Numerical Methods in MATLAB/Octave for Engineers: A course by Suddhasheel Ghosh

Technical Report · January 2018

DOI: 10.13140/RG.2.2.26528.53762

Suddhasheel Ghosh
MGM University
31 PUBLICATIONS **199** CITATIONS

Numerical Methods in MATLAB/Octave for Engineers

A course by

Suddhasheel Ghosh, PhD

Numerical Methods with MATLAB/Octave for Engineers

January 2 - 6, 2017

Department of Civil Engineering

Jawaharlal Nehru Engineering College

Preface

This material was developed in connection with the workshop titled "Numerical Methods with MATLAB/Octave for Engineers", held at Jawaharlal Nehru Engineering College, by Dr. Suddhasheel Ghosh. The workshop was organised by the Department of Civil Engineering.

The main objective of this workshop was to create the awareness about the vectorised programming techniques followed in MATLAB/Octave. While the candidate is made aware of the vectorised techniques, Dr. Suddhasheel Ghosh deals with the concept of Numerical Methods as a case study and the techniques to convert mathematical ideas into computer programs.

Apart from the popular mathematical topics like Numerical Differentiation, Numerical Integration, Initial value problems and Boundary value problems, the material generates insight about the concepts of Artifical Neural Networks and Genetic Algorithms.

It is currently planned to extend this material to a more complete form, which can then be used for a full semester course in computational programming at the post-graduate levels of Engineering education.

Who am I?

I am a blogger, humorist, researcher, motivator and a teacher. That is me! I feel and write about various topics, political, apolitical and also keep on posting various information related videos on several channels through the social network.

Programming fascinates me! I often feel that it gives me wings. Since the time I first touched a computer (the BBC Micro in 1987), I have been curious about how to simplify things on the computer i.e. writing programs that would simply do more than the ordinary.

The support and encouragement of my parents, especially my mother, towards my interest in programming pushed me to newer heights. I therefore can boast of an interesting repertoire of linguistic delights starting from BASIC (Beginner's All Purpose Symbolic Instruction Code), to the latest fourth generation programming languages like MATLAB/Octave and Python.

All the code presented in this material have been written by me. For any faults, I own their responsibility; for goodness, my parents and teachers get the credit.

My motto has beem simple - *If I can write the formula on my notepad, I can write a program for evaluating it!*

Oh yes, before I forget, all this was typeset on LaTeX.

Suddhasheel Ghosh, PhD

Contents

Preface **i**

Who am I? **iii**

1 Basic Introduction to MATLAB/Octave **1**

1.1 About MATLAB/Octave 1

1.1.1 MATLAB 1

1.1.2 Octave 2

1.2 Data types in MATLAB/Octave 2

1.2.1 Basic data types 2

1.2.2 The advanced data storage system 2

1.3 Special Matrices in MATLAB/Octave 4

1.3.1 The zero matrix 4

1.3.2 The one matrix 4

1.3.3 The Magic Square matrix 4

1.3.4 Matrix with random numbers 5

1.4 Matrix operations in MATLAB/Octave 5

1.4.1 Creating a matrix - with data 5

1.4.2 Finding the trace, determinant and inverse 7

1.4.3 Vectorized coding with matrices 7

1.4.4 Generating sequences 8

1.5 Graph Plotting in MATLAB/Octave 9

1.5.1 Sub-plots 12

1.5.2 3D surface plots 13
1.5.3 Saving a figure in MATLAB/Octave 15
1.6 Symbolic math 16
1.7 Coding functions in MATLAB/Octave 17
1.7.1 Function handles 18

2 Linear Algebra with MATLAB/Octave **21**
2.1 Linear System of equations 21
2.1.1 Solving a linear system 22
2.1.2 Gauss Siedel Method 27
2.1.3 LU Decomposition 30

3 Root finding methods in MATLAB/Octave **33**
3.1 Introduction 33
3.2 Fixed point iteration method 34
3.3 Bisection Method 35
3.4 Secant method 38
3.5 Regula-Falsi method 40
3.6 Ridders' Method 42
3.7 Newton Raphson method 43

4 Interpolation **47**
4.1 Introduction 47
4.2 Lagrange's interpolation 48
4.3 Newton's interpolation 50
4.3.1 Divided differences 50
4.3.2 Newton's polynomial 51

5 Numerical Differentiation and Integration **55**
5.1 Taylor's Series 55
5.1.1 The concept of differences 55
5.1.2 The representation of derivatives 56
5.1.3 The series 56
5.2 Finite difference approximation of derivatives 57

5.3 Richardson’s Extrapolation 57
5.3.1 MATLAB/Octave code 58
5.4 Numerical Integration 59
5.4.1 Integration by approximation 59

5 Initial and Boundary Value Problems 65
5.1 Initial Value Problem 65
5.1.1 Euler method 65
5.1.2 Stability of the Euler’s method 69
5.1.3 Runge-Kutta methods 70
5.1.4 Adam’s - Bashforth Method 74
5.2 Boundary Value Problems 77
5.2.1 Finite difference method 77
5.2.2 Shooting method 81

5 Artificial Neural Networks 85
5.1 Introduction 85
5.2 Data introduction 85
5.2.1 Data characteristics 86
5.2.2 Summary Statistics 87
5.2.3 Variable information 87
5.3 Data processing 88
5.3.1 Reading the data 88
5.3.2 Data scaling 88
5.4 Setting up the network 89
5.4.1 Layers 89
5.4.2 Existing network models 90
5.4.3 Activation functions 90
5.4.4 Configuring the network 90
5.4.5 Training, testing and validation 91
5.4.6 Random numbers and network solutions 94
5.4.7 Scaling back the outputs 96

5 Optimization with GA in MATLAB/Octave **97**

5.1 Introduction 97

5.2 Optimization Problem 97

5.2.1 Unconstrained and constrained optimization 98

5.3 Coding scenario 98

5.3.1 Designing the cost function 98

5.3.2 Running the GA without any constraints 99

5.3.3 Adding constraints 100

5.3.4 Adding bounds 101

5.4 Multi-objective coding 102

List of Figures

1.1 Two graphs in the same axis 10
1.2 MATLAB/Octave figure with subplots 13
1.3 Plot of the Rastrigin function 14

5.1 Plot of the Euler method for different values of h 70
5.2 Plot demonstrating the unstability of the Euler's method 71
5.3 Plot for the RungeKutta Method of order 4 73
5.4 Plot of the Adam's Bashforth two-step method for $y' = -15y, \quad y(0) = 1$, for different values of h. 76
5.5 Plot of the Finite Difference Method for Problem iii. 80

5.1 UCI Machine Learning Repository - Concrete data set: Page screenshot 86
5.2 Neural network training user interface - in MATLAB 92
5.3 Regression plots for the various parts of the dataset 93
5.4 Performance plot of the neural network, as presented in MATLAB 93

5.1 3D plot and contours of Himmelblau's function 98

List of Tables

1.1 Basic data types in MATLAB/Octave 3
1.2 Advanced storage types in MATLAB/Octave 3

5.1 Summary statistics for the UCI Machine Learning Concrete dataset 87
5.2 Variables in the dataset: description as per the column number . . 87

Chapter 1

Basic Introduction to MATLAB/Octave

1.1 About MATLAB/Octave

1.1.1 MATLAB

According to Wikipedia (MATLAB (matrix laboratory) is a multi-paradigm numerical computing environment and fourth-generation programming language. A proprietary programming language developed by MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, C#, Java, Fortran and Python.

Although MATLAB is intended primarily for numerical computing, an optional toolbox uses the MuPAD symbolic engine, allowing access to symbolic computing abilities. An additional package, Simulink, adds graphical multi-domain simulation and model-based design for dynamic and embedded systems. (Source: `https://en.wikipedia.org/wiki/MATLAB`)

1.1.2 Octave

GNU Octave is software featuring a high-level programming language, primarily intended for numerical computations. Octave helps in solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with MATLAB. It may also be used as a batch-oriented language. Since it is part of the GNU Project, it is free software under the terms of the GNU General Public License. (Source Wikipedia: `https://en.wikipedia.org/wiki/GNU_Octave`)

1.2 Data types in MATLAB/Octave

It is to be remembered that all data in MATLAB / Octave are represented as matrices. There are some basic data types, which can be – in programming parlance – called atomic, cell arrays, structures and objects.

1.2.1 Basic data types

The basic data types present in MATLAB/Octave are presented in Table 1.1.

It is to be noted here that the default storage type is 'double' unless otherwise mentioned.

1.2.2 The advanced data storage system

The advanced mode of data storage structures are presented in Table 1.2.

S.No.	Data type	Memory size
1.	int8	8-bit signed integer
2.	uint8	8-bit unsigned integer
3.	int16	16-bit signed integer
4.	uint16	16-bit unsigned integer
5.	int32	32-bit signed integer
6.	uint32	32-bit unsigned integer
7.	int64	64-bit signed integer
8.	uint64	64-bit unsigned integer
9.	single	single precision numerical data
10.	double	double precision numerical data
11.	logical	logical values of 1 or 0, represent true and false respectively
12.	char	character data (strings are stored as vector of characters)

Table 1.1: Basic data types in MATLAB/Octave

S.No.	Data type	Description
1.	cell array	array of indexed cells, each capable of storing an array of a different dimension and data type
2.	structure	C-like structures, each structure having named fields capable of storing an array of a different dimension and data type
3.	function handle	pointer to a function
3.	user classes	objects constructed from a user-defined class
4.	java classes	objects constructed from a Java class

Table 1.2: Advanced storage types in MATLAB/Octave

1.3 Special Matrices in MATLAB/Octave

1.3.1 The zero matrix

A matrix with all its elements as zero is called the zero matrix. The function for generating zero matrices is `zeros`.

```
%this creates a 2 x 3 zero matrix
B = zeros(2,3)
%this creates a 3 x 3 zero matrix
C = zeros(3)
```

1.3.2 The one matrix

A matrix with all its elements as one is called the one matrix. The function for generating one matrices is `ones`.

```
%this creates a 2 x 3 one matrix
B = ones(3,2)
%this creates a 4 x 4 one matrix
C = ones(4);
```

1.3.3 The Magic Square matrix

A magic square is a $n \times n$ square grid (where n is the number of cells on each side) filled with distinct positive integers in the range $1, 2, ..., n^2$ such that each cell contains a different integer and the sum of the integers in each row, column and diagonal is equal. The sum is called the magic constant or magic sum of the magic square. A square grid with n cells on each side is said to have order n.

With respect to magic sum, the problem of magic squares only requires the sum of each row, column and diagonal to be equal, it does not require the sum to be a particular value. (Src. Wikipedia).

In MATLAB/Octave, magic squares are generated by using the `magic` function.

```
%this will generate a 4 x 4 magic matrix
A = magic(4)
```

1.3.4 Matrix with random numbers

Matrices with random numbers between 0 and 1 can be generated by using the `rand` function.

```
%this will generate a matrix of order 3
A = rand(3)
%this will generate a matrix of order 4 x 3
B = rand(4,3)
```

1.4 Matrix operations in MATLAB/Octave

1.4.1 Creating a matrix - with data

In MATLAB/Octave the creation of a matrix is done rowwise, with each of its elements separated by commas (optional), and each row separated by a semicolon (;). For example, in order to create a matrix:

$$A = \begin{pmatrix} 4 & 7 & 9 & 8 \\ 5 & 2 & 1 & 2 \\ 7 & 1 & 9 & 0 \end{pmatrix}$$

we have to give the command:

```
A = [4, 7, 9, 8; 5, 2, 1, 2; 7, 1, 9, 0]
```

1.4.1.1 Accessing elements of the matrix

Accessing the elements of the matrix is easy. The rows are considered as the first dimension and the columns are in the second dimension. Therefore, the following commands illustrate the various uses using the dimensions.

```
%the fourth element in the second row
A(2,4)
%the 3rd element in the first row
A(1,3)
%all elements in the third column
A(:,3)
%all elements in the second row
A(2,:)
```

We can see that the colon (:), is a representative for ‘all’ in the above examples.

1.4.1.2 Simple mathematical operations

Simple mathematical operations like addition, subtraction and multiplication happen normally. In multiplication, there is a provision for element-wise multiplication, also.

```
A = [4, 5, 6; 2, 3, 4]
B = [5, 2, 1; 8, 2, 2]
C = [2, 3; 4, 1; 6, 2]
D = A + B
E = A - B
F = A * C
G = C * A
%element-wise multiplication
H = A *. B
```

Exercise Find a technique to declare two vectors and find their dot product. Also find their cross product.

1.4.2 Finding the trace, determinant and inverse

The trace, determinant and inverse are mathematical operations which work only on square matrices.

The trace is the sum of the diagonal elements of the matrix (the principal diagonal).

```
% this will create a magic matrix of order 4
A = magic(4)
% this will create a trace
trace(A)
```

The determinant and inverse are respectively calculated using `det` and `inv` functions.

```
A = [5, 6, 2; 8, 2, 4; 6, 4, 4];
% this gives the determinant of the matrix
det(A)
% this stores the determinant of the matrix into a variable
d = det(A)
% this calculates the inverse of the matrix
inv(A)
%this stores the inverse of the matrix into a variable
a_inv = inv(A)
```

1.4.3 Vectorized coding with matrices

Unlike 3rd generation programming languages like C/C++, MATLAB/Octave have a much better programming approach. For example, if there is a matrix **A** of order $m \times n$, and from each of the elements one has to subtract 2, in C/C++ one has to write a nested loop. In MATLAB/Octave, however, one will write a single statement:

```
A = A + 2;
```

Similar codes can be written for addition and elementwise division, multiplication, power etc. Further, most of the functions which are pre-programmed in MATLAB/Octave are vectorised i.e they can have their effect on an entire matrix, rather than a single element.

1.4.4 Generating sequences

MATLAB/Octave can generate sequences of numbers given a starting number, a step size and a final number. These numbers can increase or decrease.

To generate a sequence from -5 to 5, in steps of 0.01, we have to follow the following steps, in both increasing and decreasing orders.

```
A = [-5:0.01:5]
B = [5:-0.01:-5]
```

In case we know the number of terms that we wish to divide an interval into, we use the `linspace` command. For example

```
%this will generate 100 numbers without mentioning
A = linspace(-5,5)
%this will generate 10 numbers
B = linspace(-5,5,10)
```

1.5 Graph Plotting in MATLAB/Octave

Graph plotting in MATLAB/Octave requires data for the x, y and sometimes the z axis. Function plots are drawn using `plot` and scatter plots are generated using `scatter`. For example, if one wants to draw the graph of a sine curve, the following MATLAB/Octavecode has to be written

```
%this generate a sequence of numbers
x = -2*pi:0.01:2*pi
y = sin(x)
plot(x,y)
```

Exercise Plot the following curves:

i. $\cos x$
ii. $\tan x$
iii. $\cot x$

We will now try to plot two different curves in the same plot.

```
%generate numbers from -2 pi to 2 pi in steps of 0.01
x = -2*pi:0.01:2*pi
%generate sine values
y1 = sin(x)
%generate cosine values
y2 = cos(x)
%generate the plot
plot(x, y1, x, y2)
%add a legend
legend('sine', 'cosine')
%add x title
xlabel('x values')
ylabel('y values')
```

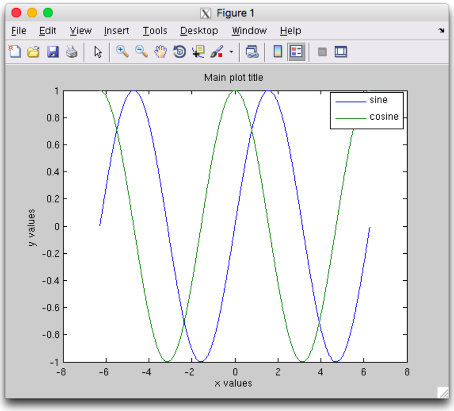


Figure 1.1: Two graphs in the same axis

```
%set main plot title
title('Main plot title')
```

Exercise Create a plot of the function $f(x) = tan(sin(x)) - sin(tan(x))$.

1.5.0.1 Line and Marker style

Line styles

Specifier	Line Style
-	Solid line (default)
- -	Dashed line
:	Dotted line
-.	Dash-dot line

Line color

Specifier	Color
y	yellow
m	magenta
c	cyan
r	red
g	green
b	blue
w	white
k	black

Marker style

Specifier	Marker
o	Circle
+	Plus sign
*	Asterisk
.	Point
x	Cross
s	Square
d	Diamond
caret	Upward pointing triangle
v	Downward pointing triangle
>	Right-pointing triangle
<	Left-pointing triangle
p	Pentagram
h	Hexagram

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));

figure
plot(x,y,'--gs',...
     'LineWidth',2,...
```

```
        'MarkerSize',10,...
        'MarkerEdgeColor','b',...
        'MarkerFaceColor',[0.5,0.5,0.5])
```

Exercise Try the given MATLAB/Octave code

1.5.1 Sub-plots

In MATLAB/Octave sub-plots are thought to be a row by column arrangement. Therefore, if in a single figure one wants four different plots, with two rows and two columns, the `subplot` function has to be used. The index of the plots varies row wise i.e. in the first row, the plots are numbered 1 and 2, and in the second row they are numbered 3 and 4.

In the following example code, we attempt to plot four different graphs in the same window.

```
x = [-2*pi:0.01:2*pi]
ya = sin(x)
yb = cos(x)
yc = tan(x)
yd = ones(size(x))./ tan(x)
%opens a new window
figure
subplot(2,2,1)
plot(x,ya)
subplot(2,2,2)
plot(x,yb)
subplot(2,2,3)
plot(x,yc)
subplot(2,2,4)
plot(x,yd)
```

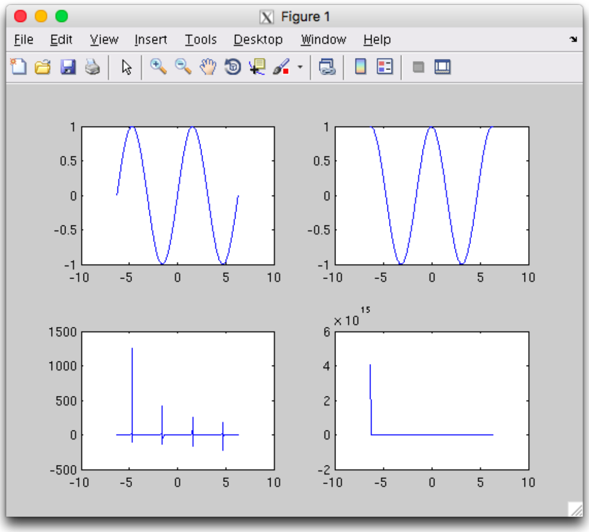


Figure 1.2: MATLAB/Octave figure with subplots

In this code, the `figure` function, creates a new figure window.

Exercise Create six different functions and create a figure with 6 plots. Each of these subplots should have a label in x and y axes and a title.

1.5.2 3D surface plots

In MATLAB/Octave the 3D surface plots are created using a function `meshgrid`. The mesh has to be generating using this function, and at each "node" of the mesh, the z value has to be evaluated. We try to plot the Rastrigin's function

$$f(x, y) = 20 + \left(x^2 - 10\cos(2\pi x)\right) + \left(y^2 - 10\cos(2\pi y)\right)$$

```
%generate the x factor
x = [-5.16:0.15:5.16];
```

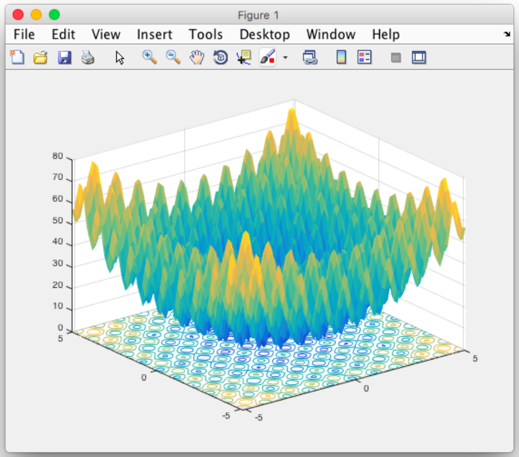


Figure 1.3: Plot of the Rastrigin function

```
%generate the y factor
y = [-5.16:0.15:5.16];
%generate the meshgrid
[xx, yy] = meshgrid(x,y);
%generate the mesh node values
zz = 20 + (xx.^2 - 10 * cos(2*pi*xx)) + (yy.^2 - 10 * cos(2*pi*yy));
%generate the surface plot
surfc(xx,yy,zz, ...
     'EdgeColor','none', ...
     'LineStyle','none', ...
     'FaceLighting','phong');
```

Exercise

i Create a sequence x from -5 to 5.

ii Create a sequence y from -5 to 5.

iii Create a meshgrid using x and y values.

iv Use the function

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

v Use surface plotting to create the plot of above function.

1.5.3 Saving a figure in MATLAB/Octave

One of the most important requirements in journals and theses is the inclusion of plotted figures. Some reputed institutes / universities require that these figures be in vectorised formats, and some other require the figures to be more than 300dpi in raster format. With the printers becoming more and more cheaper, the journal publishers and universities are becoming much more demanding on the quality of the images included.

1.5.3.1 Raster and Vector formats

Raster formats In simple language, raster formats are big in size. For an image of width w and height h, each pixel has a single value, which has to be represented as a combination of red, green and blue values, unless otherwise stated explicitly to be a black and white images. Some of the popular raster formats required by journals are BMP, TIFF and JPEG. Some of the journals also accept PNG.

To save an image for a particular plot, we want the following (a) filename, (b) resolution or dpi, and (c) the format. Sometimes, the format is explicitly mentioned by the filename extension, but it is better to be on the safe side.

```
%save a plot in the jpeg format
print('myPlot', '-djpeg', '-r300') %300 dpi plot
%save a plot in the png format
print('myPlot', '-dpng', '-r600') %600 dpi plot
%save a plot in the tiff format
print('myTIFPlot', '-dtiff', '-r300')
```

```
%bigger size, noncompressed TIFF
print('myTIFFPlot', '-dtiffn', '-r600')
%save a plot in BMP format
print('mybmpplot', '-dbmp', '-r600')
```

Vector formats Images in vector formats are *generally* smaller in size compared to the sizes of their raster counterparts. We say generally because if a plot contains a large number of nodes, it is much useful to save it in a raster format, to avoid problems with rendering later. Some of the examples of raster formats are PDF and EPS.

The advantage with the vector formats is that they can be scaled, without any distortion in the appearance of the image.

```
%save a plot in the PDF format
print('meraPDFWalaPlot', '-dpdf')

%save a plot in the EPS format
print('meraEPSPlot', '-deps') %l3 b & w
print('meraEPSPlot', '-depsc') %l3 colour
print('meraEPSPlot', '-deps2') %l2 b & w
print('meraEPSPlot', '-depsc2') %l2 colour
```

Exercise Plot the Rastrigin's and the Himmelblau's functions and save the images in raster and vector formats.

1.6 Symbolic math

One of the interesting features of MATLAB/Octave is the symbolic toolbox. This toolbox enables the user to type algebraic expressions. This enables operations like differentiation and integration. Also, the variables in the expression can be substituted with values using the `subs` functions.

```
%this declares two symbols x and y
syms x y
%this declares an expression
s = x^3 + 3 * x^2 - 5 * x + 15
%this differentiates the expression s
sd = diff(s)
%this integrates the expression s
si = int(s)
%this substitutes the value 2 in place of x and
%evaluates the expression
sde = eval(subs(sd, x, 2))
sie = eval(subs(si, x, 2))
```

Exercise Create two functional expressions g and h. Use MATLAB/Octave help for the `jacobian` function to compute the Jacobian of both the functional expressions. For this, take the following steps:

i) Create two symbols x and y
ii) Create expressions f and g using x and y terms.
iii) Create a matrix X using f and g, e.g. $\texttt{X} = [\texttt{f}; \texttt{g}]$
iv) Create a matrix of variables using x and y, e.g. $\texttt{V} = [\texttt{x}, \texttt{y}]$
v) Use the `jacobian` function to calculate the Jacobian.

1.7 Coding functions in MATLAB/Octave

In MATLAB/Octave the functions a written using the following syntax:

```
function [out_vars_list] = function_names(in_vars)
```

In the above definition, `out_vars` denotes the list of output variables separated by commas, and `in_vars` denotes the list of input variables separated by commas.

We give the following code to exemplify function coding in MATLAB/Octave.

```
function [f, g] = first_function(x, y)
f = x^2 * y + y^2 * x;
g = x^3 * y^2 - x^2 * y;
end
```

We can observe here that unlike earlier, we have put semicolons (;) at the end of statements. This avoids the "echo" of the output into the terminal, during the execution of the function. This function can be saved into a file named `first_function.m`.

This MATLAB/Octave function can be called from the terminal as follows:

```
[f,g] = first_function(x,y)
```

In professional practice, each MATLAB/Octave function is saved in a separate file named after the function name. However, sometimes it is convenient to put multiple functions in a single file, while working on a large project, depending on whether other parts of the project would be using the functions or not.

1.7.1 Function handles

The function saved at `first_function.m`, can be converted to a function handle. For example:

```
%this creates the function handle
f = @first_function
%to use the function handle
x = 2; y = 2;
[F, G] = f(x,y)
```

Sometimes some variable values are generated during the execution of the function. Therefore, in such cases, it makes more sense to pass the function handles rather than the variables themselves.

1.7.1.1 Anonymous function handles

Anonymous function handles can be created in MATLAB/Octave, as well. These functions are those which are not stored in a MATLAB/Octave program file. For example, we can create an anonymous function, for calculating the value of the function $f(x) = \frac{x^2-25}{x-5}$:

```
BB = @(x) (x.^2 - 25)/(x-5);
```

Now, for using this anonymous function, to calculate the value of the function, we can use:

```
SS = BB(4)
```

Chapter 2

Linear Algebra with MATLAB/Octave

In the last chapter, we learnt how to operate matrices in MATLAB/Octave. We will now take advantage of the this

2.1 Linear System of equations

A linear system of equations is given as follows:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\
\vdots &= \vdots \\
a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= b_n
\end{aligned}$$

This system of equations can be written in the matrix form as

$$\mathbf{Ax} = \mathbf{B}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \dots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

2.1.1 Solving a linear system

The linear system can be solved for **x** if one can compute the inverse of **A**. Therefore, we would have

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

2.1.1.1 Gaussian Elimination Method

The Gaussian Elimination Method involves the computation of the inverse using row-transformations and then multiplication of the inverse matrix with the "result vector" to compute the values of the variables.

To compute the inverse of a given matrix A, we have to juxtapose an identity matrix of the same order on the right side of A. So, if **A** is a square matrix of order n and **I** is the identity matrix of order n, then we create a new matrix, by juxtaposing **A** with **I**. We call this new matrix as **B**. The elements of the new matrix are referred to as b_{ij}, and it is known that the order of **B** is $n \times (2n)$.

$$\mathbf{B} = \left[\begin{array}{cccc|cccc} a_{11} & a_{12} & \dots & a_{1n} & 1 & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & a_{2n} & 0 & 1 & \dots & 0 \\ \vdots & \dots & \ddots & \vdots & \vdots & \dots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & 0 & 0 & \dots & 1 \end{array}\right]$$

The aim is to use simple row transformations, and convert the first left n columns of $\mathbf{B}$ to an identity matrix of order n. We will begin with converting the first element b_{11} to 1. To do this, we will have to divide the entire row by b_{11}. This is achieved by the following mathematical transformation:

$$b_{1j} = \frac{b_{1j}}{b_{11}}, \forall j = 1, \ldots, 2n$$

In order to transform the elements below b_{11} to zero, we have to use the following mathematical transformation, for each row beginning from the second.

$$\text{For each row } i > 1,\ b_{ij} = b_{ij} - b_{i1} * b_{1j}, \forall j = 1, \ldots, 2n$$

To do this in MATLAB/Octave, we have to write

```
for i=2:n
B(i,:) = B(i,:) - B(i,1) * B(1,j)
```

In this entire process, row 1 is called the pivot row, and the element a_{11} is called the pivot element. In the complete algorithm, every diagonal element is taken as the pivot element and the row transformations are conducted. These steps convert the first n columns of the matrix $\mathbf{B}$ into an identity matrix of order n. The last n columns represent the inverse of the matrix $\mathbf{A}$.

This entire algorithm is summarised in the following steps.

```
Require: $\mathbf{A}, \mathbf{b}$
1: $n \leftarrow$ Size($\mathbf{A}$)
2: $\mathbf{I} \leftarrow$ identity matrix of order $n$
3: Create $\mathbf{B} = [\mathbf{A}, \mathbf{I}]$
4: for $i = 1$ to $n$ do
5:     $b_{ij} = b_{ij}/b_{ii}, \forall j = 1, \ldots, 2n$
6:     for $k = 1$ to $n$ do
7:         if $i \neq k$ then
```

```
8:          $b_{kj} = b_{kj} - b_{ij} * b_{ki}\ \forall j = 1, \ldots, 2n$
9:        **end if**
10:     **end for**
11: **end for**
12: $\mathbf{A}^{-1} \leftarrow$ last $n$ columns of $\mathbf{B}$
13: $\mathbf{X} \leftarrow \mathbf{A}^{-1}\mathbf{b}$
14: **return** $\mathbf{X}$
```

Therefore, we would have to write the following MATLAB/Octave code for Gaussian Elimination, to solve the given system of equations

```
function X = GaussianElimination(A, b)
%determine the size of A
n = size(A, 1);
%create an identity matrix of order $n$
I = eye(n);
%create B
B = [A, I];

%elimination algorithm
for i = 1:n
B(i,:) = B(i,:) / B(i,i);
for j=[1:i-1, i+1:n]
B(j,:) = B(j,:) -  B(i,:) * B(j,i);
end
end

%calculate the inverse
invA = B(:,n+1:2*n);

%calculate X
X = invA * b;
end
```

2.1.1.2 Gauss Jordan Method

The computation of the inverse is rather tedious, and the above operation involves the computation of the inverse and then the multiplication. The Gauss Jordan Method simplified this computation.

We are given the above system of equations, and we write it in the matrix form as follows:

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \dots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array}\right]$$

We introduce the following notation: $a_{i,n+1} = b_i, \forall i = 1, \dots, n$. We therefore have the following matrix arrangement:

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} & \dots & a_{2n} & a_{2,n+1} \\ \vdots & \dots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & a_{n,n+1} \end{array}\right]$$

The aim now is to reduce the left part of the vertical line, through matrix row transformations, to an identity matrix of order n. In order to convert the element a_{11} to one, we have to divide the entire row by a_{11}. Therefore, we use the transformation:

$$a_{1j} = \frac{a_{1j}}{a_{11}}, \quad \forall j = 1, \dots, n+1$$

If we were to do this in MATLAB/Octave, then we have to write the following command:

```
A(1,:) = A(1,:) / A(1,1)
```

Now for each of the rows, all elements below the new a_{11}, will have to be converted to zero. This is done by the following mathematical transform:

$$\text{For each row } i > 1, \qquad a_{i,j} = a_{i,j} - a_{1,j} * a_{i,1}, \forall j = 1, \ldots, n+1$$

To do this in MATLAB/Octave, we have to write

```
for i=2:n
A(i,:) = A(i,:) - A(1,:) * A(i,1)
end
```

Now, the above steps have to be repeated for every diagonal element $a_{ii}, \forall i = 1, \ldots, n$. Therefore, the following algorithm is achieved.

```
Require: A, b
 1: $n$ = number of rows in $A$
 2: Create $\mathbf{B} = [\mathbf{A}, \mathbf{b}]$
 3: for $i = 1$ to $n$ do
 4:     $b_{ij} = b_{ij}/b_{ii}$, $\forall j = 1, \ldots, n+1$
 5:     for $k = 1$ to $n$ do
 6:         if $i \neq k$ then
 7:             $b_{kj} = b_{kj} - b_{ij} * b_{ki}$ $\forall j = 1, \ldots, n+1$
 8:         end if
 9:     end for
10: end for
11: $\mathbf{X} \leftarrow$ last column of $\mathbf{B}$
12: return $\mathbf{X}$
```

Therefore, we would have to write the following MATLAB/Octave code for Gauss Jordan Elimination, to solve the given system of equations

```
function [Y] = GaussJordan(A, b)
```

```
n = size(A, 1);
%club A and b
C = [A b];

for i=1:n
I = eye(n);
I([1:i-1 i+1:n],i) = -C([1:i-1 i+1:n],i) / C(i,i);
C = I * C;
end
Y = C(:,end);
end
```

2.1.2 Gauss Siedel Method

The Gauss Siedel Method is an iterative process of solving the system of equations. Let us assume that we are given the following set of equations:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3 \\
a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= b_4 \qquad (2.1)
\end{aligned}$$

From these, we can derive the following set of equations:

$$\begin{aligned}
x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - a_{14}x_4) \\
x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - a_{24}x_4) \\
x_3 &= \frac{1}{a_{33}}(b_3 - a_{31}x_1 - a_{32}x_2 - a_{34}x_4) \\
x_4 &= \frac{1}{a_{44}}(b_4 - a_{41}x_1 - a_{42}x_2 - a_{43}x_3) \qquad (2.2)
\end{aligned}$$

It is to be remembered that, the Gauss - Siedel method is used for case when the following matrix is diagonally dominant. In other cases, the Gauss-Siedel method is not convergent.

In the initial scenario, the value of $[x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, x_4^{(0)}]^T = [0,0,0,0]^T$. First, $x_1^{(1)}$ is calculated using the initial values of $x_2^{(0)}, x_3^{(0)}$ and $x_4^{(0)}$, and updated. Next, $x_2^{(1)}$ is calculated using the updated value of $x_1^{(1)}$ and the original values of $x_3^{(0)}$ and $x_4^{(0)}$. The entire set is updated in this manner, and the values for the next iteration is obtained.

We can summarise this in the following set of equations,

$$\begin{aligned}
x_1^{(1)} &= \frac{1}{a_{11}}\left(b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)}\right)\\
x_2^{(1)} &= \frac{1}{a_{22}}\left(b_2 - a_{21}x_1^{(1)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)}\right)\\
x_3^{(1)} &= \frac{1}{a_{33}}\left(b_3 - a_{31}x_1^{(1)} - a_{32}x_2^{(1)} - a_{34}x_4^{(0)}\right)\\
x_4^{(1)} &= \frac{1}{a_{44}}\left(b_4 - a_{41}x_1^{(1)} - a_{42}x_2^{(1)} - a_{43}x_3^{(1)}\right) \qquad (2.3)
\end{aligned}$$

In the generic sense, we can summarise the above equations, with the following.

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \qquad i = 1, 2, \ldots, n.$$

This is usually conducted until a fixed number of iterations, or till a sufficiently small residual value is achieved.

Exercise Solve the following linear system of equations using the Gauss-Siedel iterative method:

$$\begin{aligned}
16x + 3y &= 11\\
7x - 11 &= 13
\end{aligned}$$

The MATLAB/Octave code for the Gauss-Siedel method, is written as follows:

```
function [X] = GaussSiedel(A, b)
maxIter = 10; %will do for 10 iterations
%find the size of A
n = size(A, 1)
%create a zero matrix (initial value)
x = zeros(n,1)
c=0;
while (c < maxIter)
  for i=1:n
    r = 1 / A(i,i);
    f = b(i) - A(i,1:i-1) * x(1:i-1) - A(i, i+1:n) * x(i+1:n)
    x(i) = r * f;
  end
  c = c+1;
end
end
```

Exercise Solve the following set of equations using calculators, and using the code developed.

$$\begin{aligned} 10x_1 - x_2 + 2x_3 &= 6 \\ -x_1 + 11x_2 - x_3 + 3x_4 &= 25 \\ 2x_1 - x_2 + 10x_3 - x_4 &= -11 \\ 3x_2 - x_3 + 8x_4 &= 15 \end{aligned}$$

2.1.3 LU Decomposition

2.1.3.1 Doolittle decomposition

The Doolittle decomposition algorithm begins with a given matrix A of given dimension $N \times N$.

The following steps are taken in the algorithm: We first have a matrix A such that:

$$A = (a_{ij})_{N\times N}$$

In the first iteration, we have to make all the elements below a_{11} to be zero. Let us take the case of the second row. If we have to make the value of a_{21} as zero, we will have to multiply row 1, by a_{21}, divide it by a_{11} and then subtract it from the second row. Therefore, we have the following transformation:

$$a_{2,.} = a_{2,.} - \left(\frac{a_{21}}{a_{11}}\right) a_{1,.}$$

Therefore, for any ith row, if one wants to make the element below a_{11} as zero, the transformation would be:

$$a_{i,.} = a_{i,.} - \left(\frac{a_{i1}}{a_{11}}\right) a_{1,.}$$

If we look for generalization, then if someone wants to make the elements below any diagonal element say a_{jj} zero, then we have to use the transformation:

$$a_{i,.} = a_{i,.} - \left(\frac{a_{ij}}{a_{jj}}\right) a_{j,.}, \qquad i = j+1, \ldots N, \quad j = 1, \ldots, N-1$$

The above process will create an upper triangular matrix. Now focus on the term

$$\frac{a_{ij}}{a_{jj}} \qquad i = j+1, \ldots N, \quad j = 1, \ldots, N-1$$

Thus, for every row operation in one matrix, we create a column in a lower triangular matrix L. Therefore:

$$l_{ij} = \frac{a_{ij}}{a_{jj}} \qquad i = j+1, \dots N, \quad j = 1, \dots, N-1$$

Exercise Solve the following problems

1. Let
$$A = \begin{pmatrix} 3 & 2 & 5 \\ 6 & 8 & 9 \\ 3 & 5 & 2 \end{pmatrix}$$
Find the LU decomposition of the matrix A.

2. Let
$$B = \begin{pmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{pmatrix}$$

We therefore write the MATLAB/Octave code for the Doolittle algorithm.

```
function [L, U] = Doolittle(A)
        %determine the size of A
        N = size(A);
        %declare two matrices L and U
        L = eye(N);
        U = A;
        for j = 1:N-1 %j is the column and row - id
                %declare L_n
                for i = j+1:N
                        ratio = U(i,j) / U(j,j);
                        L(i,j) = ratio;
                        U(i,:) = U(i,:) - ratio * U(j,:);
                end
        end
```

```
end
```

Chapter 3

Root finding methods in MATLAB/Octave

3.1 Introduction

According to mathematics, an **algebraic equation** is a mathematical expression of the form

$$P = Q$$

where P and Q are polynomials with integer or rational coefficients. A more generalized definitions would be having coefficients from a *field*, but that is beyond the scope of this course.

A transcendental equation is a mathematical expression which is non-algebraic or not polynomial. A few examples might be $x = e^x$ or $x = \cos x$.

This chapter illustrates the various root finding methods and explores non-linear system of equations, as well.

3.2 Fixed point iteration method

The point x is called a fixed point if

$$x = g(x).$$

One of the first steps in the fixed point iteration method is to convert the transcendental form $f(x) = 0$, into the form $x = g(x)$. The algorithm for the fixed point iteration method is then given as follows:

```
**Require:** $g(x), x_0$
  $\epsilon \leftarrow$ A very small number
  $I_{max} \leftarrow$ A fixed number of iterations
  $i \leftarrow 0$
  **while** $i < I_{max}$ **and** $|g(x_i)| > \epsilon$ **do**
    $x_{i+1} = g(x_i)$
    $i \leftarrow i + 1$
    lastFound $\leftarrow x_{i+1}$
  **end while**
  **return** lastFound
```

Exercise Solve the following problems manually.

i Find a root of $\cos(x) - x * e^x = 0$.
ii Find a root of $x^4 - x - 10 = 0$.
iii Find a root of $x - e^{-x} = 0$.
iv Find a root of $e^{-x} \cdot (x^2 - 5x + 2) + 1 = 0$. Solution
v Find a root of $x - \sin(x) - (1/2) = 0$.
vi Find a root of $e^{-x} = 3\log(x)$.

The MATLAB/Octave code is given as follows:

```
% the fixed point iteration method
clear
syms x
g = sym(input('Enter the expresssion g(x) in x = g(x) : ', 's'));
% enter the first estimate of the root
x0 = input('Enter the first estimate of the root x_0 :' );

%epsilon and max iteration
epsilon = 1E-05;
maxIter = 10000;
i = 0;
while ((i < maxIter) && (abs(eval(subs(g, x, x0))) > epsilon))
  %find next root
  x0 = subs(g, x, x0);
  i = i + 1;
end

fprintf('Root of the equation is %f\n', x0);
```

Exercise Try to solve the given problems using the MATLAB/Octave code.

3.3 Bisection Method

The Bolzano's theorem states that - if $f(x)$ is continuous on the closed interval $[a, b]$, and suppose that $f(a)$ and $f(b)$ have opposite signs, then there exists a number c in the interval $[a, b]$, for which $f(c) = 0$.

The bisection method uses this theorem and computes the roots of a function. The algorithm is as follows:

```
Require: $f(x), a, b, f(a) \times f(b) < 0$
  $i \leftarrow 0$
  $I_{max} < 10000$
  $eps \leftarrow 10^{-5}$
  $x \leftarrow (a+b)/2$
  while $|f(x)| > \epsilon$ and $i < I_{max}$ do
    $x \leftarrow (a+b)/2$
    if $f(x) \times f(a) < 0$ then
      $b \leftarrow x$
    else
      $a \leftarrow x$
    end if
  end while
```

Exercise

i Consider the function $f(x) = x^2 - 2x - 35$. Do five iterations of the Bisection method to find at least one root of the given function. Tabulate your calculations upto the fourth place of decimal.

Sample solution: A sample solution for this problem is given here, in the following table.

i	a	$f(a)$	b	$f(b)$	c	$f(c)$
0	6	-11	7.5	6.25	6.75	-2.9375
1	6.75	-2.9375	7.5	6.25	7.125	1.5156
2	6.75	-2.9375	7.125	1.5156	6.9375	-0.7461
3	6.9375	-0.7461	7.125	1.5156	7.0313	0.3766
4	6.9375	-0.7461	7.0313	0.3766	6.9844	-0.187
5	6.9844	-0.187	7.0313	0.3766	7.0079	0.0949

ii Find the root of the function $g(x) = x^3 - 35x^2 + 375x - 1125$ using the

Bisection method to find at least one root. Tabulate your calculations upto the fourth place of decimal.

The MATLAB/Octave code for the algorithm presented for the Bisection Method is given as follows:

```
clear
%create symbols
syms x
%enter the function which will be required for finding roots
f = sym(input('Enter the function f(x) :', 's'));
%enter a
a = input('Enter the first value of x i.e. a: ');
%enter b
b = input('Enter the second value of x i.e. b: ');
%evaluation f(a) and f(b)
fa = eval(subs(f, x, a));
fb = eval(subs(f, x, b));
I_max = 10000;
epsilon = 1E-7;

if (fa * fb < 0)
  i = 0;
  c = (a + b) /2;
  fc = eval(subs(f,x,c));
  while ((i < I_max) && ( abs(fc) > epsilon))
    c = (a + b) / 2;
    fc = eval(subs(f,x,c));
    if (fa * fc < 0)
       b = c;
       fb = fc;
    else
       a = c;
       fa = fc;
    end
    i = i + 1;
  end

  fprintf('The root of the given function is %f\n', c);
else
  fprintf('The entered values of a and b are not proper\n');
end
```

Exercise Attempt to write the MATLAB/Octave code for the bisection method, such that it generates a table, and then prints the final root of the given function.

3.4 Secant method

The secant method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. It is defined by the following recurrence relation:

$$x_i = x_{i-1} - f(x_{i-1})\frac{x_{i-1} - x_{i-2}}{f(x_{i-1}) - f(x_{i-2})}$$

```
Require: $f(x), a, b$
Ensure: $f(a) \times f(b) < 0, a < b$
  $i \leftarrow 0$
  $I_{max} < 10000$
  $eps \leftarrow 10^{-5}$
  $x \leftarrow a - f(a)\frac{a-b}{f(a)-f(b)}$
  while $|f(x)| > \epsilon$ and $i < I_{max}$ do
    $x \leftarrow a - f(a)\frac{a-b}{f(a)-f(b)}$
    $a \leftarrow b$
    $b \leftarrow x$
```

end while

Exercise Attempt to solve the problems given in the earlier exercise using the Secant method.

```
clear
%create symbols
syms x
%enter the function which will be required for finding roots
f = sym(input('Enter the function f(x) :', 's'));
%enter a
a = input('Enter the first value of x i.e. a: ');
%enter b
b = input('Enter the second value of x i.e. b: ');
%evaluation f(a) and f(b)
fa = eval(subs(f, x, a));
fb = eval(subs(f, x, b));
I_max = 10000;
epsilon = 1E-7;

if (fa * fb < 0)
        i = 0;
        c = a - fa * (a-b)/(fa-fb);
        fc = eval(subs(f,x,c));
        while ((i < I_max) && ( abs(fc) > epsilon))
                c = a - fa * (a-b)/(fa-fb);
                fc = eval(subs(f,x,c));
                a = b;
                fa = fb;
                b = c;
                fb = fc;
```

```
                    i = i + 1;
        end

        fprintf('The root of the given function is %f\n', c);
else
        fprintf('The entered values of a and b are not proper\n');
end
```

Exercise Modify the given code to generate a table and then print the solution to the screen.

3.5 Regula-Falsi method

The Regula-Falsi or the false position method is a proper bracketing method, which ensures that the choice of the interval in the next generation ensures that the value of function at the endpoints in the interval are opposite in sign.

```
Require: $f(x), a, b, f(a) \times f(b) < 0$
  $i \leftarrow 0$
  $I_{max} < 10000$
  $eps \leftarrow 10^{-5}$
  $x \leftarrow a - f(a)\dfrac{a-b}{f(a)-f(b)}$
  while $|f(x)| > \epsilon$ and $i < I_{max}$ do
    $x \leftarrow a - f(a)\dfrac{a-b}{f(a)-f(b)}$
    if $f(x) \times f(a) < 0$ then
      $b \leftarrow x$
    else
      $a \leftarrow x$
    end if
  end while
```

Exercise Solve the following problems manually

- i Find a root of $x * \cos[(x)/(x-2)] = 0$
- ii Find a root of $x2 = (\exp(-2x) - 1)/x$
- iii Find a root of $\exp(x^2 - 1) + 10\sin(2x) - 5 = 0$
- iv Find a root of $\exp(x) - 3x2 = 0$
- v Find a root of $\tan(x) - x - 1 = 0$
- vi Find a root of $\sin(2x) - \exp(x-1) = 0$

The MATLAB/Octave code for the Regula-Falsi method is given here.

```
clear
%create symbols
syms x
%enter the function which will be required for finding roots
f = sym(input('Enter the function f(x) :', 's'));
%enter a
a = input('Enter the first value of x i.e. a: ');
%enter b
b = input('Enter the second value of x i.e. b: ');
%evaluation f(a) and f(b)
fa = eval(subs(f, x, a));
fb = eval(subs(f, x, b));
I_max = 10000;
epsilon = 1E-7;

if (fa * fb < 0)
        i = 0;
        c = a - fa * (a-b)/(fa-fb);
        fc = eval(subs(f,x,c));
        while ((i < I_max) && ( abs(fc) > epsilon))
                c = a - fa * (a-b)/(fa-fb);
                fc = eval(subs(f,x,c));
                if (fa * fc < 0)
                        b = c;
                        fb = fc;
                else
                        a = c;
                        fa = fc;
                end
        i = i + 1;
        end

        fprintf('The root of the given function is %f\n', c);
else
        fprintf('The entered values of a and b are not proper\n');
end
```

3.6 Ridders' Method

The Ridder's Method was introduced by C. Ridders in his paper titled "A new algorithm for computing a single root of a real continuous function". However, Press et. al. gives a direct formula for implementing into an algorithm.

i. Get x_1 and x_2 such that $f(x_1) \times f(x_2) < 0$
ii. Calculate $x_3 = (x_1 + x_2)/2$
iii. Calculate
$$x_4 = x_3 + (x_3 - x_1)\frac{\text{sign}\left[f(x_1) - f(x_2)\right] f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}$$
iv. Keep on repeating the steps until satisfied.

Require: $x_1, x_2, f(x)$
Ensure: $f(x_1) \times f(x_2) < 0$

```
  $I_{max} \leftarrow 10000$
  $\epsilon \leftarrow 10^{-5}$ (or any smaller number)
  $x_3 \leftarrow (x_1 + x_2)/2$
  $x_4 = x_3 + (x_3 - x_1)\frac{\text{sign}[f(x_1)-f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}$
  while $|f(x_4) > \epsilon|$ and $i < I_{max}$ do
    $x_4 = x_3 + (x_3 - x_1)\frac{\text{sign}[f(x_1)-f(x_2)]f(x_3)}{\sqrt{f(x_3)^2 - f(x_1)f(x_2)}}$
    $i \leftarrow i + 1$
  end while
```

Exercise Write the MATLAB/Octave code for the Ridders' Method.

3.7 Newton Raphson method

The Newton-Raphson method for root finding needs the original function, one single initial position and the slope of the function (the derivative) at a given point. From a given initial position, the next position is calculated using an iterative equation.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

```
Require: $f(x), f'(x), x_0$
  $I_{max} \leftarrow 10000$
  $\epsilon \leftarrow 10^{-5}$
  $i \leftarrow 0$
  while $|f(x_i)| > \epsilon$ and $i < I_{max}$ do
    $x_{i+1} \leftarrow x_i - \frac{f(x_i)}{f'(x_i)}$
    $i \leftarrow i + 1$
  end while
  return $x_i$
```

For the MATLAB/Octave code, we can take advantage of the symbolic mathematics toolbox and therefore, use differentiation using the `diff` function. The MATLAB/Octave code is presented as follows:

```matlab
clear
%declare the symbols
syms x
%ask the user for the function
ff = input('Enter the function f(x): ', 's');
%ask the user for the initial value
x0 = input('Enter the value of x0: ');
%convert f to a symbolic expression
f = sym(ff);
%differentiate f
fprime = diff(f);

%maxiteration
i_max = 10000;
%epsilon
epsilon = 1E-6;

i = 0;
f_x = eval(subs(f, x, x0));
while (( i < i_max) && (abs(f_x) > epsilon) )
  fprime_x = eval(subs(fprime, x, x0));
  x0 = x0 - f_x / fprime_x;
  %calculate new f_x
  f_x = eval(subs(f, x, x0));
  i = i + 1;
end

```

```
fprintf('The root of the function is %f\n', x0);
```

Exercise

1. For the problems given for Bisection Method, solve them using the Newton-Raphson method.
2. Test the code for the Newton - Raphson Method on these functions.
3. Modify the code to generate a table rather than giving the root directly.

Chapter 4

Interpolation

4.1 Introduction

Interpolation is a method of finding the unknown value of a function, given certain known values. In science and engineering experiments, it is often not possible to obtain results for every kind of input. Therefore, given limited inputs, an attempt is made to interpolate the output. In this chapter, out of the several algorithms available in literature, we shall concentrate upon two well-known methods of interpolation, namely Lagrange's interpolation and Newton's interpolation. We would also see that their results agree with each other.

Blanket assumption The following data is given:

$$\langle \mathbf{P}_i = (x_i, y_i) \rangle_{i=1}^{n} ,$$

the aim is to find a function $F(x)$, such that $F(x_i) = y_i$. In the domain of interpolation x_i are called the node points for interpolation.

4.2 Lagrange's interpolation

We first introduce the Langrange's basis polynomial.

$$L_i^n(x) = \frac{(x-x_1)(x-x_2)\dots(x-x_{i-1})(x-x_{i+1})\dots(x-x_n)}{(x_i-x_1)(x_i-x_2)\dots(x_i-x_{i-1})(x_i-x_{i+1})\dots(x_i-x_n)} = \prod_{\substack{j=1\\ j\neq i}}^{n} \frac{x-x_j}{x_i-x_j}.$$

Based on this definition, we will have

$$\begin{aligned} L_1^n(x) &= \frac{(x-x_2)(x-x_3)\dots(x-x_n)}{(x_1-x_2)(x_1-x_3)\dots(x_1-x_n)} \\ L_2^n(x) &= \frac{(x-x_1)(x-x_3)\dots(x-x_n)}{(x_2-x_1)(x_2-x_3)\dots(x_2-x_n)} \\ \vdots \quad & \quad \vdots \\ L_n^n(x) &= \frac{(x-x_1)(x-x_2)\dots(x-x_{n-1})}{(x_n-x_1)(x_n-x_2)\dots(x_n-x_{n-1})} \end{aligned}$$

Although, it is out of the scope of this course, it is imperative to mention the followng property of the Lagrange's basis functions.

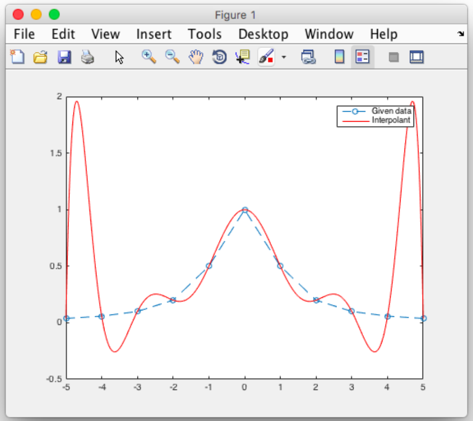
$$L_i^n(x_j) = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{4.1}$$

The calculation of the Lagrange Polynomial is defined as follows:

$$F(x) = \sum_{i=1}^{n} L_i^n(x) y_i \tag{4.2}$$

Exercise

1. There is this mathematical function, known as a complete elliptical integral,



and is defined by

$$K(k) = \int_0^{\pi/2} \frac{dx}{\left[1 - (\sin k)^2 \sin^2 x\right]^{1/2}}$$

From a table of values of these integrals, we find that for various values of k measured in degrees, we have $K(1) = 1.5709, K(4) = 1.5727$, and $K(6) = 1.5751$. Determine $K(3.5)$.

In MATLAB/Octave since the aim is code vectorization, it is wise to calculate all the differences $x - x_i$ in one single go. In our code, we not only generate the value of the interpolant, but also attempt to plot the resulting curve, by varying x from the minimum value of x_i to the maximum value of x_i.

```
function LagrangeInterpolation(xv, yv)
        %symbolic approach to Lagrange's
        syms x
        n = length(xv);

        for i = 1:n
```

```
                numerator = prod(x - xv([1:i-1 i+1:n]));
                denominator = prod(xv(i) - xv([1:i-1 i+1:n]));
                L(i) = numerator/denominator * yv(i);
        end

        P = sum(L);
        %plot the original data
        plot(xv, yv, '--o');
        hold on
        X = min(xv):0.01:max(xv);
        Y = eval(subs(P, x, X));
        plot(X, Y, 'r');
        legend('Given data', 'Interpolant');
        hold off
end
```

4.3 Newton's interpolation

4.3.1 Divided differences

For every $x_i, i = 1, \ldots, n$, the value of an unknown function f is represented by $f[x_i]$. The series of first order divided differences is given as follows:

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1} - f[x_i]}{x_{i+1} - x_i}, i = 1, \ldots, n-1.$$

The series of second order divided differences is given as follows:

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}, i = 1, \ldots, n-2.$$

Therefore, in order to generalize, we present the jth order divided difference as follows:

$$f[x_i, x_{i+1}, \dots x_{i+j}] = \frac{f[x_{i+1}, x_{i+2}, \dots, x_{i+j}] - f[x_i, x_{i+1}, \dots, x_{i+j-1}]}{x_{i+j} - x_i}, \; i = 1, \dots, n-j.$$

4.3.2 Newton's polynomial

The Newton's polynomial, the interpolant, which uses the concept of divided differences is given as

$$\begin{aligned} P(x) = f[x_1] &+ (x - x_1) f[x_1, x_2] \\ &+ (x - x_1)(x - x_2) f[x_1, x_2, x_3] + \dots \\ &+ (x - x_1) \dots (x - x_{j-1}) f[x_1, x_2, \dots, x_j] + \dots \end{aligned}$$

4.3.2.1 Divided difference table

The computation of the divided difference table is the first step in finding the interpolant.

x_i	$f[x_i]$	1st dd	2nd dd	$\dots$
x_1	$f[x_1]$	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$	$\dots$
x_2	$f[x_2]$	$f[x_2, x_3]$	$f[x_2, x_3, x_4]$	$\dots$
$\dots$	$\vdots$	$\vdots$	$\vdots$	$\dots$
x_{n-1}	$f[x_{n-1}]$	$f[x_{n-1}, x_n]$		
x_n	$f[x_n]$			

We can observe here that the first two columns are occupied by x_i and $f[x_i]$, and the first order divided difference (dd) appears on the 3rd columns, the second order difference (dd) appears on the 4th columns, and therefore we predict that the jth order divided difference will appear on the $j + 2$th column.

```
function NewtonInterpolation(xvals, yvals)
        %set interval
        h = 0.01;
        X = min(xvals):h:max(xvals);
        Y = zeros(size(X));
        numPoints = length(xvals);
        %get the table
        T = DividedDifferenceTable(xvals, yvals);
        for i = 1:length(X)
                thisX = X(i);
                Y(i) = T(1,2);
                for j = 3:numPoints+1
                        Y(i) = Y(i) + prod(thisX - ...
                        xvals(1:j-1)) * T(1,j);
                end
        end
        plot(xvals, yvals, '--gs')
        hold on
        plot(X, Y);
end

function A = DividedDifferenceTable(xvals, yvals)
        numPoints = length(xvals);
        A = zeros(numPoints, numPoints+1);
        A(:, 1) = xvals';
        A(:, 2) = yvals';
        %the counting starts from column 3
        for i = 1:numPoints
                for j = 1:numPoints-i
                        A(j, i+2) = (A(j+1, i+1) - A(j, i+1)) /...
                         (xvals(i+j) - xvals(i));
                end
        end
end
```

Exercise Test the interpolation using the Runge's function: $f(x) = 1/(1 + x^2)$

Chapter 5

Numerical Differentiation and Integration

5.1 Taylor's Series

5.1.1 The concept of differences

There are three kinds of differences namely, forward difference, backward difference and the central difference.

Mathematically, the forward difference is given by

$$f(x+h) - f(x),$$

the backward difference is given by

$$f(x) - f(x-h),$$

and the central difference is given by

$$f(x+h) - f(x-h)$$

5.1.2 The representation of derivatives

Therefore the determination of the first derivative of a function can be given by all the above three methods, for example by forward difference we have

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h},$$

by the backward difference we have

$$f'(x) = \lim_{h\to 0} \frac{f(x) - f(x-h)}{h},$$

and by the central difference we have

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x-h)}{2h},$$

5.1.3 The series

The Taylor's Series is given in the following form:

$$f(x) = f(a) + (x-a)f'(a) + (x-a)^2 \frac{f''(a)}{2!} + (x-a)^3 \frac{f'''(a)}{3!} + \dots \tag{5.1}$$

The Taylor's series is one of the major tools in Numerical Analysis and will be used often for derivation of different formulae.

5.2 Finite difference approximation of derivatives

If we use central differences, we can easily see that

$$f'(x) \approxeq \frac{f(x+h) - f(x-h)}{2h}$$

If we take $h/2$ and attempt to approximate the second derivative we shall obtain:

$$f''(x) \approxeq \frac{f'(x+\frac{h}{2}) - f'(x-\frac{h}{2})}{h} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

5.3 Richardson's Extrapolation

Richardson's extrapolation is a method of estimating the value of derivative accurately.

Using the Taylor's series formula given above, and replacing x by (x+h) and a by x, we have the following two equations:

$$f(x+h) = f(x) + hf'(x) + h^2\frac{f'(x)}{2!} + h^3\frac{f'''(x)}{3!} + h^4\frac{f^{(4)}(x)}{4!} + \dots \tag{5.2}$$

$$f(x-h) = f(x) - hf'(x) + h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!} + h^4\frac{f^{(4)}(x)}{4!} + \dots \tag{5.3}$$

Therefore, by subtraction 5.3 from 5.2, we have, the following

$$f(x+h) - f(x-h) = 2hf'(x) + 2h^3\frac{f'''(x)}{3!} + 2h^5\frac{f^{(5)}(x)}{5!} + \dots$$

Therefore, we have

$$N(h) = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + h^2\frac{f'''(x)}{6} + h^4\frac{f^{(5)}(x)}{120} + \dots \tag{5.4}$$

If in the above equation, we replace h by $\frac{h}{2}$,

$$N(h/2) = \frac{f(x+\frac{h}{2}) - f(x-\frac{h}{2})}{h} = f'(x) + h^2\frac{f'''(x)}{24} + \left(\frac{h}{2}\right)^4 \frac{f^{(5)}(x)}{5!} + \dots \quad (5.5)$$

Therefore, we multiply equation 5.5 by 4, from that we take out equation 5.4. Thus,

$$4N(h/2) - N(h) = 4f'(x) - f('x) + \text{some smaller terms}$$

Giving

$$f'(x) \approxeq N(h/2) + \frac{N(h/2) - N(h)}{3} \quad (5.6)$$

Exercise Using $f(x) = 3x^2e^{-7x}$, find the approximation of the derivative using Richardson's extrapolation, at the point $x = 0.55$ Use the calculator here. Use $h_1 = 0.25$ and $h_2 = h_1/2 = 0.125$.

5.3.1 MATLAB/Octave code

```
function V = RichardsonExtrapolation(f, xval, h)

%calculating N(h)
N_h = (subs(f, xval+h) - subs(f, xval-h)) / (2 * h);

%calculating N(h/2)
N_h2 = (subs(f, xval+h/2) - subs(f, xval-h/2)) / (h);

%calculating the approximation here
V = eval(N_h2 + (N_h2 - N_h) / 3);
end
```

Exercise

i. Using the same function, in the earlier exercise, use the MATLAB/Octave function to evaluate the approximate value of the differential.

```
syms x
f = 3 * x^2 * exp(-7 * x);
i = RichardsonExtrapolation(f, 0.55, 0.25)
```

ii. Use the function $f(x) = 5xe^{-2x}$, to evaluate the value of the differential.

5.4 Numerical Integration

The aim of Numerical Integration, also available in some modern day calculators, is to estimate the definite integral:

$$I(f) = \int_a^b f(x)\,dx \tag{5.7}$$

There are two different broad techniques by which Numerical Integration can be done. One is by using approximation and the second is by using the method of quadrature.

5.4.1 Integration by approximation

5.4.1.1 The Rectangle Rule

The simplest form of **Rectangle Rule** is

$$I(f) \approx R = (b-a)f(a)$$

However, if there were about $n+1$ nodes created by dividing the interval $[a, b]$ into n equal parts, then the rectangle rule would be generalised into the following:

$$I(f) \approx R = \sum_{i=1}^{n} (x_{i+1} - xi) f(x_i) \tag{5.8}$$

5.4.1.2 Midpoint rule

The midpoint rule in the simplest form is given as

$$I(f) \approx R = (b-a) f(a+b)/2$$

Again, like the previous rule, if there were $n+1$ nodes created by dividing the interval $[a, b]$ into n equal parts, then the midpoint rule can be generalised into the following

$$I(f) \approx R = \sum_{i=1}^{n} (x_{i+1} - x_i) f(\frac{x_{i+1} + x_i}{2})$$

5.4.1.3 Trapezoidal rule

We can also approximate the integration of a function, by approximating the area under a curve by small trapeziums. The area under the curve can be approximated by a single trapezium

$$I(f) \approx R = \frac{1}{2}[f(a) + f(b)](b-a)$$

If we were to generalize this into the $n+1$ nodes, we would have

$$\begin{aligned} I(f) \approx R = \frac{1}{2}\sum_{i=1}^{n}[f(x_i) + f(x_{i+1})]h &= \frac{h}{2}\sum_{i=1}^{n}[f(x_i) + f(x_{i+1})] \\ &= \frac{h}{2}\left[f(x_1) + 2\sum_{i=2}^{n} f(x_i) + f(x_{n+1})\right] \end{aligned}$$

Therefore, at all the nodes, the function has to be evaluated, and all the values have to be multiplied by 2, except the first and the last one. Then, all the values are added and multiplied by $h/2$.

We formulate this into a MATLAB/Octave code.

```matlab
%n denotes the number of segments
function I = TrapezoidalRule(s, x0, xn, n)
syms x
X = linspace(x0, xn, n+1);
%calculate the function values
Y = eval(subs(s, x, X));
h = (X(2) - X(1));
%calculate the mult matrix
M = [1 2*ones(1, n-1) 1];
I = sum(Y.*M) * h/2;
```

5.4.1.4 Simpson's Rule

To find the area under the curve using the Simpson's Rule, in the interval $[a,b]$ the formula is:

$$I(f) \approx R = \frac{b-a}{6}\left\{f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right\}$$

Therefore, if there are $n+1$ nodes, and the difference between each of the nodes is h, we will have the following:

$$\begin{aligned} I(f) \approx R = \frac{h}{6}\Big[& f(x_1) + 4f(x_2) + f(x_3)] \\ & + [f(x_3) + 4f(x_4) + f(x_5)] \\ & + \cdots + [f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\Big] \\ = \frac{h}{6}\Big[& f(x_1) + 4f(x_2) + 2f(x_3) + \cdots + 4f(x_{n-1}) + f(x_n)\Big] \end{aligned}$$

We see in this expression, that all the odd numbered terms have coefficient 4, and all the even numbered terms have coeffient 2, except for the first and the last terms.

Therefore, we code the following in MATLAB/Octave.

```
function I = SimpsonsRule(s, x0, xn, n)
syms x
X = linspace(x0, xn, n+1);
%evaluate function values here
Y = eval(subs(s, x, X));
h = X(2) - X(1);

%create a ones arry
A = ones(1,n+1);

%create coefficients
A(2:2:n) = 4;
A(3:2:n) = 2;

%multiply
I = sum(Y .* A) * h/3;
end
```

Exercise

i. Write the MATLAB/Octave code for the Rectangle rule for $n+1$ nodes.
ii. Write the MATLAB/Octave code for the Midpoint rule for $n+1$ nodes.
iii. Modify the MATLAB/Octave code given here to generate a table alongwith the integral.
iv. Evaluate the integral of the function $8xe^{x(9-x)}$, using the trapezoidal rule and the Simpson's rule, between the limits 0 and 3.

v. Evaluate the integral of

$$\int_{-5}^{5} \frac{dx}{1+x^2}$$

for even number of segments 10, 20, 30, 40. Compare the values of the integrals, between trapezoidal rule and the Simpson's Rule.

vi. Evaluate $\int_0^1 e^{-x^2} dx$, for 16, 32, 64, 128 segments.

vii. Evaluate

$$I = \int_0^1 (1-x^2)^{3/2} dx$$

Chapter 5

Initial and Boundary Value Problems

5.1 Initial Value Problem

An initial value problem is of the type

$$y' = F(x, y) \qquad y(x_0) = y_0 = v$$

5.1.1 Euler method

From the Taylor's series, we have

$$f(x+h) = f(x) + h \cdot f'(x) + \frac{h^2}{2} f''(x) + \dots$$

If we assume a small value of h and ignore the higher order terms (beginning from h^2), we will have

$$f(x+h) \approxeq f(x) + h \cdot f'(x)$$

In an iterative mode, we can assume this as

$$f(x_{i+1}) = f(x_i) + h \cdot f'(x_i) \tag{5.1}$$

Problem specification in the Euler Method In an Euler Method setting, the problem is given as follows:

i. $\dfrac{dy}{dx} = F(x, y)$

ii. $y_0 = y(x_0) = v$

iii. Find the value of y at some other x which will be given.

The solution to the Euler's problem is given in the following algorithm:

```
Require: $x_0, x_n, h, y_0, F(x, y)$
1: $i \leftarrow 0$
2: while $x < x_n$ do
3:     $y_{i+1} = y_i + h \cdot F(x_i, y_i)$
4:     $x_{i+1} \leftarrow x_i + h$
5:     $x \leftarrow x_{i+1}$
6:     $i \leftarrow i + 1$
7: end while
```

Exercise [1]

i. Find $y(0.5)$ if $y' = -2x - y, y(0) = -1$, using Euler's Method with $h = 0.1$.

Solution: In the given problem $y_0 = -1$, and $F(x, y) = -2x - y$

[1] Taken from the notes of Dr. YVSS Sanyasiraju, IITM

i	x_i	y_i	$F(x_i, y_i) = -2x_i - y_i$	y_{i+1}
0	0	-1.0000	1.0000	-0.9000
1	0.1	-0.9000	0.7000	-0.8300
2	0.2	-0.8300	0.4300	-0.7870
3	0.3	-0.7870	0.1870	-0.7683
4	0.4	-0.7683	-0.0317	-0.7715
5	0.5	-0.7715		

ii. If $y' = x + y + xy$, $y(0) = 1$, then determine the value of $y(0.1)$, with the step size of 0.01.

iii. It is given that $y' = x/y$, $y(0) = 1$. Find $y(1)$ using the Euler's method, with different step lengths 0.1 and 0.2. Compare the results with the analytical solution $y^2 = 1 + x^2$.

iv. Using the Euler's method, find the approximate solution of $y' = (y-x)/(y+x)$, $y(0) = 1.0$ at $x = 0.1$, by taking $h = 0.02$.

v. When $y' = y - 2x/y$, $y(0) = 1$, find $y = 0.8$, when $h = 0.1$

Before we proceed to design the MATLAB/Octave code for the Euler's Method, we need to understand the concept of cell arrays in MATLAB/Octave. Normal arrays contain same type of data, whereas cell arrays are designed to handle multiple types of data. We illustrate the declaration of a cell array by the following example:

```
%declaration of a normal array with zero content
A = zeros(2,3) %will create a 2 x 3 matrix with zeros

%declaration of a 2 x 3 cell array with no content
C = cell(2,3)

%declaration of a cell array with explicit contents
C = {[4, 5, 3; 1, 4, 3], 2, 'stray string'}
```

We now proceed to convert the given algorithm into a MATLAB/Octave code.

```matlab
%s is the expression containing x and y
function Y = EulerMethod(x_0, y_0, x_n, h, s)
  syms x y %this makes sure that the symbols are recognised
  X = x_0;  %this is the first value of x
  Y = y_0; %this is the first value of y (y_0)
  while (X < x_n)
    Y = Y + h * eval(subs(s, {x, y}, {X, Y}));
    X = X + h;
  end
  fprintf('At x = %f, the value of y is %f\n', x_n, Y)
end
```

Exercise Modify the MATLAB/Octave code to generate a table like the one presented for the solution of problem (i).

5.1.1.1 Plotting as the solutions are determined

We will now try to plot the solution of the IVP, using the MATLAB/Octave plotting engine. This exercise is also to see how the Euler's method behaves with the same differential equation, with different steps of h.

```matlab
function EulerMethodPlotter(s, x0, xn, y0, h)
% here h is a vector containing different values of step sizes
% h = [0.1, 0.01, 0.001] is a sample entry

%find the length of h
num_h = length(h);

%create empty figure
figure

```

```
for i = 1:num_h
  this_h = h(i);
  %generate X array
  X = x0:h:xn;
  %get the Y array from the Euler matrix method
  Y = generateEulerMatrix(s, X, y0, this_h);
  plot(X,Y);
  hold on
end
end

function Y = generateEulerMatrix(s, X, y0, th)
  syms x y
  %generate empty Y array same as the size of X
  Y = zeros(size(X));
  Y(1) = y0;
  for i = 1:length(X)-1
   Y(i+1) = Y(i) + th * eval(subs(s, {x, y}, {X(i), Y(i)}));
  end
end
```

5.1.2 Stability of the Euler's method

For the stability of the Euler's method, we shall see the following differential equation:

$$y' = -15y, \qquad y(0) = 1$$

It is to be mentioned here that the original solution here is $y = e^{-15x}$, with $y(0) = 1$

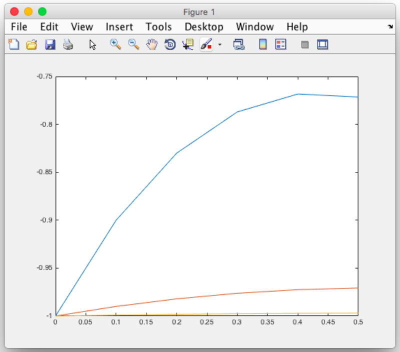


Figure 5.1: Plot of the Euler method for different values of h

Exercise Use the `EulerPlotter` function, to plot the various solutions of the differential equations. Take $h = [0.1, 0.125, 0.25]$ and see what happens.

5.1.3 Runge-Kutta methods

The Runge-Kutta methods are a series of methods which are used to solve the differential equations of the type:

$$y' = F(x, y), \qquad y(x_0) = y_0$$

It has to be remembered that the Runge-Kutta methods draw inspiration from the Euler's method, but are supposed to be more accurate.

5.1.3.1 Fourth order Runge-Kutta method

The fourth-order Runge-Kutta method, also known as RK4, is the classical method. Therefore, if the initial value problem is given, and the step size h is known, then

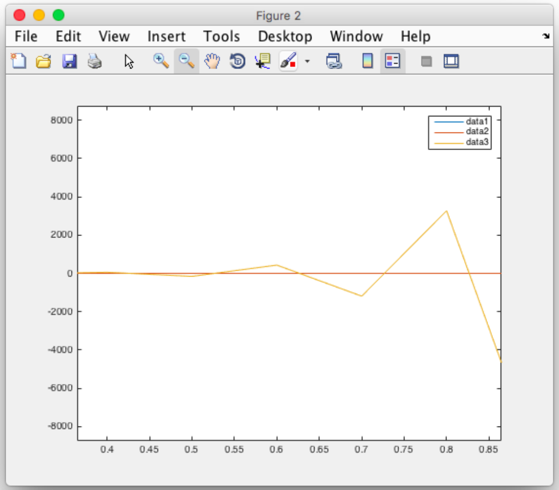


Figure 5.2: Plot demonstrating the unstability of the Euler's method

the value of y, for the next x is given in the following manner:

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

- $k_1 = f(x_i, y_i)$
- $k_2 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_1)$
- $k_3 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_2)$
- $k_4 = f(x_i + h, y_i + hk_3)$

We will now proceed to write the code for the Runge Kutta method:

```
function [X, Y] = RungeKuttaMethod(s, x0, xn, y0, h)
syms x y %making sure that the symbols are recognised

```

```
%generate the array for X
X = x0:h:xn;
%generate the array for Y
Y = zeros(size(X));
%set the initial value
Y(1) = y0;

for i=2:length(X)
  k1 = eval(subs(s, {x, y}, {X(i-1), Y(i-1)}));
  k2 = eval(subs(s, {x, y}, {X(i-1) + h/2, Y(i-1) + h/2 * k1}));
  k3 = eval(subs(s, {x, y}, {X(i-1) + h/2, Y(i-1) + h/2 * k2}));
  k4 = eval(subs(s, {x,y}, {X(i-1) + h, Y(i-1) + h * k3}));
  Y(i) = Y(i-1) + h/6 * (k1 + 2 * k2 + 2 * k3 + k4);
end
end
```

Exercise

i. For all the ODEs in the earlier exercise, attempt to use the Runge Kutta Method and solve them manually.
ii. Use the MATLAB/Octave code to attempt the solution of the ODEs and compare them with your manual solutions.
iii. Take inspiration from the `EulerMethodPlotter` code and write a similar code for the Runge-Kutta method for various values of h. (See plot 5.3).

5.1.3.2 Second order Runge Kutta method

The second order Runge Kutta method for a given ODE is given by two different methods:

1. Midpoint method
2. Heun's method

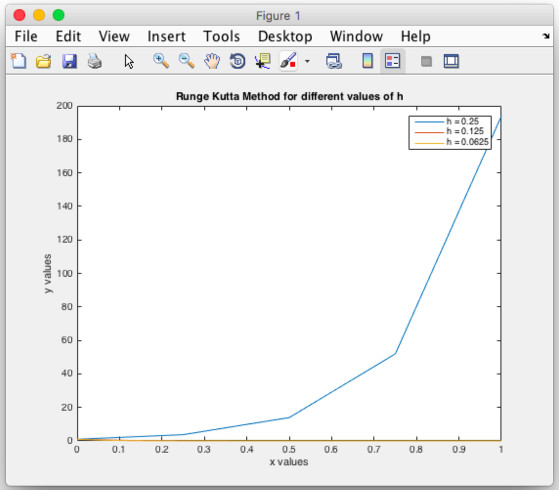


Figure 5.3: Plot for the RungeKutta Method of order 4

Midpoint method The midpoint method explicitly follows the iterative equation:

$$y_{i+1} = y_n + hf\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}F(x_i, y_i)\right),$$

In order to simplify this, we take the following steps:

- Calculate $k_1 = F(x_i, y_i)$
- Calculate $k_2 = F(x_i + \frac{h}{2}, y_i + k_1\frac{h}{2})$
- Calculate $y_{i+1} = y_i + hk_2$

Exercise Write the MATLAB/Octave code for the midpoint method.

Heun's method The Heun's method for solving the IVP is explicitly given as follows:

$$y_{i+1} = y_i + \frac{h}{2}\left[F(x_i, y_i) + F(x_{i+1}, y_i + hF(x_i, y_i))\right]$$

In order to simplify the above expression, we as follows:

- $k_1 = F(x_i, y_i)$
- $k_2 = F(x_i + h, y_i + k_1 * h)$
- $y_{i+1} = y_i + \frac{h}{2}[k_1 + k_2]$

Exercise Write the MATLAB/Octave code for the Heun's method.

5.1.4 Adam's - Bashforth Method

If you recall the Euler's method explained earlier, it is a typical single step method, and required one previous value of y to compute the present value. In turn, the Adam's Bashforth method is a two step method, where there is a requirement of two previous values.

Therefore, if the initial value problem is of the type:

$$y' = F(x, y), \qquad y(x_0) = y_0 = v$$

then the step for the Adam's Bashforth method is given by

$$y_{n+2} = y_{n+1} + \frac{3}{2}hF(x_{n+1}, y_{n+1}) - \frac{1}{2}hF(x_n, y_n)$$

One can clearly see here that there is a requirement of two previous values here, and one is given the values x_0 and y_0, y_1 is first calculated using the Euler's method.

We will now write a MATLAB/Octave code, which will solve an ODE using the Adam's Bashforth method. In addition, the code will also plot the solutions for different values of h.

```matlab
function AdamsBashforthPlotter(s, x0, xn, y0, h)
lText = cell(length(h), 1);
%generate empty figure
figure
%for each h
for i = 1:length(h)
thisH = h(i);
[X, Y] = AdamsBashforthODE(s, x0, xn, y0, thisH);
plot(X,Y)
hold on
lText{i} = ['h = ', num2str(thisH)];
end
title('Adams Bashforth Method');
xlabel('x values');
ylabel('y values');
legend(lText);
end

function [X, Y] = AdamsBashforthODE(s, x0, xn, y0, h)
syms x y %ensure that symbols are recognised.

%create linear space for X
X = [x0:h:xn];
%create empty space of Y
Y = zeros(size(X));
%set initial solution of Y
Y(1) = y0;
%calculate the first solution by Euler
Y(2) = Y(1) + h * eval(subs(s, {x, y}, {X(1), Y(1)}));
%for the remaining parts calculate using Adam's Bashforth
for i=3:length(X)
Y(i) = Y(i-1) + ...
```

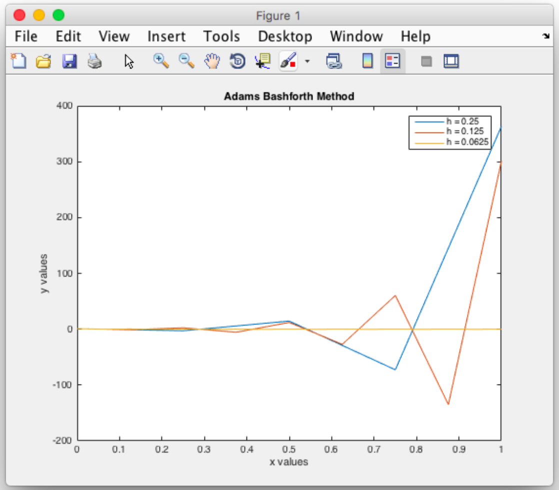


Figure 5.4: Plot of the Adam's Bashforth two-step method for $y' = -15y, \quad y(0) = 1$, for different values of h.

```
        3/2 * h * eval(subs(s, {x, y}, {X(i-1), Y(i-1)})) - ...
        1/2 * h * eval(subs(s, {x, y}, {X(i-2), Y(i-2)}));
end
end
```

Exercise

i. For each of the earlier problems, use the Adam's Bashforth method to solve them.
ii. Modify the given code to generate tabular output, instead of the graphical outputs.

5.2 Boundary Value Problems

A boundary value problem (BVP) is of the type

$$\frac{d^2y}{dx^2} + P(x)\frac{dy}{dx} + Q(x)y = R(x), \qquad y(a) = y_0, \quad y(b) = y_n. \tag{5.2}$$

In the prime notation form, this equation can be written as

$$y'' + P(x)y' + Q(x)y = R(x), \qquad y(a) = y_0, \quad y(b) = y_n.$$

5.2.1 Finite difference method

If we consider this differential equation over a finite number of nodes, then this would be written as

$$y_i'' + P(x_i)y_i' + Q(x_i)y_i = R(x_i). \tag{A}$$

If you would recall the numerical differentiation through finite differences, we would have the following approximations:

$$y_i'' \approxeq \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \qquad y_i' \approxeq \frac{y_{i+1} - y_{i-1}}{2h}. \tag{B}$$

Substituting these in (A) we have

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} + \frac{y_{i+1} - y_{i-1}}{2h}P(x_i) + Q(x_i)y_i = R(x_i).$$

Multiplying this by $2h^2$, we have

$$2y_{i+1} - 4y_i + y_{i-1} + hy_{i+1}P(x_i) - hy_{i-1}P(x_i) + 2h^2Q(x_i)y_i = 2h^2R(x_i)$$

Rearranging these terms, we have

$$\left[2 - hP(x_i)\right]y_{i-1} + \left[-4 + 2h^2Q(x_i)\right]y_i + \left[2 + hP(x_i)\right]y_{i+1} = 2h^2R(x_i). \tag{C}$$

If $i = 1$, we have the following equation

$$\left[-4 + 2h^2 Q(x_1)\right] y_1 + \left[2 + hP(x_1)\right] y_2 = 2h^2 R(x_i) - \left[2 - hP(x_1)\right] y_0. \tag{D}$$

If $i = n - 1$, we have

$$\left[2 - hP(x_{n-1})\right] y_{n-2} + \left[-4 + 2h^2 Q(x_{n-1})\right] y_{n-1} = 2h^2 R(x_{n-1}) - \left[2 + hP(x_{n-1})\right] y_n. \tag{E}$$

Therefore, we have

$$\begin{bmatrix} -4 + 2h^2 Q(x_1) & 2 + hP(x_1) & \cdots & \cdots & 0 \\ 2 - hP(x_2) & -4 + 2h^2 Q(x_2) & 2 + hP(x_2) & \cdots & 0 \\ \vdots & & & \ddots & \vdots \\ & \cdots & \cdots & & \\ 0 & 0 & \cdots & 2 - hP(x_{n-1}) & -4 + 2h^2 Q(x_{n-1}) \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \left[\begin{matrix} 2h^2 R(x_1) - [2 \\ 2h^2 R \\ \\ 2h^2 R(x_{n-1}) - [2 \end{matrix} \right.$$

```matlab
clear
%define the symbols
syms x

%define the expressions for P(x)
p_x = sym(input('Entering the expression for P(x): ', 's'));

%define the expression for Q(x)
q_x = sym(input('Entering the expression for Q(x): ', 's'));

%define the expression for R(x)
r_x = sym(input('Entering the expression for R(x): ', 's'));

%enter the first value of x
x0 = input('Enter the first value of x: ');
%enter the last value of x
xn = input('Enter the last value of x: ');
%number of nodes
h = input('Enter the step size: ');
```

```

%compute the x array
X = (x0:h:xn)';
%compute the number of nodes
numNodes = length(X);
n = numNodes - 1;

%enter the first value of y
y1 = input('Enter the first value of y: ');
%enter the last value of y
ynp1 = input('Enter the last value of y: ');

%create the y array
Y = zeros(size(X));
Y(1) = y1;
Y(numNodes) = ynp1;

%evaluate all R
R = subs(r_x, x, X);
%evaluate all P
P = subs(p_x, x, X);
%evaluate all Q
Q = subs(q_x, x, X);

%produce the B matrix
B = 2 * h^2 * R;
B(2) = B(2) - (2 - h * P(2)) * Y(1);
B(n) = B(n) - (2 + h * P(n)) * Y(n+1);

%produce the A matrix
A = zeros(n-1);
A(1,[1 2]) = [-4 + 2 * h^2 * Q(2), 2 + h * P(2)];
```

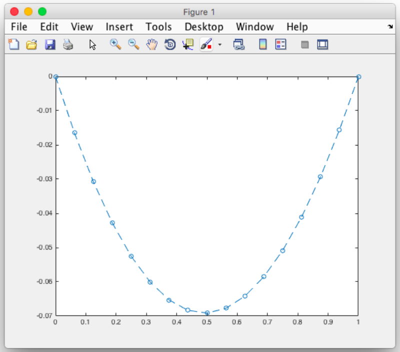


Figure 5.5: Plot of the Finite Difference Method for Problem iii.

```
for i = 2:n-2
A(i,[i-1:i+1]) = [2-P(i+1)*h, -4 + 2 * h^2 * Q(i+1), 2+h*P(i+1)];
end
A(n-1, [end-1 end]) = [2 - h * P(n), -4 + 2 * h^2 * Q(n)];

Y(2:n) =  inv(A) * eval(B(2:n));
plot(X,Y, 'o--')
```

Exercise: Solve manually and using the computer code

i) Solve the following boundary-value problem

$$\frac{d^2y}{dx^2} + y = 0, \qquad y(0) = 0, \quad y(1) = 1$$

Take $h = 1/4, 1/8, 1/6$.

ii) Solve the following BVP by using the Finite Difference Method:

$$y'' + xy' + y = 2x, \qquad y(0) = 1, \quad y(1) = 0$$

Take $h = 1/4, 1/8, 1/6$.

iii) Solve the following BVP by using the difference method

$$y'' + 2y' + y = x, \qquad y(0) = 0, \quad y(1) = 0$$

Take $h = 1/4, 1/8, 1/6$.

5.2.2 Shooting method

The shooting method solves a general second order boundary value problem.

$$y'' = F(x, y, y'), \qquad y(x_0) = y_0, \quad y(x_n) = y_n. \tag{5.3}$$

The step-wise algorithm for the second order boundary value problem (BVP) using the shooting method is given as follows:

Step I: Let α_k be an approximation to the unknown first derivative $y'(x_0) = \alpha$.

Step II: Solve the initial value problem

$$y'' = F(x, y, y') \qquad y(x_0) = y_0, y'(x_0) = \alpha_k$$

from x_0 to x_n using any of the methods of IVP (e.g. Runge Kutta, Euler's method, Adam's Bashforth method etc.). Call the solution $y(\alpha_k; x_n)$ at $x = x_n$.

Step III: Obtain the next approximation for α i.e. α_{k+1} using the linear interpolation:

$$\alpha_{k+1} = \alpha_{k-1} + (\alpha_k - \alpha_{k-1}) \frac{y_n - y(\alpha_{k-1}; b)}{y(\alpha_k; b) - y(\alpha_{k-1}; b)}, \forall\ k = 1, 2, \ldots$$

Step IV: Repeat the steps II and III until $|y(\alpha_k; b) - y_n| < \epsilon$ for a prescribed ϵ.

We illustrate the use of the Shooting Method, using the Euler's Method as a tool, and take a numerical example from Prof. Autar Kaw's YouTube channel.

Example: Solve the differential equation for the second order BVP:

$$y'' - 2y = 8x(9 - x), \qquad y(0) = 0, \quad y(9) = 0.$$

for $h = 3$.

Solution: We rearrange the differential equation into

$$y'' = 2y + 8x(9 - x), \qquad y(0) = 0, \quad y(9) = 0$$

Introduce the notation: $y' = z$, and assume that $z(0) = 4$. Therefore, we have two different differential equations:

$$\frac{dz}{dx} = 2y + 8x(9 - x), \qquad \frac{dy}{dx} = z \qquad y(0) = 0, \quad y(9) = 0 \quad z(0) = 4$$

We can therefore write:

$$\begin{aligned} y' &= F_1(x, y, z) &&= z \\ z' &= F_2(x, y, z) &&= 2y - 8x(9 - x) \end{aligned}$$

Now, for $i = 0$,

$$\begin{aligned} y_1 &= y_0 + h * F_1(x_0, y_0, z_0) &&= 0 + 3 * F_1(0, 0, 4) = 3 \times 4 = 12 \\ z_1 &= z_0 + h * F_2(x_0, y_0, z_0) &&= 0 + 3 * F_2(0, 0, 4) = 4 + 0 * 3 = 4 \end{aligned}$$

Again, for $i = 1$,

$$\begin{aligned} y_2 &= y_1 + h * F_1(x_1, y_1, z_1) &&= 12 + 3 * F_1(3, 12, 4) = 24 \\ z_2 &= z_1 + h * F_2(x_1, y_1, z_1) &&= 4 + 3 * F_2(3, 12, 4) = 4 + 504 = 508 \end{aligned}$$

For, $i = 2$,

$$y_3 = y_2 + h * F_1(x_2, y_2, z_2) = 24 + 3 * F_1(6, 24, 508) = 1548$$

So, we see that if we use the Euler's method with the initial value of z at 4, at $x = 0$, then we have $y(9) = 1548$.

If we now make another choice say $y'(0) = -24$, then we will have the following table:

i	x_i	y_i	z_i
0	0	0	-24
1	3	-72	-24
2	6	-144	-24
3	9	-216	

Better initial guess: We have the following table after Step I.

	Assumption	$y_3 \approxeq y_9$	
p_0	$y'(0) = 4$	1548	q_0
p_1	$y'(0) = -24$	-216	q_1

Therefore, we use the formula:

$$p = p_0 + \frac{p_1 - p_0}{q_1 - q_0}(q - q_0)$$

Therefore,

$$p = 4 + \frac{-24 - 0}{-216 - 1548}(0 - 1548) = -20.57.$$

Now, we can carry out the second iteration with -20.57 as an estimate of $y'(0)$.

i	x_i	y_i	z_i
0	0	0	-20.57
1	3	-61.7	-20.57
2	6	-123.42	41.17
3	9	0.09	

Although the value has come near to zero, we can again interpolate and prepare a better guess using the last two values i.e $z(0) = -24$ and $z(0) = -20.57$.

Exercise Write the MATLAB/Octave code for this method.

Chapter 5

Artificial Neural Networks

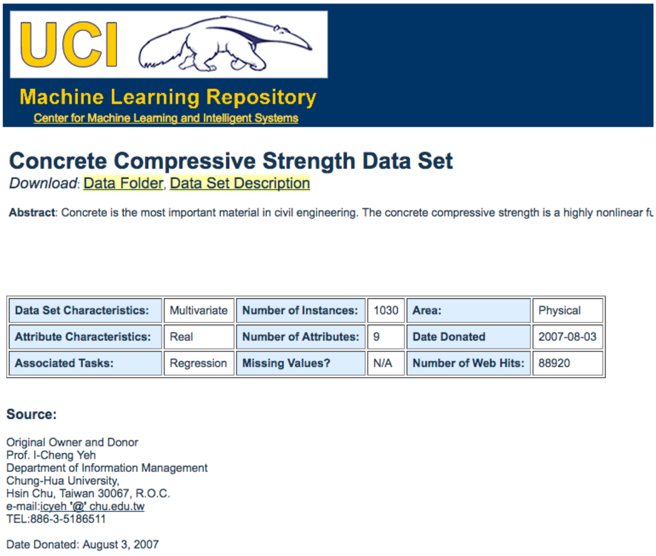
5.1 Introduction

In this exercise, we choose a very specific dataset, and do a complete programming of training a neural network based on various parameters. An approach to parallelize the training, testing and validation of the neural network is also presented at the end of the exercise. The use of multicore processors is imperative for parallelization.

5.2 Data introduction

Concrete is the most important material in civil engineering. The concrete compressive strength is a highly nonlinear function of age and ingredients. These ingredients include cement, blast furnace slag, fly ash, water, superplasticizer, coarse aggregate, and fine aggregate.

Prof. I-Cheng Yeh, Department of Information Management, Chung-Hua University, Hsin Chu, Taiwan 30067, R.O.C. donated this dataset to UCI, Machine



UCI Machine Learning Repository
Center for Machine Learning and Intelligent Systems

Concrete Compressive Strength Data Set
Download: Data Folder, Data Set Description

Abstract: Concrete is the most important material in civil engineering. The concrete compressive strength is a highly nonlinear fu

Data Set Characteristics:	Multivariate	**Number of Instances:**	1030	**Area:**	Physical
Attribute Characteristics:	Real	**Number of Attributes:**	9	**Date Donated**	2007-08-03
Associated Tasks:	Regression	**Missing Values?**	N/A	**Number of Web Hits:**	88920

Source:

Original Owner and Donor
Prof. I-Cheng Yeh
Department of Information Management
Chung-Hua University,
Hsin Chu, Taiwan 30067, R.O.C.
e-mail:icyeh '@' chu.edu.tw
TEL:886-3-5186511

Date Donated: August 3, 2007

Figure 5.1: UCI Machine Learning Repository - Concrete data set: Page screenshot

Learning Repository in 2007. Many publications have been made using this dataset.

5.2.1 Data characteristics

In Civil Engineering practice, a concrete mixture is prepared as per the ISO provisions. However, these ISO provisions usually contain a range of values rather that a single appropriate value. Therefore the perfect combination / ratio of components of concrete are often determined through multiple experiments in the laboratory. This dataset has also been generated by testing the compressive strength of the various combinations of concrete components in the laboratory. Data is in raw form (not scaled).

Number of instances (observations)	1030
Number of Attributes	9
Attribute breakdown	8 quantitative input variables, and 1 quantitative output variable
Missing Attribute Values	None

Table 5.1: Summary statistics for the UCI Machine Learning Concrete dataset

Comp. No	Name	Data type	Measurement	Description
1	Cement	quantitative	kg in a m^3 mixture	Input
2	Blast furnace slag	quantitative	kg in a m^3 mixture	Input
3	Fly ash	quantitative	kg in a m^3 mixture	Input
4	Water	quantitative	kg in a m^3 mixture	Input
5	Superplasticizer	quantitative	kg in a m^3 mixture	Input
6	Coarse aggregate	quantitative	kg in a m^3 mixture	Input
7	Fine aggregate	quantitative	kg in a m^3 mixture	Input
8	Age	quantitative	Day (1 3̃65)	Input
9	Compressive strength	quantitative	MPa	Output

Table 5.2: Variables in the dataset: description as per the column number

5.2.2 Summary Statistics

The summary statistics for this dataset, are as presented in Table 5.1. It is seen here that the data attributes presented here are not scaled i.e. they have been presented as is.

5.2.3 Variable information

Given is the variable name, variable type, the measurement unit and a brief description. The concrete compressive strength is the regression problem. The order of this listing corresponds to the order of numerals along the rows of the database.

5.3 Data processing

5.3.1 Reading the data

On the website, the data is provided in an Excel file, which is pretty old. For the convenience of the workshop, the file has been converted to a tab delimited file named `concdata.txt`. This file contains a header line as well, with fields named from I1 through I8 and output column O1.

Therefore, the file can be read by the following command.

```
%load the file
%comma delimited and with header line, so skipping one line
A = dlmread('concdata.txt', ',', 1, 0);
```

We will separate the input and the output now

```
%separate the input and output matter
outputData = A(:,end);
inputData = A(:,[1:end-1]);
```

5.3.2 Data scaling

For scaling the data, we have to first determine the minimum and maximum of the outputs and inputs. This can be done by the commands:

```
%determine the minimum and maximum of the columns
minOutput = min(outputData);
maxOutput = max(outputData);
minInput = min(inputData);
maxInput = max(inputData);
```

The scaling of the data has to be done between 0 and 1. The standard transform for scaling the data between 0 and 1 is

$$\frac{v - v_{\text{min}}}{v_{\text{max}} - v_{\text{min}}}$$

Implementing this scaling in MATLAB/Octave, will require the following piece of code:

```
%scale data
diffInput = (maxInput - minInput);
diffOutput = maxOutput - minOutput;

%calculate the number of rows
n = size(inputData,1);

scaledInput = (inputData - minInput(ones(n,1),:)) ./ ...
diffInput(ones(n,1), :);
scaledOutput = (outputData - minOutput) / diffOutput;
```

5.4 Setting up the network

In this section, we will learn about setting up the neural networks through various commands.

5.4.1 Layers

A neural net is composed of three kinds of layers, viz. (a) input layer, (b) hidden layer, and (c) output layer. Each layer is composed of neurons. The number of neurons in the input layer is equal to the number of input variables. Similarly, the number of neurons in the output layer is equal to the number of output variables. In our case, there are 8 input variables and therefore, there should be 8 neurons in the input layer. There is one output variable and therefore there would be 1 neuron in the output layer.

5.4.2 Existing network models

There are several neural network models, which are listed in the Wikipedia page. The classical neural network model is the Feedforward neural network model. Other neural network models include the RBF, self organising networks etc.

5.4.3 Activation functions

The activation function is a mathematical function associated with a layer which will define its output as OFF and ON. In computational networks this value is `logsigmoidal` or `tansigmoidal` denoted as **logsig** and **tansig**. While 'creating' the neural network, one has to mention these activation functions, for the input and output channels.

5.4.3.1 Defining the network

We will define the network, with the `feedforwardnet` command. This command requires the number of hidden neurons.

```
%initialise the network now
nnet = feedforwardnet(10);
```

5.4.4 Configuring the network

The input parameter and the output parameters need to be configured and checked first. We do this in MATLAB/Octave using the `configure` command.

```
%configure the network.
nnet = configure(nnet, scaledInput', scaledOutput');
```

5.4.5 Training, testing and validation

It is good to divide the data into various parts viz. training, testing and validation sets. By default, this is done by the dividerand function already in MATLAB/Octave. However, users are more demanding in their wish to customize this feature – "It should be randomized, but I will decide the ratio". Therefore, in order to declare the ratio of separation into training, testing and validation data, we can issue the following commands:

```
%set the partitioning ratio
nnet.divideParam.trainRatio = 0.6;
nnet.divideParam.valRatio = 0.2;
nnet.divideParam.testRatio = 0.2;
```

5.4.5.1 Training the network

After all settings for input, configuration and data sharing has been done, the network can be asked to train itself. This is done by issuing the `train` command.

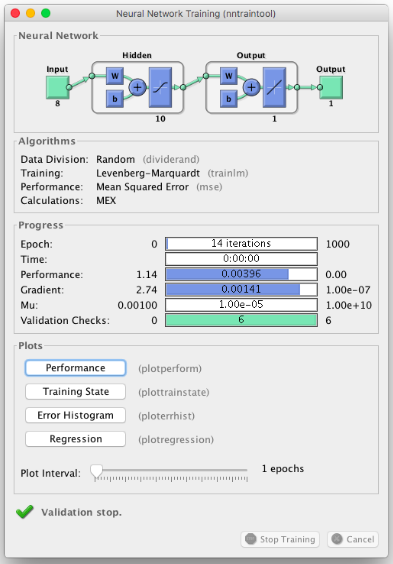
```
[nnet, tr] = train(nnet, scaledInput', scaledOutput');
```

In this code, the `tr` variable maintains the record of training. This can be used later for evaluating the performance of the network.

While training the network a dialog opens up, which shows various parameters of the network, and displays a certain progress bar, while the network training goes on. (see figure 5.2).

5.4.5.2 Performance

For evaluating the performance of the network error-wise, we use the `perform` function. The performance is measured for the validation data only. The validation data can be extracted from the original data by using the training record `tr`.



Algorithms

Data Division: Random (dividerand)
Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Calculations: MEX

Progress

```
Epoch:                0   14 iterations   1000
Time:                     0:00:00
Performance:       1.14   0.00396         0.00
Gradient:          2.74   0.00141         1.00e-07
Mu:             0.00100   1.00e-05        1.00e+10
Validation Checks:    0   6               6
```

Plots

Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Regression (plotregression)

Plot Interval: 1 epochs

✓ Validation stop.

Figure 5.2: Neural network training user interface - in MATLAB

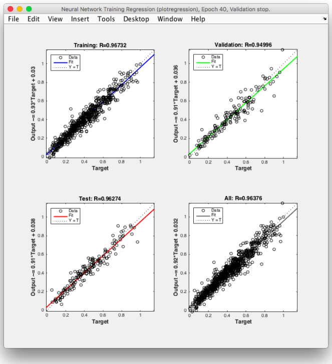


Figure 5.3: Regression plots for the various parts of the dataset

```
y = nnet(scaledInput(tr.valInd,:)');
perf = perform(nnet, scaledOutput(tr.valInd,:)', y);
```

If one clicks on the `Regression` button of the GUI presented earlier, then a figure is generated where one can see the various regression plots (see figure 5.3).

Clicking on the performance plot, generates a time series plot of how the error in the neural network has decreased since the first epoch (see figure 5.4).

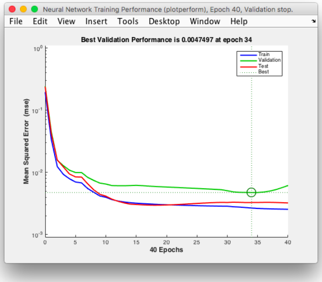


Figure 5.4: Performance plot of the neural network, as presented in MATLAB

5.4.6 Random numbers and network solutions

The neural network strongly depends on the generation of random numbers by MATLAB/Octave. Therefore, it becomes difficult to handle neural networks as it will never generate the same result or the same best output. As a result, it is practicable to run the training several times, say 10,000 times, to generate a very good solution.

5.4.6.1 Parallelization

We have made several uses of the `for` loop in MATLAB/Octave, and this runs the program sequentially. Since, one neural network is independent of another, it is possible to *parallelize* this code. We can keep on storing all the performances, networks and training records and then pick the best network.

To parallelize this code, one uses a cell-array based storage, and a maximum number of trials of the network. Therefore, the entire parallelised code is presented here.

```
clear
%load the file
A = dlmread('concdata.txt', ',', 1, 0);

%separate the input and output matter
outputData = A(:,end);
inputData = A(:,1:end-1);

%determine the minimum and maximum of the columns
minOutput = min(outputData);
maxOutput = max(outputData);
minInput = min(inputData);
maxInput = max(inputData);

%scale data
```

```
diffInput = (maxInput - minInput);
diffOutput = maxOutput - minOutput;

%calculate the number of rows
n = size(inputData,1);

scaledInput = (inputData - minInput(ones(n,1),:)) ./ ...
diffInput(ones(n,1), :);
scaledOutput = (outputData - minOutput) / diffOutput;

maxTrials = 10000;

%declare cell arrays for storing performances
networkCell = cell(maxTrials, 1);
perfCell = zeros(maxTrials,1); %because these are numbers
trCell = cell(maxTrials, 1);

parfor ii = 1:maxTrials
        %initialise the network now
        nnet = feedforwardnet(10);

        %set the partitioning ratio
        nnet.divideParam.trainRatio = 0.6;
        nnet.divideParam.valRatio = 0.2;
        nnet.divideParam.testRatio = 0.2;

        nnet = configure(nnet, scaledInput', scaledOutput');
        [nnet, tr] = train(nnet, scaledInput', scaledOutput');

        y = nnet(scaledInput(tr.valInd,:)');
        perf = perform(nnet, scaledOutput(tr.valInd,:)', y);
        networkCell{ii} = nnet;
        trCell{ii} = tr;
        perfCell(ii) = perf;
end
```

5.4.7 Scaling back the outputs

Since the minimums and maximums of the dataset are known, the scaling can be done by vectorising the following formula:

$$D_{\text{original}} = D_{\text{min}} + D_{\text{scaled}} * (D_{\text{max}} - D_{\text{min}})$$

Exercise Write the MATLAB/Octave code for scaling back the data.

Chapter 5

Optimization with GA in MATLAB/Octave

5.1 Introduction

This chapter is going to typically focus on the use of GA as an optimization tool in MATLAB/Octave. There are two variations of GA in MATLAB/Octave, i.e. single objective and multi-objective.

5.2 Optimization Problem

In general an optimization problem there are two components:

1. cost function or performance function: A cost function is usually minimized, whereas a performance function is usually maximized. [mandatory]

2. Constraints: These are a set of linear or nonlinear inequalities and even sometime equalities. [optional]

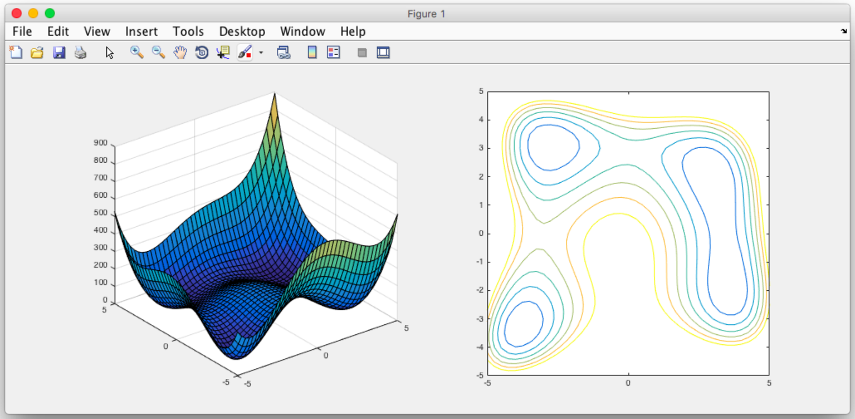


Figure 5.1: 3D plot and contours of Himmelblau's function

3. Range of the variable values.

5.2.1 Unconstrained and constrained optimization

If the constraints to a problem are not given, then the problem is *un-constrained*. Otherwise, it is called an *constrained* problem.

5.3 Coding scenario

5.3.1 Designing the cost function

In MATLAB/Octave, the first part in optimization is to design the cost function. We use the Himmelblau's function to design the cost function (see figure 5.1)

```
function z = costfn(X)
x = X(1);
```

```
y = X(2);
z = (x.^2 + y - 11).^2 + (x + y.^2 - 7).^2;
end
```

It is imperative to note that the number of variables here are 2, which are to be provided in an array.

5.3.2 Running the GA without any constraints

To run the GA, without any constraints, we will use the concept of a *function handle* which is required by the `ga` function. So for example, we can run the following code, where we specify the number of variables.

```
function x = gaTraining

%define handle for the fitness function
fitnessfn = @costfn;

%basic finding of minimum
%put the number of the variables to 2
nvars = 2;
x = ga(fitnessfn,nvars);
end
```

If we run this function as follows

```
X = gaTraining
```

then we will see that each run will give a different output. This is because there are multiple minima in the given cost function.

5.3.3 Adding constraints

There are two types of constraints, viz. linear and non-linear.

5.3.3.1 Linear constraints

Linear constraints are easier to represent in the form of

$$\mathbf{A}x \leq \mathbf{b}$$

For example if we wish to add the constraint

$$4x + y \leq 20,$$

we will put $A = \begin{bmatrix} 4 & 1 \end{bmatrix}$ and $B = [20]$. Therefore, we can add the following lines in our code and modify some.

```
A = [4 1];
b = [20];
%modified line
x = ga(fitnessfn, nvars, A, b);
```

5.3.3.2 Non-linear constraints

Since non-linear constraints are not so direct, they have to be added as function handles in the program. Therefore, if we plan to use the non-linear constraint option

$$(x - 5)^2 - y^2 \leq 26,$$

we have to program it as

$$(x - 5)^2 - y^2 - 26 \leq 0.$$

Thus we write a non-linear constraint function as follows:

```
function [C, Ceq] = nonlincon(X)
x = X(1);
y = X(2);
C = (x - 5).^2 + y.^2 - 26;
Ceq = [];
end
```

To include this function, as a part of MATLAB/Octave coding for GA optimization, we have to add / modify the following lines:

```
%adding nonlinear constraint
nlfunc = @nonlincon;
%modification for non-linear optimization along with linear
x = ga(fitnessfn, nvars, A, b, [], [], [], [], nlfunc)
end
```

5.3.4 Adding bounds

This is about adding lower and upper bounds of a variable. Since in our problem there are two variables, there would two upper and two lower bounds. The lower and upper bounds are coded as vectors. The following code adds bounds to the variables, and runs the GA.

```
LB = [0 0];
UB = [5 5];
%modification for non-linear optimization along with linear,
%and bounds;
x = ga(fitnessfn, nvars, A, b, [], [], LB, UB, nlfunc)
```

5.4 Multi-objective coding

In multiobjective coding, the `ga` function is replaced by `gamultiobj` function. The other parameters remain the same.

Exercise

i. Attempt changing the linear and non-linear constraints.
ii. Write code for two linear and two non-linear constraints.
iii. Study about factors like `crossover`, `generations` etc. and modify the `ga` code.