COMPUTER PROGRAMMING IN MATLAB
Covenant University Workshop

Department of Mathematics,
Covenant University, Ota, Nigeria

By

DR TIRI CHINYOKA
Department of Mathematics and Applied Mathematics
.and.
Center for Research in Computational & Applied Mechanics
Department of Mechanical Engineering

University of Cape Town

May 24, 2011

Contents

1 Introduction to MATLAB **5**

1.1 Scalar Mathematics . . . 5
1.1.1 Numbers . . . 5
1.1.2 Number Display Options . . . 5
1.1.3 Arithmetic Operations . . . 6
1.1.4 Variables . . . 6
1.1.5 Special variables . . . 6
1.1.6 Basic Mathematical Functions . . . 7
1.2 Operator Precedence . . . 7
1.3 Display . . . 9
1.4 FPRINTF . . . 9
1.4.1 Format Specifiers . . . 9
1.5 Script M-files . . . 11
1.6 Vectors . . . 13
1.6.1 Vector Creation . . . 13
1.6.2 Vector Orientation . . . 14
1.6.3 Vector Length & Size . . . 14
1.6.4 Vector Addressing . . . 15
1.7 Matrices . . . 15
1.7.1 Matrix Orientation . . . 16
1.7.2 Matrix Size . . . 16
1.7.3 Matrix Addressing . . . 16
1.7.4 Some Special Matrices: . . . 17
1.8 Array Operations . . . 18
1.8.1 Scalar-Array Mathematics . . . 18
1.8.2 Function-Array Mathematics . . . 18
1.8.3 Element-by-element Array-Array Mathematics . . . 18
1.8.4 Matrix products & inverses . . . 19
1.9 Graphing with Plot . . . 19
1.10 Changing the axes . . . 22
1.11 Grid Lines . . . 24

1.12 Titles & labels . . . 24
1.13 Line thickness & Font size . . . 25
1.14 Several plots together . . . 25
1.14.1 Holding a plot . . . 27
1.14.2 Line Type and colour: . . . 29
1.14.3 Multiple plots in one window: . . . 30
1.15 Parametric Plots & Animation . . . 32
1.15.1 2-D Parametric Plots . . . 32
1.16 3-D Parametric Curves . . . 33
1.16.1 Animation . . . 35
1.17 FOR loop . . . 36
1.18 If Statement . . . 40
1.18.1 Logical operators . . . 42
1.19 WHILE loop . . . 43
1.19.1 Execution of WHILE loop . . . 44
1.19.2 Interrupting infinite loops . . . 44
1.19.3 Examples . . . 44
1.20 Breaking from a loop . . . 46
1.20.1 The BREAK command . . . 46
1.21 Timing a program . . . 48
1.21.1 The TIC & TOC commands . . . 48
1.21.2 CLOCK, ETIME & CPUTIME . . . 49
1.22 Function M-files . . . 50
1.23 Boundary Value Problems (BVP's) . . . 52
1.23.1 Finite Difference Method (FDM) . . . 53
1.23.2 Case 1: $\mathbf{y}(\mathbf{a}) = \alpha,\ \mathbf{y}(\mathbf{b}) = \beta$ . . . 53
1.23.3 Case 2: $\mathbf{y}'(\mathbf{a}) = \alpha,\ \mathbf{y}(\mathbf{b}) = \beta$ . . . 56
1.23.4 Case 3: $\mathbf{y}(\mathbf{a}) = \alpha,\ \mathbf{y}'(\mathbf{b}) = \beta$ . . . 58

2 Fluid flow, heat and mass transfer 60
2.1 Fluid flow and heat transfer . . . 60
2.1.1 Rectangular coordinates; 1-D flow . . . 60
2.1.2 Cylindrical coordinates; axisymetric flow . . . 60
2.2 Fluid flow and mass transfer . . . 61
2.2.1 Rectangular coordinates; 1-D flow . . . 61
2.2.2 Cylindrical coordinates; axisymetric flow . . . 61
2.3 Numerical Solution . . . 61

3 Thermal combustion and environmental problems 63
3.1 CO_2 emission & O_2 depletion . . . 63

4 Epidemiology and Finance **64**
4.1 Initial Value Problems 64
4.1.1 Euler's Method 64
4.1.2 Runge-Kutta Methods 65
4.1.3 Higher order IVP's 65
4.2 Problems in Finance, Random Numbers 67
4.2.1 Rounding 67
4.2.2 Truncating 68
4.2.3 Random numbers: 69
4.2.4 Remainders: 71
4.3 Stochastic ODE's in finance 72
4.3.1 Some Stochastic ODE Models 73

Lecture 1

Introduction to MATLAB

The name MATLAB stands for **MAT**RIX **LAB**ORATORY because the system was primarily designed to make matrix computations particulary easy. The MATLAB environment is both an *interactive environment* and an *executable program*, developed in a *high level language*, which interprets (English language) user commands.

MATLAB has proven to be extraordinarily versatile in its ability to solve problems in any area that deals with complex numerical calculations: Applied Mathematics, Applied Sciences, Engineering, Finance, etc.

1.1 Scalar Mathematics

Scalar mathematics involves **simple arithmetic** operations on single valued variables. MATLAB provides support for scalar mathematics similar to that provided by a calculator.

for more information, type **help ops**

1.1.1 Numbers

MATLAB represents numbers in two formats: *fixed point* and *floating point.*

Fixed Point: Decimal form, with an optional decimal point, for example:

$$276.349 \qquad -481 \qquad 0.0053$$

Floating Point: Scientific notation, $A \times 10^B$ (MATLAB:- AeB), e.g.

2.76349×10^2 is represented as $2.76349e2$

0.00053 is represented as $5.3e-4$

In IEEE double precision standard used by MATLAB: $(-1)^{sign} \times 2^{exponent-1023} \times (1.mantissa)$ there is one bit in the sign, 52 bits in the mantissa and 11 bits in the exponent for a total of 64 bits to represent a scalar number.

This provides for a range of values extending from 10^{-308} to 10^{308}

Two MATLAB functions **realmin** and **realmax** display the *smallest* and *largest* numbers respectively, that can be represented.

1.1.2 Number Display Options

MATLAB's rules for displaying numerical results:

- **Integers:** displayed as integers as long as they contain 9 digits or less. For 10 digits or more, they are displayed in scientific notation.
- **Short fixed point:** Up to 3 digits to the left of the decimal point. Such numbers will be rounded and displayed to 4 decimal places, this is called **short format**.
- **Short floating point:** when the result has more that 3 digits to the left of the decimal point or if a nonzero entry only appears after the third decimal place, then the result is displayed in scientific notation. This is called **short e format**.

The default behavior can be changed by specifying a different numerical format:

For more information type **help format**

1.	format short	4 decimal digits
2.	format long	14 decimal digits
3.	format short e	4 decimal digits plus exponent
4.	format long e	14 decimal digits plus exponent
5.	format short g	better of 1. or 3.
6.	format long g	better of 2. or 4.
7.	format bank	2 decimal digits
7.	format +	positive, negative or zero

1.1.3 Arithmetic Operations

Operation	Algebraic form	MATLAB
Addition	$a + b$	$a + b$
Subtraction	$a - b$	$a - b$
Multiplication	$a \times b$	$a * b$
Right division	$a \div b$	a/b
Left division	$b \div a$	$a\backslash b$
Exponentiation	a^b	$a \hat{} b$

1.1.4 Variables

Variable names can be assigned to represent numerical values in MATLAB. The rules for these names are:

- **Must** start with an alphabetical letter.
- May consist only of the letters a - z (or A - Z), the digits 0 - 9 and the underscore character _
- May be as long as you would like.
- Is case sensitive.

1.1.5 Special variables

ans:	whatever the last answer was
pi:	circumference/diameter of circle
eps:	smallest amount by which two machine numbers can differ
inf or Inf:	infinity e.g. 1/0
nan or NaN:	not-a-number e.g. 0/0

Commands involving variables:

who:	lists names of defined variables
whos:	lists names and sizes of defined variables
clear var:	clears variable var
clear:	clears all variables, resets default values of special variables
clc:	clears command window but does not affect variables

Command re-use and variable redefinition:

Use up arrow key to call back previous commands, redefine these by editing their values, press enter to execute.

1.1.6 Basic Mathematical Functions

MATLAB support many mathematical functions most of which are used in the same way as you would write them mathematically:

For a more complete list type **help elfun**

abs(x)	$\|x\|$
exp(x)	e^x
log(x)	$\ln x$
log10(x)	$\log_{10} x$
sqrt(x)	$\sqrt{x}$
sin(x), cos(x), tan(x)	same as in math but default angle is radians
sind(x), cosd(x), tand(x)	default angle is degrees

Exercise 1.1 *After you have covered Lecturer 3, use vector concepts to calculate a maximum possible double precision number in MATLAB given that the eleven bits in the exponent are such that* $10000000000 \leq exponent \leq 01111111111$.*

*Hint: A **bit** is a binary digit, taking value of 0 or 1.*

1.2 Operator Precedence

Since several arithmetic operations can be combined in one expression, there are rules under which these operations are performed. The operator precedence is:

1. Parentheses ( )
   Innermost first, then left to right.
2. Exponentiation ^
   left to right.
3. Multiplication * and Division / or \
   with equal precedence, left to right.

*a value of zero in the exponent leads to a subnormal number and all bits in the exponent being 1 represents inf or nan

4. Addition + and Subtraction –

with equal precedence, left to right.

Example 1.1 *Each of the following is either an algebraic or MATLAB expression. If it is algebraic, convert it to the correct MATLAB syntax. Similarly, if you encounter a MATLAB expression convert it to the correct algebraic representation.*

$$\frac{8+2^{2\times 3}}{17-12+3}, \qquad 2\,\hat{}\,3\,\hat{}\,2, \qquad 2^{3^2}, \qquad -4\,\hat{}\,2, \qquad 27^{\frac{1}{3}}, \qquad \frac{1-\frac{2}{3+2}}{1+\frac{2}{3-2}}$$

Solutions:

$$\begin{aligned}
\frac{8+2^{2\times 3}}{17-12+3} &\equiv \text{(8 + 2\^{}(2 * 3)) / (17 - 12 + 3)} \\
&= \text{(8 + 2\^{}6) / (17 - 12 + 3)} \\
&= \text{(8 + 64) / (17 - 12 + 3)} \\
&= \text{72 / (17 - 12 + 3)} \\
&= 72 \,/\, (5+3) \\
&= 72/8 \\
&= 9.
\end{aligned}$$

$$\begin{aligned}
2\,\hat{}\,3\,\hat{}\,2 &\equiv (2^3)^2 \\
&= 8^2 \\
&= 64.
\end{aligned}$$

$$\begin{aligned}
2^{3^2} &\equiv 2\,\hat{}\,(3\,\hat{}\,2) \\
&= 2^9 \\
&= 512.
\end{aligned}$$

$$\begin{aligned}
-4\,\hat{}\,2 &\equiv -\,(4^2) \\
&= -16.
\end{aligned}$$

$$\begin{aligned}
27^{\frac{1}{3}} &\equiv \text{27\^{}(1 / 3)} \\
&= \sqrt[3]{27} \\
&= 3.
\end{aligned}$$

$$\begin{aligned}
\frac{1-\frac{2}{3+2}}{1+\frac{2}{3-2}} &\equiv \text{(1 - 2 / (3 + 2)) / (1 + 2 / (3 - 2))} \\
&= \text{(1 - 2 / 5) / (1 + 2 / (3 - 2))} \\
&= \text{0.6 / (1 + 2 / (3 - 2))} \\
&= \text{0.6 / (1 + 2 / 1)} \\
&= \text{0.6 / 3} \\
&= 0.2
\end{aligned}$$

1.3 Display

There are three ways to display *numerical values* and *text* in MATLAB:

1. By entering the variable name without a semi-colon, and the corresponding text inside single quotes: *'text'*.
2. By use of the command **disp**.
3. By use of the command **fprintf**.

Example 1.2 *The syntax for* ***disp*** *is* disp(string)*, where* string *is a* predefined variable*, a* number *or* 'text'.

```
>> disp('Blood temp is'), disp(BloodTemp), disp('Degrees Kelvin')
Blood temp is
   310

Degrees Kelvin
```

Example 1.3 *It is possible to use the* ***disp*** *command to embed numerical values inside text. In this case the syntax is* disp([string,string,...,string])*, where as before if* string *is plain text, then you use enclose it in single quotes* 'text'*. If however* string *is a predefined variable, you will need to use the* ***num2str*** *(number to string) command.*

```
>> disp(['Blood temp is ', num2str(BloodTemp),' Degrees Kelvin'])
Blood temp is 310 Degrees Kelvin
```

Notice how we have manually put spaces inside the text string, one after is *and another before* Degrees*.*

1.4 FPRINTF

As was briefly shown in section 1.1, the **fprintf** command allows for the displaying/embedding of variable values within text.

The **fprintf** command can also be used to control the **number format** to be used in the display including the specification to skip to a new line. The general form of this command is:

fprintf('format string', list of values)

The *format string* contains the text to be displayed together with *format specifiers* that will control how the variable values listed are to be embedded and displayed in the string.

The *list of values* should be separated by commas in case more than one value is specified.

1.4.1 Format Specifiers

Using *specific non-negative integers* W & D and the **alphabetical letters** d, e, f, g, s and n: the format specifiers include:

\n	Go to next line. *If you leave out \n, notice that the MATLAB prompt will be at the end of the output and not on the next line!*
%Wd	for integers with a width of W characters.
%W.De	for scientific notation with a width of W characters (including the decimal point, a possible minus sign & five for the exponent) and D digits after the decimal point.
%W.Df	for fixed point numbers with a width of W characters (including the decimal point & a possible minus sign) and D digits after the decimal point.
%W.Dg	better of W.De & W.Df – MATLAB decides format.
%Ws	for strings with a width of W characters.

Example 1.4

```
>> fprintf('Without a %2s at end of string, prompt appears here','\n')
>> fprintf('Use %2s at end of this string & prompt moves to new line\n','\n')
>> fprintf('%4s has width 4, larger W %15s leads to spaces\n','tiri','tiri')
```

Example 1.5

Display the 10 digit integer 1234567897 exactly

```
>> TenDigInt = 1234567897;
>> fprintf('The required number is %10d exactly \n', TenDigInt)
```

Example 1.6

Write 2.76958126e-187, using 3dp and scientific notation.

```
>> a = 2.76958126e-187;
>> fprintf('Required number is %10.3e to 3dp & scientific notation \n', a)
```

Notice that the exponent always takes up a width of 5:

```
>> b = 0.000012548; c = 1.26485e32;
>> fprintf('Exponent always has width 5: try %10.4e and %10.3e \n', b , c)
```

Spaces are filled in from the left if necessary: try %12.1e to display 27.69187

```
>> drtc = 27.69187;
>> fprintf('Displayed number is %12.1e with spaces to the left \n', drtc)
```

Example 1.7

Represent the number 2794.69187 exactly:

```
>> x = 2794.69187;
>> fprintf('Required number is %10.5f in fixed point notation \n', x)
```

Spaces are again filled in from the left if necessary: try %12.3f to display 2794.69187

```
>> fprintf('Required number is %12.3f in fixed point notation \n', x)
```

Set D to 0 if you do not want any decimal places: try %7.0 to display 2794691.87

```
>> y = 2794691.87;
>> fprintf('Required number is %7.0f with no decimal places \n', y)
```

The W.D specifiers are optional. If they are left out, default values are used.

- %d for integers
- %e for floating point numbers (scientific notation)
- %f for fixed point numbers
- %g for any kind of numbers – MATLAB decides format.
- %s for strings

Example 1.8

One line fprintf command:

```
% fpf1.m
% One line fprintf command
clc
clear

r = 5;
A = pi*r^2;
fprintf('radius of circle is %g metres, its area is %6.4f m^2 \n', r, A)
```

Example 1.9

Multi-line fprintf command:

```
% fpf2.m
% Multi-line fprintf command
clc
clear

r = 125.32154;
A = pi*r^2;
fprintf('radius of circle is %e metres. \narea is thus %8.2f m^2. \n', r, A)
```

1.5 Script M-files

Suppose one wants to repeat a calculation many times with different inputs. it would be inconvenient to have to re-type common statements in the Command Window over and over again. In such cases, it is convenient to use m-files. Here, we group the collection of statements needed to do a particular calculation in one file, *MATLAB program commonly known as an m-file since any such file always has the* ***.m*** *suffix.*

MATLAB m-file names follow the same rules as variable names. A MATLAB script m-file can be run by:

- Simply typing its name on the Command Window and then pressing Enter or Return. You have to make sure, however, that the *Current Directory* on the Command Window is the same as that in which your m-file has been saved.
- Clicking on the bold green arrow in the *Editor* window (MATLAB 2009).
- Pressing F5 in the *Editor* window.

Example 1.10

Quadratic root finding

The quadratic equation

$$Ax^2 + Bx + C = 0,$$

has two roots

$$r_1,\ r_2 = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

If we define the intermediate values:

$$x = \frac{-B}{2A} \qquad \text{and} \qquad y = \frac{\sqrt{B^2 - 4AC}}{2A},$$

then we can write:

$$r_1 = x + y \qquad \text{and} \qquad r_2 = x - y,$$

and proceed to use a MATLAB script m-file to compute these roots.

```
% Quadratic root finding script.
% A*x^2 + B*x + C = 0

% clean out command window and clear all variables from memory
clc
clear

% prompt for coefficients A, B & C
A = input('Enter leading coefficient A: ');
B = input('Enter middle coefficient B: ');
C = input('Enter constant C: ');

disp(' ')  % This forces a blank space here

% Compute intermediate values x & y
x = -B/(2*A);
y = sqrt(B^2-4*A*C)/2/A;

% Compute the two quadratic roots
r1 = x + y;
r2 = x - y;

% Display the roots on the command window
disp(['The first  quadratic root r_1 = ', num2str(r1)])
disp(['The second quadratic root r_2 = ', num2str(r2)])
```

It is always good practice to have the first two uncommented lines in **every script m-file** as

```
clc
clear
```

The *clc* command reduces clutter on the command window so that errors are easier to detect each time you run a script m-file.

Since predefined variables interfere with script m-files, it is advisable to use the *clear* command so that your script does not take on any unneeded values from the command window.

Notice how we have also generously used comments (preceded by %) to explain what our program does for future reference and for others to easily follow. MATLAB ignores anything to the right of %

We have seen that a **scalar** is a quantity that is represented by a single number. On the other hand, an **array** is represented by more than one number. Each number is called an **element** of the array. Rather than performing the same operation on one number at a time, array operations allow operating on multiple numbers at once.

One-Dimensional Arrays: These are also called **vectors** and occur in two forms (i) a single **row** of numbers called a **row vector** or (ii) a single **column** of numbers called a **column vector**.

Two-Dimensional Arrays: Also called **matrices** (or **matrix** if only one such array), these consist of multiple rows and columns of numbers arranged in a two-dimensional table.

1.6 Vectors

We look at vector creation, orientation, length and addressing.

1.6.1 Vector Creation

There are four ways to create a vector. We describe each one in items (1) through (4) below.

1. $\mathbf{x} = [\mathbf{x_1}\ \mathbf{x_2}\ \mathbf{x_3}\ \cdots\ \mathbf{x_n}]$ or $\mathbf{x} = [\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \cdots, \mathbf{x_n}]$

   is a **row vector** of n elements created by an *explicit list* of numbers: starting with a left square bracket, entering the values separated by spaces, commas (or mixture of both) and closing the vector with a right square bracket.

   ```
   >> x  = [-0.13 7 56 9.82 67.23 56.521 -95 71]
   >> x2 = [-0.13,7,56,9.82,67.23,56.521,-95,71]
   >> x3 = [-0.13 7,56 9.82,67.23 56.521,-95 71]
   ```

   If instead of spaces and/or commas we use **semi-colons** to separate the elements, then the result is a **column vector**.

   ```
   >> y = [-0.13 ; 7 ; 56 ; 9.82 ; 67.23 ; 56.521 ; -95 ; 71]
   ```

   *Note however that you cannot mix semicolons with either commas or spaces when creating a **vector**. Doing so will result in either a matrix (each of whose rows has the same number of elements) or error messages!*

   ```
   >> OopsMatrix = [-0.13  7 ; 56  9.82 ; 67.23  56.521 ; -95  71]
   >> OopsMatrix2 = [-0.13  7  56  9.82 ; 67.23  56.521  -95  71]
   >> OopsError = [-0.13 ; 7  56 ; 9.82 ; 67.23 ; 56.521  -95  71]
   >> OopsError2 = [-0.13 ; 7  56 , 9.82 ; 67.23 ; 56.521 , -95 , 71]
   ```

All the semicolons included in an array creation have to be equally spaced (since they separate adjacent rows) otherwise you will get error messages.

2. **xrow = start : end**

   Create a **row** vector **xrow** starting with **start**, counting by **one**, ending at or before **end**.

```
>> v1 = -2 : 8
>> v2 = 13 : 22.7
```

3. **xrow = start : increment : end**

   Create a **row** vector **xrow** starting with **start**, counting by **increment**, ending at or before **end**.

```
>> w11 = -5 : 0.5 : 0
>> MyVec = 1 : 21 : 240
>> MyVec2 = 1 : 21 : 233
>> NegVec = 23 : -6 : -39
```

4. **xrow = linspace(start , end , N)**

   Create a **row** vector **xrow** starting with **start**, ending at **end**, having **N** elements.

   In this case the increment is not specified but can be calculated from

$$\textbf{increment} \;=\; \frac{\textbf{end} \;-\; \textbf{start}}{N-1}.$$

```
>> w11_b = linspace(-5 , 0 , 11)
```

1.6.2 Vector Orientation

The **transpose** operator ' is used to convert a row vector into a column vector (and vice-versa).

```
>> x'
>> MyVec'
>> y'
```

1.6.3 Vector Length & Size

The command **length(x)** gives the length (number of elements) of the vector **x**.

```
>> length(w11_b)
```

Recall that a row vector has *exactly one row* and at least one column and similarly a column vector has at least one row and *exactly one column*.

The command **size(x)** returns two numbers in the form:

```
r     c
```

which specify the exact dimensions of the vector **x** in terms of number of rows **r** and columns **c**. Since **x** is a vector, one of **r** or **c** has to take the value 1.

```
>> size(w11_b)
>> size(w11_b')
```

1.6.4 Vector Addressing

The command **x(k)** gives the k^{th} element of the vector **x**.

```
>> NegVec(8)
```

To address a block of elements of the vector **x**, specify the block as another vector:

Example 1.11
x(Start : increment : End) *where* ***Start, increment, End*** *are all integers.*

```
>> NegVec(2 : 9)
```

```
>> NegVec(1 : 2 : 11)
```

```
>> NegVec(11 : -2 : 1)
```

Example 1.12
x(explicit vector) *where* ***explicit vector*** *is a vector created via the explicit method described earlier.*

```
>> NegVec([1 5, 6, 7 9 11])
```

1.7 Matrices

A matrix can be explicitly created as follows:

- Start with the left bracket [
- Elements in a row are separated by spaces or commas
- A semicolon or **Enter** is used to separate rows
- End with a right bracket]

```
>> ABC123 = [1 3 5 7 9 11
>> 2 4 6 8 10 12
>> 3,6,9,12,15,18
>> 4 8,12 16 20,24
>> 5,10,15 20 25,30]
```

The output will look like the following:

```
ABC123 =

     1     3     5     7     9    11
     2     4     6     8    10    12
     3     6     9    12    15    18
     4     8    12    16    20    24
     5    10    15    20    25    30
```

We notice that the rows of the matrix are separated by pressing enter. The semicolon (**;**) can also be used to separate the rows of a matrix.

```
>> M = [1 3 5 7 9 11 ; 2 4 6 8 10 12 ; 3,6,9,12,15,18
>> 4 8,12 16 20,24 ; 5,10,15 20 25,30 ; 6,12 18 24 30 36]
```

1.7.1 Matrix Orientation

The **transpose** operator ' is used to convert rows of a matrix into columns (and vice-versa).

```
>> ABC123'
>> M'
```

1.7.2 Matrix Size

The command **size(A)** gives the size (number of rows & columns respectively) of the matrix **A**.

[r , c] = size(A) returns two scalars **r** and **c** containing the number of rows and columns in **A**, respectively.

r = size(A , 1) returns the number of rows in **A** in the variable **r**.

c = size(A , 2) returns the number of columns in **A** in the variable **c**.

```
>> size(ABC123)
>> size(ABC123')
>> [MM , NN] = size(ABC123)
>> M2 = size(ABC123 , 1)
>> N2 = size(ABC123 , 2)
```

1.7.3 Matrix Addressing

The command **A(i , j)** gives the $(\mathrm{i}\,,\mathrm{j})^{th}$ element of the matrix **A**.

To address a block of elements of the matrix **A** (i.e. a submatrix of **A**) use

A(vector1 , vector2)

where **vector1** is a vector containing the positions of the required rows of matrix **A** and similarly **vector2** contains the positions of the required columns of **A**.

Example 1.13

$$\boldsymbol{A(r_1 \,:\, increment_1 \,:\, r_2 \;,\; c_1 \,:\, increment_2 \,:\, c_2)}$$

where $\boldsymbol{r_k}$, $\boldsymbol{increment_k}$ *are all integers.*

In particular, the command

$$\boldsymbol{A(:\;,\; c_1 \,:\, increment_2 \,:\, c_2)}$$

picks out the prescribed column entries among ***all rows*** *and similarly*

$$\boldsymbol{A(r_1 \,:\, increment_1 \,:\, r_2 \;,\; :)}$$

picks out the prescribed row entries among ***all columns***

```
>> ABC123(4,5)
>> ABC123(1 : 2 : 5 , 2 : 5)
>> ABC123(2 : 5 , 1 : 2 : 6)
>> ABC123(: , 3 : 6)
>> ABC123(1 : 2 , :)
```

1.7.4 Some Special Matrices:

The command **zeros(m,n)** creates an $m \times n$ matrix of **zeros** and similarly, the command **ones(m,n)** creates an $m \times n$ matrix of **ones**. If either **m** or **n** is 1, then the output is a actually vector.

```
>> B0 = zeros(5 , 12)
>> B1 = ones(11 , 7)
>> B2 = ones(size(ABC123))
```

The command **eye(n)** gives an $n \times n$ **identity matrix**.

```
>> I8 = eye(8)
```

If x is a vector, then the command **diag(x)** gives a **diagonal matrix** whose diagonal elements are the elements of vector x.

```
>> diag(NegVec)
```

or give it a name:

```
>> Dnv = diag(NegVec)
```

On the other hand if A is a matrix, **diag(A)** returns a vector whose elements are the diagonal entries of A.

```
>> diag(M)
```

Thus if you want to create a diagonal matrix whose diagonal elements are the same as the diagonal entries of A, you type:

```
>> diag(diag(M))
```

or give it a name:

```
>> DM = diag(diag(M))
```

The following MATLAB commands are used to create **triangular matrices:**

tril(A)	lower triangular part of A
tril(A,k)	lower triangular part of A below & including k^{th} diagonal
triu(A)	upper triangular part of A
triu(A,k)	upper triangular part of A above & including k^{th} diagonal

```
>> tril(M)     >> tril(M,0)     >> tril(M,3)     >> tril(M,-2)
>> triu(M)     >> triu(M,0)     >> triu(M,1)     >> triu(M,-1)
```

To create other special diagonal matrices (matrices with several diagonals of zeros) e.g. **tri-diagonal matrices:**, use the command **spdiags**:

spdiags([column_1 column_2 ... column_M] , Diagonals , N , N)

N is the required number of rows and columns of the matrix,

Diagonals is a vector of length **M** containing the positions of the nonzero diagonals,

column_1 through **column_M** are respectively the column vectors that will be "fitted" into the diagonals whose positions appear in the vector **Diagonals**. Ideally **M**<**N**.

Example 1.14

```
>> c1 = 1 : 7;        c1 = c1'
>> c2 = -7 : -1;      c2 = c2'
>> c3 = 8 : 14;       c3 = c3'
>> MyTriDiag = spdiags([c1 c2 c3], -1 : 1 , 7 , 7)
```

You should notice that only the nonzero diagonals' elements and their corresponding addresses are displayed. To display the full matrix, use the command **full**:

```
>> full(MyTriDiag)
>> full(spdiags([c1 c2 c3], -1 : 1 , 7 , 7))
```

1.8 Array Operations

1.8.1 Scalar-Array Mathematics

Addition, subtraction, multiplication and division of an array by a scalar simply applies the operation to all the elements in the array. Output is thus the same dimensions as the original array.

```
>> x6 = 0 : 15 : 90          >> y6 = linspace(-9 , 40 , length(x6))
>> x6 / 3                    >> y6 * 10
>> ABC123 + 20               >> ABC123 / 2 - 3
```

1.8.2 Function-Array Mathematics

Applying the Basic Mathematical Functions to an array also simply applies the function to all the elements in the array. Output is thus the same dimensions as the original array.

```
>> sind(x6)
>> exp(y6/10)
>> sqrt(ABC123)
```

1.8.3 Element-by-element Array-Array Mathematics

When two arrays have the same dimensions, some operator notation becomes unconventional:

Operation	Algebraic form	MATLAB
Addition	$a + b$	$a + b$
Subtraction	$a - b$	$a - b$
Multiplication	$a \times b$	$a. * b$
Division	$a \div b$	$a./b$
Exponentiation	a^b	$a \ .\hat{} \ b$

```
>> y6 .* x6
>> x6 ./ y6
>> ABC123 .^ 2
>> M .* M                >> M .^ 2
```

The operator precedence is the same as that for scalar mathematics.

1.8.4 Matrix products & inverses

Given an $m \times n$ matrix **A** and Given an $n \times k$ matrix **B** then the product **A*B** gives the usual linear-algebraic $m \times k$ matrix.

```
>> x6 * x6'
>> ABC123 * M
>> M * M                    >> M ^ 2
```

If **A** is a non-singular[†] square matrix, then its inverse can be calculated from the command **inv(A)** or from the exponent $\mathbf{A}^{-1}$ i.e. **A**^(-1). Note that this is not the same as **A**.^(-1)

Applications: Linear systems; Solve $\mathbf{A\ x = b}$

where **A** is non-singular say $n \times n$ and **b** is $n \times 1$. Solution:

$\mathbf{x = inv(A)\ *\ b}$ or $\mathbf{x = A^{-1}\ *\ b}$ or $\mathbf{x = A \setminus b}$

```
>> det(MyTriDiag)

>> s1 = MyTriDiag \ x6'

>> s2 = inv(MyTriDiag) * x6'

>> s3 = MyTriDiag^(-1) * x6'

>> det(M)
```

1.9 Graphing with Plot

For more information type >>**help plot**. The **plot** command works on vectors of numerical data.

The syntax is **plot(x,y)** where **x** and **y** are vectors of the same **length**.

Example 1.15 *Plot the graph of:*

$$y = \sin x, \qquad 0 \le x \le 30 \qquad \text{use} \quad \Delta x = 0.01$$

```
>> x = 0 : 0.01 : 30;
>> plot(x , sin(x))
```

Notice that the commands:

```
>> y = sin(x);
>> plot(x , y)
```

will work the same way.

[†]If a matrix is non-singular, then it has a **nonzero determinant**. MATLAB has the command **det** to calculate the determinant, type **det(A)** to get the determinant of matrix **A**.

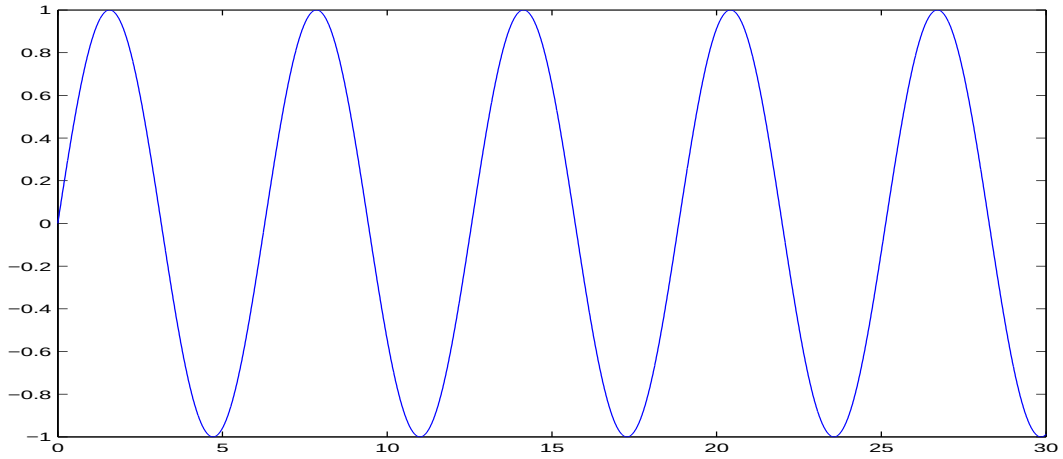


Figure 1.1: Graph of $y = \sin x$.

Example 1.16 *Plot the graph of:*

$$y = e^{-x/20}(1 + \sin x), \qquad 0 \le x \le 100 \qquad \text{use} \quad 1000 \text{ points.}$$

```
>> x = linspace(0 , 100, 1000);
>> y = exp(-x/20) .* (1 + sin(x));
>> plot(x , y)
```

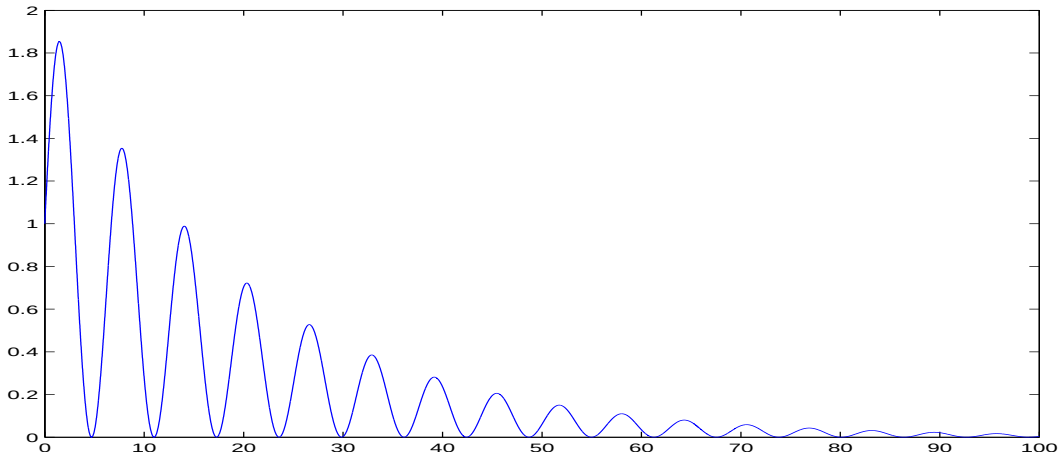


Figure 1.2: Graph of $y = e^{-x/20}(1 + \sin x)$.

The command **plot(x,y)** considers the vectors **x** and **y** as lists of the x and y coordinates of successive points on the graph and joins the points with line segments. It is possible to view these coordinate points:

Example 1.17

```
>> w = 1 : 0.5 : 10;
>> plot(w , log(w), '.')
```

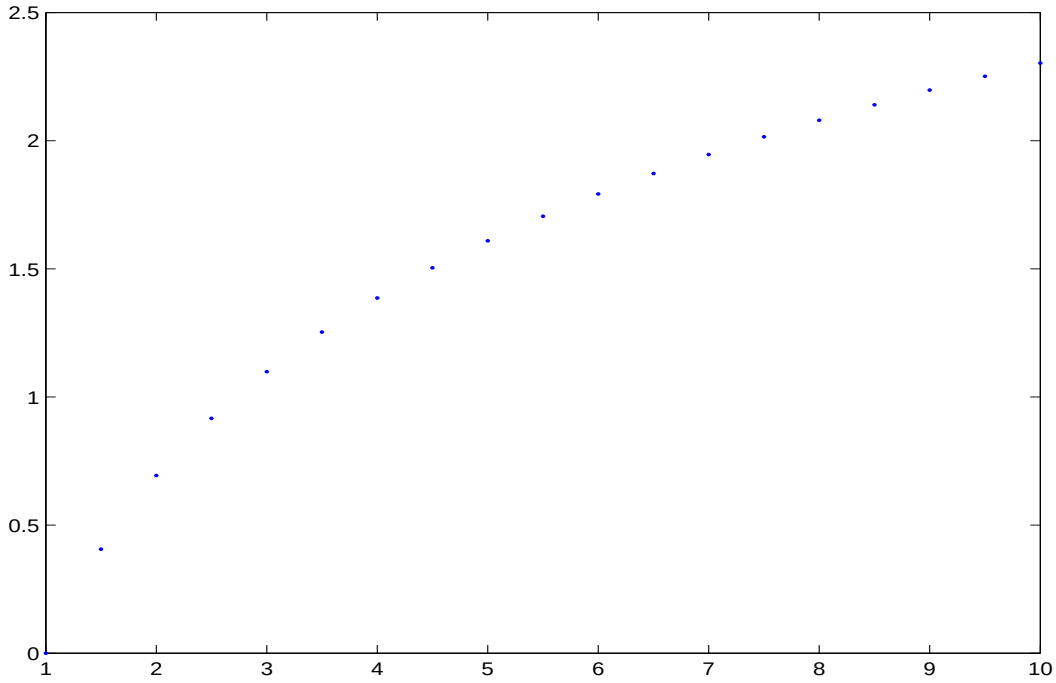


Figure 1.3: The dots $\ln w$ are plotted at corresponding values of w.

We need to use enough points to ensure that the resulting graph drawn by **"connecting the dots"** looks smooth.

Consider the graph of $\cos z$ in the interval $z \in [0, 10]$.

Example 1.18

```
>> z = linspace(0 , 10 , 10);
>> plot(z , cos(z))
```

We notice that the few plot points used here makes the curve far from smooth. How about increasing the number of plot points?

Example 1.19

```
>> z = linspace(0 , 10 , 21);
>> plot(z , cos(z))
```

The curve has definitely improved as an approximation to $\cos z$ but is still not smooth enough.

Example 1.20

```
>> z = linspace(0 , 10 , 1000);
>> plot(z , cos(z))
```

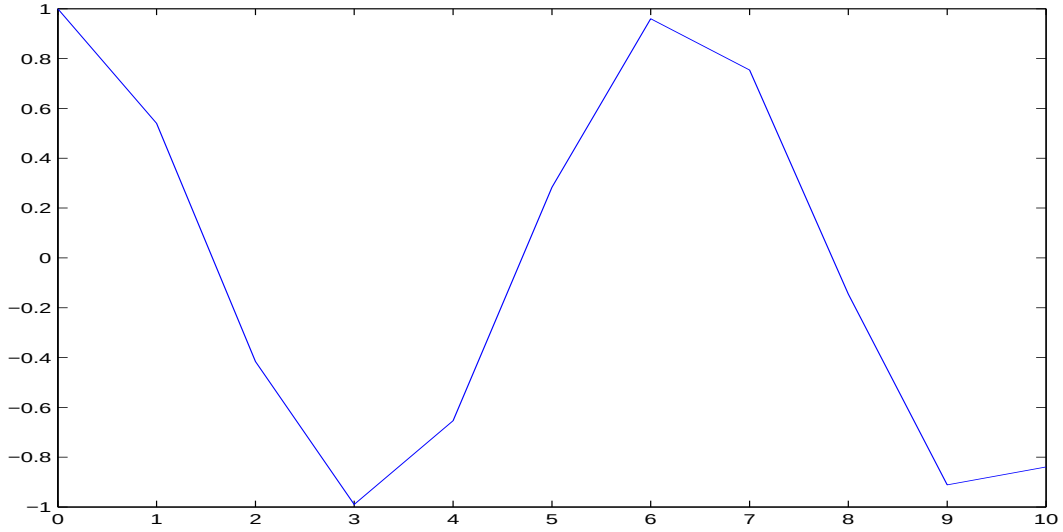


Figure 1.4: $\cos z$ plotted using only 10 points.

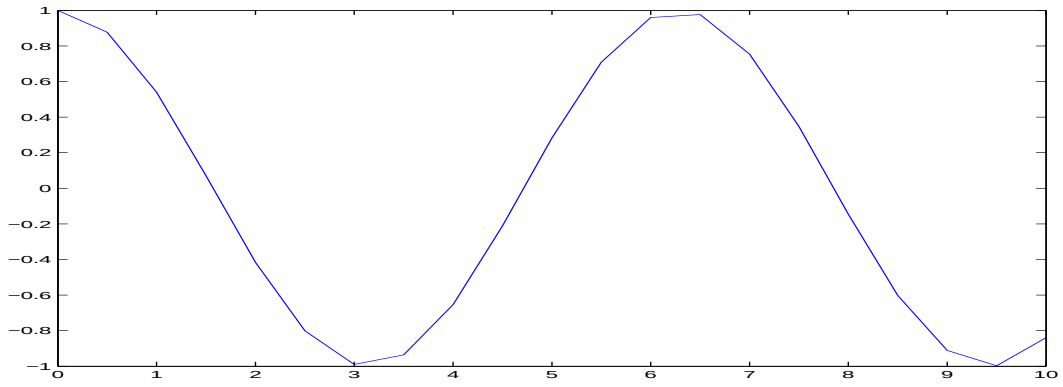


Figure 1.5: $\cos z$ plotted using 21 points.

1.10 Changing the axes

>> **help axes**

Example 1.21 *Plot a graph with an asymptote such as* $y = 1/x$ *in* $-3 \leq x \leq 3$. *Write a small m-file* ***asympt.m*** *as follows:*

```
% asympt1.m
clc
clear

x = -3 : 0.02 : 3;
y = 1 ./ x;

plot(x , y)
```

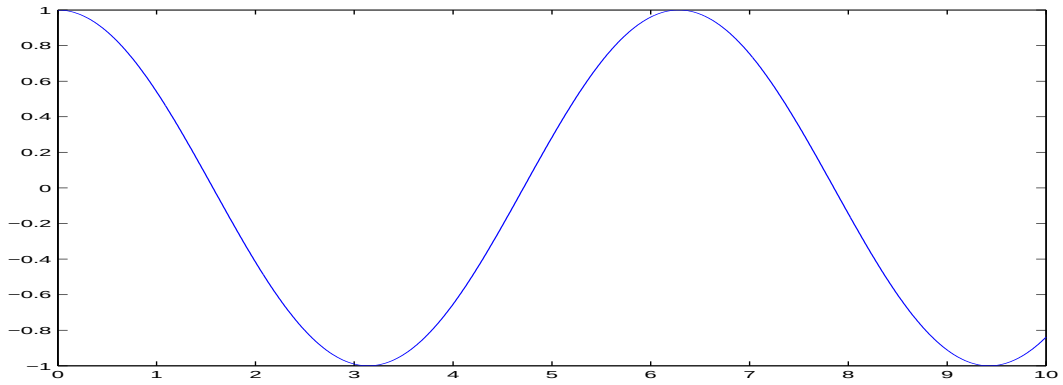
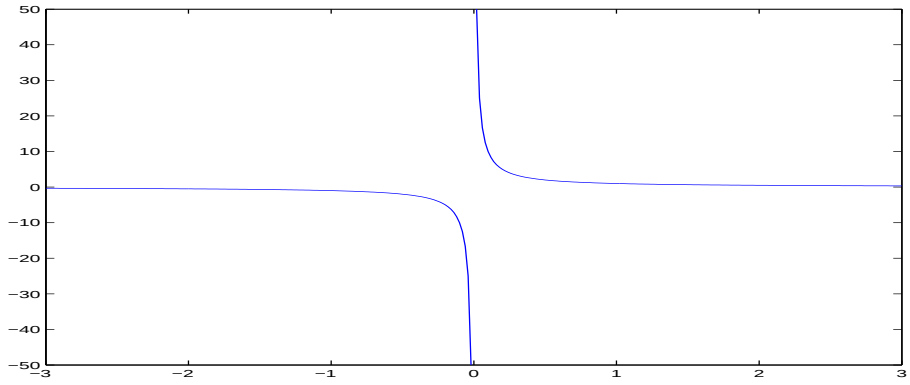


Figure 1.6: $\cos z$ plotted using 1000 points.



Example 1.22 *Suppose you want to change the y axis to run from* -3 *to 3 instead. Modify* ***asympt.m*** *so that below the plot command you have the line:* *axis([-3 , 3 , -3 , 3])*

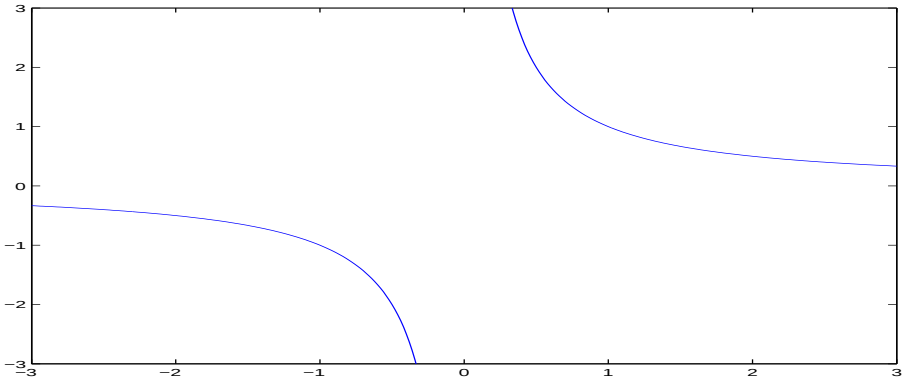
```
% asympt2.m
clc
clear

x = -3 : 0.02 : 3;
y = 1 ./ x;

plot(x , y)
axis([-3 , 3 , -3 , 3])
```

In general you can change the axes with the command

```
axis([xmin, xmax, ymin, ymax])
```

which specifies the required ranges in the x and y directions. The square brackets are needed because the argument of **axis** must be a vector. Since it is a vector, you could omit the commas;

```
axis([xmin xmax ymin ymax])
```

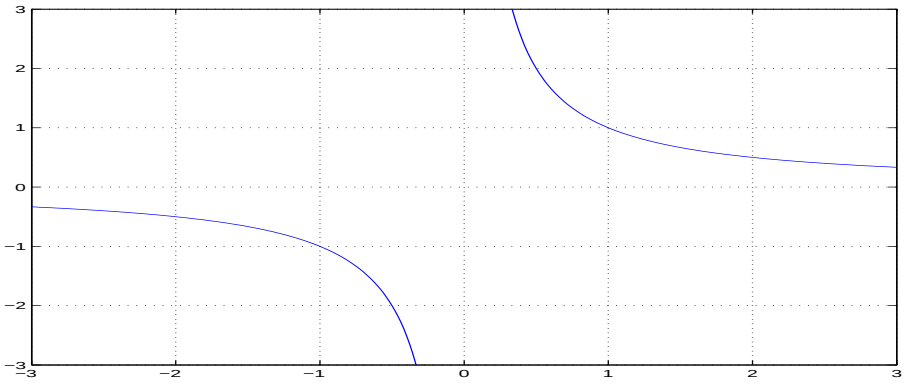


1.11 Grid Lines

You can turn on the grid lines using the command **grid**:

Example 1.23

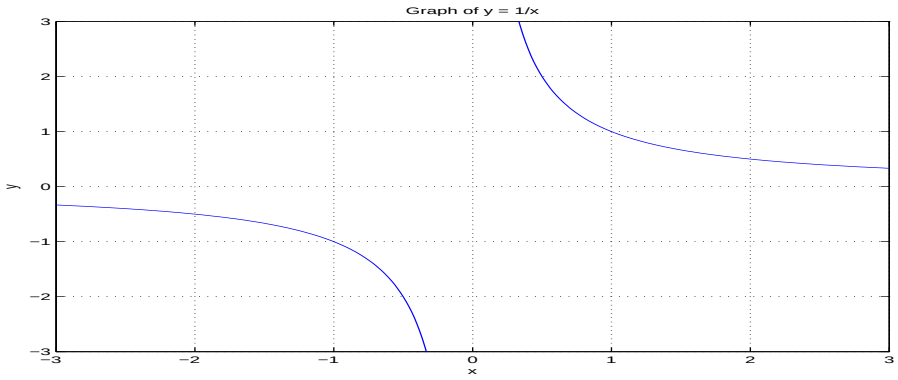
```
% asympt3.m
clc
clear

x = -3 : 0.02 : 3;
y = 1 ./ x;

plot(x , y) , grid
axis([-3 , 3 , -3 , 3])
```



1.12 Titles & labels

Example 1.24 *Modify* ***asympt.m*** *to add a title and axis labels:*

```
% asympt4.m
clc
clear
```




```
x = -3 : 0.02 : 3;
y = 1 ./ x;

plot(x , y) , grid
axis([-3 , 3 , -3 , 3])

title('Graph of y = 1/x')
xlabel('x')
ylabel('y')
```

1.13 Line thickness & Font size

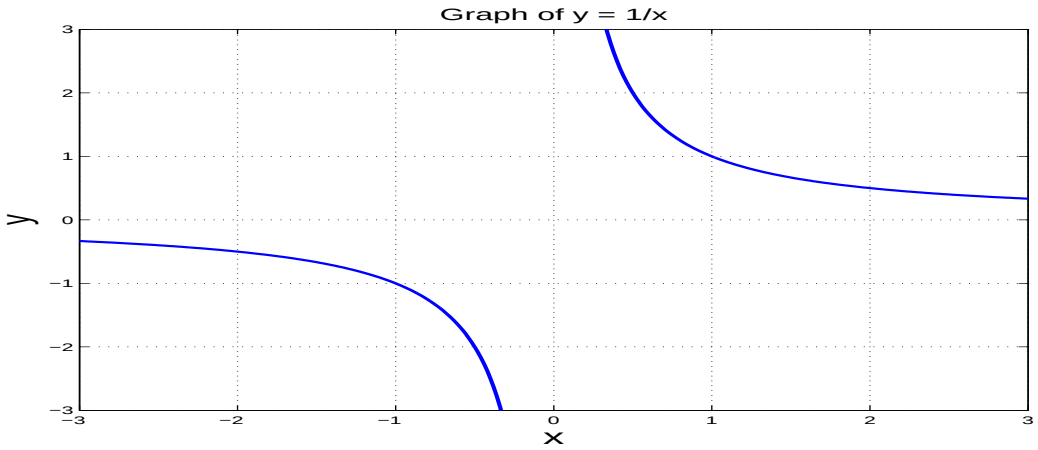
For improved visibility, you can modify the thicknesses of the lines appearing in your graphs using the **LineWidth** command. Similarly, the **FontSize** command can be used to increase the size of the font appearing in the labels.

Example 1.25 *Modify **asympt.m** to increase the line thicknesses and font sizes:*

```
% asympt.m
clc
clear

x = -3 : 0.02 : 3;
y = 1 ./ x;

plot(x , y , 'LineWidth' , 2), grid
axis([-3 , 3 , -3 , 3])

title('Graph of y = 1/x' , 'FontSize' , 15)
xlabel('x' , 'FontSize' , 20)
ylabel('y' , 'FontSize' , 20)
```

1.14 Several plots together

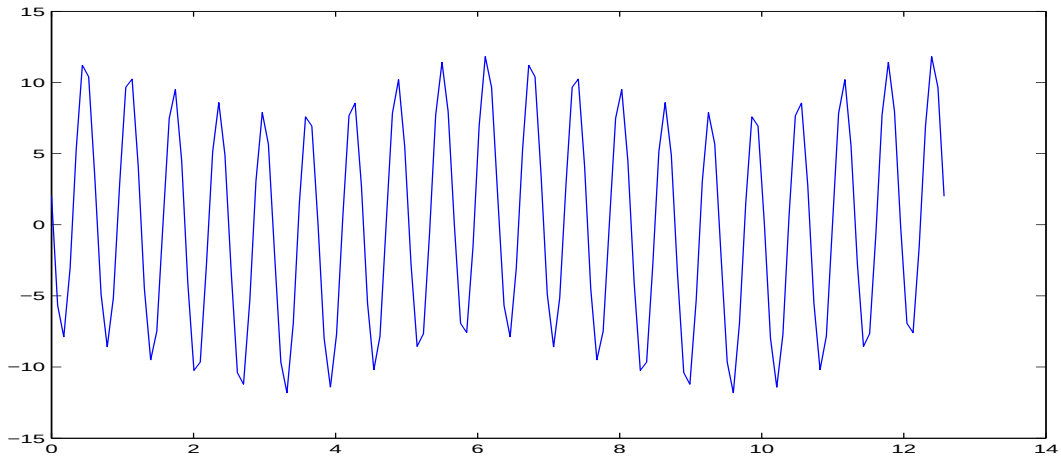
Suppose we want to plot the graph of $y = 2\sin x + cos(10x)$ and its derivative $dydx = 2\cos x - 10\sin(10x)$ on the same plot.



Example 1.26 *If you do this:*

```
% multplt1.m
clc
clear

x = linspace(0 , 4*pi , 200);
y = 2*sin(x) + cos(10*x);
dydx = 2*cos(x) - 10*sin(10*x);

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

The first graph will be lost when the second is plotted:

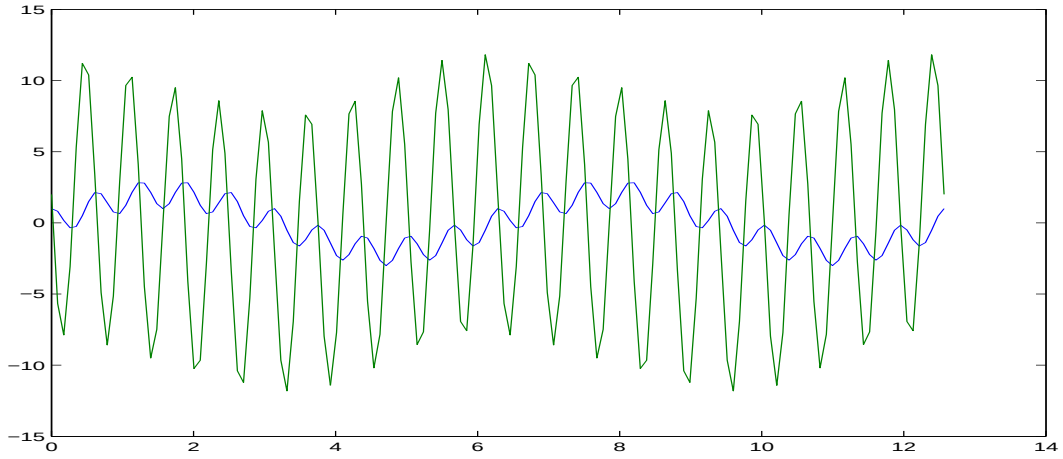


Example 1.27

Rather do:

```
% multplt2.m
clc
```

```
clear

x = linspace(0 , 4*pi , 200);
y = 2*sin(x) + cos(10*x);
dydx = 2*cos(x) - 10*sin(10*x);

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



Example 1.28

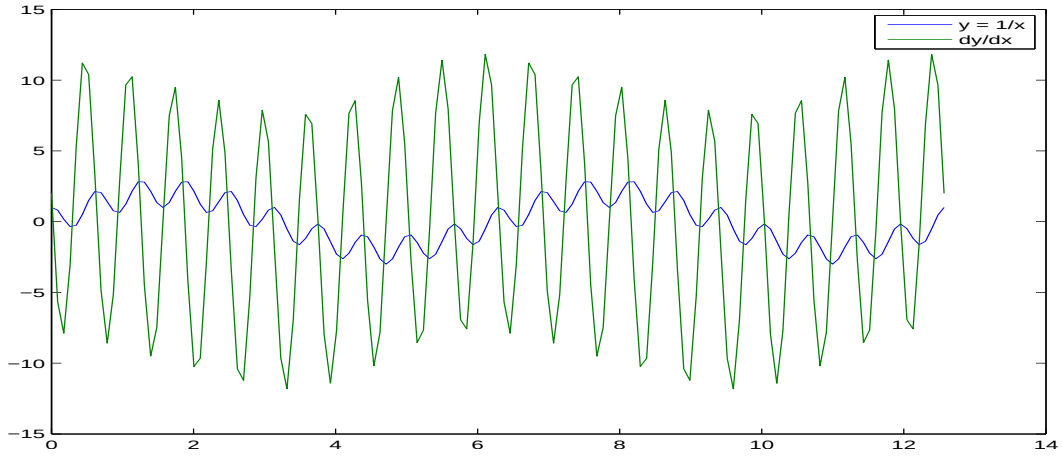
In this case, you can use ***legend*** *to label (and hence identify) the curves:*

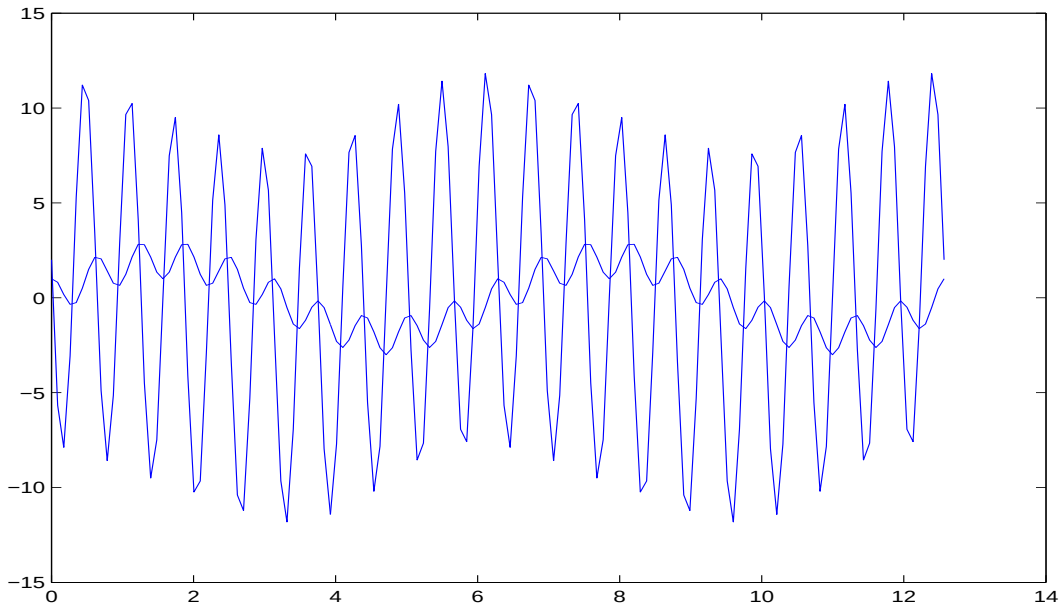
```
% multplt3.m
clc
clear

x = linspace(0 , 4*pi , 200);
y = 2*sin(x) + cos(10*x);
dydx = 2*cos(x) - 10*sin(10*x);

plot(x , y , x , dydx)
legend('y = 1/x' , 'dy/dx')
```

1.14.1 Holding a plot

Example 1.29 *Alternatively, you could first plot one curve, then tell MATLAB to hold it, then plot the second curve:*

```
% multplt4.m
clc
clear

x = linspace(0 , 4*pi , 200);
y = 2*sin(x) + cos(10*x);
dydx = 2*cos(x) - 10*sin(10*x);
```



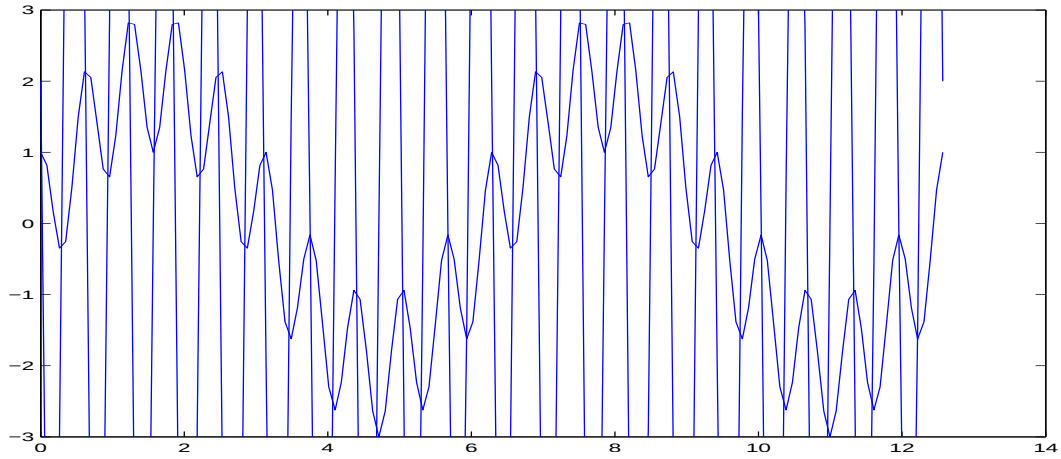



```
hold on
plot(x , y)
plot(x , dydx)
hold off
```

Example 1.30

Notice that the axis get re-scaled by the second plot command. Suppose you want the axes to be determined by the first plot and then plot the second curve with the same axis range:

```
plot(x, y)
axis(axis)
hold on

plot(x, dydx)
hold off
```



1.14.2 Line Type and colour:

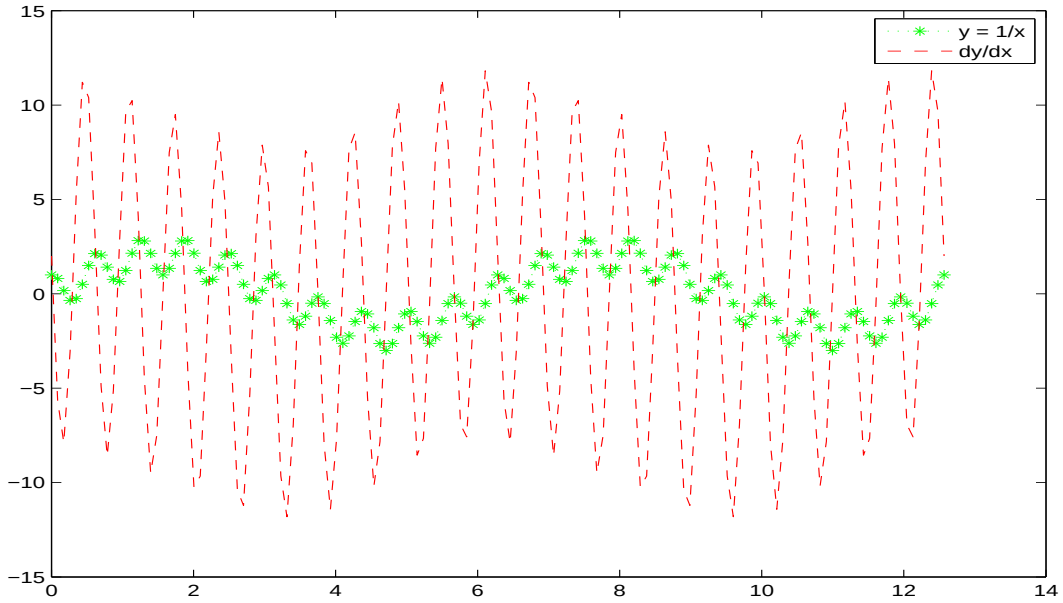
Notice that holding a graph makes all curves the same colour, while plotting several curves with one plot command makes the different colours.

```
Various line types, plot symbols and colors may be obtained with
    plot(x,y,'s') where s is a character string made from one element
    from any or all the following 3 columns:

          b     blue          .     point              -     solid
          g     green         o     circle             :     dotted
          r     red           x     x-mark             -.    dashdot
          c     cyan          +     plus               --    dashed
          m     magenta       *     star             (none)  no line
          y     yellow        s     square
          k     black         d     diamond
                              v     triangle (down)
                              ^     triangle (up)
                              <     triangle (left)
                              >     triangle (right)
                              p     pentagram
                              h     hexagram

    For example, plot(x,y,'c+:') plots a cyan dotted line with a plus
    at each data point; plot(x,y,'bd') plots blue diamond at each data
    point but does not draw any line, plot(x,y,'r') plots a red line.
```

Example 1.31

```
% multplt5.m
clc
clear

x = linspace(0 , 4*pi , 200);
y = 2*sin(x) + cos(10*x);
dydx = 2*cos(x) - 10*sin(10*x);
```

```
plot(x, y, 'g*:', x, dydx, 'r--')
legend('y = 1/x' , 'dy/dx')
```



1.14.3 Multiple plots in one window:

>> **help subplot**

subplot(m,n,p) or subplot(mnp) breaks the Figure window into an m-by-n matrix of small axes and selects the p-th axes for for the current plot. The plots (p) are counted left to right, top to bottom e.g.

```
subplot(321)
    plot(x , y1)
subplot(322)
    plot(x , y2)
subplot(323)
    plot(x , y3)
subplot(324)
    plot(x , y4)
subplot(325)
    plot(x , y5)
subplot(326)
    plot(x , y6)
```

Gives six windows arranged into a 3×2 matrix and plots:

- **y1** in the (1,1) window,
- **y2** in the (1,2) window,
- **y3** in the (2,1) window,
- **y4** in the (2,2) window,

- **y5** in the (3,1) window &
- **y6** in the (3,2) window.

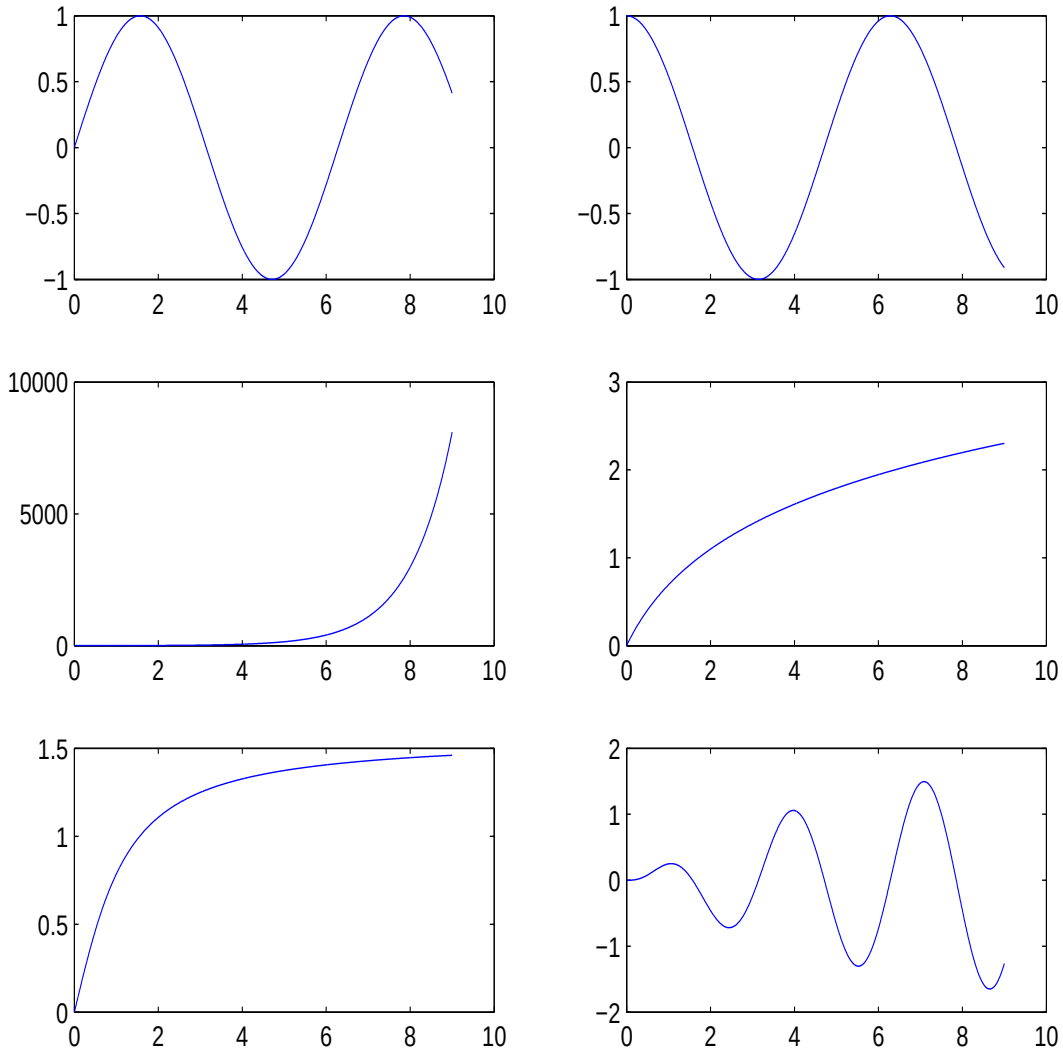


Figure 1.7: Illustrating the **subplot** command

Example 1.32 *Let us illustrate this (Fig. 1.7) using:*
$y_1 = \sin x,\ y_2 = \cos x,\ y_3 = e^x,\ y_4 = \ln(1+x),\ y_5 = \tan^{-1} x$, *and*
$y_6 = \sin x \cos x \ln(1+x) \tan^{-1} x$, *in the range* $0 \le x \le 9$.
Recall that in MATLAB:
e^x *is written exp(x),*
$\ln(1+x)$ *is written log(1+x), &*
$\tan^{-1} x$ *is written atan(x).*

```
% subplot32.m
clc
```

```
clear

x = linspace(0 , 9, 100);

y1 = sin(x);
y2 = cos(x);
y3 = exp(x);
y4 = log(1+x);
y5 = atan(x);
y6 = sin(x).*cos(x).*log(1+x).*atan(x);

subplot(321)
    plot(x , y1)
subplot(322)
    plot(x , y2)
subplot(323)
    plot(x , y3)
subplot(324)
    plot(x , y4)
subplot(325)
    plot(x , y5)
subplot(326)
    plot(x , y6)
```

1.15 Parametric Plots & Animation

1.15.1 2-D Parametric Plots

Previously, we generated 2-D graphs of functions of the form $y = f(x)$.

In this case, the (independent) variable x was created as an array of numbers in a finite domain $[x_{\min}, x_{\max}]$ and the dependent variable y was generated as an array of equal length via the function $f(x)$.

In the present case, we will think of both x and y as both dependent functions of a third independent parameter. This method of curve representation, known as *parametric representation* is described by $(x(s)\,,\, y(s))$, where the parameter s is created as an array of numbers in a finite domain $[s_{\min}, s_{\max}]$.

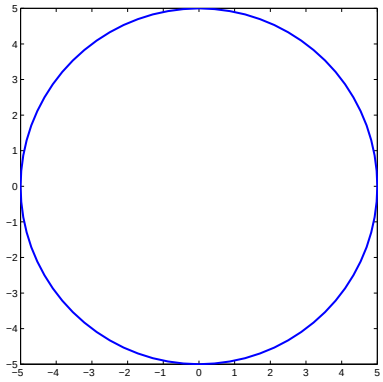
Example 1.33

The equation of a circle of radius r in Cartesian coordinates is $x^2 + y^2 = r^2$. A way to plot this would be via the two functions $y = \sqrt{r^2 - x^2},\ -r \le x \le r$ and $y = -\sqrt{r^2 - x^2},\ -r \le x \le r$.

A more straightforward method would be to plot the trigonometric circle, with parametric representation: $x = r\,\cos\theta$, $y = r\,\sin\theta$ where the parameter θ is confined to the interval $0 \le \theta \le 2\pi$.

```
% TrigCirc.m
% Plotting the trigonometric circle x = r*cos(theta), y = r*sin(theta)
clc
clear

th = 0 : pi/36 : 2*pi;
```

```
r = 5;
x = r*cos(th);
y = r*sin(th);

plot(x , y, 'LineWidth', 2) , grid
axis square
```



The parametric representation makes it easier to plot sections of the curve:

In this case, going left to right and top to bottom, we have limited the parameter θ to the domains, $0 \leq \theta \leq \pi/2$, $0 \leq \theta \leq 5\pi/4$, $0 \leq \theta \leq 3\pi/2$ and $0 \leq \theta \leq 7\pi/4$, respectively.

1.16 3-D Parametric Curves

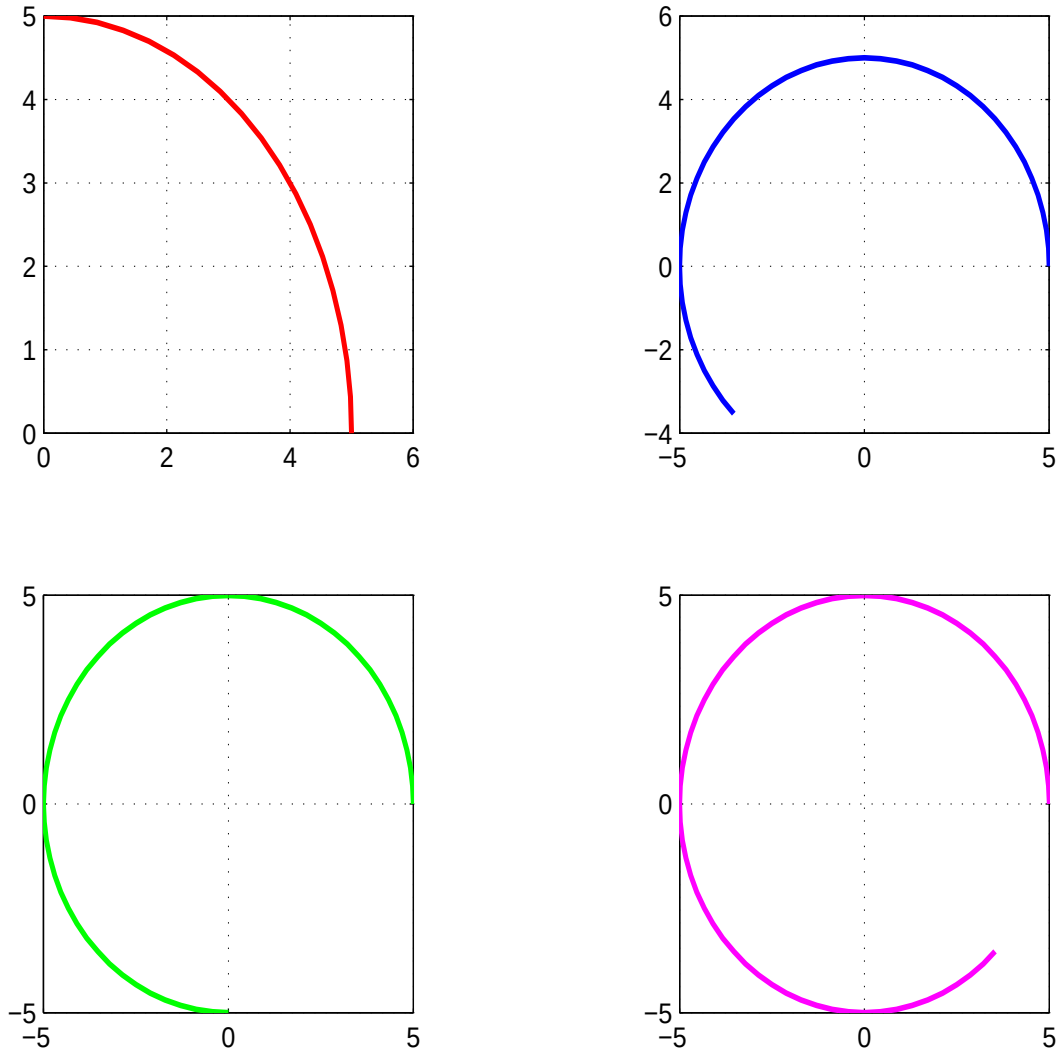
Plotting 3-D curves is done in a similar way to that described above for the 2-D case except we now have to use the MATLAB command **plot3**.

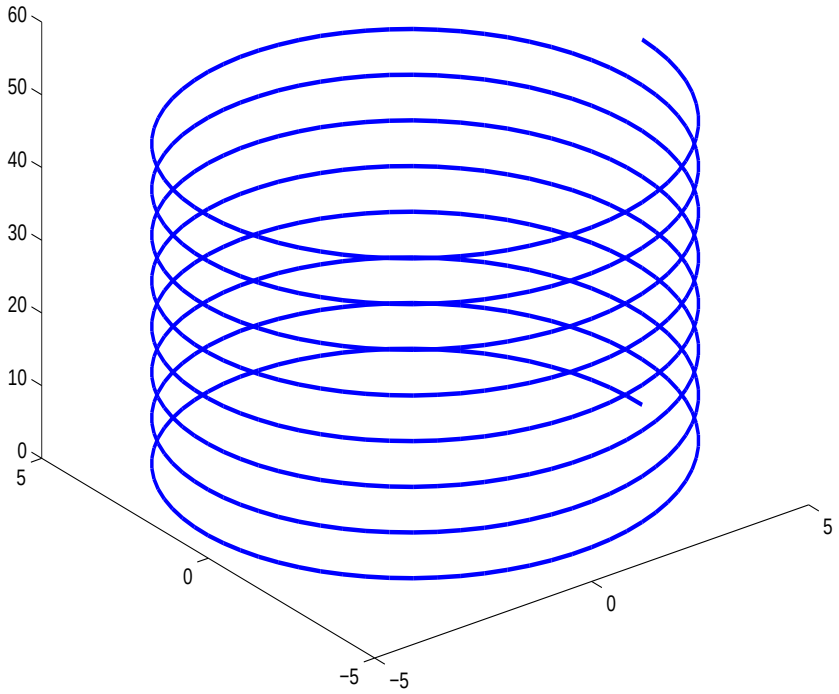
Example 1.34

Plot the circular helix:

In this case, imagine initially a point moving at a uniform speed around a circle in the x-y-plane. Now imagine that as the circular motion is continuing, the point is moving away from the x-y-plane at some constant linear speed. The parametric representation of the 3-D curve (circular helix) generated by the trajectory of the point can be written as:

$$x = r\,\cos\theta, \quad y = r\,\sin\theta, \quad z = \theta.$$


5
4
3
2
1
0
0
2
4
6
6
4
2
0
−2
−4
−5
0
5
5
0
−5
−5
0
5
5
0
−5
−5
0
5


60
50
40
30
20
10
0
5
0
−5
−5
0
5

```
% Helix3D.m
% Plotting the circular helix

clc
clear
‘
th = 0 : pi/36 : 16*pi;
r = 5;
x = r*cos(th);
y = r*sin(th);
z = th;

plot3(x , y, z , ’LineWidth’, 2) , grid
```

1.16.1 Animation

A very powerful feature of MATLAB is its ability to render animations.

Oscillations of a Spring

As an example, the helix above can be thought of as an ordinary spring. Suppose you initially pull the top of the spring up and then release it, we demonstrate the ability of MATLAB to visualize the resulting oscillations of the spring. Simply implement the following steps:

- Determine the (parametric) equations that describe the curve at a fixed time. *In this case, it is the helix parametric equations as given earlier.*
- Introduce the time dependence in the appropriate curve parameters. *In this case, make the helix pitch to be oscillatory in time*
- Generate (3-D) plots of the curve at different times.
- Use the **movie** commands to display consecutively the frames obtained in the previous step.

The following script implements the above plan:

```
% VibrSpring.m
% Animation of a vibrating spring
th = 0 : pi/60 : 32*pi;
r = 1;
A = 0.25;
w = 2*pi/15;
M = moviein(16);
for t = 1 : 16;
    x = r*cos(th);
    y = r*sin(th);
    z = (1 + A*cos(w*(t-1))) * th; %oscillatory pitch
    plot3(x , y , z , ’r’);
    axis([-2 2 -2 2 0 40*pi]);
    M(t) = getframe;
end
movie(M , 10)
```

- **M = moviein(16)** creates an array (vector) that stores in each element the data corresponding to a frame at a specific time.

  The frames themselves are generated within the **for** loop.

- The **getframe** function returns a pixel image of the image of the different frames.

- **movie(M , 10)** plays the movie the specified number of times, 10 in this case.

Moving rectangular Pulse

Consider a rectangular pulse that is moving to the right at constant speed, say 1 m/s. We want to write a program that will illustrate the time development of the pulse and then play the resulting movie:

```
% MovePulse.m
% Animation of a moving rectangular pulse
LHSrect = 0;
RHSrect = 2;
x = linspace(0,10,200);
t = linspace(0,8,40);
M = moviein(40);
for m = 1 : 40
    y = ( ( x >= LHSrect + t(m) ) & ( x <= RHSrect + t(m)) );
    plot(x , y , 'g')
    axis([-2 12 0 1.2]);
    M(m) = getframe;
end
movie(M , 12)
```

In this section, we investigate ways of (i) repeating common calculations many times over and (ii) making decisions in programs.

1.17 FOR loop

The **for** loop repeats a group of commands a fixed, predetermined number of times and has the following structure:

```
for loop_variable = vector
    statement
    statement
    .........
end
```

The *statements* between **for** and **end** are executed for each value of *loop_variable* in *vector*.

These statements can use *loop_variable* in their calculations[‡]. In other words, if MATLAB encounters a FOR loop

1. it sets *loop_variable* to the first element of *vector*,

[‡]However, the way MATLAB was designed, FOR loops tend to be inefficient in terms of computing time. If the *statements* contain the *loop_variable* it would be more advisable to avoid the FOR loop and instead vectorize the *statements* using array operations where necessary

2. executes each *statement* between FOR and END in turn,
3. returns to the beginning of the FOR loop and sets *loop_variable* to the next element and again executes each *statement* between FOR and END in turn,
4. repeats this for every element of the *vector* and after it finishes, leaves the loop and proceeds to commands below the loop.

Example 1.35

Compute the finite sum

$$\sum_{k=1}^{100} k = 1 + 2 + 3 + \ldots + 99 + 100,$$

using both a ***for*** *loop and by vectorizing.*

```
% ForLoop1.m
% recursive computations with for loop.

clc
clear
% % compute 1 + 2 + 3 + ... + 99 + 100 using for loop:

disp(' ')

tic
ForSum = 0;
for k = 1 : 100
    ForSum = ForSum + k;
end
toc
disp(['required sum is ', num2str(ForSum),' using the for loop'])

disp(' ')

% % compute 1 + 2 + 3 + ... + 99 + 100 by vectorizing
tic
k = 1 : 100;
VecSum = sum(k);
toc
disp(['vectorization also gives sum as ', num2str(VecSum)])
```

- - - - - - - - - - -

Example 1.36

Compute the finite sum

$$\sum_{k=1}^{100,000} \frac{1}{k^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots + \frac{1}{100,000^2},$$

using both a ***for*** *loop and by vectorizing.*

```
% ForLoop2.m
% recursive computations with for loop.

clc
clear
% % compute 1 + 1/2^2 + 1/3^2 + ... + 1/100000^2 using for loop:

disp(' ')

tic
ForSum = 0;
for k = 1 : 100000
    ForSum = ForSum + 1/k^2;
end
toc
disp(['required sum is ', num2str(ForSum),' using the for loop'])

disp(' ')


% % compute 1 + 1/2^2 + 1/3^2 + ... + 1/100000^2 by vectorizing:
tic
k = 1 : 100000;
VecSum = sum(1./k.^2);
toc
disp(['vectorization also gives sum as ', num2str(VecSum)])
```

- - - - - - - - - - -

Example 1.37

Compute the alternating sum

$$\sum_{k=1}^{501} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots - \frac{1}{1003},$$

using both a ***for*** *loop and by vectorizing.*

```
% ForLoop3.m
% recursive computations with for loop.

clc
clear
% % compute 1 - 1/3 + 1/5 - 1/7 + 1/9 - ... - 1/1003 using for loop:

disp(' ')

tic
ForAns = 0;
for i = 0 : 501
    ForAns = ForAns + (-1)^i/(2*i+1);
end
toc
disp(['the sum is ', num2str(ForAns),' using the (-1)^i method'])
```

```
disp(' ')


% % alternative method:
tic
ForAns2 = 0;
sign = -1;
for i = 0 : 501
    sign = -sign;
    ForAns2 = ForAns2 + sign/(2*i+1);
end
toc
disp(['the sum is ', num2str(ForAns2),' using the sign method'])

disp(' ')


% % compute 1 - 1/3 + 1/5 - 1/7 + 1/9 - ... - 1/1003 by vectorizing:
tic
i = 0 : 501;
VecSum = sum((-1).^i./(2*i+1));
toc
disp(['vectorization also gives sum as ', num2str(VecSum)])
```

- - - - - - - - - - -

For loops are also important in computations involving recursion formulas:

Example 1.38

Refrigerator Cooling: *A can of soft drink at temperature 25°C is placed in a refrigerator, where the ambient temperature F is 10°C. A standard way of determining how the soda temperature changes over a period of time is to subdivide the time interval into a number of small steps, each of duration Δt. If $T(i)$ is the temperature at the beginning of step i, the following model can be used to determine $T(i+1)$:*

$$T(i+1) = T(i) + K\,\Delta t\,(F - T(i)),$$

where K is the conduction coefficient, a parameter that depends on the insulating properties of the can and thermal properties of the soda. Compute and plot the soda temperature with $K = 0.05$ and $\Delta t = 1 minute$. See Worksheet 5.

- - - - - - - - - - -

Example 1.39

Signal Filtering: *Denoting a noisy input sample signal by $x(k)$ and a filtered output sample signal by $y(k)$, a 3-point moving average filter is represented by:*

$$\begin{aligned} y(1) &= x(1), \\ y(2) &= \frac{1}{2}[x(1) + x(2)], \\ y(k) &= \frac{1}{3}[x(k) + x(k-1) + x(k-2)], \qquad \text{if} \qquad k \geq 3. \end{aligned}$$

Write a MATLAB script file to test this filtering method. Use

$$x = \sin\left(\frac{2\pi}{5}t\right) + Noise,$$

where t is a vector with 400 elements starting from $t = 0$ & ending at $t = 10$ and where $Noise$ is a set of normally distributed random numbers with mean zero and standard deviation 0.1: the MATLAB command to create a vector of such random numbers with the same length as vector t is:

$$Noise = 0.1 * \text{randn}(1, \text{length}(t)), \qquad \text{or} \qquad Noise = 0.1 * \text{randn}(\text{size}(t)).$$

Use subplot commands to plot (i)the noisy input signal x against t and (ii)the filtered signal y against t. See Worksheet 5.

- - - - - - - - - -

Example 1.40

```
% ForLoop4.m
% recursive computations with for loop.

% % Vandermonde-type matrix

clc
clear
A = zeros(5,5); % initialize the matrix A with zero elements
for i = 1 : 5
    for j = 1 : 5
        A(j,i) = j^(i-1); % each element of A has value row^(column-1)
    end
end
disp('A = ')
disp(A)
```

1.18 If Statement

Making decisions in programs can be achieved via the **if** statement:

```
if condition1
    statement
    statement
    .........
elseif condition2
    statement
    statement
    .........
elseif condition3
    statement
    statement
    .........
    .........
    .........
```

```
else
    statement
    statement
    .........
end
```

When MATLAB encounters an **if** statement:

1. *conditions* are usually logical expressions, i.e. an expression containing a relational operator, and which is either true or false.

Relational operator	Meaning
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
~=	not equal

2. *condition1* is tested. If it is true, the corresponding statements are executed and MATLAB then moves out of the if loop to the commands below **end**.
3. If *condition1* is false, MATLAB then checks *condition2* and repeats the procedure described in (1).
4. In this way, all the conditions are tested until a true condition is found. As soon as a true condition is found, no other conditions (or elseifs) are examined and MATLAB jumps out of the loop.
5. If none of the conditions is true, the statements corresponding to **else** are executed.
6. The logic should be arranged so that not more than one of the conditions is true.
7. There can be any number of **elseifs** (or none at all), but at most one **else**.

Example 1.41

```
% IfStat1.m
% making decisions with if statement.

clc
clear

if 20 > sqrt(200)
    disp('This Statement should display')
elseif 6 == 3 * 2
    disp('True but will not display')
elseif -5 <= 7 - 12
    disp('Still true but still not displayed')
elseif 10 ~= 11
    disp('Again true and not displayed')
end
```

- - - - - - - - - - -

Example 1.42

```
% IfStat2.m
% making decisions with if statement.

clc
clear

if 20 < sqrt(200)
    disp('false, go to next condition')
elseif 6 ~= 3 * 2
    disp('again false, check next condition')
elseif -5 >= 7 - 12
    disp('true, display this statement')
elseif 10 == 11
    disp('false, but inconsequential')
end
```

1.18.1 Logical operators

Compound logical expressions can be constructed using the three **logical operators:**

logical operator	Meaning
&	and
|	or
~	not

For example here is the script **leap.m**. Can you figure out on paper what the program does?

```
% leap.m
% Determines whether a given (input) year is a leap year.
clc
clear

year = input('Give the year:  ');
% ly = 0 --> not a leap year,   ly = 1 --> is a leap year
if ((rem(year,4) == 0) &&...           % condition1
        (rem(year,100) ~= 0)) ||...    % condition1 continued
        (rem(year, 400) == 0)         % condition1 continued
    ly = 1;                           % statement1
else
    ly = 0;                           % statement2
end
if  ly == 1
  disp('This is a leap year.')
else
  disp('This is not a leap year.')
end
```

Example 1.43

```
% Qroots2.m
% Quadratic root finding script.
```

```
% A*x^2 + B*x + C = 0
% prompt for coefficients A, B & C
clc
clear

A = input('Enter NonZero leading coefficient A: ');
B = input('Enter middle coefficient B: ');
C = input('Enter constant C: ');
disp('')

if (B^2 - 4*A*C == 0)
    r1 = -B/2/A;
    r2 = r1;
    disp(['Repeated  quadratic root r_1 = r_2 = ', num2str(r1)])

elseif (B^2 - 4*A*C > 0)
    % Compute intermediate values x & y
    x = -B/(2*A);
    y = sqrt(B^2 - 4*A*C)/2/A;
    % Compute and display roots
    r1 = x + y;
    r2 = x - y;
    disp(['The 1st real quadratic root r_1 = ', num2str(r1)])
    disp(['The 2nd real quadratic root r_2 = ', num2str(r2)])

else
    x = -B/(2*A);
    y = sqrt(B^2 - 4*A*C)/2/A;
    r1 = x + y;
    r2 = x - y;
    disp(['The 1st complex root r_1 = ', num2str(r1)])
    disp(['The 2nd complex root r_2 = ', num2str(r2)])
end
```

Recall that a **for** loop repeats a set of commands for a fixed and predetermined number of times. A **while** loop, on the other hand, repeats a group of commands an indefinite number of times, i.e. the commands are executed as long as a specified condition is true.

1.19 WHILE loop

The **while** loop has the following general structure:

```
initialise condition

while condition <is appropriately specified>
    statement
    statement
    ..........
    update condition
end
```

The types of **conditions** allowed and the method of evaluation for a **while** loop are exactly the same as for an **if** statement. The difference comes in that the **if** statement is not capable

of looping.

1.19.1 Execution of WHILE loop

When MATLAB encounters a **while** loop:

- it checks the **condition(s)**
- If false, it does not execute the loop **statement(s)** at all but instead skips to commands below the **end** & continues with the rest of the program.
- If true, the statements between **while** and **end** are executed.

  When it reaches **end**, it jumps back to **while** and checks the **condition(s)** again.
- Eventually the **condition** should become false so that MATLAB can jump out of the while loop.

Thus the idea is that the variables modified within the **while** loop should include the variables in the **condition(s)**, otherwise the values in the **condition(s)** will never change! In particular, if the **condition** is always true, the loop becomes an infinite loop!

1.19.2 Interrupting infinite loops

You can generally interrupt the execution of an infinite loop by typing **CTRL+C**; otherwise you may have to shut down MATLAB **CTRL+ALT+DEL**.

1.19.3 Examples

Example 1.44

```
% cos_calc.m
% Calculates cosine of  x  using the Taylor series: and while loop
% cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - x^10/10! +.....
clc
clear

xdeg = input('Give angle in degrees:  ');
x = pi*xdeg/180;   % convert angle to radians
k = 0;             % initialize counter
oldsum = -1;       % initialize condition
newsum =  0;       % initialize condition

while  ~(newsum == oldsum) % or newsum ~= oldsum
    k2 = 2*k;
    oldsum = newsum;
    newsum = newsum + (-1)^k * x^(k2)/factorial(k2);
    k = k + 1;
end

disp(['cosine of ', num2str(xdeg),' degrees converges to ', num2str(newsum)...
    ,' using ', num2str(k-1) ,' terms in Taylor series '])
```

```
disp(' ')

cosx = cos(x);
disp(['MATLAB''s value is ', num2str(cosx)])

% Here, MATLAB uses WHILE to compute cos(x) using Taylor series, stopping
% only when the terms become so small (compared to the machine precision)
% that the numerical sum stops changing
```

- - - - - - - - - - -

Example 1.45

```
% cos_calc2.m
% Calculates cosine of  x  using the Taylor series:
% cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - x^10/10! +.....
clc
clear

x = input('Give angle in degrees:  ');
x = pi*x/180;   % convert angle to radians
err = 1e-4;     % Accuracy needed in result
k = 0;          % initialize for first term in series

term = 1;
CosSum = term;

while  abs(term) > err
  k = k + 1;
  k2 = 2*k;
  term = - term * x^2 / k2 / (k2 - 1); % New term as a multiple of previous
  CosSum = CosSum + term;
end

disp('Result of program for cos(x): ')
disp(CosSum)
disp('Result using Matlab''s built-in cos function: ')
disp(cos(x))
```

- - - - - - - - - - -

Example 1.46

```
% maxfact.m
% Finds the smallest integer whose factorial is >= 10,000.
% Simple WHILE demo.
clc
clear

i = 0;
max = 10000;

while  factorial(i) < max
  i = i + 1;
```

```
end

disp(['smallest required integer is ', num2str(i)])
disp(' ')

n = 0 : i;
FactN = factorial(n);
disp('           i           i! ')
disp([n' FactN']) %display column vectors side by side
```

```
                                  ----------
```

Example 1.47

```
% infloop.m
clc
clear

k = 0;
Counter = 0;

while k <= 1        % k not updated leading to infinite loop
  Counter = Counter + 1;
end
```

1.20 Breaking from a loop

Sometimes you may want MATLAB to jump out of a (**for** or **while**) loop prematurely, for example if a certain condition is met.

In a **while** loop, there may be an auxiliary condition that you may want to check in addition to the main condition of the **while** loop.

1.20.1 The BREAK command

Inside either type of loop, you can use the command **break** to tell MATLAB to stop running the loop and skip to the next line after the end of the loop. The command **break** is generally used in conjunction with an **if** statement. The command **return** on the other hand stops the program from running, i.e immediately terminated execution of the program.

Example 1.48

```
% WhileBreak.m
% Script to compute a*x^2 + b*x + c
clc
clear

disp('Quadratic expression a*x^2 + b*x + c evaluated')
disp('for user input a, b, c, x')

a=1; b=1; c=1; x=0;
```

```
while a ~= 0 || b ~= 0 || c ~= 0 || x ~= 0
    disp('Enter a = b = c = x = 0 to terminate')
    a = input('Enter value of a: ');
    b = input('Enter value of b: ');
    c = input('Enter value of c: ');
    x = input('Enter value of x: ');
    if a == 0 && b == 0 && c == 0 && x == 0
        disp('Congratulations you managed to jump out of the loop')
        break
    end
   quadratic = a*x^2 + b*x + c;
   disp(['Quadratic result = ', num2str(quadratic)])
end
```

- - - - - - - - - - -

Example 1.49

```
% coscalc_for.m
% Calculates cosine of  x  using the Taylor series: and for loop
% cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - x^10/10! +.....
clc
clear

xdeg = input('Give angle in degrees:  ');
x = pi*xdeg/180;   % convert angle to radians
newsum =  0;

for k = 0 : 10000
    k2 = 2*k;
    oldsum = newsum;
    newsum = newsum + (-1)^k * x^(k2)/factorial(k2);
    if newsum == oldsum
        break
    end
end

disp(['cosine of ', num2str(xdeg),' degrees converges to ', num2str(newsum)...
    ,' using ', num2str(k) ,' terms in Taylor series '])

disp(' ')

cosx = cos(x);
disp(['MATLAB''s value is ', num2str(cosx)])

% Here, the FOR loop stops when k reaches 10,000 or when the
% variable newsum stops changing, whichever comes first.
```

- - - - - - - - - - -

Example 1.50

```
% numguess.m
clc
```

```
clear

aim = 1 + floor(99.5*rand);
ok = 0;
disp('I''m thinking of a number in the range 1 to 100')
disp('see if you can guess it in seven tries')

for n = 1 : 7
    if ok == 0
        disp(['Guess number: ', num2str(n)'])
        guess = input('your guess: ');
        num = n;
        if guess == aim
            ok = 1;
            disp(['Congratulation! You win in: ', num2str(num) ,' tries'])
            break
        elseif guess > aim
            disp('Too large')
        else
            disp('Too small')
        end
    end
end

if ok == 0
    disp ('You lose. Why not try again')
end
```

- - - - - - - - - - -

Example 1.51

```
% infloop_break.m
% Shows how to avoid endless loops in while statements.
clc
clear

cntr = 0;                                % The counter
while (1 == 1)                           % This would give an infinite loop
%while (1 == 1) & (cntr < 100000000)     % Alternative 2
  cntr = cntr + 1;
  if  cntr > 10000                       % Alternative 1
      break
  end
end
disp(['managed to break out of loop at counter = ', num2str(cntr)])
```

1.21 Timing a program

1.21.1 The TIC & TOC commands

>> help tic >> help toc

To obtain the time elapsed between the commencement and conclusion of a program, type **tic** at the start of the program and **toc** at the end.

Example 1.52

```
% Lsum1.m
% Calculates 1/3 +  1/6 + 1/9 + 1/12 + 1/15 + ...
% up to a sum L  = 1, 2, 3, 4, 5
clc
clear

L = input('Input a value for the sum L:  ');
k = 0;             % initialise counter
mysum =  0;

tic
while  mysum < L
    k = k + 1;
    k3 = 3*k;
    mysum = mysum + 1/k3;
    if rem(k , 1000000) == 0
        disp(['sum after ', num2str(k) ,' terms is ', num2str(mysum)])
    end
end
toc

disp(['The required sum converges to ', num2str(mysum)...
    ,' after ', num2str(k) ,' terms of the series '])
```

The **tic** and **toc** commands here will modify the program **Lsum1.m** to print out at the end how long it took to run.

1.21.2 CLOCK, ETIME & CPUTIME

>> help clock >> help etime >> help cputime

We could change this program further so that every 1,000,000 terms it also prints the number of seconds since the start of the **while** loop. Add a statement just outside the **while** loop and another one inside the **if** statement.

Example 1.53

```
% Lsum2.m
% Calculates 1/3 +  1/6 + 1/9 + 1/12 + 1/15 + ...
% up to a sum L  = 1, 2, 3, 4, 5
clc
clear

L = input('Input a value for the sum L:  ');
k = 0;             % initialize counter
mysum =  0;

timer = cputime;
```

```
tic
while  mysum < L
    k = k + 1;
    k3 = 3*k;
    mysum = mysum + 1/k3;
    if rem(k, 1000000) == 0
        TimeSinceStart = cputime - timer
        disp(['sum after ', num2str(k) ,' terms is ', num2str(mysum)])
    end
end
toc

disp(['The required sum converges to ', num2str(mysum)...
    ,' after ', num2str(k) ,' terms of the series '])
```

1.22 Function M-files

The functions **sin, cos, sqrt, abs** are examples of built-in MATLAB functions. You can develop your own user-defined functions that can be used in a similar way to the built-in MATLAB functions.

Such user-defined functions are called **function m-files** and as with script m-files, have the **.m** extension.

The **first uncommented line** of a function m-file **must** have the following syntax:

function [output variables] = Function_Name(input variables)

Thus the first uncommented line must

- start with the word **function**,
- followed by **(optional) output variables**, *[enclosed in square brackets and separated by commas]. In case of only a single output variable, the square brackets should be omitted. These are dummy variables and can be assigned any name during function call*
- an **equal sign**, *(if no output arguments are listed, the equal sign should be omitted as well)*
- and the **function name**.
- The **input variables**, called arguments, of the function must follow the function name and are enclosed in parentheses. *Multiple inputs must also be separated by commas.*

This first line distinguishes a function m-file from a script m-file. After this first line, the remainder of a function m-file is similar to a script m-file with subtle differences:

- When you execute a script m-file, the variables you use and define belong to your Workspace; i.e., they take on any **uncleared values** you assigned earlier in your MATLAB session, and they persist after the script has finished executing.

  Variables used in a function m-file are usually *local*, meaning that they are unaffected by, and have no effect on the variables in your Workspace.

- Script m-files can call other script m-files as well as function m-files. Function m-files can call other function m-files – but it isn't sensible to use them to call script m-files.

Function call: A function m-file is called *by the name of the m-file in which it is defined and not by the Function_Name in the first line.* To avoid confusion, use the same name for both the Function_Name and M-file.

Example 1.54

```
% fx.m
% function m-file

function y = fx(x)
y = 3*x + sin(x);
```

Note in particular that the input and output variables **x** & **y** respectively are dummy variables. Thus the function file **fx.m** is exactly the same as the file **F3PlusSin.m** below:

Example 1.55

```
% F3PlusSin.m
% function m-file

function MyOutPut = F3PlusSin(MyInPut)
MyOutPut = 3*MyInPut + sin(MyInPut);
```

Check that these two functions are in fact the same:

```
>> fx(pi)                      >> F3PlusSin(pi)
>> w1 = fx(pi/2)               >> w2 = F3PlusSin(pi/2)
>> L1 = fx(pi/6) - pi/2        >> L2 = F3PlusSin(pi/6) - pi/2
>> s1 = fx(pi)/pi              >> s2 = F3PlusSin(pi)/pi
```

Example 1.56 *Write a function m-file that calculates the derivative of the function* ***fx****:*

```
% dfx.m
% function m-file

function y = dfx(w)
y = 3 + cos(w);
```

As before, the input & output variables are dummy arguments and thus can be used again in the function call:

```
>> w = dfx(pi/2)                  >> x = dfx(pi/3) - 1/2
>> y = fx(pi) / dfx(pi/2)         >> z = -6 : 0.01 : 6;
>> plot(z , fx(z) , 'r:' , z , dfx(z) , 'g-.')
```

Example 1.57

```
% tablex.m
% function m-file calling other function m-files

function [t, f, df, tfdf] = tablex(a, b, dt)
t = (a : dt : b)';
f = fx(t);
df = dfx(t);
tfdf = t - f./df;
```

The function **tablex.m** has *three inputs* and *four outputs*. The 1st output is a column vector from **a** to **b** by increments of **dt**. The 2nd output is thus also a column vector of the values calculated by from applying the function **fx** on the 1st output. The 3rd output is similarly a column vector calculated from the function **dfx**. The 4th and final output is a column vector calculated from the function:

$$4th\ output \quad = \quad 1st\ output \quad - \quad \frac{\mathbf{fx}(1st\ output)}{\mathbf{dfx}(1st\ output)}$$

Notice that the division is done element by element and thus the result is a vector of the same length.

Example 1.58

Display only the first output on the screen:

```
>> tablex(0, 1, 0.2)
```

Display only the 1st output on the screen, but now assign it a variable name (say p*):*

```
>> p = tablex(0, 1, 0.2)
```

Display the first and second outputs on the screen assigned to variable names m & n *respectively:*

```
>> [m, n] = tablex(0, 1, 0.2)
```

You do not have to put commas between variables that receive the output.

```
>> [df Xz L2] = tablex(0, 1, 0.2)
```

Display all outputs on the screen assigned to variable names tfdf, df, f & t *respectively:*

```
>> [tfdf, df, f, t] = tablex(0, 1, 0.2)
```

Notice that the variable names used in the last case are in the same as the output variables in the function file but in reverse order. This example further illustrates that the input and output arguments inside a function m-file are simply place holders (i.e. dummy variables).

1.23 Boundary Value Problems (BVP's)

We limit our attention to **two-point boundary value problems:** Solve

$$\frac{d^2y}{dx^2} = f(x, y, y'), \tag{1.1}$$

subject to the boundary conditions:

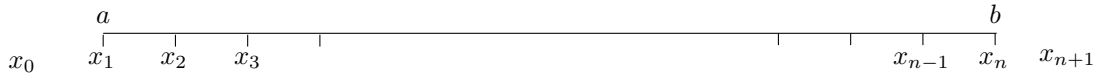
$$y(a) = \alpha \quad \text{or} \quad y'(a) = \alpha, \quad \text{and} \quad y(b) = \beta \quad \text{or} \quad y'(b) = \beta. \tag{1.2}$$

In this case, the independent variable x lies in a closed interval: $x \in [a, b]$. Boundary conditions are then specified on the solution, y, at the end-points (boundaries) of this interval.

The problem is then to use the differential equation together with the supplied conditions to find the solution, y, at all other point inside the interval where it is not known.

1.23.1 Finite Difference Method (FDM)

In the *finite difference method* we subdivide the interval $[a, b]$ into $(n-1)$ equal subintervals:



As before, we will write $y_i = y(x_i)$ to represent the solution values at the corresponding mesh point.

Here x_0 and x_{n+1} are superficial points, we will see later why they may be needed.

The idea behind the FDM is that the derivatives of y in the differential equation are replaced by finite differences (forward, backward or central differences) obtained from Taylor series.

Increased accuracy is achieved by using central difference approximations. In this case, the derivatives of y at each mesh point are calculated as:

$$y_i' = \frac{y_{i+1} - y_{i-1}}{2\Delta x}, \qquad y_i'' = \frac{y_{i+1} - 2\,y_i + y_{i-1}}{\Delta x^2}. \tag{1.3}$$

The differential equation (1.1) thus reduces to

$$\frac{y_{i+1} - 2\,y_i + y_{i-1}}{\Delta x^2} = f\left(x_i, y_i, \frac{y_{i+1} - y_{i-1}}{2\Delta x}\right), \tag{1.4}$$

and should be solved subject to one of the following cases.

1.23.2 Case 1: $\mathbf{y(a) = \alpha,\ y(b) = \beta}$

Here y_1 and y_n are **known** and thus equations (1.4) need only be solved for the $n-2$ unknown values $y_2,\ y_3,\ \ldots,\ y_{n-1}$.

In other words, the index i runs from 2 through $n-1$ and equations (1.4) represents a system of $n-2$ simultaneous algebraic equations in $n-2$ unknowns.

If the differential equation is linear, then the algebraic equations will also be linear: $\mathbf{Ay = B}$ – easy to solve!

If on the other hand the differential equation is nonlinear, the algebraic equations will also be nonlinear and must be solved via a root finding method, say Newton's method or the bisection method.

Example 1.59 *Write a MATLAB script file to compute the finite difference solution of the following BVP:*

$$c_2\frac{d^2y}{dx^2} + c_1\frac{dy}{dx} + c_0 y = f(x), \qquad y(a) = \alpha, \quad y(b) = \beta.$$

This boundary value problem can be written in the alternative notation:

$$c_2 y'' + c_1 y' + c_0 y = f(x), \qquad y(a) = \alpha, \quad y(b) = \beta.$$

Solution 1.1 *The solutions at $i = 1$ (i.e. at $x = a$) and at $i = n$ (i.e. at $x = b$) are know and will be denoted respectively as:*

$$y_1 = \alpha \qquad \text{and} \qquad y_n = \beta.$$

The remaining values of the solution (i.e. y_2, y_3, y_4, ..., y_{n-1}) can then be found from the finite difference equations (1.4) for the BVP. For our current BVP, these equations are constructed as follows:

$$c_2 \frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} + c_1 \frac{y_{i+1} - y_{i-1}}{2\Delta x} + c_0 y_i = f(x_i), \qquad i = 2, 3, 4 \ldots, n-1.$$

Multiplying through by $2\Delta x^2$ yields

$$2c_2\,(y_{i+1} - 2y_i + y_{i-1}) + \Delta x\, c_1\,(y_{i+1} - y_{i-1}) + 2\Delta x^2\, c_0\, y_i = 2\Delta x^2\, f(x_i), \qquad i = 2, 3, 4 \ldots, n-1.$$

Grouping like terms leads to the desired equations:

$$(2c_2 - \Delta x\, c_1)\, y_{i-1} + (2\Delta x^2\, c_0 - 4c_2)\, y_i + (2c_2 + \Delta x\, c_1)\, y_{i+1} = 2\Delta x^2\, f(x_i). \qquad i = 2, 3, 4 \ldots, n-1.$$

Taking

$$A_L = (2c_2 - \Delta x\, c_1), \quad A_C = (2\Delta x^2\, c_0 - 4c_2), \quad A_R = (2c_2 + \Delta x\, c_1),$$

the finite difference equations can then be re-written as

$$A_L\, y_{i-1} + A_C\, y_i + A_R\, y_{i+1} = 2\Delta x^2 f(x_i). \qquad i = 2, 3, 4 \ldots, n-1.$$

Or in matrix form: $\mathbf{A}\,\mathbf{y} = \mathbf{B}$*, where*

$$\mathbf{A} = \begin{bmatrix} A_C & A_R & & & \\ A_L & A_C & A_R & & \\ & \ddots & \ddots & \ddots & \\ & & A_L & A_C & A_R \\ & & & A_L & A_C \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} y_2 \\ y_3 \\ \vdots \\ y_{99} \\ y_{100} \end{bmatrix}$$

and

$$\mathbf{B} = \begin{bmatrix} 2\Delta x^2 f(x_2) - A_L y_1 \\ 2\Delta x^2 f(x_3) \\ \vdots \\ 2\Delta x^2 f(x_{n-2}) \\ 2\Delta x^2 f(x_{n-1}) - A_R y_n \end{bmatrix}$$

The $(n-2) \times (n-2)$ matrix **A** can be created using the MATLAB command **spdiags**, *see tutorial 2.*

```
% BvpScript.m

clc
clear

a = ....; b = ....; n = ....; dx = (b - a)/(n - 1);
c2 = ....; c1 = ....; c0 = ....;

AL = (2*c2 - dx*c1);
AC = (2*dx^2*c0 - 4*c2);
AR = (2*c2 + dx*c1);
e1 = ones(n-2,1);

A = spdiags([AL*e1 AC*e1 AR*e1], -1 : 1, n-2, n-2);

% The vector B of length (n-2) is created from x_2, x_3,..., x_(n-1)
i = 1 : n;
x = a + (i-1)*dx;

B = 2 * dx^2 * f(x); % type in the function here
B = B(2:n-1);
B = B';

% Input boundary conditions & correct the rhs
y1 = ....; yn = ....;
B(1) = B(1) - AL*y1;
B(n-2) = B(n-2) - AR*yn;

                        % The solution vector y can now be calculated
yint = A\B;             % solution at interior points
y = [y1 ; yint ; yn];   % full solution

plot( x , y , '....' , 'LineWidth' , 2), grid
xlabel('x' , 'FontSize' , 20)
ylabel('y' , 'FontSize' , 20)
```

In this case **yint** represents the interrior solution vector $y_2,\ y_3,\ \ldots,\ y_{n-1}$ and the full solution, **y**, is formed by appending y_1 and y_n to this.

Example 1.60 *Write out equations (1.4) for the following linear BVP using* $n = 101$ *and solve these equations.*

$$y'' = -sinx, \qquad y(0) = 0, \quad y(8\pi) = 0.$$

Compare the finite difference solution with the exact solution $y_{exact} = \sin x$ *(use graphs).*

Solution 1.2 *In this case* $c_2 = 1$, $c_1 = c_0 = 0$ *and* $f(x) = -sin(x)$ *hence*

```
% Bvp1.m
```

```
clc
clear

a = 0; b = 8*pi; n = 101; dx = (b - a)/(n - 1);
c2 = 1; c1 = 0; c0 = 0;

AL = (2*c2 - dx*c1);
AC = (2*dx^2*c0 - 4*c2);
AR = (2*c2 + dx*c1);
e1 = ones(n-2,1);

A = spdiags([AL*e1 AC*e1 AR*e1], -1 : 1, n-2, n-2);

% The vector B of length (n-2) is created from x_2, x_3,..., x_(n-1)
i = 1 : n;
x = a + (i-1)*dx;

B = - 2*dx^2 * sin(x);
B = B(2:n-1);
B = B';

% Input boundary conditions & correct the rhs
y1 = 0; yn = 0;
B(1) = B(1) - AL*y1;
B(n-2) = B(n-2) - AR*yn;

% The solution vector y can now be calculated
yint = A\B;
y = [y1 ; yint ; yn];

y_exact = sin(x);

plot( x , y , 'r' , x , y_exact , 'go'  , 'LineWidth' , 2), grid
legend('Numerical','sin(x)')
xlabel('x' , 'FontSize' , 20)
ylabel('y' , 'FontSize' , 20)
title('Finite Difference Solution of y'' '' = -sin x' , 'FontSize' , 15)
```

1.23.3 Case 2: $\mathbf{y'(a) = \alpha,\ y(b) = \beta}$

Now y_1 is **unknown** but y_n is.

The index i now runs from 1 through $n-1$ and equations (1.4) represents a system of $n-1$ simultaneous algebraic equations in the $n-1$ unknowns: $y_1,\ y_2,\ y_3,\ \ldots,\ y_{n-1}$.

Notice however that substituting $i = 1$ in equations (1.4) introduces the superficial value y_0. This "spill over" can be eliminated by using the left hand side boundary condition:

$$y_1' = \alpha \qquad \Rightarrow \qquad \frac{y_2 - y_0}{2\,\Delta x} = \alpha, \qquad \text{thus,} \qquad y_0 = y_2 - 2\,\alpha\,\Delta x.$$

```
% Bvp Template Case 2

clc
```

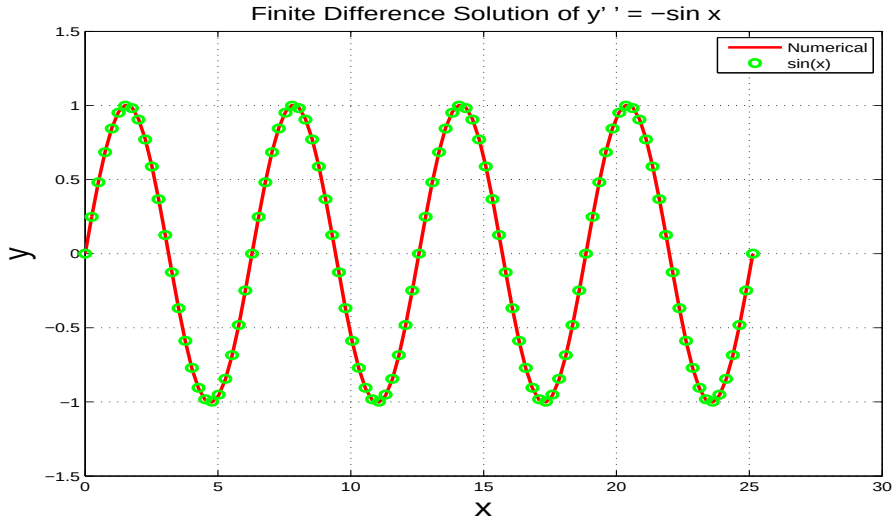


Figure 1.8: Graphical representation of solution.

```
clear

a = ....; b = ....; n = ....; dx = (b - a)/(n - 1);
c2 = ....; c1 = ....; c0 = ....; alpha = ....;

AL = (2*c2 - dx*c1);
AC = (2*dx^2*c0 - 4*c2);
AR = (2*c2 + dx*c1);
e1 = ones(n-1,1);

A = spdiags([AL*e1 AC*e1 AR*e1], -1 : 1, n-1, n-1);
A(1 , 2) = AL + AR;

% The vector B of length (n-1) is created from x_1, x_3,..., x_(n-1)
i = 1 : n;
x = a + (i-1)*dx;

B = 2 * dx^2 * f(x); % type in the function here
B = B(1:n-1);
B = B(:);

% Input boundary conditions & correct the rhs
yn = ....;
B(1) = B(1) + 2*alpha*AL*dx;
B(n-1) = B(n-1) - AR*yn;

                        % The solution vector y can now be calculated
ylhs = A\B;             % solution at interior points
y = [ylhs ; yn];        % full solution

plot( x , y , '....' , 'LineWidth' , 2), grid
xlabel('x' , 'FontSize' , 20)
```

```
ylabel('y' , 'FontSize' , 20)
```

Example 1.61 *Using* $n = 101$ *and solve the BVP:*

$$y'' - y' - 2y = -2\exp(x), \qquad y'(0) = 5, \quad y(\ln 2) = 5.$$

via the finite difference method.

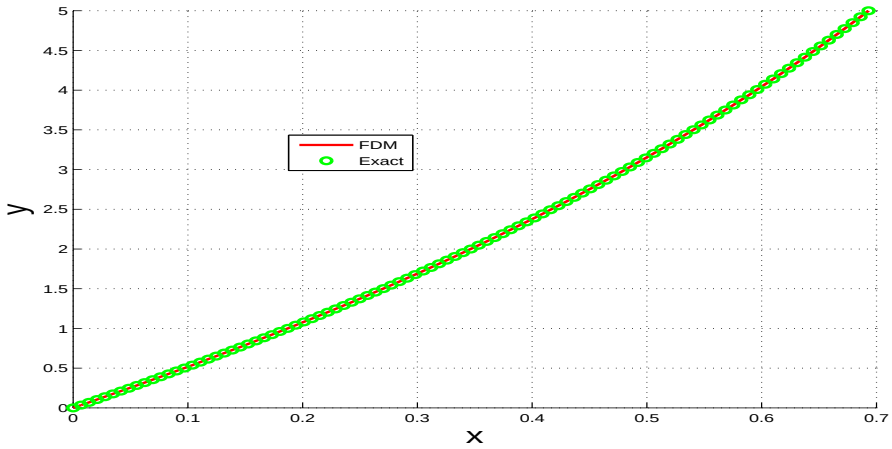


Figure 1.9: Graphical representation of solution.

1.23.4 Case 3: $\mathbf{y(a) = \alpha,\ y'(b) = \beta}$

In this case y_n is **unknown**, but y_1 is.

The index i now runs from 2 through n and equations (1.4) represents a system of $n-1$ simultaneous algebraic equations in the $n-1$ unknowns: $y_2,\ y_3,\ \ldots,\ y_{n-1},\ y_n$.

Again substituting $i = n$ in equations (1.4) introduces the superficial value y_{n+1} and this "spill over" is eliminated by using the right hand side boundary condition:

$$y'_n = \beta \qquad \Rightarrow \qquad \frac{y_{n+1} - y_{n-1}}{2\,\Delta x} = \beta, \qquad \text{thus,} \qquad y_{n+1} = y_{n-1} + 2\,\beta\,\Delta x.$$

```
% Bvp Template Case 3

clc
clear

a = ....; b = ....; n = ....; dx = (b - a)/(n - 1);
c2 = ....; c1 = ....; c0 = ....; beta = ....;

AL = (2*c2 - dx*c1);
AC = (2*dx^2*c0 - 4*c2);
AR = (2*c2 + dx*c1);
e1 = ones(n-1,1);

A = spdiags([AL*e1 AC*e1 AR*e1], -1 : 1, n-1, n-1);
```

```
A(n-1 , n-2) = AL + AR;

% The vector B of length (n-1) is created from x_2, x_3,..., x_n
i = 1 : n;
x = a + (i-1)*dx;

B = 2 * dx^2 * f(x); % type in the function here
B = B(2:n);
B = B(:);

% Input boundary conditions & correct the rhs
y1 = ....;
B(1) = B(1) - AL*y1;
B(n-1) = B(n-1) - 2*beta*AR*dx;

                          % The solution vector y can now be calculated
yrhs = A\B;               % solution at interior points
y = [y1 ; yrhs];          % full solution

plot( x , y , '....' , 'LineWidth' , 2), grid
xlabel('x' , 'FontSize' , 20)
ylabel('y' , 'FontSize' , 20)
```

Example 1.62 *Using* $n = 101$ *and solve the BVP:*

$$y'' = 5y' - 6y + \sin x, \qquad y(0) = 0.1, \quad y'(\pi/2) = 1.$$

via the finite difference method.

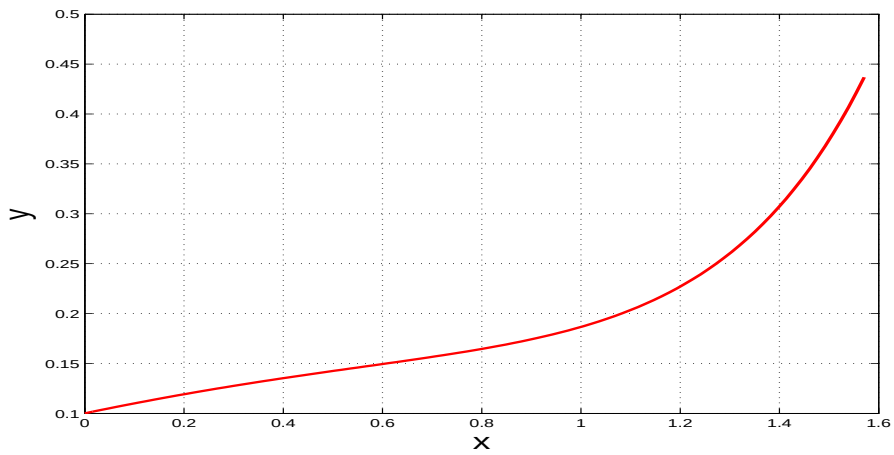


Figure 1.10: Graphical representation of solution.

Lecture 2

Fluid flow, heat and mass transfer

2.1 Fluid flow and heat transfer

2.1.1 Rectangular coordinates; 1-D flow

$$\mathrm{Re}\,\frac{\partial u}{\partial t} = -\frac{\partial p}{\partial x} + \frac{\partial}{\partial y}\left(\mu(T)\frac{\partial u}{\partial y}\right), \tag{2.1}$$

$$\mathrm{Re}\ \mathrm{Pr}\,\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial y^2} + \mu(T)\,\mathrm{Br}\left(\frac{\partial u}{\partial y}\right)^2 + \lambda\,H(T), \tag{2.2}$$

No-slip boundary conditions are employed for u and the problem can be formulated as a (i) Couette flow ($p \equiv 0$ and moving walls) (ii) Plane Poiseuille flow ($p \neq 0$ and non-moving walls) or (iii) combined Couette-Poiseuille flow. The walls can either be kept isothermal or be exposed to convective heat transfer processes. This will determine the type of temperature boundary conditions, either Dirichlet, Neumann or Robin conditions. The walls can either be considered porous or impermeable and the governing equations would have to be modified accordingly.

The temperature dependent variable viscosity $\mu(T)$ and the possible chemical kinetics $H(T)$ can be modeled as follows;

$$\mu(T) = \exp\left(-\frac{T}{1+\beta T}\right), \quad H(T) = \exp\left(\frac{T}{1+\beta T}\right), \tag{2.3}$$

2.1.2 Cylindrical coordinates; axisymetric flow

$$\mathrm{Re}\,\frac{\partial w}{\partial t} = -\frac{\partial p}{\partial z} + \frac{1}{r}\,\frac{\partial}{\partial r}\left(r\,\mu(T)\frac{\partial w}{\partial r}\right), \tag{2.4}$$

$$\mathrm{Re}\ \mathrm{Pr}\,\frac{\partial T}{\partial t} = \frac{1}{r}\,\frac{\partial}{\partial r}\left(r\,\frac{\partial T}{\partial r}\right) + \mu(T)\,\mathrm{Br}\left(\frac{\partial w}{\partial r}\right)^2 + \lambda\,H(T), \tag{2.5}$$

2.2 Fluid flow and mass transfer

Governing equations for velocity and, say, pollutant concentration (ϕ)

2.2.1 Rectangular coordinates; 1-D flow

$$\mathrm{Re}\,\frac{\partial u}{\partial t} = K + \frac{\partial}{\partial y}\left(e^{\alpha\phi}\frac{\partial u}{\partial y}\right) + \mathrm{Gr}\;\phi, \tag{2.6}$$

$$\mathrm{Pe}\,\frac{\partial \phi}{\partial t} = \frac{\partial}{\partial y}\left(e^{\gamma\phi}\frac{\partial \phi}{\partial y}\right) + \lambda\, e^{n\phi}, \tag{2.7}$$

2.2.2 Cylindrical coordinates; axisymetric flow

$$\mathrm{Re}\,\frac{\partial w}{\partial t} = K + \frac{1}{r}\frac{\partial}{\partial r}\left(r\, e^{\alpha\phi}\frac{\partial w}{\partial r}\right) + \mathrm{Gr}\;\phi, \tag{2.8}$$

$$\mathrm{Pe}\,\frac{\partial \phi}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\, e^{\gamma\phi}\frac{\partial \phi}{\partial r}\right) + \lambda\, e^{n\phi}. \tag{2.9}$$

2.3 Numerical Solution

We notice that each set of equations has a similar structure and hence the solution process for one set can be generalized to the others. We illustrate such a numerical solution process for equations (2.4 - 2.5).

We implement a numerical algorithm based on the semi-implicit finite difference scheme. The discretization of the governing equations is based on a linear Cartesian mesh and uniform grid on which finite-differences are taken. We approximate both the second and first spatial derivatives with second-order central differences. The equations corresponding to the first and last grid points are modified to incorporate the boundary conditions. The semi-implicit scheme for the velocity component reads:

$$\mathrm{Re}\;\frac{u^{(n+1)} - u^{(n)}}{\Delta t} = G + \left[\frac{\partial \mu}{\partial r}\frac{\partial u}{\partial r} + \frac{\mu}{r}\frac{\partial u}{\partial r}\right]^{(n)} + \mu^{(n)}\,\frac{\partial^2 u}{\partial r^2}^{(n+\alpha)}, \tag{2.10}$$

where $0 \le \alpha \le 1$. Here, as in the Crank-Nicolson scheme, terms given at $(n+\alpha)$ are taken as the averages of the corresponding terms at $(n+1)$ and n, i.e. $\alpha\,\#^{(n+1)} + (1-\alpha)\,\#^{(n)}$. The equation for $u^{(n+1)}$ then becomes:

$$\begin{aligned} -\Gamma_1\, u_{j-1}^{(n+1)} + (\,\mathrm{Re} + 2\Gamma_1)\, u_j^{(n+1)} - \Gamma_1\, u_{j+1}^{(n+1)} &= \Delta t\, G \\ +\Gamma_2\, u_{j-1}^{(n)} + (\,\mathrm{Re} - 2\Gamma_2)\, u_j^{(n)} + \Gamma_2\, u_{j+1}^{(n)} & \\ + \frac{\Delta t}{2\Delta r}\left(u_{i+1}^{(n)} - u_{i-1}^{(n)}\right)\left[\frac{\left(\mu_{i+1}^{(n)} - \mu_{i-1}^{(n)}\right)}{2\Delta r} - \frac{\mu_i^{(n)}}{r_i}\right] & \end{aligned} \tag{2.11}$$

where $\Gamma_1 = \alpha\,\mu^{(n)}\,\Delta t/(\Delta r)^2$ and $\Gamma_2 = (1-\alpha)\,\mu^{(n)}\,\Delta t/(\Delta r)^2$. The solution procedure for $u^{(n+1)}$ thus reduces to inversion of tri-diagonal matrices which is an advantage over a full

implicit scheme. The semi-implicit integration scheme for the temperature equation is similar to that for the velocity component. Unmixed second partial derivatives of the temperature are treated implicitly:

$$\text{Re Pr}\ \frac{T^{(n+1)}-T^{(n)}}{\Delta t}=\frac{\partial^2 T}{\partial r^2}^{(n+\alpha)}+\frac{1}{r}\frac{\partial T}{\partial r}^{(n)}+\lambda H(T)^{(n)}+\mu^{(n)}\text{Br}\left(\frac{\partial u}{\partial r}^{(n)}\right)^2. \tag{2.12}$$

The equation for $T^{(n+1)}$ thus becomes:

$$\begin{aligned}&-\Gamma_3\,T_{j-1}^{(n+1)}+(\,\text{Re Pr}\,+2\Gamma_3)\,T_j^{(n+1)}-\Gamma_3\,T_{j+1}^{(n+1)}\\&=\Gamma_4\,T_{j-1}^{(n+1)}+(\,\text{Re Pr}\,-2\Gamma_4)\,T_j^{(n+1)}+\Gamma_4\,T_{j+1}^{(n+1)}\\&\quad+\lambda H(T)^{(n)}+\frac{\Delta t}{2r_i\Delta r}\left(T_{i+1}^{(n)}-T_{i-1}^{(n)}\right)+\frac{\text{Br}}{4\Delta r^2}\,\mu_i^{(n)}\left(u_{i+1}^{(n)}-u_{i-1}^{(n)}\right)^2\end{aligned} \tag{2.13}$$

where $\Gamma_3=\alpha\,\Delta t/(\Delta r)^2$ and $\Gamma_4=(1-\alpha)\,\Delta t/(\Delta r)^2$. The remaining task is to write MATLAB codes (using ideas from the boundary value problems of the previous lecture) to solve the discretized equations.

References

1. T. Chinyoka, O.D. Makinde, Computational dynamics of unsteady flow of a variable viscosity reactive fluid in a porous pipe, *Mechanics Research Communications, Volume 37 (2010) 347-353.*

2. O.D. Makinde, T. Chinyoka, Analysis of unsteady flow of a variable viscosity reactive fluid in a slit with wall suction or injection, *in review.*

3. O. D. Makinde: Thermal ignition in a reactive viscous flow through a channel filled with a porous medium. Transactions of ASME, Journal of Heat Transfer, Vol. 128 (2006) 601-604.

4. O.D. Makinde, T. Chinyoka, Transient analysis of pollutant dispersion in a cylindrical pipe with a nonlinear waste discharge concentration, Computers and Mathematics with Applications, 60 (2010) 642-652.

5. T. Chinyoka, O.D. Makinde, Analysis of nonlinear dispersion of a pollutant ejected by an external source into a channel flow, *Mathematical Problems in Engineering Volume 2010 (2010), Article ID 827363, 17 pages doi:10.1155/2010/827363*

Lecture 3

Thermal combustion and environmental problems

3.1 CO_2 emission & O_2 depletion

$$\frac{\partial \theta}{\partial t} = \frac{\partial^2 \theta}{\partial y^2} + \lambda(1+\varepsilon\theta)^m \, \Phi^n \, \exp\left(\frac{\theta}{1+\varepsilon\theta}\right), \tag{3.1}$$

$$\frac{\partial \Phi}{\partial t} = \alpha \, \frac{\partial^2 \Phi}{\partial y^2} - \lambda \, \beta_1 (1+\varepsilon\theta)^m \, \Phi^n \, \exp\left(\frac{\theta}{1+\varepsilon\theta}\right), \tag{3.2}$$

$$\frac{\partial \Psi}{\partial t} = \sigma \, \frac{\partial^2 \Psi}{\partial y^2} + \lambda \, \beta_2 (1+\varepsilon\theta)^m \, \Phi^n \, \exp\left(\frac{\theta}{1+\varepsilon\theta}\right). \tag{3.3}$$

$$\frac{\partial \theta}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \theta}{\partial r}\right) + \lambda(1+\varepsilon\theta)^m \, \Phi^n \, \exp\left(\frac{\theta}{1+\varepsilon\theta}\right), \tag{3.4}$$

$$\frac{\partial \Phi}{\partial t} = \frac{\alpha}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \Phi}{\partial r}\right) - \lambda \, \beta_1 (1+\varepsilon\theta)^m \, \Phi^n \, \exp\left(\frac{\theta}{1+\varepsilon\theta}\right), \tag{3.5}$$

$$\frac{\partial \Psi}{\partial t} = \frac{\sigma}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \Psi}{\partial r}\right) + \lambda \, \beta_2 (1+\varepsilon\theta)^m \, \Phi^n \, \exp\left(\frac{\theta}{1+\varepsilon\theta}\right). \tag{3.6}$$

The (partial differential) equations have a similar structure to those in the previous lecture and hence the solution process, say via the finite difference method, is similar.

References

1. O.D. Makinde, T. Chinyoka & R.S. Lebelo, Numerical investigation into CO_2 emission, O_2 depletion and thermal decomposition in a reacting slab, *in review.*

2. T. Chinyoka, O.D. Makinde, Computational dynamics of CO_2 emission, O_2 depletion and thermal decomposition in a cylindrical pipe filled with reactive materials *in review.*

Lecture 4

Epidemiology and Finance

The dynamical systems used to model epidemiological problems are examples of Initial Value Problems (IVP's) and can be solved say by Euler's method or the Runge-Kutta methods.

4.1 Initial Value Problems

Consider the system of initial value problems:

$$\frac{d\mathbf{y}}{dt} = \mathbf{F}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0, \tag{4.1}$$

where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}, \qquad \text{and} \qquad \mathbf{F}(t, \mathbf{y}) = \begin{bmatrix} F_1(t, y_1, y_2, y_3, \ldots, y_n) \\ F_2(t, y_1, y_2, y_3, \ldots, y_n) \\ F_3(t, y_1, y_2, y_3, \ldots, y_n) \\ \vdots \\ F_{n-1}(t, y_1, y_2, y_3, \ldots, y_n) \\ F_n(t, y_1, y_2, y_3, \ldots, y_n) \end{bmatrix} \tag{4.2}$$

The solution method for the system of IVP's (4.1) can be found numerically via either Euler's method or Runge-Kutta methods.

4.1.1 Euler's Method

This reads:

$$\begin{aligned} \mathbf{y}_{i+1} &= \mathbf{y}_i + \Delta t\, \mathbf{F}(t_i, \mathbf{y}_i), \\ t_{i+1} &= t_i + \Delta t, \\ i &= 0, 1, 2, \ldots. \end{aligned} \tag{4.3}$$

It is usually convenient to write

$$\dot{\mathbf{y}} \qquad \text{or} \qquad \mathbf{y}' \qquad \text{instead of} \qquad \frac{d\mathbf{y}}{dt}$$

4.1.2 Runge-Kutta Methods

We could get more accurate results if we use Runge-Kutta methods instead. In particular the *fourth-order Runge-Kutta methods* for systems now reads

$$\begin{aligned}
\mathbf{K}_1 &= \Delta t\, \mathbf{F}(t, \mathbf{y}) \\
\mathbf{K}_2 &= \Delta t\, \mathbf{F}\left(t + \frac{\Delta t}{2}, \mathbf{y} + \frac{\mathbf{K}_1}{2}\right) \\
\mathbf{K}_3 &= \Delta t\, \mathbf{F}\left(t + \frac{\Delta t}{2}, \mathbf{y} + \frac{\mathbf{K}_2}{2}\right) \\
\mathbf{K}_4 &= \Delta t\, \mathbf{F}(t + \Delta t, \mathbf{y} + \mathbf{K}_3) \\
\mathbf{y}(t + \Delta t) &= \mathbf{y}(t) + \frac{1}{6}(\mathbf{K}_1 + 2\mathbf{K}_2 + 2\mathbf{K}_3 + \mathbf{K}_4)
\end{aligned} \tag{4.4}$$

Example 4.1 *Repeat example (**??**) with the Runge-Kutta method.*

4.1.3 Higher order IVP's

Consider an ordinary differential equation of order n

$$y^{(n)} = f(t, y, y', y'', \ldots, y^{(n-1)}), \tag{4.5}$$

with initial conditions

$$y(t_0) = \alpha_1, \quad y'(t_0) = \alpha_2, \quad y''(t_0) = \alpha_3, \quad \ldots, \quad y^{(n-1)}(t_0) = \alpha_n. \tag{4.6}$$

Since all the conditions are specified at the same value of t, the equation (4.5) together with conditions (4.6) is an IVP.

Any *nth* order equation of the form (4.5) can always be transformed into a system of n first-order differential equations.

Using the notation

$$y_1 = y, \quad y_2 = y', \quad y_3 = y'', \quad \ldots, \quad y_n = y^{(n-1)}, \tag{4.7}$$

the equivalent system of 1st order equations is

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \\ \vdots \\ \dot{y}_{n-1} \\ \dot{y}_n \end{bmatrix} = \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_n \\ f(t, y_1, y_2, \ldots, y_n) \end{bmatrix}. \tag{4.8}$$

This is the same as the system in (4.1) and hence can be solved using the Euler or Runge-Kutta methods for systems.

Example 4.2 *The differential equations*

$$\ddot{r} = -r\,\dot{\theta}^2 + \frac{G\,M_e}{r^2}, \qquad \ddot{\theta} = -\frac{2\dot{r}\dot{\theta}}{r},$$

where $G\,M_e = 3.9860 \times 10^{14}$*, describe the motion of a spacecraft (falling from space to the Earth) in polar coordinates* (r, θ)*. Solve these equations subject to*

$$\begin{bmatrix} r(0) \\ \dot{r}(0) \\ \theta(0) \\ \dot{\theta}(0) \end{bmatrix} = \begin{bmatrix} 7.15014 \times 10^6 \\ 0 \\ 0 \\ 0.937045 \times 10^{-3} \end{bmatrix}. \tag{4.9}$$

Letting

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} r \\ \dot{r} \\ \theta \\ \dot{\theta} \end{bmatrix}$$

leads to the system (**??**) of example (**??**):

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \\ \dot{y}_4 \end{bmatrix} = \begin{bmatrix} y_2 \\ -y_1\,y_4^2 + G\,M_e\,/\,y_1^2 \\ y_4 \\ -2y_2\,y_4\,/\,y_1 \end{bmatrix}. \tag{4.10}$$

Relevant M-files

```
% RK_Sys.m
clc
clear all
t_0 = 0;     t_n = 1050;     dt = 0.1;
y_0 = [7.15014e14, 0, 0, 0.937045e-3];
%
n = (t_n - t_0)/dt;
% time vector
t = t_0 : dt : t_n;
% initialize solution vector & correct the initial value
y = zeros( length(t) , 4 );
   % y(:,1) = r,
   % y(:,2) = r',
   % y(:,3) = theta
   % y(:,3) = theta'
y(1,:) = y_0;
%
for i = 1 : length(t) - 1 % length(t) here = n+1
    k1 = dt * SysFunc( t(i), y(i,:) );
    k2 = dt * SysFunc( t(i) + dt/2, y(i,:) + k1/2 );
    k3 = dt * SysFunc( t(i) + dt/2, y(i,:) + k2/2 );
    k4 = dt * SysFunc( t(i) + dt, y(i,:) + k3 );
    y(i+1,:) = y(i,:) + 1/6 * ( k1 + 2*k2 + 2*k3 + k4 );
end
```

```
subplot(2 , 2 , 1)
plot(t , y(: , 1) , 'r' , 'LineWidth' , 2 )
ylabel('r(t)' , 'FontSize' , 20)
subplot(2 , 2 , 2)
plot(t , y(: , 2) , 'g' , 'LineWidth' , 2)
ylabel('r''(t)' , 'FontSize' , 20)

subplot(2 , 2 , 3)
plot(t , y(: , 3) , 'm' , 'LineWidth' , 2)
xlabel('t' , 'FontSize' , 20)
ylabel('\theta(t)' , 'FontSize' , 20)
subplot(2 , 2 , 4)
plot(t , y(: , 4) , 'LineWidth' , 2)
xlabel('t' , 'FontSize' , 20)
ylabel('\theta''(t)' , 'FontSize' , 20)

% SysFunc.m
function F = SysFunc( t , w )
% rhs of system of differential equations
F = zeros(1,4);
F(1) = w(2);
F(2) = - w(1) * w(4)^2 + 3.9860e14 / w(1)^2;
F(3) = w(4);
F(4) = - 2*w(2) * w(4) / w(1);
```

4.2 Problems in Finance, Random Numbers

Black-Scholes models are parabolic type partial differential equations similar to the pde's of the previous lecture. For example, the one-dimensional Black-Scholes equation reads:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - D)S\frac{\partial V}{\partial S} - rV = 0, \tag{4.11}$$

where V is the derivative type, S is the underlying asset (or stock), σ is the constant volatility, r is the interest rate and D is a dividend.

The solution process via finite difference methods is the same. Random numbers play an important role in modeling and numerical solution of probabilistic problems in finance. We give a preview of random number generation in MATLAB.

Get MATLAB's help on **round, floor, ceil, fix, rand** and **rem**, with e.g.

```
>> help floor
```

4.2.1 Rounding

round(m) rounds the number m to the nearest integer.

```
>> round(19.41)
>> round(19.67)
>> round(19.47)
>> round(19.5)
```

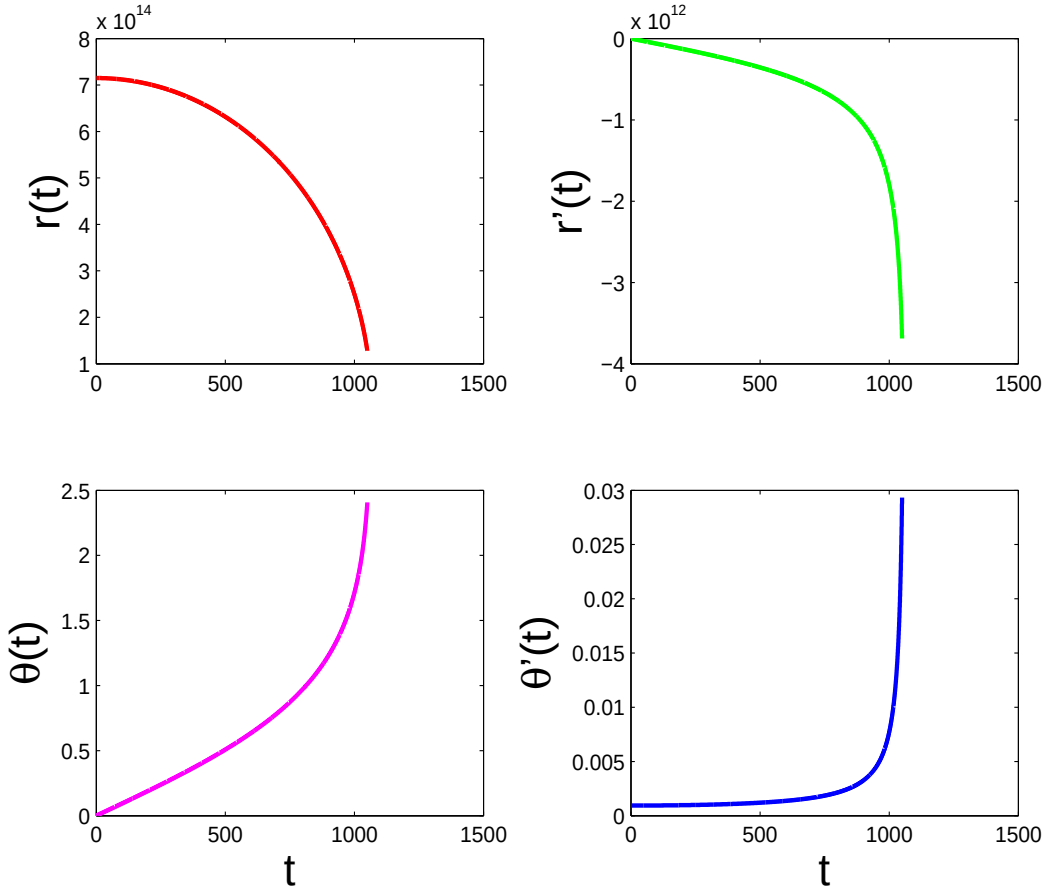


Figure 4.1: Graphical representation of solution.

How about say rounding m to N decimal places? Simply multiply m by 10^N, round this new value then divide the answer by 10^N. For example round 13.125785216 to 3 decimal places:

```
>> m = 13.125785216
>> m1 = 13.125785216*1000;
>> m2 = round(m1);
>> m_3dp = m2/1000
```

You could combine all these steps in one command by typing:

```
>> round(13.125785216*1000)/1000
```

4.2.2 Truncating

floor

floor(m) truncates m to the nearest integer less than or equal to m: i.e. floor(m) $\leq m$.

```
>> floor(3.9)
>> floor(-3.1)
```

ceil

ceil(m) truncates m to the nearest integer greater than or equal to m: i.e. ceil$(m) \geq m$.

```
>> ceil(3.9)
>> ceil(-3.1)
```

fix

fix(m) truncates m to the nearest integer whose absolute value is less than or equal to the absolute value of m: $|\text{fix}(m)| \leq |m|$. Thus if $m > 0$ fix is the same as floor whereas if $m < 0$ fix acts like ceil.

```
>> fix(3.9)
>> fix(3.1)
>> fix(-3.1)
```

4.2.3 Random numbers:

To obtain help on MATLAB commands related to random number generation, type >> help rand

The command **rand** generates uniformly distributed pseudo-random numbers in the interval [0,1), including 0 but **not including 1**.

On the other hand **rand(m,n)** returns an $m \times n$ matrix with random entries, thus it is equally easy to create vectors and matrices of random elements in MATLAB.

```
>> rand(1,20)
>> rand(5,8)
```

It is usually desirable to generate random numbers in more general intervals of the form (L, R), $[L, R)$, $(L, R]$ and $[L, R]$ rather than the standard interval $[0, 1)$.

By definition, for any positive number b, the command **b*rand** gives a random number in the interval $[0, b)$ and similarly **b*rand(m,n)** returns an $m \times n$ array of numbers, each of which is in the interval $[0, b)$.

```
>> 13*rand(15,1)
>> 13*rand(9,7)
```

If on the other hand, b is negative, then the random numbers lie in the interval $(-|b|, 0]$

```
>> -13*rand(15,1)
>> -13*rand(9,7)
```

It follows that **a + b*rand** gives a random number in the interval $[a, a + b)$ and that **a + b*rand(m,n)** returns an $m \times n$ array of numbers, each of which is in the interval $[a, a + b)$.

```
>> 6 + 7*rand(1,30)
>> 6 + 7*rand(11,8)
```

Random Integers

To get random integers use **floor, ceil** or **fix**.

Random non-negative integers with floor & fix

Suppose $P > 0$ is an integer: Since **P*rand** returns decimal numbers from zero to just below P, **floor(P*rand)** gives random integers from the set $\{0, 1, 2, \ldots, P-1\}$.

```
>> floor(5*rand(1,30))
```

If P is a decimal, then **floor(P*rand)** gives random integers from the set $\{0, 1, 2, \ldots, floor(P)\}$.

```
>> floor(5.3*rand(1,30))
```

If L is any given integer, we can easily create random integers in the interval $[L, L+P)$ using **L+floor(P*rand)**

```
>> 2 + floor(9*rand(10,7))
>> 2 + floor(9.4*rand(10,7))
```

Since $P > 0$ the command **fix** will work in the same way as **floor**.

```
>> fix(5*rand(1,30))
>> fix(5.3*rand(1,30))
>> 2 + fix(9*rand(10,7))
>> 2 + fix(9.4*rand(10,7))
```

Random non-negative integers with ceil

Suppose again that $P > 0$ is an integer. Since **P*rand** returns decimal numbers from zero to just below P, **ceil(P*rand)** gives random integers from the set $\{0, 1, 2, \ldots, P\}$.

```
>> ceil(5*rand(1,30))
```

If P is a decimal, then **ceil(P*rand)** gives random integers from the set {0, 1, 2, ..., ceil(P)}.

```
>> ceil(5.3*rand(1,30))
```

If L is any given integer, we can easily create random integers in the interval $(L, L + ceil(P)]$ or in $[L, L + ceil(P)]$ using **L+1 + ceil(P*rand)** or **L + ceil(P*rand)** respectively.

```
>> 2 + ceil(9*rand(10,7))
>> 2 + ceil(9.4*rand(10,7))
```

Random negative integers with ceil & fix

Suppose $N < 0$ is an integer: Since **N*rand** returns decimal numbers from just to the right of below N up to zero, **ceil(N*rand)** gives random integers from the set $\{N+1, N+2, N+3, \ldots, 0\}$.

```
>> ceil(-5*rand(1,30))
```

If N is a decimal, then **ceil(N*rand)** gives random integers from the set $\{ceil(N), ceil(N)+1, ceil(N)+2, \ldots, 0\}$.

```
>> floor(-5.3*rand(1,30))
```

If L is any given integer, we can easily create random integers in the interval $[ceil(N) + L, L]$ using **L+ceil(P*rand)**

```
>> 2 + ceil(-9*rand(10,7))
>> -2 + ceil(-9*rand(10,7))
>> 2 + ceil(-9.4*rand(10,7))
```

Since $N < 0$ the command **fix** will work in the same way as **ceil**.

```
>> fix(-5*rand(1,30))
>> fix(-5.3*rand(1,30))
>> 2 + fix(-9*rand(10,7))
>> 2 + fix(-9.4*rand(10,7))
```

Random negative integers with floor

Suppose again that $N < 0$ is an integer. Since **N*rand** returns decimal numbers from just to the right of N up to zero, **floor(N*rand)** gives random integers from the set $\{N, N+1, N+2, \ldots, 0\}$.

```
>> floor(-5*rand(1,30))
```

If N is a decimal, then **floor(N*rand)** gives random integers from the set $\{floor(N), floor(N)+1, floor(N)+2, \ldots, 0\}$.

```
>> floor(-5.3*rand(1,30))
```

If L is any given integer, we can easily create random integers in the interval $[floor(N) + L, L]$ using **L + floor(N*rand)**.

```
>> 2 + floor(-9*rand(10,7))
>> -2 + floor(-9*rand(10,7))
>> 2 + floor(-9.4*rand(10,7))
```

randn

m + s.*randn generates Normally distributed random numbers with mean m and standard deviation s.

See graphical illustration below.

4.2.4 Remainders:

```
>> help rem
```

rem(b,a) gives the remainder when b (numerator) is divided by a (denominator).

```
>> rem(13,6)
>> rem(2,9)
>> rem(0,27)
>> rem(5,0)
```

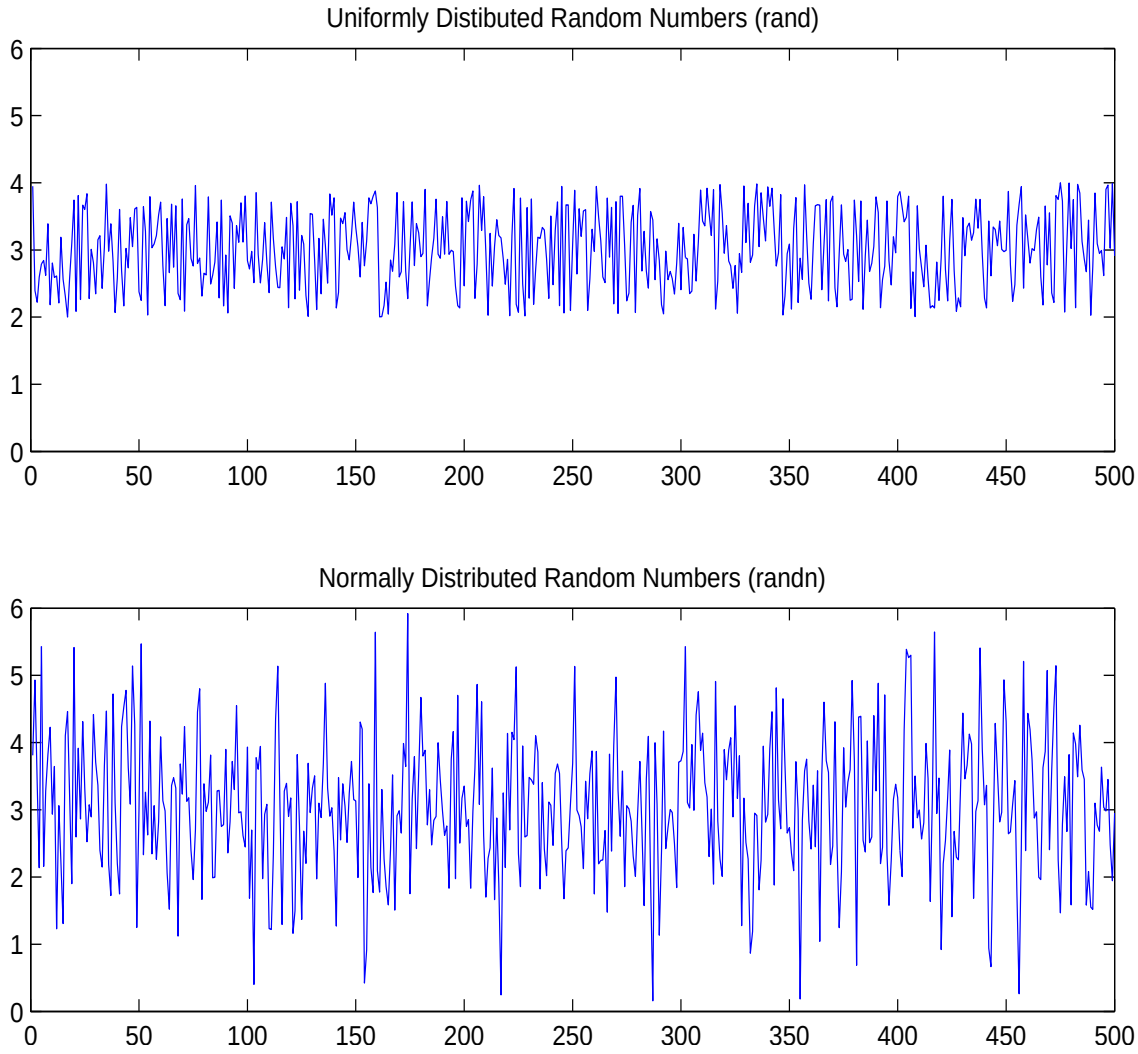


Figure 4.2: Graphical illustration of **rand** and **randn**

4.3 Stochastic ODE's in finance

The stock price S is modelled using a stochastic differential equation (SDE):

$$dS = \mu(S,t)\,S\,dt + \sigma(S,t)\,dX_1, \tag{4.12}$$

where dX_1 is a normally distributed random variable (mean 0 and variance dt), μ is the drift and σ is the volatility.

The SDE for the interest rate r is given by:

$$dr = u(r,t)\,dt + w(r,t)\,dX_2 \tag{4.13}$$

where dX_2 is the normally distributed random variable (mean 0 and variable dt) and $u(r,t)$ and $w(r,t)$ are (as of now) unspecified functions. The stock market and the fixed income market are related to each other and the correlation is given by the relationship:

$$E(dX1, dX2) = \rho(S,r,t)\,dt.$$

Furthermore, we assume continuous dividends and that the bond pays a coupon at an annual rate k and that at expiry the convertible returns Z unless it has been converted into n shares in the meantime. Finally, we assume that there are no transaction costs. Then the PDE for the convertible bond V becomes

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + \rho\sigma S w \frac{\partial^2 V}{\partial S \partial r} + \frac{1}{2}\omega^2 \frac{\partial^2 V}{\partial r^2} (r - D) S \frac{\partial V}{\partial S} (u - \lambda w) \frac{\partial V}{\partial r} - rV + kV = 0, \quad (4.14)$$

where $\lambda = \lambda(S, r, t)$ is the market price of risk. The payoff function is given by:

$$V(S, r, T) = \max(nS, Z).$$

Since the bond can be converted into n shares of the underlying stock at any time before expiry, the price V must satisfy the so-called conversion constraint:

$$V(S, r, t) \geq nS.$$

4.3.1 Some Stochastic ODE Models

The Merton model

$$dr(t) = \mu, dt + \sigma\, dW(t),$$

where μ and σ are constant, $W(t)$ is the standard Brownian motion and the risk market premium λ is constant.

The Vasicek model

$$dr(t) = K(\theta - r(t)), dt + \sigma\, dW(t),$$

where K, θ and σ are constant, $W(t)$ is the standard Brownian motion and the risk market premium λ is constant.

Cox, Ingersoll and Ross (CIR)

$$dr(t) = K(\theta - r(t)), dt + \sigma \sqrt{r(t)}\, dW(t),$$

where K, θ and σ are constant, $W(t)$ is the standard Brownian motion and the risk market premium $\lambda = \lambda_0 \sqrt{r(t)}$.

The HullWhite model

$$dr(t) = (\theta(t) - K(t))\, r(t), dt + \sigma(t)\, r^{\beta}(t)\, dW(t),$$

The risk market premium $\lambda = \lambda_0 r^{\gamma}(t)$ with $\lambda_0, \gamma \geq 0$.

Reference

D.J. Duffy, Finite Difference Methods in Financial Engineering, Wiley, 2006.