of Y.
- Similarly, if X and Y are both $m \times n$, **plot(X,Y)** superimposes the plots created by corresponding columns of X and Y. If nonreal numbers are supplied to plot then imaginary parts are generally ignored.
- The only exception to this rule arises when `plot` is given a single argument.
- If Y is nonreal, **plot(Y)** is equivalent to **plot(real(Y),imag(Y))**. In the case where Y is real, **plot(Y)** plots the columns of Y against their index.

- You can exert further control by supplying more arguments to plot.
- The properties LineWidth (default 0.5 points) and MarkerSize (default 6 points) can be specified in points, where a point is 1/72 inch (from the days of "hot metal" printing).
- For example, the commands

```
plot(x,y,'LineWidth',2)
```

```
plot(x,y,'p','MarkerSize',10)
```

  produce plots with a 2-point line width and 5-pointed star markers in 10-point size, respectively.
- For markers that have a well-defined interior, *MarkerEdgeColor* and *MarkerFaceColor* can be set to one of the colors in Table 9.

- So, for example,

```
plot(x,y,'o','MarkerEdgeColor','m')
```

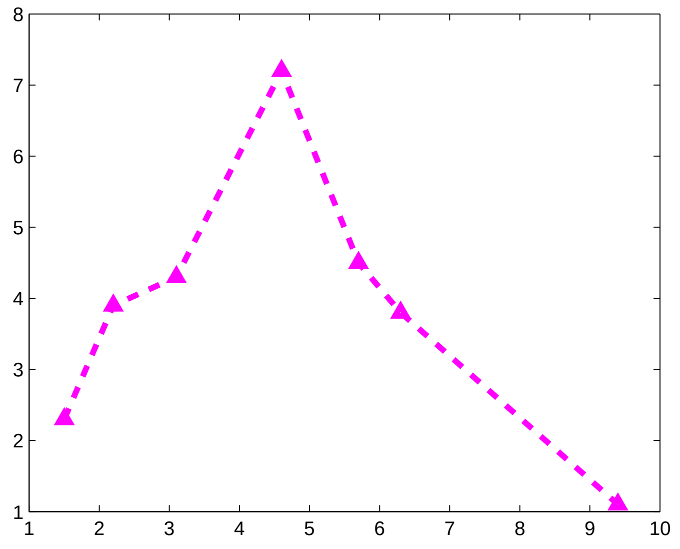
gives magenta edges to the circles.

- The upper plot in Figure 5 was produced with

```
plot(x,y,'m--^ ','LineWidth',3,'MarkerSize',5)
```

and the lower plot with

```
plot(x,y,'--rs','MarkerSize',20,'MarkerFaceColor','g')
```



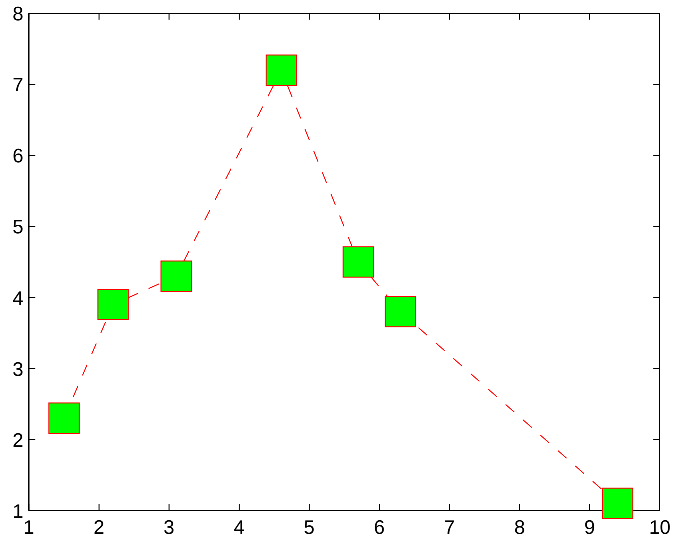


Figure 5: Non-default x-y Plots

- Using **loglog** instead of **plot** causes the axes to be scaled logarithmically.
- This feature is useful for revealing power-law relationships as straight lines.
- In the example below we plot $|1+h+h^2/2-\exp(h)|$ against h for $h = 1, 10{-}1, 10^{-2}, 10^{-3}, 10^{-4}$.
- Taylor's Theorem tells us that this quantity behaves like a multiple of h^3 when h is small, and hence on a log-log scale the values should lie close to a straight line of slope 3.
- To confirm this, we also plot a dashed reference line with the predicted slope, exploiting the fact that more than one set of data can be passed to the plot commands.
- The **Matlab** code is shown in Listing 13 and the output in Figure 6.

Listing 13: loglogex.m

```
 % LOGLOGEX
 % Plots the remainder in a 2nd order Taylor
 % expansion as a function of h
 % on a LogLog scale

h = 10.^[0:-1:-4];
taylerr = abs((1+h+h.^2/2) - exp(h));
loglog(h,taylerr,'-',h,h.^3,'--')
xlabel('h')
ylabel('abs(error)')
title('Error in quadratic Taylor series approx. to exp(h)')
box off
```

- Load the file from http://jkcray.maths.ul.ie/ms4024/M-Files/loglogex.m.

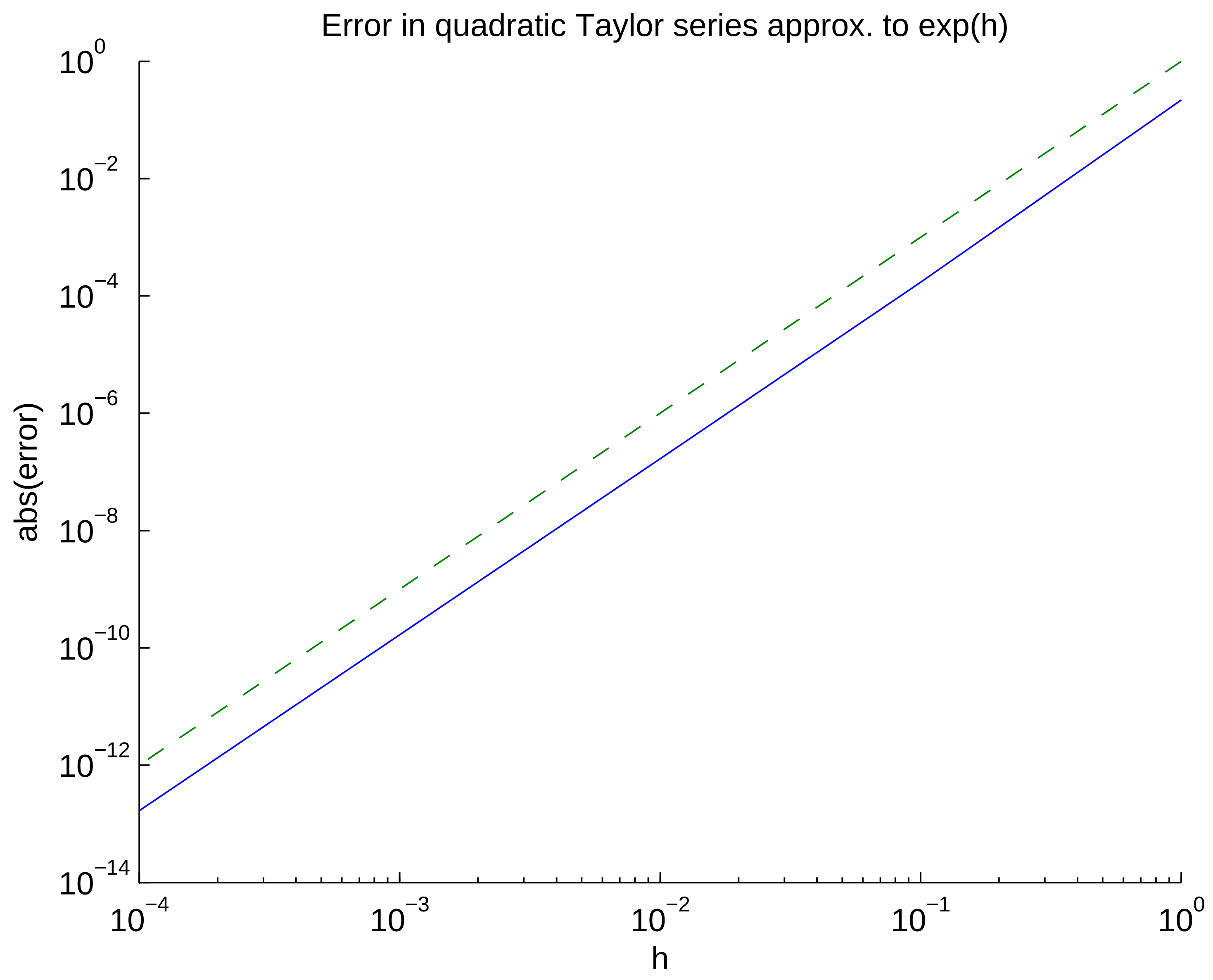


Figure 6: LogLog x-y Plots

- In this example, we used **title**, **xlabel** and **ylabel**.
- These functions reproduce their input string above the plot and on the x- and y-axes, respectively.
- We also used the command **box off**, which removes the box from the current plot, leaving just the x- and y-axes.
- **Matlab** will, of course, complain if nonpositive data is sent to loglog (it displays a warning and plots only the positive data).
- Related functions are **semilogx** and **semilogy** for which only the x- or y-axis, respectively, is logarithmically scaled.

- If one plotting command is later followed by another then the new picture will either replace or be superimposed on the old picture, depending on the current `hold` state (`on` or `off`).
- Typing **hold on** causes subsequent plots to be superimposed on the current one, whereas **hold off** specifies that each new plot should start afresh.
- The default status corresponds to **hold off**.

- The command **clf** clears the current figure window, while **close** closes it.
- It is possible to have several figure windows on the screen.
- The simplest way to create a new figure window is to type **figure**.
- The $\mathtt{n}^{\mathtt{th}}$ figure window (where n is displayed in the title bar) can be made current by typing **figure(n)**.
- The command **close n** causes the **nth** figure window to be closed.
- The command **close all** causes all the figure windows to be closed.

- Many aspects of a figure can be changed interactively, after the figure has been displayed, by using the items on the toolbar of the figure window or on the Tools pull-down menu.
- In particular, it is possible to zoom in on a particular region of the plot using mouse clicks (see **help zoom**).

7.1.2 Axes and Annotation

- Various aspects of the axes of a plot can be controlled with the axis command.
- The axes are removed from a plot with **axis off**.
- The aspect ratio can be set to unity, so that, for example, a circle appears circular rather than elliptical, by typing **axis equal**.
- The axis box can be made square with **axis square**.

- To illustrate, the left-hand plot in Figure 7 was produced by

```
t=0:0.01:2*pi; x=cos(t); y=sin(t);
plot(x,y)
axis equal, axis square
```

- Since the plot obviously lies inside the unit circle the axes are hardly necessary.
- The right-hand plot in Figure 7 was produced with

```
t=0:0.01:2*pi; x=cos(t); y=sin(t);
plot(x,y)
axis normal, axis off
```

  The command **axis normal** resets the aspect ratio so circles may look like ellipses.

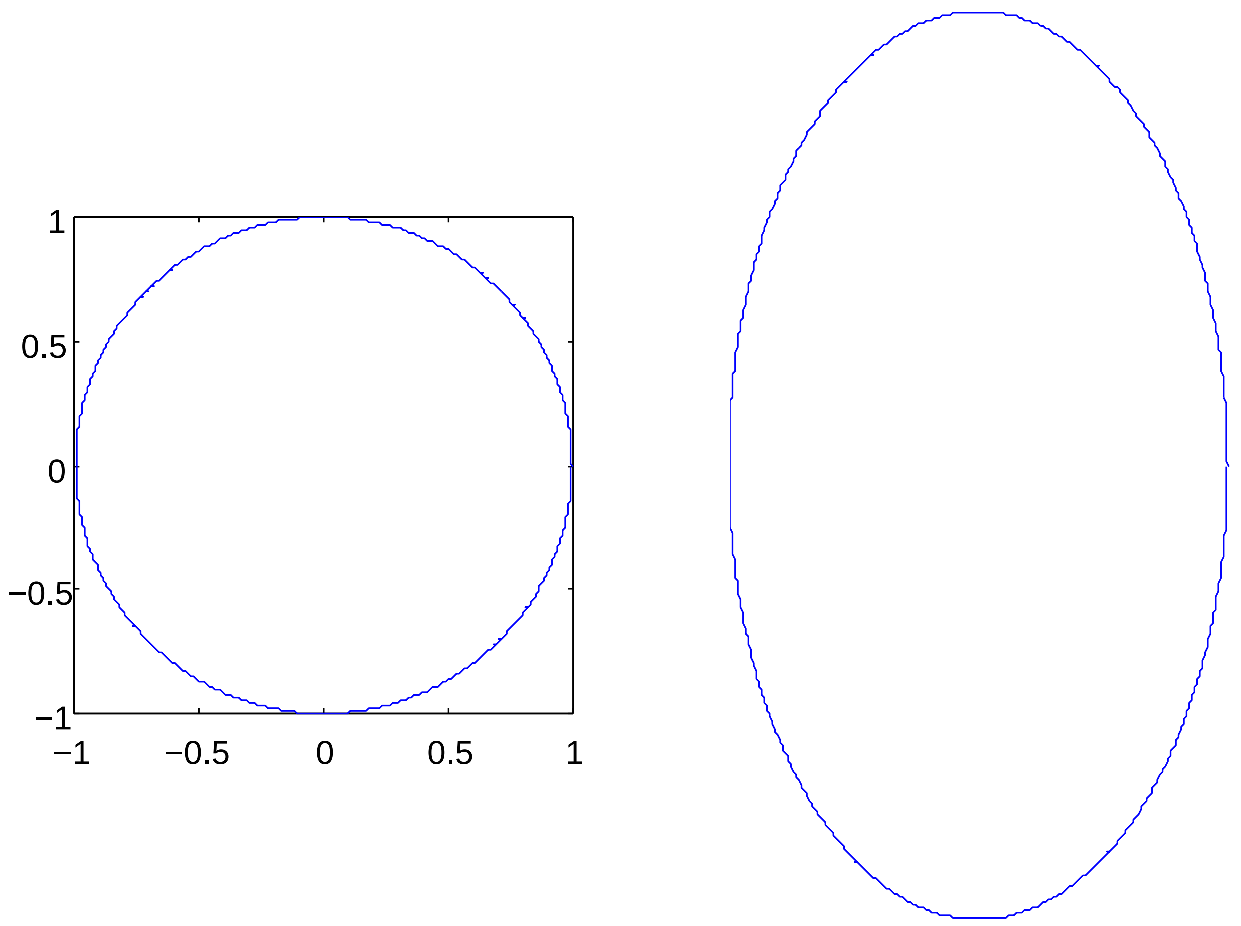


Figure 7: Boxed circle plot

- Setting **axis([xmin xmax ymin ymax])** causes the x-axis to run from xmin to xmax and the y-axis from ymin to ymax.
- To return to the default axis scaling, which **Matlab** chooses automatically based on the data being plotted, type **axis auto**.
- If you want one of the limits to be chosen automatically by **Matlab** , set it to -inf or inf;for example, **axis([-1 1 -inf 0])**.
- The x-axis and y-axis limits can be set individually with **xlim([xmin xmax])** and **ylim([ymin ymax])**.

- The next example plots the function $1/(x-1)^2 + 3/(x-2)^2$ over the interval $[0,3]$:

```
x = linspace(0,3,500);
plot(x,1./(x-1).^ 2 + 3./(x-2).^ 2)
grid on
```

- We specified **grid on**, which introduces a light horizontal and vertical grid that extends from the axis ticks.
- The result is shown in the left-hand plot of Figure 8.
- Because of the singularities at $x = 1, 2$, the plot is uninformative.

- However, by executing the additional command

```
ylim([0 50])
```

  the right-hand plot of Figure 8 is produced, which focuses on the interesting part of the first plot.

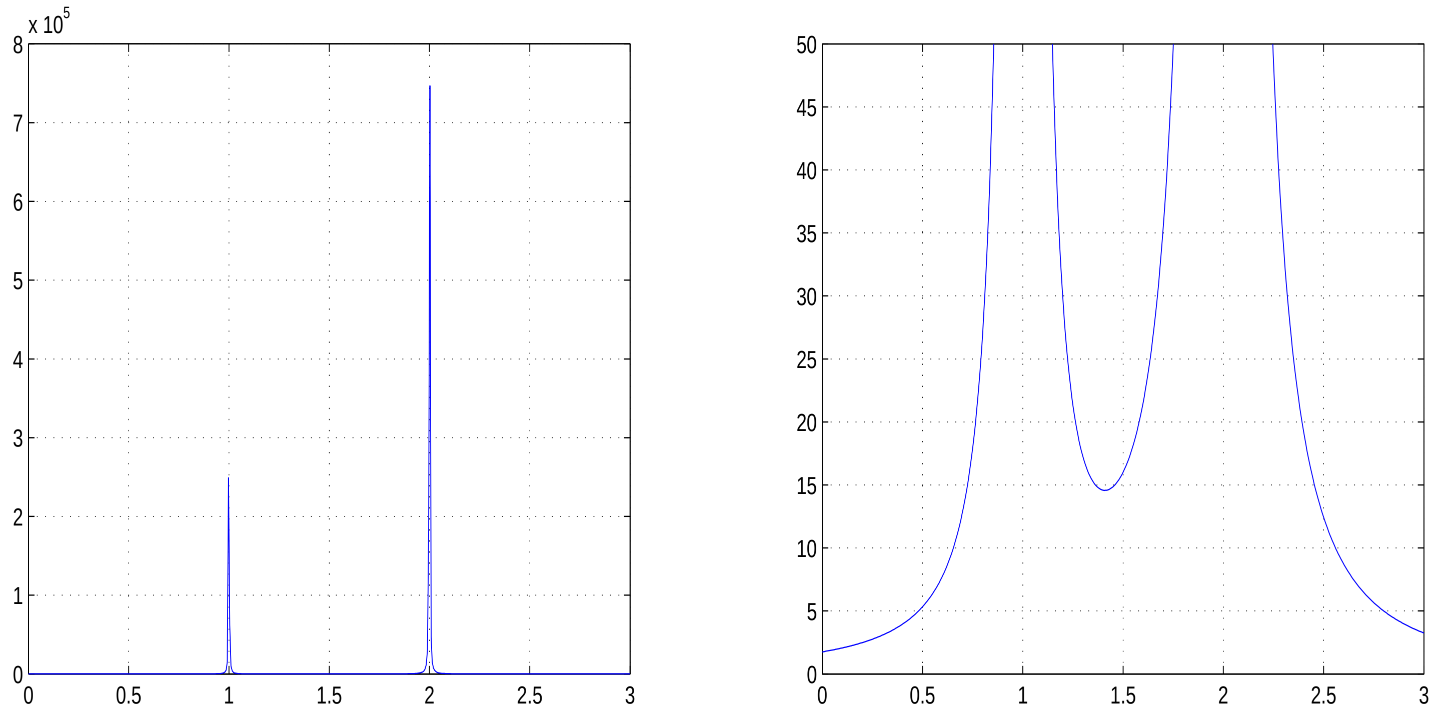


Figure 8: Use of **ylim** to change auto y-axis limits

- In the following example listed in Listing 14 we plot the "epicycloid"

$$0 \le t \le 10\pi \begin{cases} x(t) = (a+b)\cos(t) - b\cos((a/b+1)t) \\ y(t) = (a+b)\sin(t) - b\sin((a/b+1)t)) \end{cases}$$

  for a = 12 and b = 5 in Figure 9.
- Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/epicyc.m`.
- The axis limits were chosen to put some space around the epicycloid.

Listing 14: epicyc.m

```
% EPICYC Plots an epicycloid curve

a = 12; b = 5;
t = 0:0.05:10*pi;
x = (a+b)*cos(t) - b*cos((a/b+1)*t);
y = (a+b)*sin(t) - b*sin((a/b+1)*t);
plot(x,y)
axis equal, axis([-25 25 -25 25]),grid on
title('Epicycloid: a=12, b=5')
xlabel('x(t)'), ylabel('y(t)')
```

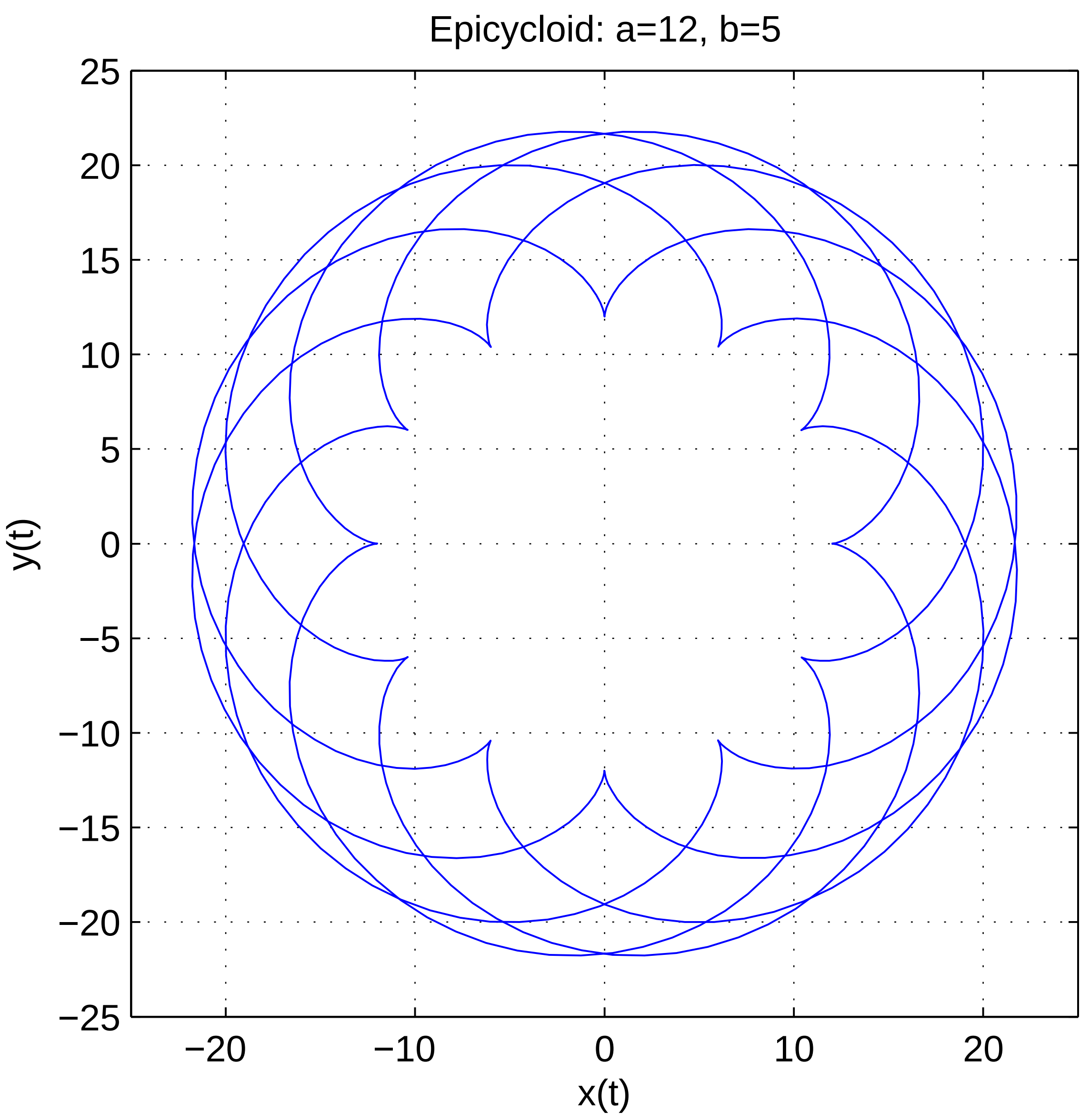


Figure 9: Epicycloid Example

- Next we plot the Legendre polynomials of degrees 1 to 4.
- We use the **legend** function to add a box that explains the line styles.
- The listing is in Listing 15.
- The output is in Fig. 10.
- Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/legpol.m`.

Listing 15: legpol.m

```
% LEGPOL Plots the first 4 Legendre polynomials

x = -1:.01:1; p1 = x; p2 = (3/2)*x.^ 2 - 1/2;
p3 = (5/2)*x.^ 3 - (3/2)*x;
p4 = (35/8)*x.^ 4 - (15/4)*x.^ 2 + 3/8;
plot(x,p1,'r:',x,p2,'g--',x,p3,'b-.',x,p4,'m-'), box off
legend('n=1','n=2','n=3','n=4',4)
xlabel('x','FontSize',12,'FontAngle','italic')
ylabel('P_n','FontSize',12,'FontAngle','italic')
title('Legendre Polynomials','FontSize',14)
text(-.6,.7,...
    '(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - n P_{n-1}(x)',...
    'FontSize',12,'FontAngle','italic')
```

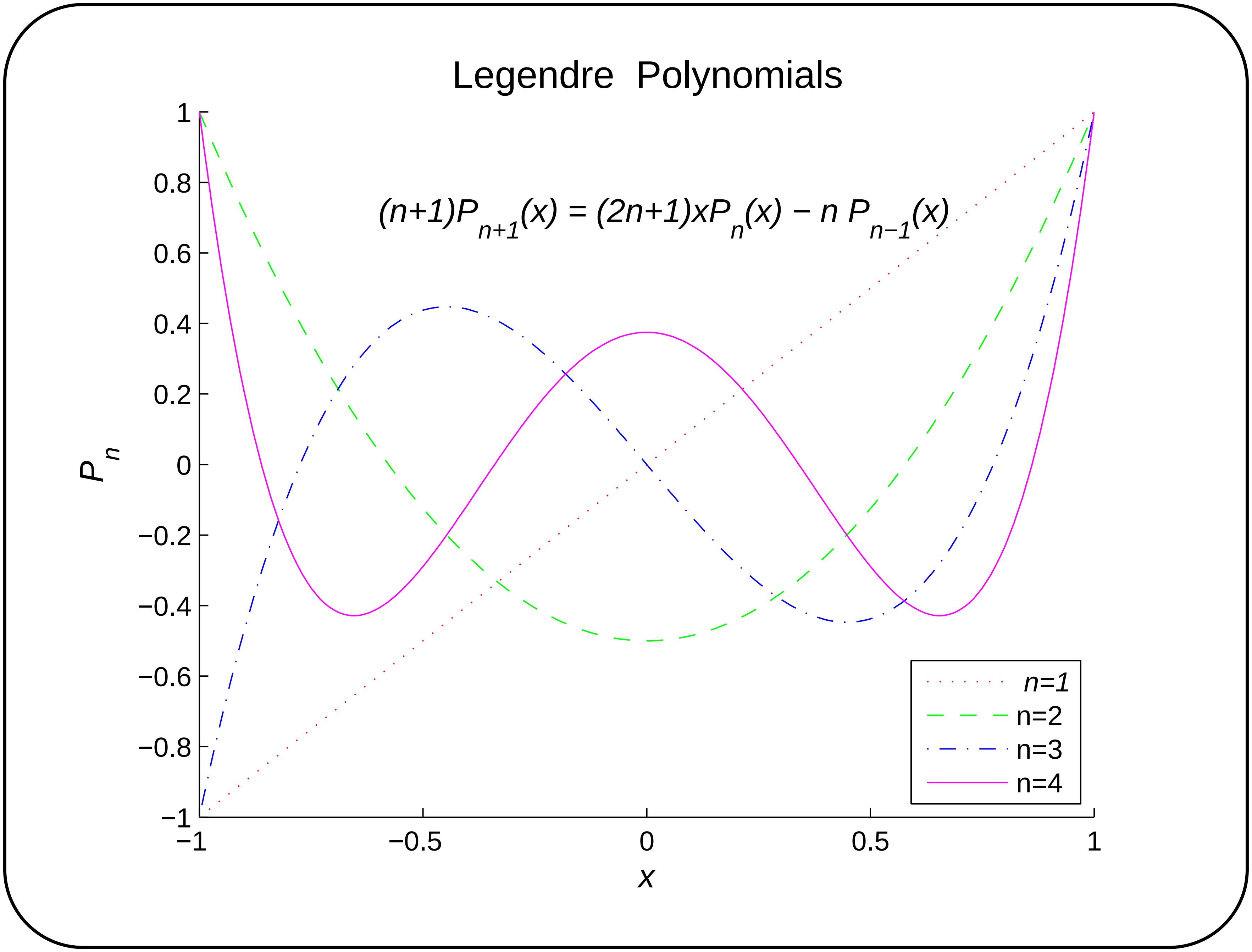


Figure 10: Legendre Polynomial Example

- Generally, typing **legend('string1','string2',...,'stringn')** will create a legend box that puts the $\mathtt{i}^{\mathtt{th}}$ string next to the color/marker/line style information for the $\mathtt{i}^{\mathtt{th}}$ plot.
- By default, the box appears in the top right-hand corner of the axis area.
- The location of the box can be specified by adding an extra argument as follows:

-1	to the right of the plot
0	automatically chosen "best" location
1	top right-hand corner (default)
2	top left-hand corner
3	bottom left-hand corner
4	bottom right-hand corner

- In the Legendre polynomial example we chose the bottom right-hand corner.
- Once the plot has been drawn, the legend box can be repositioned by putting the cursor over it and dragging it using the left mouse button.
- This example uses the **text** command: generally, **text(x,y,'string')** places 'string' at the position whose coordinates are given by x and y.
- You can specify Greek letters, mathematical symbols, fonts and superscripts and subscripts by using the notation of the typesetting system LaTeX(which you know a bit about) in the **xlabel**, **ylabel** and **text** commands.

- For example

```
title('\int_0^\infty f(x) dx')
```

produces a title of the form $\int_0^\infty f(x)dx$.

- Another example:

```
text(–.6,–.7,'\alpha^{-3/2}+\beta^{12}-\sigma_i')
```

produces a title of the form: $\alpha^{-3/2} + \beta^{12} - \sigma_i$ at location $(-0.6, -0.7)$ on the plot.

7.1.3 Multiple Plots in a Single Figure

- **Matlab** 's subplot allows you to place a number of plots in a grid pattern together on the same figure. Typing **subplot(m,n,p)** or, equivalently, **subplot(mnp)** (no spaces) splits the figure window into an $m \times n$ array of regions, each having its own axes.
- The current plotting commands will then apply to the p^{th} of these regions, where the count moves along the first row, and then along the second row, and so on.
- I recommend that you use the more readable **subplot(m,n,p)** notation.
- So, for example, **subplot(4,2,5)** splits the figure window into a 4-by-2 matrix of regions and specifies that plotting commands apply to the fifth region, that is, the first region in the third row.

- If **subplot(4,2,7)** appears later, then the region in the (4,1) position becomes active.
- Several examples in which **subplot** is used appear below.

- For plotting mathematical functions the **fplot** command is useful.
- It adaptively samples a function at enough points to produce a representative graph.
- The following example `http://jkcray.maths.ul.ie/ms4024/M-Files/subplots.m` generates the graphs in Fig. 11:

Listing 16: subplots.m

```
    % SUBPLOTS Demo of SUBPLOT plotting command

subplot(2,2,1), fplot('exp(sqrt(x)*sin(12*x))',[0 2*pi])
subplot(2,2,2), fplot('sin(round(x))',[0 10],'--')
subplot(2,2,3),fplot('cos(30*x)/x',[0.01 1 -15 20],'-.')
subplot(2,2,4), fplot('[sin(x),cos(2*x),1/(1+x)]',...
[0 5*pi -1.5 1.5])
```

- In this example,
  - The first call to **fplot** produces a graph of the function $e^{\sqrt{x} sin(12*x)}$ over the interval $0 \leq x \leq 2\pi$.
  - In the second call, we override the default solid line style and specify a dashed line with '–'.
  - The argument [0.01 1 -15 20] in the third call forces limits in both the x and y directions, $0.01 \leq x \leq 1$ and $-15 \leq y \leq 20$. The '-.' asks for a dash-dot line style.
  - The final **fplot** example illustrates how more than one function can be plotted in the same call.

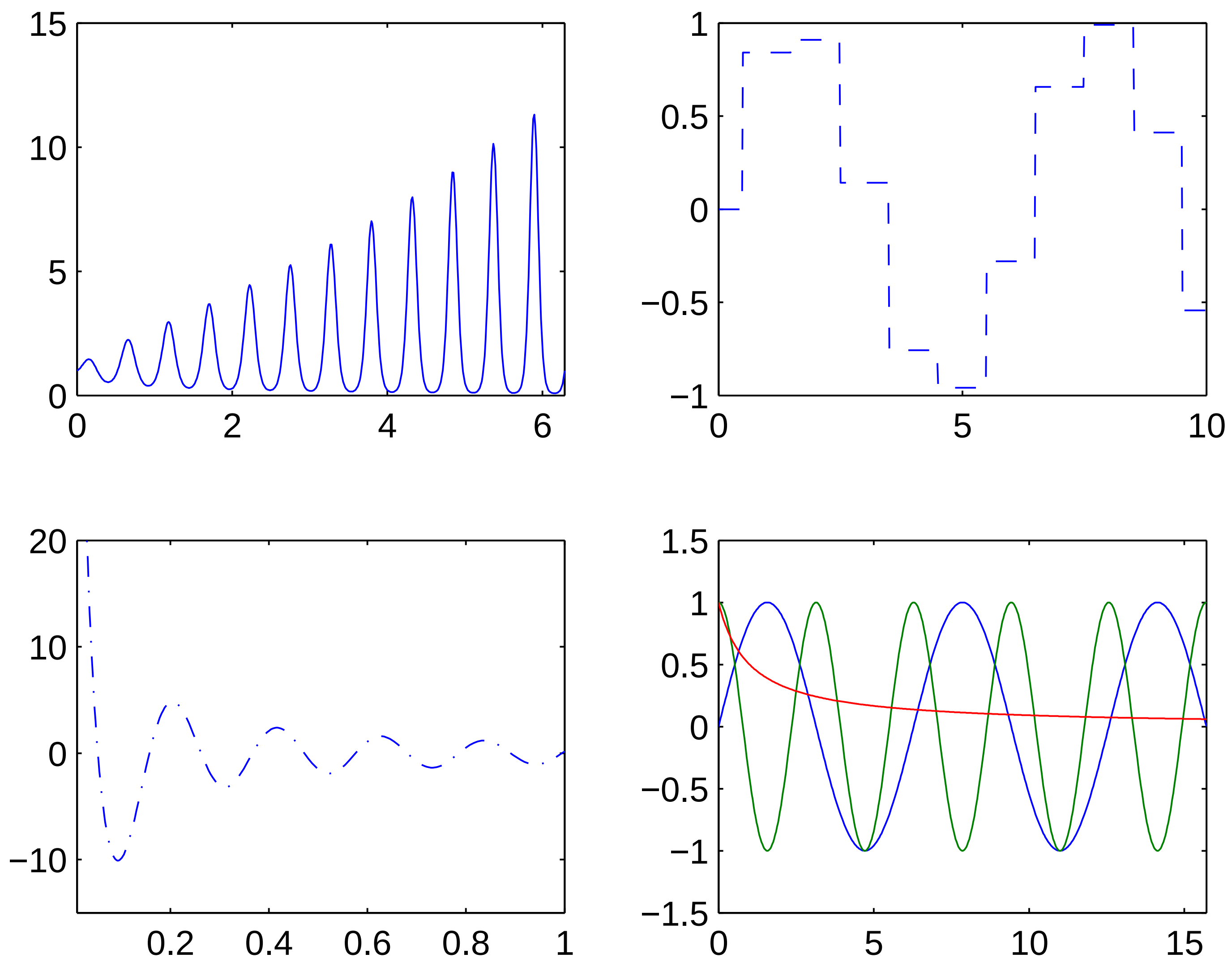


Figure 11: Example with **subplot** and **fplot**

- It is possible to supply further arguments to **fplot**.
- The general pattern is **fplot('fun',lims,tol,N,'LineSpec',p1,p2,...)** where **p1, p2, ...** are user-supplied parameters to be passed to the function specified.
- Unused parameters must be replaced by empty matrices [] as placeholders, see **help fplot** for details.
- A user function defined in an M-file can also be plotted with **fplot**.
- Take the function **cheby.m** listed in Listing 8.
- The following code produces the pictures in Figure 12.

```
subplot(2,1,1), fplot('cheby',[−1 1],[],[],[],5)
subplot(2,1,2), fplot('cheby',[−1 1],[],[],[],35)
```

- Here, the first 5 and first 35 Chebyshev polynomials are plotted in the upper and lower regions, respectively.
- Note the use of empty matrices to allow the unused parameters in **fplot** to be skipped and still pass 5 and 35 to **cheby**.

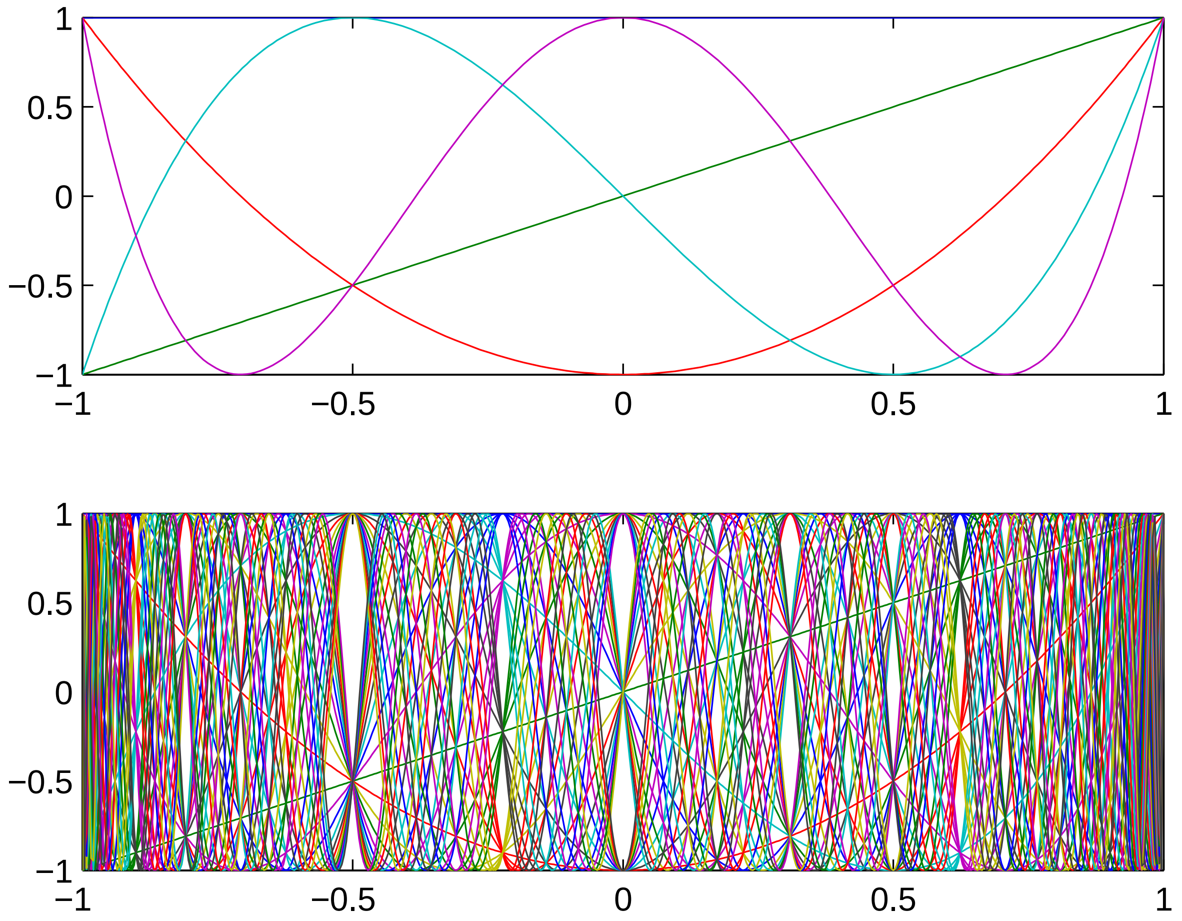


Figure 12: First 5 (upper) and 35 (lower) Chebyshev polynomials.

7.2 Three-Dimensional Graphics

- A very brief treatment of this more advanced topic.
- The function **plot3** is the three-dimensional analogue of **plot.**
- The following example illustrates the simplest usage: **plot3(x,y,z)** draws a "join-the-dots" **surface** by taking the points x(i), y(i), z(i) in order.

Listing 17: plot3ex.m

```
% PLOT3EX Generates a simple surface plot

t = -5:.005:5;
x = (1+t.^ 2).*sin(20*t); y = (1+t.^ 2).*cos(20*t); z = t;
plot3(x,y,z) ,grid on
xlabel('x(t)'), ylabel('y(t)'),zlabel('z(t)')
title('\it{plot3 example}"','FontSize',14)
```

- Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/plot3ex.m`.
- This example also uses the functions **xlabel, ylabel** and **title,** which were discussed in the previous section, and the analogous **zlabel.**
- Note that we have used the LaTeXnotation `\it` in the title command to produce italic text.
- The color, marker and line styles for **plot3** can be controlled in the same way as for **plot.** So, for example, **plot3(x,y,z,'rx–')** would use a red dashed line and place a cross at each point.
- Note that for 3D plots the default is **box off**; specifying **box on** adds a box that bounds the plot.
- The 3-D plot generated by this code is displayed in Fig. 13.

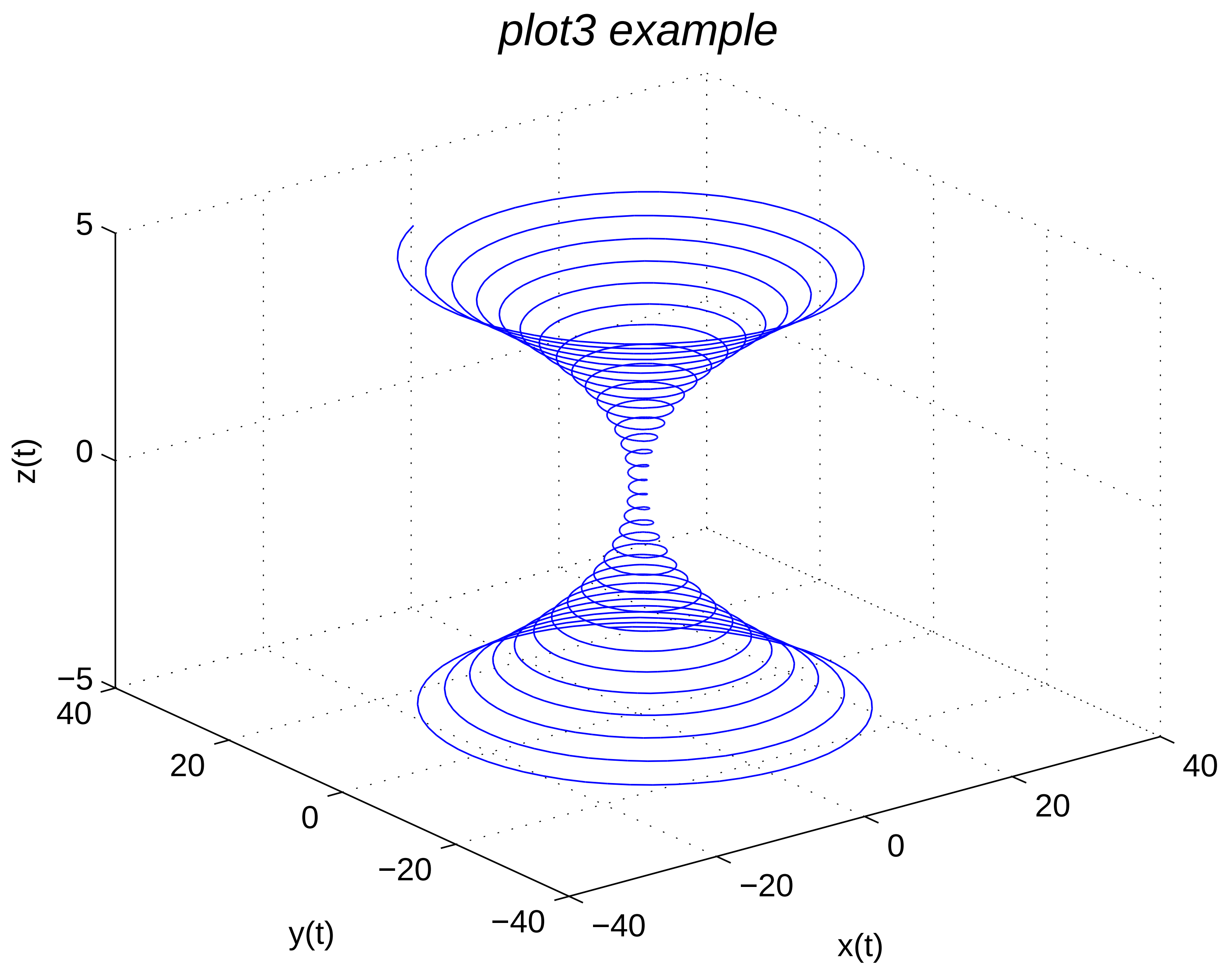


Figure 13: 3-D plot generated with **plot3**

7.2.1 Contour Plots

- A simple contour plotting facility is provided by **ezcontour.**
- The call to **ezcontour** in **lines 1 & 2** of the following example produces contours for the function $\sin(3y - x^2 + 1) + \cos(2y^2 - 2x)$ over the range $-2 \le x \le 2$ and $-1 \le y \le 1$.
- The result can be seen in the upper half of Figure 14.
- Note that the contour levels have been chosen automatically.

Listing 18: contourex.m

```
% CONTOUREX Generates two contour plots
% One with ezcontour and one
% with contour

subplot(211)
ezcontour('sin(3*y-x^2+1)+cos(2*y^2-2*x)',[-2 2 -1 1]);
x = -2:.01:2;y = -1:.01:1;
[X,Y] = meshgrid(x,y);
Z = sin(3*Y-X.^2+1)+cos(2*Y.^ 2-2*X);
subplot(212)
contour(x,y,Z,20)
```

- Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/contourex.m`.

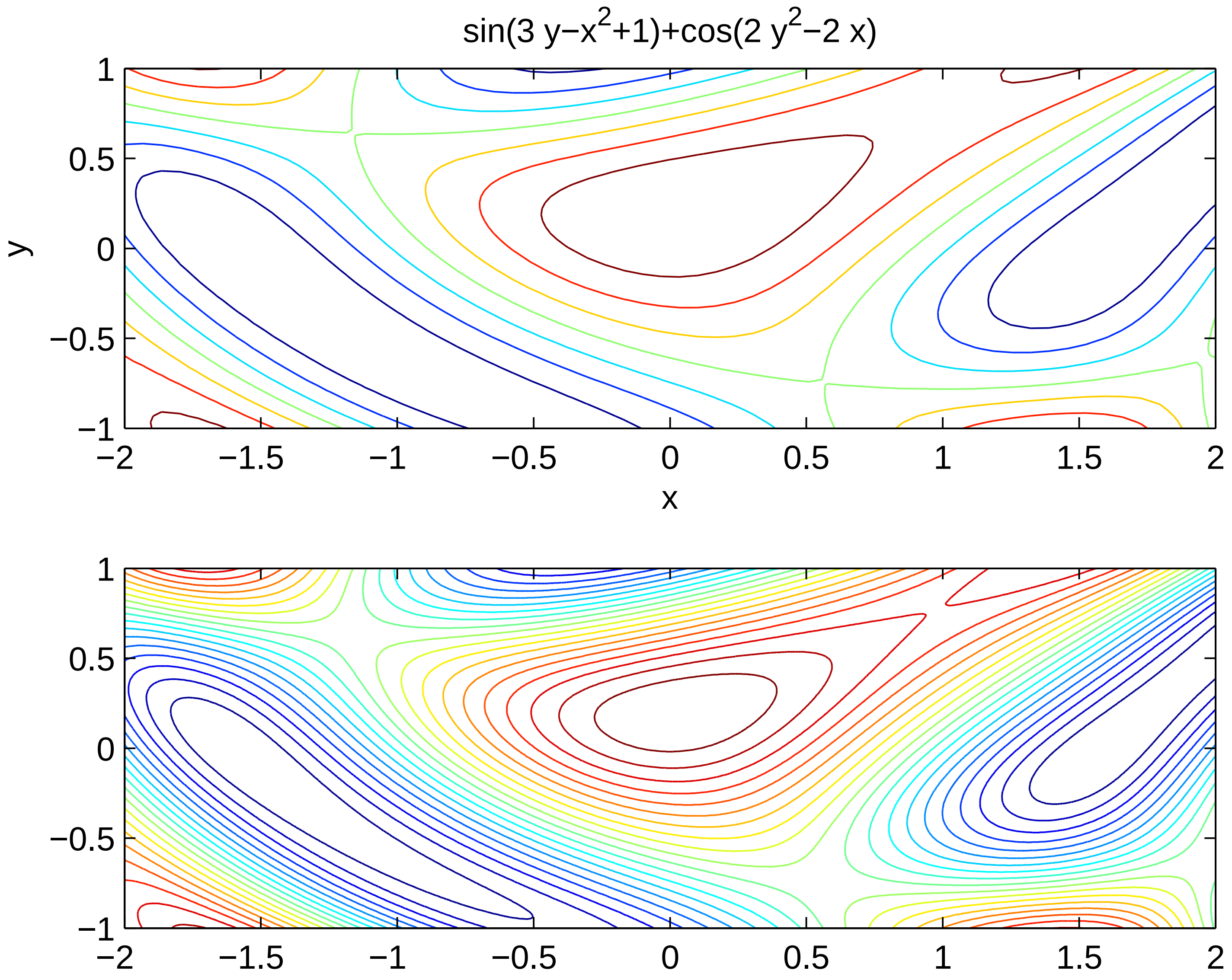


Figure 14: Contour plots with **ezcontour** (upper) and **contour** (below)

- For the lower half of Figure 14 we use the more general function **contour.**
- We first assign **x = -2:.01:2** and **y = -1:.01:1** to obtain closely spaced points in the appropriate range.
- We then set **[X,Y] = meshgrid(x,y)**, which produces matrices X and Y such that each row of X is a copy of the vector x and each column of Y is a copy of the vector y.
- The function **meshgrid** is extremely useful for setting up data for many of **Matlab** 's 3D plotting tools.
- The matrix Z is then generated from array operations on X and Y, with the result that Z(i,j) stores the function value corresponding to x(j), y(i).
- This is precisely the form required by **contour.**

- Typing **contour(x,y,Z,20)** tells **Matlab** to regard Z as defining heights above the x-y plane with spacing given by x and y.
- The final input argument specifies that 20 contour levels are to be used.
- If this argument is omitted **Matlab** automatically chooses the number of contour levels.

- The next example illustrates the use of clabel to label contours, with the result shown in Figure 15:

Listing 19: sixhumps.m

```
% SIXHUMPS Generates a contour plot with labelled contours

[X,Y] = meshgrid(-3:.05:3, -1.5:.025:1.5);
Z = 4*X.^2 - 2.1*X.^4 + X.^6/3 + X.*Y - 4*Y.^2 + 4*Y.^4;
cvals = [linspace(-2,5,14) linspace(5,10,3)];
[C,h] = contour(X,Y,Z,cvals);
clabel(C,h,cvals([3 5 7 9 13 17]))
xlabel('x'), ylabel('y')
title('Six humps function','FontSize',16)
```

Load the file from
http://jkcray.maths.ul.ie/ms4024/M-Files/sixhumps.m.

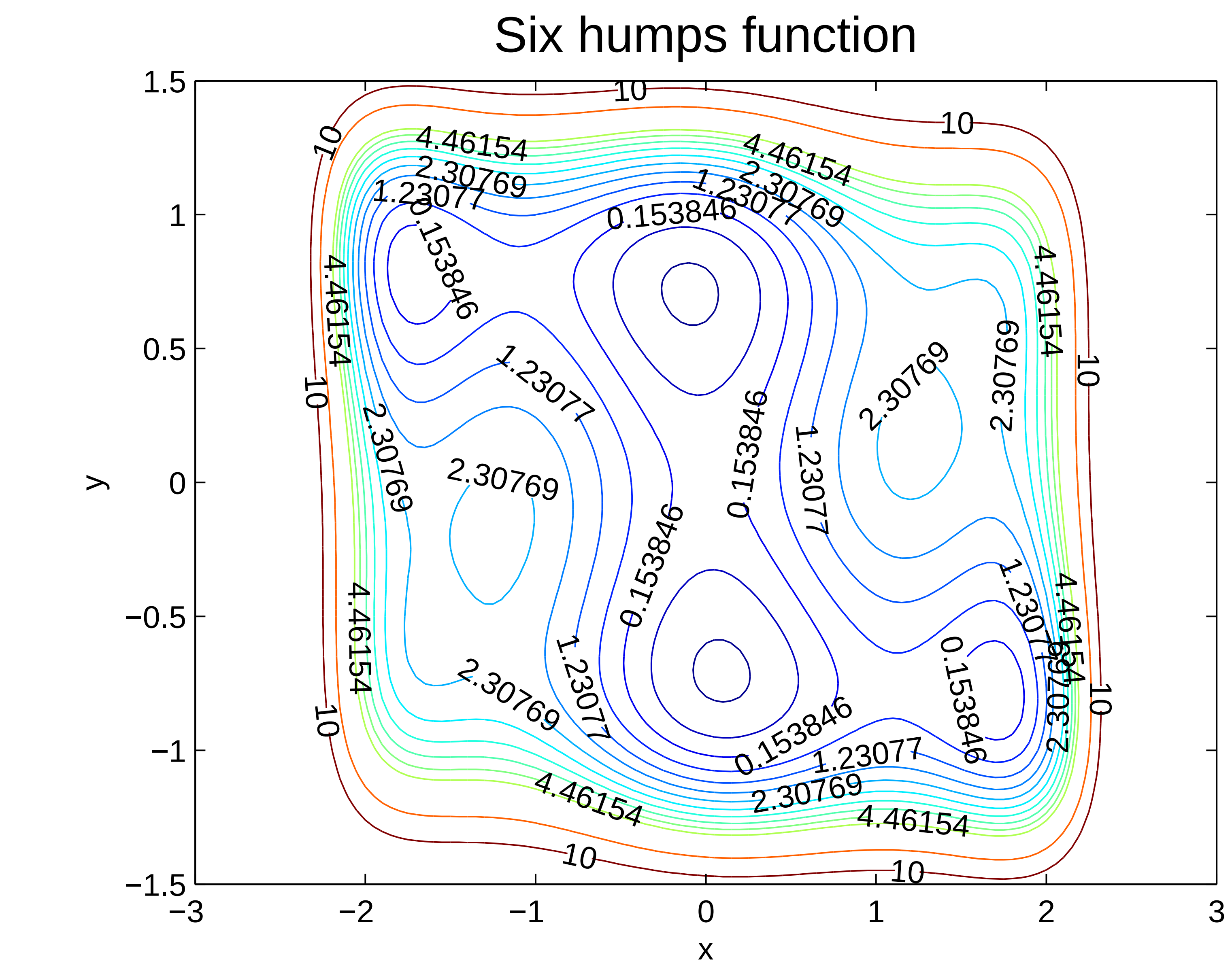


Figure 15: Contour plot labelled with **clabel**

- Here, we are using an interesting function having a number of maxima, minima and saddle points.
- **Matlab** 's default choice of contour levels does not produce an attractive picture, so we specify the levels (chosen by trial and error) in the vector cvals.
- The **clabel** command takes as input the output from contour (C contains the contour data and h is a "graphics object handle") and adds labels to the contour levels specified in its third input argument.
- Again the contour levels need not be specified, but the default of labelling all contours produces a cluttered plot in this example.

- An alternative form of **clabel** is

```
clabel(C,h,'manual')
```

  which allows you to specify with the mouse the contours to be labelled: click to label a contour and press return to finish.

- The h argument of **clabel** can be omitted, in which case the labels are placed close to each contour with a plus sign marking the contour.

7.2.2 Surface Plots

- Our final topic under the heading of 3-D plots is that of "wire-frame" surface plots.
- The function **mesh** accepts data in a similar form to **contour** and produces wireframe surface plots.
- If **meshc** is used in place of **mesh,** a contour plot is appended below the surface.
- The Example below, which produces Figure 16, involves the surface defined by $z = \sin(y^2 + x) - \cos(y - x^2)$ for $0 \leq x, y \leq \pi$.
- I've used the alternative (confusing) `subplot(mnp)` notation here.

- The first subplot is produced by **mesh(Z)**.
- Since no x, y information is supplied to **mesh,** row and column indices are used for the axis ranges.
- The second subplot shows the effect of **meshc(Z)**.
- For the third subplot, we use **mesh(x,y,Z)**, so the tick labels on the x- and y-axes correspond to the values of x and y.
- We also specify the axis limits with **axis([0 pi 0 pi -5 5])**, which gives $0 \leq x, y \leq \pi$ and $-5 \leq z \leq 5$.
- For the final subplot, we use **mesh(Z)** again, followed by **hidden off**, which causes hidden lines to be shown.

Listing 20: meshex.m

```
% MESHEX Generates four versions of the same mesh plot

x = 0:.1:pi;y = 0:.1:pi; [X,Y] = meshgrid(x,y);
Z = sin(Y.^2+X)-cos(Y-X.^ 2);
subplot(221), mesh(Z)
subplot(222),meshc(Z)
subplot(223),mesh(x,y,Z)
axis([0 pi 0 pi -5 5])
subplot(224),mesh(Z) ,hidden off
```

- Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/meshex.m`.

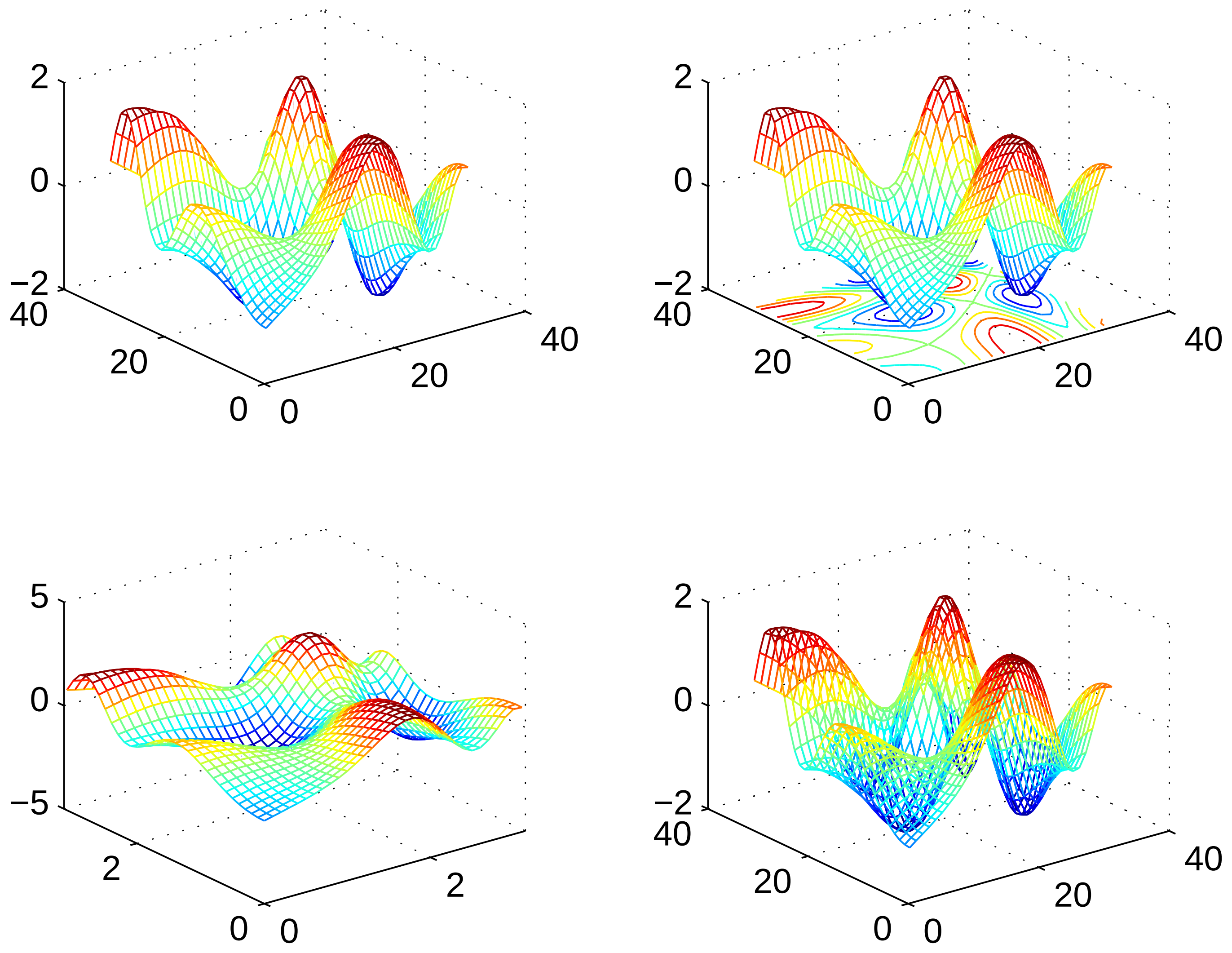


Figure 16: Surface plots with **mesh** and **meshc**

7.3 Saving and Printing Figures

- If you need to save a Figure as a graphics file, the best choice is to save it as a Portable network Graphics (**png**) file.
- There are two ways of doing this:
  1. From the command prompt type:

     ```
     print -dpng myfig.png
     ```

  2. or select the **F**ile **S**ave As menu item in the Figure window and save your Figure as a file with file extension **.png**, e.g. **myfig.png**.

- This file can be incorporated into a LaTeXdocument by inserting the following (or similar) LaTeXcode into your document: (you'll also need to insert `\usepackage{graphicx}` into the document preamble)

```
\begin{figure}[ht]\centering
  % Requires \usepackage{graphicx}
\includegraphics[width=8cm]{myfile.png}
\caption{A Remarkably Fine Plot}
\label{fig:fine-plot}
\end{figure}
```

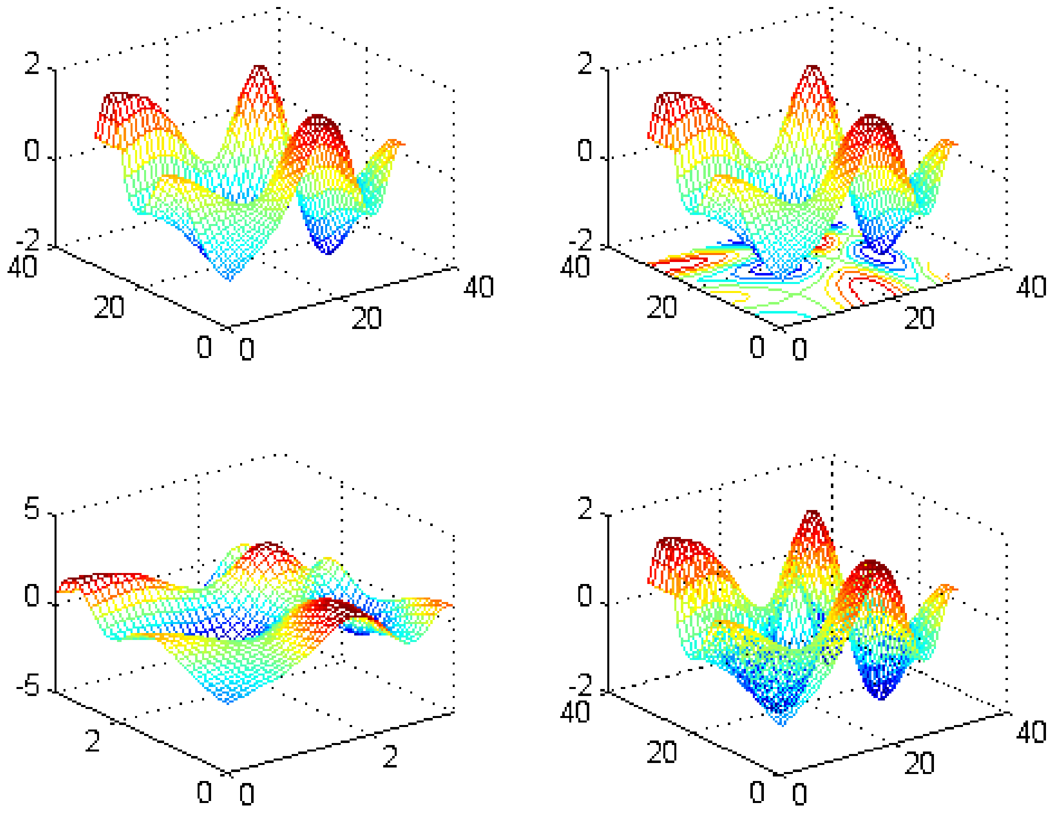


Figure 17: A Remarkably Fine Plot

- The many options of the **print** command can be seen with **help print**.
- The **print** command also has a functional form, illustrated by

```
print('-dpng','myfig.png')
```

  (an example of command/function duality — discussed in Sec. 6.5).
- To illustrate the usefulness of the functional form, the next example generates a sequence of five figures and saves them to files **fig1.eps,...,fig5.eps**:

- Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/printex.m`.

Listing 21: printex.m

```
% PRINTEX Generates 5 figures and saves to 5 files

x = linspace(0,2*pi,50);
for i=1:5
     plot(x,sin(i*x))
     print('-dpng',['fig' int2str(i) '.png'])
end
```

- The second argument to the **print** command in **printex.m** is formed by string concatenation, making use of the function int2str, which converts its integer argument to a string.
- Thus when i=1, for example, the print statement is equivalent to **print('-deps2','fig1.eps')**.

- Finally, the **saveas** command saves a figure to a file in a form that can be reloaded into **Matlab** . For example,

```
saveas(gcf,'myfig','fig')
```

saves the current figure as a binary FIG-file, which can be reloaded into **Matlab** with

```
open('myfig.fig')
```

- The term **gcf** is a reference to the **gcf** command that "**G**ets a handle to the **C**urrent **F**igure" — so that **Matlab** knows which Figure to save.
- Or just save and print from the File menu in the figure window.

8 Linear Algebra

- **Matlab** was originally designed for linear algebra computations, so it not surprising that it has a very complete set of functions for solving linear equation and eigenvalue problems.
- We will only discuss a very small subset of **Matlab** 's linear algebra functions in this short Chapter — corresponding to the material you studied in Linear Algebra 1.
- Most of the linear algebra functions work for both real and complex matrices.
- In mathematics we write A^* for the conjugate transpose of A and A^{T} for the transpose of A.
- You should remember that a square matrix A is Hermitian if $A^* = A$ and symmetric if $A^{\mathsf{T}} = A$.

- Similarly A is unitary if $A^*A = \mathrm{I}$, where I is the identity matrix and orthonormal if $A^\mathsf{T}A = \mathrm{I}$.
- The **Matlab** command for the hermitian conjugate A^* is **A'** and the command for the transpose A^T is **A.'**.
- To avoid repetition, we will use complex matrix terms even when the matrix is real.
- So, when the matrix is real, "Hermitian" can be read as "symmetric" and "unitary" can be read as "orthogonal".
- When A is real the results are the same so we may as well use the **A'** notation — we will always use **A'** rather than **A.'** in the rest of this Chapter.

8.1 Norms and Condition Numbers

- A norm is a scalar measure of the size of a vector or matrix.
- The p-norm of an n-vector x is defined by

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}, \quad 1 \le p < \infty$$

- For $p = \infty$, the norm is defined by

$$\|x\|_\infty = \max_{1 \le i \le n} |x_i|.$$

- As a special case, for $p = -\infty$, the norm is defined to be $\min_{1 \le i \le n} |x_i|$.
- The **Matlab** **norm** function can compute any p-norm and is invoked as **norm(x,p),** with default $p = 2$.

- For example:

```
x = 1:4;
[norm(x,1), norm(x,2), norm(x,inf), norm(x,-inf)]
```

- The p-norm of an $m \times n$ **matrix** is defined by (strictly speaking, `sup` rather than `max`)

$$\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}.$$

- The 1- and ∞-norms can be simplified and written down explicitly as

$$\|A\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}|, \quad \text{the maximum column sum}$$

$$\|A\|_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}|, \quad \text{the maximum row sum}$$

- For matrices the **norm** function is invoked as **norm(A,p)** and supports **p = 1,2,inf** and **p = 'fro'**, the "Frobenius" norm.
- It can be shown (in Linear Algebra 2) that the 2-norm of A can be expressed as the square root of largest of the eigenvalues of A^*A (or AA^*)

```
A=rand(3,5);
norm(A,2)
AtA=A'*A;
AAt=A*A';
sqrt(max(eig(AtA)))
sqrt(max(eig(AAt)))
```

- Another example: the Frobenius norm is defined mathematically as

$$A_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{\frac{1}{2}}.$$

- It can be shown that

$$A_F = \left(\text{trace}(A^* A) \right)^{\frac{1}{2}}.$$

```
A = [1 2 3;4 5 6;7 8 9] ;
[norm(A,1) norm(A,2) norm(A,inf) norm(A,'fro')]
```

- Finally for a nonsingular square matrix A, $\kappa(A) = \|A\| \|A^{-1}\|$ is the **condition number**.
- It measures the sensitivity of the solution of a linear system $Ax = b$ to perturbations in A and b.
- The matrix A is said to be "well conditioned" or "ill conditioned" according as $\kappa(A)$ is small or large.
- A large condition number implies that A is "nearly singular" — solutions to $Ax = b$ will be inaccurate.
- The condition number is computed by the **cond** function as **cond(A,p)**.
- The p-norm choices p = 1,2,inf,'fro' are supported, with default **p = 2**.

- For p = 2, non-square matrices are allowed, in which case the condition number is defined by $\kappa_2(A) = \|A\|_2 \|A^+\|_2$ where A^+ is the “pseudo-inverse “ to be mentioned later.
- Computing the exact condition number is expensive, so **Matlab** provides two functions for estimating the 1-norm condition number of a square matrix A, **rcond** and **condest**.
- The function **rcond** estimates the reciprocal of $\kappa_1(A)$ and **condest** estimates $\kappa_1(A)$.

```
A = hilb(5);
[cond(A,1) cond(A,2) condest(A) 1/rcond(A)]
```

- We will revisit the condition number after discussing the solution of Linear Equations.

8.2 Linear Equations

- The fundamental tool for solving a linear system of equations $Ax = b$ is the backslash operator, \.
- We enter `x=A\b` to solve the linear system.
- The three possible shapes for A lead to square, overdetermined and underdetermined systems, as described below.
- More generally, the \operator can be used to solve $AX = B$, where B is a matrix with p columns — in this case **Matlab** solves $AX(:, j) = B(:, j)$ for $j = 1 : p$.

8.2.1 Square Systems

- If A is an $n \times n$ nonsingular matrix then A**b**is the solution x to $Ax = b$, computed by LU factorization with partial pivoting.
- During the solution process **Matlab** computes **rcond(A)**, and it prints a warning message if the result is smaller than about **eps**:

```
x = hilb(15) \ ones(15,1);
```

Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 1.567841e-018.

```
cond( hilb(15))
```

- When **cond(A)** is large, (or equivalently, **rcond** is small) the accuracy of the solution is likely to be low.
- In the worst case, as many digits of accuracy can be lost in the solution as $\log_{10} \text{cond}(A)$ — so in the example just mentioned, as many as 18 digits of accuracy could be lost making the solution of little use.
- More precisely, it can be shown that if there is any error in b then the corresponding uncertainty in the solution may be the error in b multiplied, in the worst case, by $\text{cond}(A)$.

- For example:

Listing 22: condex.m

```
% CONDEX Demonstrates effects of large condition number on the
% accuracy of solutions to a linear system

    A=hilb(8); cond(A)
    b=ones(8,1);
    x=A\b;
    err=norm(A*x-b) % error is about 10^{-12}; pretty good!
    b1=b+1.0e-6*rand(8,1); % make a small change in b
    x1=A\b1; % x1 is the solution to the ``perturbed'' system
    norm(x1-x,inf) % measures the largest change in x(i), i=1..8,
    % can be as big as 1000!
```

- Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/condex.m`.
- We have $\text{cond}(A) \approx 10^{10}$ and we made a change of order 10^{-6} in b — and sure enough, the error in x is roughly equal to the product of these two factors.
- The moral is that ill-conditioned systems can have inaccurate solutions, **no matter what solution method is used**,

8.2.2 Overdetermined Systems

- If A has dimension $m \times n$ with $m > n$ then $Ax = b$ is an overdetermined system: there are more equations than unknowns.
- In general, there is no x satisfying the system.
- **Matlab** 's **A\b** gives a least squares solution to the system, that is, it minimizes **norm(A*x-b)** (the 2-norm of the residual) over all vectors x.
- If A has full rank n there is a unique least squares solution.
- If A has rank k less than n then **A\b** is a basic solution — one with at most k nonzero elements (k is determined, and x computed, using the QR factorization with column pivoting).
- In the latter case **Matlab** displays a warning message.

- A least squares solution to $Ax = b$ can also be computed as **x_min = pinv(A)*b**, where the function pinv computes the "pseudo-inverse".
- In the case where A is rank-deficient **x_min** is the unique solution of minimal 2-norm.
- A vector that minimizes the 2-norm of $Ax - b$ over all nonnegative vectors x, for real A and b, is computed by **lsqnonneg.**
- The simplest usage is **x = lsqnonneg(A,b)**, and several other input and output arguments can be specified, including a starting vector for the iterative algorithm that is used.

Example:

```
  A = gallery('lauchli',3,0.25), b = [1 2 4 8]';
  x = A \ b; % Least squares solution.}
  xn = lsqnonneg(A,b); % Nonnegative least squares solution.
[x xn], [norm(A*x-b) norm(A*xn-b)]
```

8.2.3 Underdetermined Systems

- If A has dimension $m \times n$ with $m < n$ then $Ax = b$ is an underdetermined system: there are fewer equations than unknowns.
- The system has either no solution or infinitely many.
- In the latter case **A\b** produces a basic solution, one with at most k nonzero elements, where k is the rank of A.
- This solution is generally not the solution of minimal 2-norm, which can be computed as **pinv(A)*b**.
- If the system has no solution (that is, it is inconsistent) then **A\b** is a least squares solution.

- Here is an example that illustrates the difference between the \and **pinv** solutions:

```
A = [1 1 1;1 1 −1], b = [3;1]
x = A \b;y = pinv(A)∗b;
[x y]
[norm(x) norm(y)]
[norm(A∗x−b) norm(A∗y−b)]
```

8.3 Eigenvalue problems

- Eigenvalue problems are straightforward to define but their efficient and reliable numerical solution is a complicated subject.
- **Matlab** 's **eig** function simplifies the solution process by recognizing and taking advantage of the number of input matrices, as well as their structure and the output requested.
- It automatically chooses among many different algorithms or algorithmic variants depending on the details of the problem!

8.3.1 Eigenvalues

- The scalar λ and nonzero vector x are an eigenvalue and corresponding eigenvector of the $n \times n$ matrix A if $Ax = \lambda x$.
- The eigenvalues are the n roots of the degree n characteristic polynomial $\det(\lambda I - A)$.
- The $n + 1$ coefficients of this polynomial are computed by **p = poly(A)**:

$$\det(\lambda I - A) = p_1 \lambda^n + p_2 \lambda^{n-1} n + \cdots + p_n \lambda + p_{n+1}.$$

- The eigenvalues of A are computed with the **eig** function: **e = eig(A)** assigns the eigenvalues to the vector **e**.

- More generally, **[V,D] = eig(A)** computes an n-by-n diagonal matrix D and an n-by-n matrix V such that **A*V = V*D**.
- Thus D contains eigenvalues on the diagonal and the columns of V are eigenvectors.
- Not every matrix has n linearly independent eigenvectors, so the matrix V returned by **eig** may be singular (or, because of roundoff, nonsingular but very ill conditioned).
- The matrix in the following example has two eigenvalues 1 and only one eigenvector (a "defective" matrix):

```
[V,D] = eig([2 -1;1 0])
```

- The scaling of eigenvectors is arbitrary (if x is an eigenvector then so is any nonzero multiple of x).
- As the previous example illustrates, **Matlab** normalizes so that each column of V has unit 2-norm.

- Note that eigenvalues and eigenvectors can be complex, even for a real (non-Hermitian) matrix.
- A Hermitian matrix has real eigenvalues and its eigenvectors can be taken to be mutually orthogonal.
- For Hermitian matrices **Matlab** returns eigenvalues sorted in increasing order and the matrix of eigenvectors is unitary to working precision.

```
[V,D] = eig([2 -1;-1 1])
norm(V'*V-eye(2))
```

- In the following example eig is applied to the (non-Hermitian) Frank matrix:

```
  F = gallery('frank',5)
  e = eig(F)'
```

- This matrix has some special properties, one of which we can see by looking at the reciprocals of the eigenvalues:

```
1./e
```

- Thus if λ is an eigenvalue then so is $1/\lambda$.
- The reason is that the characteristic polynomial is "anti-palindromic":

```
poly(F)
```

so that $\det(F - \lambda I) = -\lambda^5 \det(F - \lambda^{-1} I)$.

- As we said at the start of this short Chapter, there is much more to **Matlab** 's suite of Linear Algebra functions than the small sample presented here — but you already have enough to solve any problem likely to crop up in your U.L. course.

9 Input and Output

- In this chapter we discuss how to obtain input from the user, how to display information on the screen, and how to read and write text files.
- Note that textual output can be captured into a file (perhaps for subsequent printing) using the **diary** command.
- How to print and save figures was discussed in Section 7.3.

9.1 User Input

- User input can be obtained with the **input** function, which displays a prompt and waits for a user response:

```
x = input('Starting point: ')
```

```
Starting point:  0.5
x =
0.500000000000000
```

- Here, the user has responded by typing "0.5", which is assigned to x.
- The input is interpreted as a string when an argument 's' is appended:

```
mytitle = input('Title for plot: ','s')
```

- The function **ginput** collects data via mouse clicks.
- The command [**x,y**] = **ginput(n)** returns in the vectors x and y the coordinates of the next n mouse clicks from the current figure window.
- Input can be terminated before the nth mouse click by pressing the return key.
- One use of **ginput** is to find the approximate location of points on a graph.
- For example, with Fig. 10 in the current figure window (by running **legpol.m** — listed in Listing 15), you might type [**x,y**] = **ginput(1)** and click on one of the places where the curves intersect.

- The **pause** command suspends execution until a key is pressed, while pause(n) waits for n seconds before continuing.
- A typical use of pause is to wait for user input, as in:

```
warning('Silly Mistake - press a key to continue')
pause
```

- **Matlab** will display the message and wait for a keyboard input.

9.2 Output to the Screen

- The results of **Matlab** computations are displayed on the screen whenever a semicolon is omitted after an assignment and the format of the output can be varied using the format command.
- But much greater control over the output is available with the use of several functions.
- The **disp** function displays the value of a variable, according to the current format, without first printing the variable name and "=".

- If its argument is a string, **disp** displays the string.
- Example:

```
disp('Here is a 3-by-3 magic square')
disp(magic(3))
```

- More sophisticated formatting can be done with the **fprintf** function.
- The syntax is **fprintf(***format* **,list-of-expressions)**, where *format* is a string that specifies the precise output format for each expression in the list.
- In the example

```
fprintf('%6.3f\n', pi)
```

the `%` character denotes the start of a format specifier requesting a field width of 6 with 3 digits after the decimal point and \n denotes a new line (without which subsequent output would continue on the same line). If the specified field width is not large enough **Matlab** expands it as necessary:

```
fprintf('%6.3f\n', pi^ 10)
```

- The fixed point notation produced by **f** is suitable for displaying integers (using **%n.0f** — where **n** is the number of places before the decimal point) and when a fixed number of decimal places are required, such as when displaying euro and cents (using say **%6.2f**).
- If **f** is replaced by **e** then the digit after the period denotes the number of significant digits to display in exponential notation:

```
fprintf('%12.3e\n', pi)
```

- When choosing the field width remember that for a negative number a minus sign occupies one position:

```
fprintf('%5.2f\n%5.2f\n',exp(1),-exp(1))
```

- A minus sign just after the % character causes the field to be left-justified.
- Compare

```
fprintf('%5.0f\n%5.0f\n',9,103)
fprintf('%-5.0f\n%-5.0f\n',9,103)
```

- The format string can contain characters to be printed literally, as the following example shows:

```
m = 5;iter = 11;U = orth(randn(m)) + 1e-10;
fprintf('iter = %2.0f\n', iter)
fprintf('norm(U''*U-I) = %11.4e\n', norm(U'*U - eye(m)))
```

- Note that, within a string, ” represents a single quote.
- To print % and \use \% and \\in the format string.

- Another useful format specifier is g, which uses whichever of e and f produces the shorter result:

```
fprintf('%g %g\n', exp(1), exp(20))
```

- If more numbers are supplied to be printed than there are format specifiers in the fprintf statement then the format specifiers are reused, with elements being taken from a matrix down the first column, then down the second column, and so on.
- This feature can be used to avoid a loop.
- Example:

```
A = [30 40 60 70];
fprintf('%g miles/hour = %g kilometers/hour\n', ...
[A;8*A/5])
```

- The function **sprintf** is analogous to **fprintf** but returns its output as a string. It is useful for producing labels for plots.
- A simpler to use but less versatile alternative is **num2str**: num2str(x,n)**converts** x to a string with n significant digits, with n defaulting to 4.
- For converting integers to strings, **int2str** can be used.
- Here are three examples, the second and third of which make use of string concatenation

```
n = 16;
err_msg = sprintf('Must supply a %d-by-%d matrix',n,n)
disp(['Pi is given to 6 significant figures by ' ...
num2str(pi,6)])
i =3; title_str = ['Result of experiment ' int2str(i)]
```

9.3 File Input and Output

A number of functions are provided for reading and writing binary and formatted text files;type **help iofun** to see the complete list.

We show by example how to write data to a formatted text file and then read it back in. Before operating on a file it must be opened with the **fopen** function, whose first argument is the filename and whose second argument is a file permission, which has several possible values including 'r' for read and 'w' for write. A file identifier is returned by **fopen**; it is used in subsequent read and write statements to specify the file. Data is written using the **fprintf** function, which takes as its first argument the file identifier.

The code

```
A = [30 40 60 70];
fid = fopen('myoutput','w');
fprintf(fid,'%g miles/hour = %g kilometers/hour\n',[A;8*A/5]);
fclose(fid);
```

creates a file **myoutput** containing

```
30 miles/hour = 48 kilometers/hour
40 miles/hour = 64 kilometers/hour
60 miles/hour = 96 kilometers/hour
70 miles/hour = 112 kilometers/hour
```

The file can be read in as follows.

```
fid = fopen('myoutput','r');
X = fscanf(fid,'%g miles/hour = %g kilometers/hour')
```

The **fscanf** function reads data formatted according to the specified format string, which in this example says

- read a general floating point number (**%g**), skip over the string ' miles/hour = ', read another general floating point number and skip over the string ' kilometers/hour' .

The format string is recycled until the entire file has been read and the output is returned in a vector. We can convert the vector to the original matrix format using

```
X = reshape(X,2,4)'
```

Alternatively, a matrix of the required shape can be obtained directly:

```
X = fscanf(fid,'%g miles/hour = %g kilometers/hour', [2 inf]);
X = X'
```

The third argument to **fprintf** specifies the dimensions of the output matrix, which is filled column by column. We specify **inf** for the number of columns, to allow for any number of lines in the file, and transpose to recover the original format.

Finally, binary files are created and read using the functions **fread** and **fwrite**. See the online help for details of their usage.

10 Using Matlab Efficiently

- Most users of **Matlab** find that computations are completed fast enough that execution time is not usually a cause for concern.
- Some computations, though, particularly when the problems are large, require a significant time and it is natural to ask whether anything can be done to speed them up.
- This chapter describes some techniques that produce better performance from M-files.
- They all exploit the fact that **Matlab** is an interpreted language with dynamic memory allocation.
- Another approach to optimization is to compile rather than interpret **Matlab** code.

- The **Matlab** Compiler, available from The MathWorks as a separate product, translates **Matlab** code into C and compiles it with a C compiler.
- External C or Fortran codes can also be called from **Matlab** via the MEX interface.
- Vectorisation, discussed in the first section below, has benefits beyond simply increasing speed of execution.
- It can lead to shorter and more readable **Matlab** code.
- Furthermore, it expresses algorithms in terms of high-level constructs that are more appropriate for high-performance computing.
- **Matlab** 's profiler is a useful tool when you are optimizing M-files, as it can help you decide which parts of the code to optimize.

10.1 Vectorisation

- Since **Matlab** is a matrix language, many of the matrix-level operations and functions are carried out internally using compiled C or assembly code and are therefore executed at near optimum efficiency.
- This is true of the arithmetic operators *****, **+**, **-**, **”**, **/** and of relational and logical operators.
- However, **for** loops are executed relatively slowly.
- One of most important tips for producing efficient M-files is to avoid **for** loops in favor of vectorised constructs, that is, to convert **for** loops into equivalent vector or matrix operations.

- Consider the following example:

```
n = 5e5; x = randn(n,1);
tic, s = 0; for i=1:n, s = s + x(i)^ 2;end, toc
```

and

```
tic, s = sum(x.^ 2);toc
```

- In this example we compute the sum of squares of the elements in a random vector in two ways: with a **for** loop and with an elementwise squaring followed by a call to **sum**.
- The latter vectorised approach is two orders of magnitude faster.
- Another example: computing the graph of a function— say $\sin x$ on the range $[0, \pi]$.

- We need to calculate x varying from 0 to π and $y = \sin(x(i))$ as i ranges over the index of x. We could write

```
n=10000; tic; for i=1:n x(i)=pi*i/n; y(i)=sin(x(i)); end; toc
plot(x,y)
```

- The elapsed time (using **tic** & **toc**) to form the vectors x and y is about 0.06 seconds.
- But a much simpler (and better) method is

```
tic;x=pi/n*(1:n);y=sin(x);toc,plot(x,y)
```

- Now the elapsed time is about 0.002 seconds — 30 times faster!
- The timings will vary from machine to machine but the lesson is clear; **for** loops should be avoided.

- For a non-trivial example of vectorisation, consider the inner loop of Gaussian elimination applied to an $n \times n$ matrix A, which can be written

```
for i = k+1:n
   for j = k+1:n
      A(i,j) = A(i,j) - A(i,k)*A(k,j)/A(k,k);
   end
end
```

- Both loops can be avoided, simply by deleting the two `for`s and `end`s:

```
i = k+1:n;
j = k+1:n;
A(i,j) = A(i,j) - A(i,k)*A(k,j)/A(k,k);
```

- The approximately $(n-k)^2$ scalar multiplications and additions have now been expressed as one matrix multiplication and one matrix addition.
- Using a random $n \times n$ matrix A with n = 1600 and k = 1 the double loop takes about 3 times as long — vectorisation yields a substantial improvement.
- Load the file from `http://jkcray.maths.ul.ie/ms4024/M-Files/ticktock.m`

Listing 23: ticktock.m

```
% TICKTOCK compares time for inner loop of G.E.
% using FOR loops vs not.
k=1;n=1600;
AA=rand(n,n);
A=AA;
[m,n]=size(A);
tic;
for i = k+1:n
    for j = k+1:n
        A(i,j) = A(i,j) - A(i,k)*A(k,j)/A(k,k);
    end
end
fprintf('Time using FOR loops %g seconds \n', toc)
A=AA;
tic;
i = k+1:n;
j = k+1:n;
A(i,j) = A(i,j) - A(i,k)*A(k,j)/A(k,k);
fprintf('Time without using FOR loops %g seconds\n', toc)
```

```
Time using FOR loops 0.64673 seconds
Time without using FOR loops 0.20288 seconds
```

10.1.1 J.I.T.

- But!
- Maybe things are not that simple.
- Let's look at another example; the (pointless) piece of code

```
function test1 (n, runs)
for r = 1:runs
   for i = 1:n
      j = i * i;
   end
end
```

`test1(200,10000)`

just calculates $1^2, 2^2, \ldots, n^2$ **runs** times (with $n = 200$ and **runs** $= 10000$).

- The run-time measured using

```
tic;test1(200,10000);toc
```

  is about 0.018 seconds. (This will vary from machine to machine.)

- It would seem like a Good Idea to set a variable **range=1:n** so as to avoid evaluating the list **1:n** over and over again **runs** times inside the outer loop.

- This gives the code:

```
function test2 (n, runs)
range = 1:n;
for r = 1:runs
   for i = range
      j= i * i;
   end
end
```

- The run-time measured using

```
   tic;test2(200,10000);toc
```

  is about 6 seconds — 300 times as long!

- What is going on?

- Starting with **Matlab** 6.5 (our version is much newer), improvements were made to the **Matlab** interpreter.
- The JIT-Accelerator can speed up **Matlab** code — in particular loop structures.
- This is done automatically — provided certain rules are respected.

- Some of the rules that determine whether JIT-Acceleration will be applied:
  1. The loop structure is a **for** loop.
  2. The loop uses arrays that are 3-d or less.
  3. All variables used in the loop are defined prior to loop execution.
  4. Memory for all variables used in the loop is allocated before entering the loop.
  5. Variables do not change in size or data type inside the loop.
  6. Loop ranges take the form M:N:P (e.g. 1:3:90).
  7. Only built-in **Matlab** functions are used in the loop.
  8. Conditional statements (e.g. **if/then/else**) only involve scalar comparisons.

- The code for **test2.m** violates Rule 6 so JIT-Acceleration is not applied.
- When JIT-Acceleration is applied, the speedups are such that hand-vectorisation may be only marginally useful.
- Consider, for example, this code to assign to **row_norm** the ∞-norms of the rows of A:

```
for i=1:n
    row_norms(i) = norm(A(i,:), inf);
end
```

- It can be replaced by the single statement

```
  row_norms = max(abs(A),[],2);
```

(see Section 4.5), which obviously much neater — though a bit obscure.

- And it doesn’t use a **for** loop.
- Frustratingly, the vectorised second version takes about twice as long!
- The **for** loop in the first version does obey all the rules listed for JIT-Acceleration which is why it is competitive with the vectorised version.
- There is no simple way to determine whether a JIT-Accelerated **for** loop will be faster than hand-vectorised code.

- For example the factorial function $n!$ is more quickly calculated by

```
n=100;tic;prod(1:n),toc
```

(2.4600e-04 seconds) than by

```
n=100;tic;p=1;for i=1:n,p=p*i;end;p,toc
```

(3.7200e-04 seconds).

- The moral is complicated — carefully written **for** loops are now at least competitive with hand-vectorised code.
- The difference is usually not very large.
- And badly written **for** loops will be **much** slower than either as JIT Acceleration will not be applied to your code.

10.2 Preallocating Arrays

- One of the attractions of **Matlab** is that arrays need not be declared before first use: assignment to an array element beyond the upper bounds of the array causes **Matlab** to extend the dimensions of the array as necessary.
- Unfortunately, if overused, this flexibility can lead to inefficiencies, however.
- Consider the following implementation of a recurrence relation:

```
% x has not so far been assigned.
n=1e4; x(1:2) = 1;
for i=3:n
    x(i) = 0.25*x(i-1)^2 - x(i-2);
end
```

- On each iteration of the loop, **Matlab** must increase the length of the vector x by 1.
- In the next version x is preallocated as a vector of precisely the length needed, so no resizing operations are required during execution of the loop:

```
% x has not so far been assigned.
n=1e4; x=ones(n,1);
for i=3:n
    x(i) = 0.25*x(i-1)^ 2 - x(i-2);
end
```

- With n = 1e4, the first piece of code took 0.1403 seconds and the second 5.8400e-04 seconds, showing that the first version spends most of its time doing memory allocation rather than floating point arithmetic.

- Preallocation has the added advantage of reducing the fragmentation of memory resulting from dynamic memory allocation and deallocation.
- You should pre-allocate memory for arrays whenever possible.

10.3 Miscellaneous Optimisations

- Suppose you wish to set up an $\mathtt{n} \times \mathtt{n}$ matrix of 2's.
- The obvious assignment is

  $>> \mathtt{A} = \mathtt{2} * \mathtt{ones}(\mathtt{n});$
- The $\mathtt{n}^2$ floating point multiplications can be avoided by using

  $>> \mathtt{A} = \mathtt{repmat}(\mathtt{2}, \mathtt{n});$
- The **repmat** approach is much faster for large n.
- This use of **repmat** is essentially the same as assigning

  $>> \mathtt{A} = \mathtt{zeros}(\mathtt{n});\ \mathtt{A}(:) = \mathtt{2};$

  in which scalar expansion is used to fill A.

- There is one final optimisation that is automatically performed by **Matlab** .
- Arguments that are passed to a function are not copied into the function's workspace unless they are altered within the function.
- Therefore there is no memory penalty for passing large variables to a function provided the function does not alter those variables.

Part X

Supplementary Material

A First Matlab Project for 10%

- You are asked to write a **Matlab** script m-file to:
  - generate a vector **r** of 5 random numbers between -1 and 1
  - generate the coefficients of the quintic (polynomial of degree 5) that has these 5 numbers as its roots (use the **poly** command)
  - generate the coefficients of the **derivative** of this quintic (use the **polyder** command)
  - plot the quintic and its derivative (using the **polyval** command) on two sub-plots of a single plot, one above the other (use the **subplot** command)
  - The x–range should be from the smallest root to the largest with intervals of 0.01.

  - Give appropriate titles to the two sub-plots such as "Plot of quintic with roots at: `root1`, `root2`, `root3`, `root4`, `root5`"" where `root1` etc are the numerical values of the 5 roots (use the **num2str** and **title** commands)

- You are asked to submit a single Adobe PDF-file prepared using LaTeX containing:
  - a listing of your **Matlab** script m-file (see `http://jkcray.maths.ul.ie/ms4024/LaTeX-Files/` and `http://jkcray.maths.ul.ie/ms4024/LaTeX-Files/HowToIncludeMatlabCode.tex` in particular)
  - the plot created by your file
  - a clear but short (at most half a page) explanation of your work.

- You will submit your work electronically during a MS4024 class.
- You may not bring project work into class on a USB stick or on paper.
- Incomplete work at the end of a class may be submitted and retrieved at the start of a subsequent class.
- You may be given a code to identify your work as your own.

B Second Matlab Project for 15%

Outline of Project

- You are asked to write **Matlab** script and function m-files to find the roots of a specified function (not necessarily a polynomial) using a particular root-finding method.
- You are also asked to create a plot of the function together with the successive root estimates.
- Finally you are asked to prepare an Adobe PDF file containing listings of your m-files, graphical output and a commentary.

Details

- You are asked to write a **Matlab** **function** m-file `halley.m` to implement Halley's method (explained on Slide 358) for root-finding.
- `halley.m` will have an input parameter list consisting of the following three parameters :
  - the name of the **function** m-file that calculates the function whose root is to be found
  - a starting guess as to the value of the root
  - a tolerance/accuracy for the final value

- `halley.m` should return
  - x, the best estimate of the root
  - a vector X of successive root estimates
  - the number N of iterations needed
  - the value $fval$ of the function at the final root estimate.
- You are also asked to write a a **Matlab** **function** m-file `fun.m` that evaluates the function $f(x)$ (to be supplied) together with its first and second derivatives at a point x, (x is passed as a parameter to `fun.m`).
- `fun.m` should return the values of the function and its first and second derivatives at x.
- You will need to use finite-difference code (see Listing 11) to evaluate the first and second derivatives of $f(x)$.

- You are also asked to write a **Matlab** script m-file `Main.m` that
  - inputs a start point from the user,
  - runs `halley.m`,
  - displays the results
  - generates a Figure showing your test function close to the start point together with the successive root estimates (displayed as suitably-sized symbols joined by lines).

- You are asked to submit a single Adobe PDF-file prepared using LaTeX containing:
  - a listing of `halley.m`
  - a listing of `fun.m`
  - a listing of `Main.m`
  - the plot created by `Main.m`
  - a sample of the output from `Main.m` — use the `diary` command
  - a clear but short (at most a page) explanation of your work.

- You will submit your work electronically during a MS4024 class.
- You may not bring project work into class on a USB stick or on paper.
- Incomplete work at the end of a class may be submitted and retrieved at the start of a subsequent class.
- You may be given a code to identify your work as your own.

Halley's Method Halley's method is an variation on the more familiar Newton's method for root-finding. Newton's method uses the formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{B.1}$$

to update the estimate x_n of the root x^* (the x-value that satisfies $f(x^*) = 0$. (It can be shown that Newton's method has quadratic convergence.)

Halley's method uses the more complicated update formula:

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2[f'(x_n)]^2 - f(x_n)f''(x_n)}. \tag{B.2}$$

(It can be shown that Halley's method has cubic convergence.)

Both methods have their limitations as you will learn in Numerical Analysis.

C Third Matlab Project for 25%

Outline of Project

- You are asked to write **Matlab** script and function m-files that, for a given set of x–y values:
  - calculates an interpolating polynomial
  - calculates a least squares polynomial fit of any degree
  - compute a cubic spline interpolant
- You are also asked to create plots of the various polynomial approximations overlaid on the given data.
- Finally you are asked to prepare an Adobe PDF file containing listings of your m-files, graphical output and a commentary.

Details

- You are asked to write a **Matlab** function m-file `bell.m` that evaluates the "bell-shaped" function $\mathsf{bell}(\mathsf{x}) = 1/(1+\mathsf{x}^2)$.
- You are asked to write a **Matlab** function m-file `makedata.m` that inputs a vector `x` and outputs a vector `y` where `y= bell(x)`.
- You are asked to write a **Matlab** function m-file `intpoly.m` that computes the polynomial of specified degree that best fits any given x-y data. (See Slide 365 for details.)
  - `intpoly.m` should input the vectors `x` and `y` together with degree `m` of the required best fit polynomial.
  - `intpoly.m` should output `pfit`, the vector of `m+1` coefficients of the required best fit polynomial.

- You are asked to write a **Matlab** function m-file `genspline.m` that generates a piece-wise cubic spline representation for any given x-y data. (See Slide 371 for details.)
  - `genspline.m` should input the vectors `x` and `y`.
  - `genspline.m` should output an $n \times 4$ array `C` of coefficients — one row for each of the n cubics that make up the piece-wise cubic spline, where x and y have length $n + 1$.

- You are asked to write a **Matlab** function m-file `splineval.m` that evaluates a cubic spline at a point in its domain. (See Slide 380 for details.)
  - `splineval.m` should input the vector `x` used in the construction of the spline, the array `C` output by `genspline.m` and a scalar or vector z at which the spline is to be evaluated.
  - `splineval.m` should output the corresponding scalar or vector `y`.

- You are also asked to write a **Matlab** script m-file `Main.m` that:
  - sets `x=(-2:0.01:2)';`
  - sets `n` equal to `length(x)-1`
  - runs `makedata.m` to output a vector `y` where `y= bell(x)`.
  - runs `intpoly.m` with $m = n$ to generate **the** interpolating polynomial.
  - runs `intpoly.m` with $m < n$ (say $m = 10, 20, 50, 100, 200$) to generate a succession of best-fit polynomials.
  - runs `genspline.m` to generate a cubic spline representation for the data
  - plots the interpolating polynomial and the five best-fit polynomials
  - plots the cubic spline representation using `splineval.m`.

- You are asked to submit a single Adobe PDF-file prepared using LaTeX containing:
  - listings for
    - `Main.m`,
    - `makedata.m`
    - `bell.m`
    - `intpoly.m`
    - `genspline.m`
    - `splineval.m`
  - the plots created by `Main.m`
  - a sample of the output from `Main.m` — use the `diary` command
  - a clear but short (at most two pages) explanation of your work and interpretation of your results.

Polynomial Interpolation and Least Squares Fits We want to find the polynomial of degree m that "best fits" the given x-y data. So we want the polynomial $p_m(x) = a_1 x^m + a_2 x^{m-1} + \cdots + a_{m+1} = \sum_{j=1}^{m+1} a_j\, x^{m+1-j}$ that satisfies

$$p_m(x_i) \equiv \sum_{j=1}^{m+1} a_j\, x_i^{m+1-j} = y_i, \quad i = 1, \ldots n+1. \tag{C.1}$$

Note: the reason we write $p_m(x) = a_1x^m + a_2x^{m-1} + \cdots + a_{m+1}$ rather than $p_m(x) = a_1 + a_2x + \cdots + a_{m+1}x^m$ is that **Matlab** calculates and stores the coefficients of polynomials in order of **decreasing** powers of x. For example:

```
>> poly([1  2])
```

returns

```
ans =
1 -3 2
```

A second note: **Matlab** does not allow vectors to be indexed by zero or negative numbes so we are forced to store $a_0, a_1, \ldots, a_m$ as `[a(1) a(2) ... a(m+1)]`.

Hence the complicated power of x_i in $\sum_{j=1}^{m+1} a_j(x_i)^{m+1-j}$. Fortunately we never need to use it — we can work with vectors instead.

The equation (C.1) is a linear system of $n+1$ equations in $m+1$ unknowns $a_1, a_2, \ldots, a_{m+1}$ — in general there is a unique solution only when $m = n$, the interpolation case. If $m < n$ then we are solving a Least-Squares problem — i.e. trying to find the values of $a_1, a_2, \ldots, a_{m+1}$ that most nearly solve the linear system.

In either case there is a neat way to write the linear system (C.1) as a matrix equation:

$$\begin{bmatrix} x_1^m & x_1^{m-1} & \dots & x_1 & 1 \\ x_2^m & x_2^{m-1} & \dots & x_2 & 1 \\ \vdots & \vdots & \dots & \vdots & \\ x_{n+1}^m & x_{n+1}^{m-1} & \dots & x_{n+1} & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{m+1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{bmatrix} \tag{C.2}$$

or just

$$Va = y \tag{C.3}$$

where the “Vandermonde” matrix V can be written

$$V = \begin{bmatrix} x^m & x^{m-1} & \dots & x & 1 \end{bmatrix} \tag{C.4}$$

in a short-hand for the matrix on the left in (C.2).

So in **Matlab** notation, we can solve for the coefficients with the simple **Matlab** command:

`>> a = V\ y`

When $m = n$ then a is the unique solution. This works even when $m < n$ and in fact when $m < n$ **Matlab** computes `a=V\y` to be precisely the value of a that makes $\|Va - y\|^2$ as small as possible — the least squares solution!

(You can check this by comparing the "textbook" formula for a; $a = (V^T V)^{-1} V^T y$ with the answer you get with `V\y`. They will be the same apart from small errors due to **Matlab** using a more accurate numerical method when computing `V\y` than `(V'*V)\V'*y`.)

How to construct V? Suppose $m = 3$, then

```
>> V = [x.^3  x.^2  x  ones(n+1,1)]
```

The challenge you face is to find a neat way to automate this command for **arbitrary** m.

Once you have calculated V you can solve for a and then evaluate $p_m(x)$ using

```
>> pm = polyval(a, x)
```

where x can be a scalar or a vector.

Cubic Splines Introduction

- We have $n+1$ points $(x_1, y_1), \dots (x_{n+1}, y_{n+1})$.
- We set out to compute n different cubics $Y_i(x)$, each of which is used only on the interval $[x_i, x_{i+1}]$ where $i = 1, 2, \dots n$.
- The advantage is that we are using (lots of) polynomials of degree 3 rather than a single polynomial of high degree.
- As you have seen, working with high-degree polynomials leads to numerical difficulties.
- In fact it is convenient to write each of the Y_i as $Y_i(t)$ where $t = \dfrac{x - x_i}{x_{i+1} - x_i}$ and $i = 1, 2, \dots n$.
- With this definition (for any given $i = 1, 2, \dots n$) the value $t = 0$ corresponds to $x = x_i$ and $t = 1$ to $x = x_{i+1}$.

- Obviously we have to make sure that the polynomials "join up" — i.e. are continuous at the $n-1$ internal points or "knots"; $x_2, \ldots, x_n$.
- This gives $n-1$ equations for the $4n$ unknowns (the coefficients of the n different cubics).
- We can also require that the first and second derivatives are continuous at the $n-1$ knots.
- This gives $2(n-1)$ extra equations.
- Finally we must require that each cubic gives the "right" answer at each of the $n+1$ points — i.e. $y_i = Y_i(0)$ and $y_{i+1} = Y_i(1)$ for $i = 1, 2, \ldots, n$.
- This seems to give $2n$ extra conditions but due to continuity at the $n-1$ internal points we only have $2n-(n-1) = n+1$ distinct equations.

- Summing we have $n-1+2(n-1)+n+1=4n-2$ equations for the $4n$ unknowns.
- We need two more equations to determine the $4n$ coefficients uniquely — the choice that **Matlab** 's built-in spline function uses is the "no-knot" rule that requires that the **third derivative** be continuous at x_2 and x_n so

$$Y_1'''(1) = Y_2'''(0) \text{ and} \tag{C.5}$$
$$Y_{n-1}'''(1) = Y_n'''(0). \tag{C.6}$$

Cubic Splines Details We can reduce the $4n$ equations for the $4n$ unknowns to a linear system of n equations for n unknowns. The details follow — you can skip to Slide 377 if you only want the formulas for now — of course you'll need to understand the details when you are writing your report.

- We have $n+1$ x-values $x_1, \ldots, x_{n+1}$ and y-values $y_1, \ldots, y_{n+1}$.
- As mentioned already, we have n "cubic splines" $Y_i(x)$, each of which is used only on the interval $[x_i, x_{i+1}]$ where $i = 1, 2, \ldots n$.
- Each $Y_i(t) = a_i t^3 + b_i t^2 + c_i t + d_i$ and must satisfy (for $i = 1, \ldots, n$):

$$\begin{aligned} Y_i(0) &\equiv d_i = y_i \\ Y_i(1) &\equiv a_i + b_i + c_i + d_i = y_{i+1} \\ Y_i'(0) &\equiv D_i \quad \text{(say)} = c_i \\ Y_i'(1) &\equiv D_{i+1} \quad \text{(say)} = 3a_i + 2b_i + c_i \end{aligned}$$

- So $d_i = y_i$ and $c_i = D_i$ for $i = 1, \ldots, n$.
- We can also solve for a_i and b_i in terms of the n parameters D_i.
- Solving for a_i and b_i we find that
  $a_i = -2(y_{i+1} - y_i) + D_i + D_{i+1}$ and
  $b_i = 3(y_{i+1} - y_i) - 2D_i - D_{i+1}$.

- Now require that second derivatives match at internal points: for $i = 2, \ldots, n$ we must have $Y''_{i-1}(1) = Y''_i(0)$ or $6a_{i-1} + 2b_{i-1} = 2b_i$.
- Substituting for a_i and b_i we find that for $i = 2, \ldots, n$: $D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1})$.
- Finally, $Y'''_1(1) = Y'''_2(0)$ means that $a_1 = a_2$. Similarly $Y'''_{n-1}(1) = Y'''_n(0)$ so $a_{n-1} = a_n$.
- Substituting for $a_i = -2(y_{i+1} - y_i) + D_i + D_{i+1}$ gives $D_3 - D_1 = 2y_1 - 4y_2 + 2y_3$ and $D_{n+1} - D_{n-1} = 2y_{n-1} - 4y_n + 2y_{n+1}$.

We now have $n+1$ equations for the $n+1$ unknowns $D_1, \ldots, D_{n+1}$.

Cubic Splines — the Practical Stuff The $n+1$ equations for the $n+1$ unknowns $D_1, \dots, D_{n+1}$ can be written as a matrix equation: $MD = Y$, where M has a nice "band" structure:

$$M = \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \ddots & \ddots & \ddots \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & -1 & 0 & 1 \end{bmatrix} \tag{C.7}$$

$$D = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_{n+1} \end{bmatrix} \tag{C.8}$$

and

$$Y = \begin{bmatrix} 2(y_1 - 2y_2 + y_3) \\ 3(y_3 - y_1) \\ 3(y_4 - y_2) \\ \vdots \\ 3(y_{n+1} - y_{n-1}) \\ 2(y_{n-1} - 2y_n + y_{n+1}) \end{bmatrix} \tag{C.9}$$

For your convenience I've put the results that you need together:

for $i = 1, \ldots, n$:

- $a_i = -2(y_{i+1} - y_i) + D_i + D_{i+1}$,
- $b_i = 3(y_{i+1} - y_i) - 2D_i - D_{i+1}$,
- $c_i = D_i$,
- $d_i = y_i$.
- $Y_i(t) = a_i t^3 + b_i t^2 + c_i t + d_i$ where $t = \dfrac{x - x_i}{x_{i+1} - x_i}$ are the n cubic splines, each defined on the interval $t \in [0, 1]$ corresponding to $x \in [x_i, x_{i+1}]$.

The challenge to you is how to construct the matrix M. Once you have, it is easy to solve the linear system $MD = Y$ and calculate the coefficients a_i, b_i, c_i and d_i for $i = 1, \ldots, n$.

Evaluating Cubic Splines Once we have calculated the coefficients a_i, b_i, c_i and d_i for $i = 1, \ldots, n$ — how do we compute and plot the n cubic splines Y_i, $i = 1, \ldots, n$?

Easy! First store the a_i, b_i, c_i and d_i in four vectors of length n, a, b, c and d. Then set up a `for` loop (to show the cubics one-by-one):

```
%% Each cubic spline is plotted on the inteval 0 <=t <=1!
step=1/n;
for i=1:n
   coeff=[a(i) b(i) c(i) d(i)];
   t=linspace(x(i),x(i+1),n+1);
   p=polyval(coeff,[0:step:1]);
   pause
   plot(t,p,'r')
end
```

Now, once you have calculated a, b, c and d; how can you use code like the above to evaluate the cubic spline at **any** point in the range $x(1), \cdots, x(n)$?

Over to you!

D More Examples

Here are the two advanced examples referred to on Slide 102 — the first unfamiliar but easy to set up in **Matlab** , the second very familiar but not as easy to program.

Both are a bit tricky but both illustrate the power of **Matlab** and are worth the effort.

An Example: Random Fibonacci Sequences

- A **random** Fibonacci sequence $\{x_n\}$ is generated by choosing x_1 and x_2 and setting $x_{n+1} = x_n \pm x_{n-1}$ for $n \geq 2$.
- Here, the $\pm$ indicates that $+$ and $-$ must have equal probability of being chosen.
- In 1999 Divakar Viswanath http://en.wikipedia.org/wiki/Random_Fibonacci_sequence analyzed this "formula" and showed that for large n, $|x_n|$ is proportional to c^n, where $c = 1 \cdot 13198824$.

We can test Viswanath's result as follows:

(The following command is broken over two lines by typing the **continuation** symbol **...** at the end of the first line.)

```
clear
x=zeros(1,1000); x = [1 2]; %Pre−allocate memory for x then initialise
for n = 2:999
x(n+1) = x(n) + sign(rand−0.5)∗x(n−1);
end
semilogy(1:1000,abs(x)) % like plot except log Y scale
c = 1.13198824;
hold on
semilogy(1:1000,c.^ [1:1000],'r')% like plot except log Y scale
                                 % so get straight line
hold off
```

- The **clear** command removes all variables from the workspace.
- The **for** loop stores a random Fibonacci sequence in the array **x**; **Matlab** automatically extends **x** each time a new element **x(n+1)** is assigned.
- The **semilogy** function then plots **abs(x)** on the y-axis against n on the x-axis with logarithmic scaling for the y-axis.
- Typing **hold on** tells **Matlab** to superimpose the next picture on top of the current one.
- The second **semilogy** plot produces a line of slope **c**.
- The overall picture, shown in Figure 18, is consistent with Viswanath's theory.

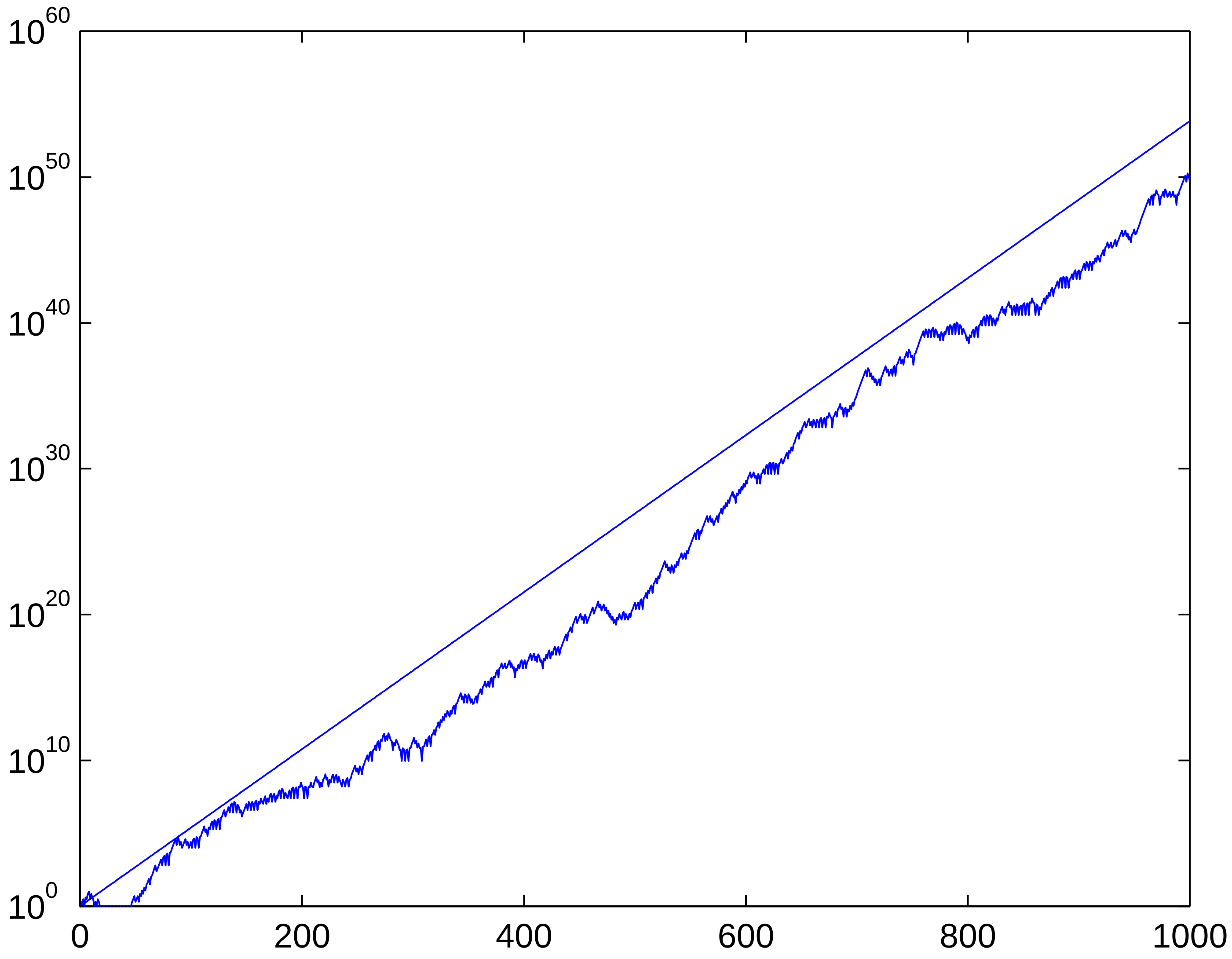


Figure 18: Growth of random Fibonacci sequence

A Second Example: The Mandlebrot Set

- The well-known and much studied Mandelbrot set can be approximated graphically in just a few lines of **Matlab** .
- It is defined as the set of points c in the complex plane for which the sequence generated by the mapping $z \longrightarrow z^2 + \mathrm{c}$, starting with $z = \mathrm{c}$, remains bounded.
- The script mandel in the file **mandel.m** produces the plot of the Mandelbrot set shown in Figure 19.
- The listing is given in Listing 24 below.
- You can load the file **mandel.m** from `http://jkcray.maths.ul.ie/ms4024/M-Files/mandel.m`.

- If your web browser is set up correctly, when you click on the link above the file will be loaded into the **Matlab** editor.
- If not, just copy from the browser and paste into the **Matlab** editor.
- A third option is to use the Save As option in your browser to save the file to a local folder, then load using the **Matlab** editor.

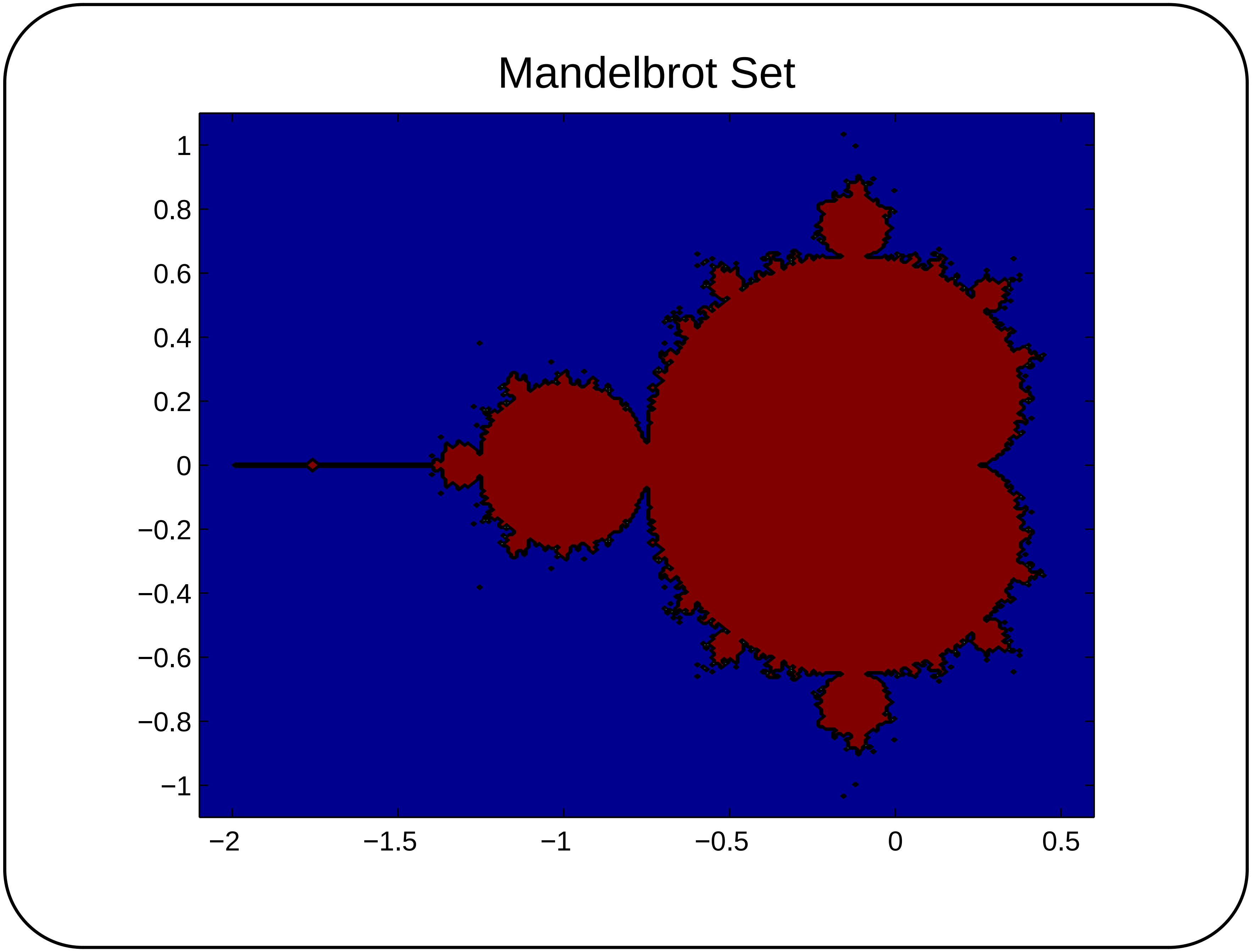


Figure 19: Mandelbrot Set

Listing 24: mandel.m

```
% MANDEL Mandelbrot set.
h = waitbar(0,'Computing...');
x = linspace(-2.1,0.6,301);
y = linspace(-1.1,1.1,301);
[X,Y] = meshgrid(x,y);
C = complex(X,Y);
Z_max = 1e6; it_max = 50;Z = C;
for k = 1:it_max
Z = Z.^2 + C;
waitbar(k/it_max)
end
close(h)
contourf(x,y,double(abs(Z)<Z_max))
title('Mandelbrot Set','FontSize',16)
```

- The script contains two calls to **linspace** of the form **linspace(a,b,n)**, which generate equally spaced vectors of n values between a and b — in this case vectors `x` & `y` of length 301.
- The **complex** function takes the two real matrices X, Y and returns the complex matrix $X + iY$.
- The **waitbar** function plots a bar showing the progress of the computation.

- The plot itself is produced by **contourf**, which plots a filled contour.
- The expression in the call to **contourf** detects points that have not exceeded the threshold **Z_max** and which are therefore assumed to lie in the Mandelbrot set.
- You can experiment with **mandel** by changing the region that is plotted, via the **linspace** calls, the number of iterations it max, and the threshold **Z_max**.