

LECTURES ON APPLIED MATHEMATICS

Part 2: Numerical Analysis

Ray M. Bowen

Professor Emeritus of Mechanical Engineering
President Emeritus
Texas A&M University
College Station, Texas

Copyright Ray M. Bowen

March, 2015

PREFACE

To Part 2

This textbook is a small addition to the long list of undergraduate textbooks on the subject of Numerical Analysis. Only time will tell whether or not it is a useful addition. It is available for free download at the site <http://rbowen.tamu.edu>. This book is designed to be a continuation of the textbook, *Lectures On Applied Mathematics Part 1: Linear Algebra* which can also be downloaded at <http://rbowen.tamu.edu/>.

This textbook evolved from my teaching an undergraduate Numerical Analysis course to Mechanical Engineering students at Texas A&M University. That course was one of the courses I was allowed to teach after my several years out of the classroom. It tries to utilize rigorous concepts in Linear Algebra in combination with the powerful computational tools of MATLAB to provide undergraduate students practical numerical analysis tools. It makes extensive use of MATLAB's graphics capabilities and, to a limited extent, its ability to animate the solutions of ordinary differential equations. It is not a textbook that tries to be comprehensive as a source of MATLAB information. It does contain a large number of links to MATLAB's extensive online resources. This information has been invaluable to me as this work was developed. The version of MATLAB used in the preparation of this textbook is MATLAB 2014b. This version implements a new graphics system. As a result, when earlier versions are utilized with this textbook, small changes in the script may be required to cause the script in the textbook to execute.

Unlike the other textbooks posted on the site <http://rbowen.tamu.edu>, the posted version of Numerical Analysis has not been used in its entirety in the classroom. Thus, because of my limitations as a typist and a proofreader, it is inevitable that the book will contain a variety of errors, typographical and otherwise. Hopefully, none will interfere with the utility of the book. Emails to rbowen@tamu.edu that identify errors will always be welcome. For as long as mind and body will allow, this information will allow me to make corrections and post updated versions of the book.

I wish to express my appreciation to my faculty colleagues in Mechanical Engineering and in Mathematics at Texas A&M for allowing me to teach their undergraduate students. I also wish to express my appreciation to the students that endured the early stages of the development of this textbook. Finally, this book would not have been possible if it had not been for the help of Dr. Waqar Malik. Several years ago, as a graduate student in Mechanical Engineering at A&M, Dr. Malik was assigned the burden of teaching me MATLAB in order that I could teach Numerical Analysis. I am deeply indebted to Dr. Malik. I value his help and his friendship over these several years.

College Station, Texas

R.M.B.

Posted March, 2015

CONTENTS

Part 2 Numerical Analysis

Selected Readings for Part II.....	582
CHAPTER 7 Elements of Numerical Linear Algebra.....	583
Section 7.1 Elementary Matrix Calculations with MATLAB.....	583
Section 7.2 Systems of Linear Equations.....	591
Section 7.3 Additional MATLAB Related Matrix Operations.....	595
Section 7.4 Ill Conditioned Matrices.....	611
Section 7.5 Additional Discussion of LU Decomposition.....	621
Section 7.6 Additional Discussion of Eigenvalue Problems.....	629
CHAPTER 8 Errors that Arise In Numerical Analysis.....	637
Section 8.1 Taylor's Theorem.....	637
Section 8.2 Round Off and Truncation Errors.....	645
Section 8.3 Computer Representation of Real Numbers, Round-Off Errors.....	651
CHAPTER 9 Roots of Nonlinear Equations.....	673
Section 9.1 Use of Graphics to Locate the Real Roots of Nonlinear Equation.....	675
Section 9.2 MATLAB's fzero Command.....	679
Section 9.3 Bracketing Methods.....	683
Section 9.4 The Newton-Raphson Method.....	705
Section 9.5 Systems of Nonlinear Equations.....	715
Section 9.6 Polynomials.....	729
CHAPTER 10 Regression.....	735
Section 10.1 Least Squares Problems and Overdetermined Systems.....	735
Section 10.2 Linear Regression.....	741
Section 10.3 Linearization of Nonlinear Relationships.....	749
Section 10.4 MATLAB Tools for Linear Regression.....	755
Section 10.5 Polynomial Regression.....	759
Section 10.6 More General Types of Regression.....	771
CHAPTER 11 Interpolation.....	791

Section 11.1	Linear Interpolation.....	787
Section 11.2	Polynomial Interpolation.....	791
Section 11.3	Monomial Interpolation with MATLAB.....	813
Section 11.4	Newton Interpolation with MATLAB.....	819
Section 11.5	Lagrange Interpolation with MATLAB.....	835
Section 11.6	Interpolation by MATLAB's polyfit Command.....	843
Section 11.7	Extrapolations of Interpolations.....	847
Section 11.8	Approximation of a Known Function: Oscillations.....	849
Section 11.9	Issues of Numerical Accuracy.....	853
Section 11.10	Piecewise Lagrange Interpolation.....	865
Section 11.11	Numerical Integration and Piecewise Interpolation.....	889
CHAPTER 12 Ordinary Differential Equations.....		915
Section 12.1	Normal Form of a System of Ordinary Differential Equations.	916
Section 12.2	Picard's Theorem.....	925
Section 12.3	Direction Field.....	927
Section 12.4	Euler's Method: A One Step Iteration Method.....	933
Section 12.5	MATLAB Implementations of the Euler Method.....	947
Section 12.6	Runge-Kutta Methods: Improved One Step Methods.....	959
Section 12.7	MATLAB Implementations of Runge-Kutta Methods.....	969
Section 12.8	MATLAB ODE Solvers.....	981
Section 12.9	More on Stiff Ordinary Differential Equations.....	997
Section 12.10	Systems of Linear Ordinary Differential Equations.....	1009
Section 12.11	Systems of Nonlinear Ordinary Differential Equations.....	1033
Section 12.12	Forced Vibrations of Nonlinear Pendulum with Damping....	1045
Section 12.13	Other Pendulum Examples.....	1067
APPENDIX A Introduction to MATLAB.....		1097
Section A.1	Components and Features of MATLAB.....	1098
Section A.2	Methods of Working with MATLAB.....	1103
Section A.3	Vectors and Matrices in MATLAB.....	1105
Section A.4	Matrix Concatenation and Matrix Addressing in MATLAB..	1113
Section A.5	Mathematical Operations in MATLAB.....	1119
Section A.6	Creating Plots in MATLAB.....	1125
Section A.7	Programming with MATLAB.....	1141
Section A.8	Control Structures.....	1147
APPENDIX B Animations.....		1155
INDEX.....		v
INDEX of MATLAB COMMANDS and SCRIPT.....		ix

PART II

NUMERICAL ANALYSIS

Selected Reading for Part II

- BOWEN, RAY M., and C.-C. WANG, *Introduction to Vectors and Tensors, Linear and Multilinear Algebra*, Volume 1, Plenum Press, New York, 1976.
- BOWEN, RAY M., and C.-C. WANG, *Introduction to Vectors and Tensors: Second Edition—Two Volumes Bound as One*, Dover Press, New York, 2009.
- BOWEN, RAY M., *Lectures On Applied Mathematics Part 1: Linear Algebra*,
<http://repository.tamu.edu/handle/1969.1/2500>, 2011.
- BUTCHER, J. C., *Numerical Methods for Ordinary Differential Equations, Second Edition*, John Wiley, 2008
- CHAPRA, STEVEN C, *Applied Numerical Methods with MATLAB, Second Edition*, McGraw Hill, 2008.
- GOLUB, GENE H., and CHARLES F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1996.
- KREIDER, DONALD L., ROBERT G. KULLER, and DONALD R. OSTBERG, *Elementary Differential Equations*, Addison-Wesley, 1968.
- MOLER, CLEVE, *Numerical Computing with MATLAB*, SIAM, Philadelphia, 2004. The electronic edition is at <http://www.mathworks.com/moler>.
- POLKING, JOHN C., and DAVID ARNOLD, *Ordinary Differential Equations using MATLAB, Third Edition*, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2004.
- PRENTER, P. M., *Spines and Variational Methods*, Dover Publications, 2008.
- SHAMPINE, L. F., I. GLADWELL and S. THOMPSON, *Solving ODEs with MATLAB*, Cambridge University Press, 2003.
- STRANG, GILBERT, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, 1986

Chapter 7

ELEMENTS OF NUMERICAL LINEAR ALGEBRA

Part 1 of these Lectures is concerned with Linear Algebra and its applications. A course in numerical analysis makes major use of these concepts. In this Chapter, we shall explain how MATLAB is used to perform many of the matrix computations encountered in the applications of Linear Algebra. It is assumed that the reader has a familiarity of MATLAB at least at the level discussed in Appendix A of this work.

Section 7.1. Elementary Matrix Calculations with MATLAB

It is explained in Section A.3 how to enter a matrix into the workspace of MATLAB. Operations such as sum, difference and multiplication by a scalar are assumed to be familiar to the reader. Next, we shall list and, in some cases, illustrate by examples elementary matrix computations.

Trace: If A is an $N \times N$ matrix, that has been entered into the workspace of MATLAB, its trace, $\text{tr } A$, is calculated by entering **trace(A)** into the command window.

Matrix Product: If A is an $M \times N$ matrix and B is an $N \times K$ matrix, both of which have been entered into the workspace of MATLAB, the *product* of B by A is the $M \times K$ matrix calculated by entering **A*B** into the command window.

Example 7.1.1: If you are given matrices A and B defined by

$$A = \begin{bmatrix} 3 & -2 \\ 2 & 4 \\ 1 & -3 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -2 & 1 & 3 \\ 4 & 1 & 6 \end{bmatrix} \quad (7.1.1)$$

they are entered into the MATLAB workspace by entering

```
>> A=[ 3,-2;2,4;1,-3]
```

```
A =
```

```
3      -2
2       4
1      -3
```

```
>> B=[ -2,1,3;4,1,6]

B =
-2     1     3
 4     1     6
```

The multiplications, AB is implemented by entering

```
>> A*B
```

and the output is

```
ans =
-14     1    -3
 12     6   30
 -14    -2   -15
```

Likewise, if $B*A$ is entered the output is

```
ans =
-1     -1
 20    -22
```

Identity Matrix: The $N \times N$ identity matrix is entered into the workspace of MATLAB by entering **eye(N)**, where **N** is a given positive integer.

Zero Matrix: The $M \times N$ zero matrix is entered into the workspace of MATLAB by entering **zeros(M,N)**, where **M** and **N** are given positive integers. If $M = N$, i.e., the matrix is square, it can be implemented by entering **zeros(N)**.

Transpose: If A is an $M \times N$ matrix, its transpose is the $N \times M$ obtained by interchanging the rows and columns. In MATLAB, the transpose is calculated by entering **A'**. A slight difference over matrix algebra occurs when the elements of the matrix are complex numbers. In this case MATLAB produces the transposed complex conjugate matrix. This result corresponds to the representation of the adjoint of a linear transformation with respect to orthonormal bases.

Example 7.1.2: If A is the matrix

$$A = \begin{bmatrix} 2i & 3 & 7+2i \\ 5 & 4+3i & i \end{bmatrix} \quad (7.1.2)$$

The MATLAB calculation of the transpose is

```

>> A=[2*i,3,7+2*i;5,4+3*i,i]

A =

    0 + 2.0000i    3.0000          7.0000 + 2.0000i
    5.0000          4.0000 + 3.0000i      0 + 1.0000i

>> A'

ans =

    0 - 2.0000i    5.0000
    3.0000          4.0000 - 3.0000i
    7.0000 - 2.0000i      0 - 1.0000i

```

Matrix Inverse: If A is a square matrix that is nonsingular, its inverse is calculated by entering **inv(A)**.

Example 7.1.3: You are given the matrix introduced in Example 1.6.5, namely

$$A = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 1 & 1 \\ -2 & 2 & -1 \end{bmatrix} \quad (7.1.3)$$

This matrix is entered into MATLAB by typing into the command window

```
>> A=[1,3,1;2,1,1;-2,2,-1]
```

The output is

```

A =

    1      3      1
    2      1      1
   -2      2     -1

```

The command **inv(A)** yields

```

>> inv(A)

ans =

    -1.0000    1.6667    0.6667
        0    0.3333    0.3333
    2.0000   -2.6667   -1.6667

```

The output of MATLAB resulting from numerical inputs is a double precision floating point number. MATLAB also has the ability to manipulate *symbols*. We shall utilize this feature at several points in this work. For an elementary matrix such as (7.1.3) it is sometimes to enter its elements as symbol objects rather than numerical objects and calculate the inverse symbolically. This version of the inverse would enter the matrix as

```
>> A=sym([1,3,1;2,1,1;-2,2,-1])
```

The syntax **sym** tells MATLAB that the quantity in the following bracket is a symbol. The output from this entry is

```
A =
[ 1, 3, 1]
[ 2, 1, 1]
[ -2, 2, -1]
```

If we now calculate **inv(A)**, the result is

```
ans =
[ -1, 5/3, 2/3]
[ 0, 1/3, 1/3]
[ 2, -8/3, -5/3]
```

which, while the same as above, displays the inverse in a form such that the entries are rational numbers. While the output is displayed as rational numbers, to MATLAB they are symbols.

In the above example, the matrix was created from symbol elements. If one wants to create a $M \times N$ symbol matrix A without specifying the elements, the following syntax is used

$$\mathbf{A} = \mathbf{sym}(\text{'A'}, [\mathbf{M}, \mathbf{N}]) \quad (7.1.4)$$

The MATLAB output from this script is

```
A =
[ A1_1, A1_2, A1_3, A1_4]
[ A2_1, A2_2, A2_3, A2_4]
[ A3_1, A3_2, A3_3, A3_4]
```

for the choices $M = 3$ and $N = 4$.

Determinant: If A is a square matrix, its determinant is calculated by entering **det(A)**.

The theoretical result developed in Section 1.10 that a matrix is nonsingular if and only if its determinant is zero suggests that one simply calculates the determinant to conclude whether or not the matrix is nonsingular. Unfortunately, for nonsingular matrices with very small determinants, the scheme utilized by MATLAB to generate numbers will place the small nonzero value to zero. We shall look into how MATLAB handles small numbers in Chapter 8. As explained in MATLAB help, the function **cond(x)** can be used to check for singular and nearly singular matrices. Section 7.4 contains a discussion of ill conditioned matrices and it is explained how the value **cond(x)** is utilized.

Adjugate: In Section 1.10 the adjugate of a square matrix A was defined as the transposed matrix of cofactors of the matrix A . MATLAB does not have a built in function that calculates the adjugate but it is not difficult to create a function file that utilizes the **det** on the various submatrices used to calculate the cofactors. One such function file is **adjugate.m** defined by the script

```
function B=adjugate(A)
% adjugate: Calculates the adjugate of the matrix A.
% adjugate=transposed matrix of cofactors
%   B=adjugate(A)=adjugate of A
% input:
%   A = Square matrix
% output:
%   B = adjugate of A
n=size(A,1);
B=zeros(n); % Preallocate
for j=1:n;
    for k=1:n;
        d=[1:n];
        q=[1:n];
        %remove j value from d and k value from q
        d(j)=[];
        q(k)=[];
        % Calculate jk element of adjugate matrix.
        % Note the switch of k,j when defining B in
        % order to built the transpose the of matrix
        % of cofactors.
        B(k,j)=(-1)^(j+k)*det(A(d,q));
    end
end
```

This function file presumes the elements of A are numerical objects, and the output is a matrix of numerical objects. In simple cases, it is convenient to enter A as a matrix of symbolic objects as illustrated with the example above. In this case one would simply replace the line **B=zeros(n)** in the above script with **B=sym(zeros(n))**.

A comment needs to be made about the line of script

```
B=zeros(n); % Preallocate
```

that appears in the script for **adjugate.m**. It reflects a good programming practice that deserves explanation. The script creates an $n \times n$ matrix of zeros and calls it **B**. The remaining script in **adjugate.m** replaces the zeros with the entries that define the adjugate of A . The purpose of preallocating space for the matrix **B** is to avoid the necessity of MATLAB expanding the size of the array repeatedly as it proceeds through the programming loops. For large matrices, resizing the array can affect the performance of the program. This because MATLAB must spend time allocating more memory each time the array size is increased. In addition, the newly allocated memory is likely to be noncontiguous, thus slowing down any operations that MATLAB needs to perform on the array. As in the script above, the preferred method for sizing an array that is expected to grow with subsequent MATLAB steps is to estimate the maximum possible size for the array, and preallocate this amount of memory for it at the time the array is created. In this way, the program performs one memory allocation that reserves one contiguous block.¹ In our many examples in this work, we shall usually preallocate as illustrated above.

Example 7.1.4: If the matrix A defined by (7.1.3) is entered into the MATLAB command window and the command **adjugate(A)** is executed, the output is

```
ans =
-3      5      2
 0      1      1
 6     -8     -5
```

For very large matrices, one would anticipate that round off errors associated with calculating the various determinants could accumulate causing the answer to be inaccurate.

Example 7.1.5: Exercise 1.10.16 involved finding the adjugate and inverse of the 3×3 Vandermonde matrix

$$V = \begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ x_1^2 & x_2^2 & x_3^2 \end{bmatrix} \quad (7.1.5)$$

It is instructive to utilize the symbolic manipulation features of MATLAB to construct the inverse of (7.1.5) and derive the answers to Exercise 1.10.16, equations (1.10.90) and (1.10.91), repeated,

¹ MATLAB's online help, for example at http://www.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html, provides more information about preallocation.

$$\text{adj}V = \begin{bmatrix} x_2x_3(x_3 - x_2) & x_2^2 - x_3^2 & x_3 - x_2 \\ x_1x_3(x_1 - x_3) & x_3^2 - x_1^2 & x_1 - x_3 \\ x_1x_2(x_2 - x_1) & x_1^2 - x_2^2 & x_2 - x_1 \end{bmatrix} \quad (7.1.6)$$

and

$$V^{-1} = \begin{bmatrix} \frac{x_2x_3}{(x_1 - x_2)(x_1 - x_3)} & -\frac{x_2 + x_3}{(x_1 - x_2)(x_1 - x_3)} & \frac{1}{(x_1 - x_2)(x_1 - x_3)} \\ -\frac{x_1x_3}{(x_1 - x_2)(x_2 - x_3)} & \frac{x_1 + x_3}{(x_1 - x_2)(x_2 - x_3)} & -\frac{1}{(x_1 - x_2)(x_2 - x_3)} \\ \frac{x_1x_2}{(x_1 - x_3)(x_2 - x_3)} & -\frac{x_1 + x_2}{(x_1 - x_3)(x_2 - x_3)} & \frac{1}{(x_1 - x_3)(x_2 - x_3)} \end{bmatrix} \quad (7.1.7)$$

The MATLAB script that will generate these answers is

```
clc
clear
syms x1 x2 x3
z=[x1,x2,x3]'
%Define matrix
V=[z.^0,z.^1,z.^2]'
adjugate(V)
adjV=simplify(adjugate(V))
inv(V)
det(V)
invV=simplify(adjV/det(V))
```

Exercises:

7.1.1: Utilize the symbolic version of **adjugate.m** and rework Exercise 6.2.2.

Section 7.2. Systems of Linear Equations

MATLAB is especially well suited for the matrix application of finding the solution, if it exists, of M linear algebraic equations in N unknowns. In Section 1.2, we wrote these equations as

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + A_{13}x_3 + \cdots + A_{1N}x_N &= b_1 \\ A_{21}x_1 + A_{22}x_2 + A_{23}x_3 + \cdots + A_{2N}x_N &= b_2 \\ \vdots & \\ \vdots & \\ A_{M1}x_1 + A_{M2}x_2 + A_{M3}x_3 + \cdots + A_{MN}x_N &= b_M \end{aligned} \quad (7.2.1)$$

and, also, in the matrix form

$$\left[\begin{array}{cccc|c} A_{11} & A_{12} & \cdots & \cdots & A_{1N} \\ A_{21} & A_{22} & & & A_{2N} \\ \vdots & & & & \vdots \\ \vdots & & & & \vdots \\ A_{M1} & A_{M2} & \cdots & \cdots & A_{MN} \end{array} \right] \left[\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_N \end{array} \right] = \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_M \end{array} \right] \quad (7.2.2)$$

The compact form of the matrix (7.2.2) was written, in Section 1.2, as

$$Ax = \mathbf{b} \quad (7.2.3)$$

Or, if we wish to enter the matrices A and \mathbf{b} into MATLAB, we would type the following two lines into the command window:

A=[A₁₁,A₁₂,...,A_{1N};A₂₁,A₂₂,...,A_{2N};...;A_{M1},A_{M2},...,A_{MN}]

b=[b₁;b₂;...;b_M]

As explained in Section 1.2, a *solution* to the $M \times N$ system is a $N \times 1$ column matrix \mathbf{x} that obeys (7.2.3). It is often the case that overdetermined systems do not have a solution. Likewise, undetermined solutions usually do not have a unique solution. If there are an equal number of unknowns as equations, i.e., $M = N$, the system may or may not have a solution. If it has a solution, it may not be unique.

In the special case where A is a square matrix that is also nonsingular, the solution of (7.2.3) is formally

$$\mathbf{x} = A^{-1}\mathbf{b} \quad (7.2.4)$$

The MATLAB form of the solution in the case where A is nonsingular is

$$\mathbf{x} = \text{inv}(\mathbf{A}) * \mathbf{b} \quad (7.2.5)$$

An alternate syntax that replaces (7.2.5) is a *backslash* or *left division* operation defined by

$$\mathbf{x} = \mathbf{A} \setminus \mathbf{b} \quad (7.2.6)$$

It turns out that the backslash method is computationally more efficient than a solution based upon (7.2.5) which requires that the inverse actually be calculated prior to the multiplication by \mathbf{b} . In terms of time of execution and numerical accuracy, (7.2.6) is superior to (7.2.5). Equation (7.2.6) utilizes a Gaussian elimination procedure and, as illustrated in Sections 1.3 through 1.5 does not explicitly calculate the inverse, A^{-1} . Many of the problems in this textbook are elementary, and the advantages of (7.2.6) over (7.2.5) will not be significant.

Example 7.2.1: As an illustration of (7.2.4) and (7.2.5), consider the system of linear equations introduced in Exercise 1.3.2.

$$\begin{aligned} x_2 + x_3 + x_4 &= 0 \\ 3x_1 + 3x_3 - 4x_4 &= 7 \\ x_1 + x_2 + x_3 + 2x_4 &= 6 \\ 2x_1 + 3x_2 + x_3 + 3x_4 &= 6 \end{aligned} \quad (7.2.7)$$

Therefore,

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 3 & 0 & 3 & -4 \\ 1 & 1 & 1 & 2 \\ 2 & 3 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 7 \\ 6 \\ 6 \end{bmatrix} \quad (7.2.8)$$

The matrices A and \mathbf{b} are entered into MATLAB by the script

```
>> A=[0,1,1,1;3,0,3,-4;1,1,1,2;2,3,1,3]
```

```
A =
```

```
0      1      1      1
```

```

3      0      3      -4
1      1      1       2
2      3      1       3

>> b=[0;7;6;6]

b =
0
7
6
6

```

The command

```
>> x=inv(A)*b
```

Produces the solution

```

x =
4.0000
-3.0000
1.0000
2.0000

```

Of course, the command **x=A\ b** yields the same result.

Exercises

7.2.1: Utilize MATLAB to find the solution of the system of equations

$$\begin{aligned}
 4x_1 - 2x_2 - 3x_3 + 6x_4 &= 12 \\
 -6x_1 + 7x_2 + 6.5x_3 - 6x_4 &= -6.5 \\
 x_1 + 7.5x_2 + 6.25x_3 + 5.5x_4 &= 16 \\
 -12x_1 + 22x_2 + 15.5x_3 - x_4 &= 17
 \end{aligned} \tag{7.2.9}$$

7.2.2: Utilize MATLAB to find the solution of the system of equations

$$\begin{aligned}
 2x_1 + x_2 + 2x_3 + x_4 &= 1 \\
 3x_1 + x_3 + x_4 &= 2 \\
 -x_1 + 2x_2 - 2x_3 + x_4 &= 3 \\
 -3x_1 + 2x_2 + 3x_3 + x_4 &= 4
 \end{aligned} \tag{7.2.10}$$

7.2.3: Utilize MATLAB to find the solution to the system of equations

$$\begin{aligned}x_2 + x_3 + x_4 &= 0 \\3x_1 + 3x_3 - 4x_4 &= 7 \\x_1 + x_2 + x_3 + 2x_4 &= 6 \\2x_1 + 3x_2 + x_3 + 3x_4 &= 6\end{aligned}\tag{7.2.11}$$

7.2.4: Utilize MATLAB to find the solution of the system of equations

$$\begin{aligned}-23x_1 + 26x_2 - 42x_3 - 32x_4 - 90x_5 &= -6 \\-2x_1 + x_2 - 3x_3 - 4x_5 &= -2 \\-17x_1 + 19x_2 - 28x_3 - 22x_4 - 63x_5 &= -3 \\-13x_1 + 14x_2 - 24x_3 - 16x_4 - 52x_5 &= -2 \\18x_1 - 20x_2 + 32x_3 + 23x_4 + 69x_5 &= 3\end{aligned}\tag{7.2.12}$$

7.2.5: Utilize MATLAB to find the solution of the system of equations

$$\left[\begin{array}{ccccccccc|c} 4 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 4 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 4 \end{array} \right] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix} = \begin{bmatrix} 27 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \end{bmatrix}\tag{7.2.13}$$

Coefficient matrices of the form (7.2.13) occur in various applications. It is called a *tridiagonal matrix*.

Section 7.3. Additional MATLAB Related Matrix Operations

In Chapter 1, Gaussian Elimination, Gauss-Jordan Elimination and the Reduced Row Echelon Form were discussed. In this Section, certain aspects of these concepts will be revisited. In Section 4.3, we discussed the *Gram-Schmidt orthogonalization* process. The objective is to connect these concepts to MATLAB functions that will perform these operations. As we shall see, in some cases these operations are built into MATLAB and in some cases it is instructive to create the script sufficient to perform the operations.

In Section 1.3, we introduced and illustrated the process of Gaussian Elimination as a technique for solving systems of linear algebraic equations. We first illustrated this technique with Example 1.3.2. We shall repeat this example here except, in this case, utilize MATLAB to do the elimination

Example 7.3.1: Utilize Gaussian Elimination to find the solution of the system

$$\begin{aligned}x_1 + 2x_2 - x_3 &= 1 \\2x_1 - x_2 + x_3 &= 3 \\-x_1 + 2x_2 + 3x_3 &= 7\end{aligned}\tag{7.3.1}$$

It is elementary to recreate the steps used in Section 1.3 to solve this system but, in this case, cause MATLAB to do the elimination. The following MATLAB script achieves this result.

```
%Elementary Gaussian Elimination for N=3.
clc
clear
%Enter the Matrix A a 3X3
A=[1,3,1;2,1,1;-2,2,-1]
%Enter the Matrix b a 3X1
b=[1;5;-8]
%Form the Augmented Matrix <A|b>
M=[A,b]

%Step 1: Build zeros below M(1,1)
%The next line checks if M(1,1) is zero. If so, the
%calculation stops.
if M(1,1)==0,error('You tried to divide by zero'),end
for n=[2 3]
    M(n,:)=M(n,:)-(M(n,1)/M(1,1))*M(1,:);
end

%Step 2: Build zeros below the new M(2,2)
%Again, check if the pivot coefficient is zero.
```

```

if M(2,2)==0,error('You tried to divide by zero'),end
for n=3
    M(n,:)=M(n,:)-(M(n,2)/M(2,2))*M(2,:);
end

%Start back substitution procedure
%Check if the M(3,3) coefficient is zero
if M(3,3)==0,error('You tried to divide by zero'),end
%Calculate the three unknowns
x3=M(3,4)/M(3,3)
x2=(M(2,4)-M(2,3)*x3)/M(2,2)
x1=(M(1,4)-M(1,3)*x3-M(1,2)*x2)/M(1,1)

```

If these commands are pasted into an m-file and executed, the results of Example 1.3.2 are obtained, namely,

```

x3 =
2.0000

x2 =
-1.0000

x1 =
2.0000

```

The above script can easily be generalized to square matrices of arbitrary size. It is an elementary implementation of the Gaussian Elimination because it simply stops when a zero pivot coefficient is encountered. In other words, it does not implement partial pivoting.

It is often convenient to organize MATLAB calculations into function files that will perform important calculations. One such file that generalizes the above calculation is the file **ElementaryGauss.m** which contains the script

```

function x=ElementaryGauss(A,b)
%ElementaryGauss: Gaussian Elimination without partial
%pivoting.
%x=ElementaryGauss(A,b):
%A=NxN coefficient matrix
%b=Nx1 right side column vector
%Output:
% x=solution vector

```

```
%Confirm that A is square and b is the right side
[M,N]=size(A);
Q=size(b);
if M~=N, error('The Matrix A must be square.' );end
if Q~=N, error('The Matrix b is the wrong size' );end
%Form the augmented matrix
M=[A,b];

%Forward Elimination Process: Build zeros below M(k,k)
for k=1:N-1;
    if M(k,k)==0
        M
        error('Cannot divide by zero. Forward elimination fails.')
    end
    for i=k+1:N;
        M(i,k:N+1)=M(i,k:N+1)-(M(i,k)/M(k,k))*M(k,k:N+1);
    end
end
%Back Substitution Process
x=zeros(N,1);
%Check if the M(N,N) coefficient is zero
if M(N,N)==0
    M
    error('Cannot divide by zero. Back substitution fails.')
end
%Calculate the N unknowns
x(N)=M(N,N+1)/M(N,N);
for k=N-1:-1:1
    if M(k,k)==0
        M
        error('Cannot divide by zero. Back substitution fails.')
    end
    x(k)=(M(k,N+1)-M(k,k+1:N)*x(k+1:N))/M(k,k);
end
```

It is instructive to illustrate when the above function file does not work. Example 1.3.3 is such an example. This example concerned finding the solution of the system (1.3.32), repeated,

$$\begin{aligned} 2x_2 + 3x_3 &= 8 \\ 4x_1 + 6x_2 + 7x_3 &= -3 \\ 2x_1 - 3x_2 + 6x_3 &= 5 \end{aligned} \tag{7.3.2}$$

The discussion of this example in Section 1.3 illustrated that partial pivoting was required. If we enter into MATLAB the following

```
>> A=[0,2,3;4,6,7;2,-3,6]
```

```

A =
0      2      3
4      6      7
2     -3      6

>> b=[8;-3;5]

b =
8
-3
5

>> ElementaryGauss(A,b)

```

The output is

```

M =
0      2      3      8
4      6      7     -3
2     -3      6      5

??? Error using ==> ElementaryGauss at 20
Cannot divide by zero. Forward elimination fails.

```

The calculation stopped when it confronted a division by $M_{11} = 0$. A similar result occurs when we attempt to use **ElementaryGauss.m** to work Example 1.3.4. This example involved finding the solution of (1.3.35), repeated,

$$\begin{aligned}
 2x_1 + 3x_2 + x_3 &= 1 \\
 x_1 + x_2 + x_3 &= 3 \\
 3x_1 + 4x_2 + 2x_3 &= 4
 \end{aligned} \tag{7.3.3}$$

As the solution to this example illustrates, the problem arises with the appearance of a zero in the 33 position during the elimination process. This zero appears if we enter into MATLAB the following

```
>> A=[2,3,1;1,1,1;3,4,2]
```

```

A =
2      3      1
1      1      1

```

```

3      4      2

>> b=[1;3;4]

b =

1
3
4

>> ElementaryGauss(A,b)

M =

2.0000    3.0000    1.0000    1.0000
0    -0.5000    0.5000    2.5000
0        0        0        0

??? Error using ==> ElementaryGauss at 28
Cannot divide by zero. Back substitution fails.

```

As pointed out above, the back substitution process failed because of the presence of the zero in the 33 position of the augmented matrix at the end of the forward elimination process.

In Section 1.5, we discussed the Gauss-Jordan Elimination process. It should not be difficult to modify the script above to implement the Gauss-Jordan process in those cases where partial pivoting is not necessary. The further refinement of our script is not necessary at this point because MATLAB has the function **rref** that calculates the *reduced row echelon form* of a matrix. As our many examples in Chapter 1 and later illustrate, this matrix implements the Gauss-Jordan elimination process with partial pivoting. The following examples illustrate the use of **rref**.

Example 7.3.2: We again seek the solution of (7.3.1). The augmented matrix in this case is

$$M = (A|b) = \begin{bmatrix} 1 & 2 & -1 & 1 \\ 2 & -1 & 1 & 3 \\ -1 & 2 & 3 & 7 \end{bmatrix} \quad (7.3.4)$$

If the following script is entered into MATLAB

```

>> A=[1,2,-1;2,-1,1;-1,2,3]

A =

1      2      -1
2     -1       1
-1      2       3

```

```

-1      2      3
>> b=[1;3;7]

b =
1
3
7

>> M=[A,b]

M =
1      2      -1      1
2      -1      1      3
-1      2      3      7

```

The command

```

>> rref(A)

ans =
1      0      0
0      1      0
0      0      1

>> rref(M)

ans =
1      0      0      1
0      1      0      1
0      0      1      2

```

The last output is the augmented matrix corresponding to the solution above, namely,

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \quad (7.3.5)$$

Example 7.3.3: The augmented matrix based upon the equations (7.3.2) is

$$M = (A|b) = \begin{bmatrix} 0 & 2 & 3 & 8 \\ 4 & 6 & 7 & -3 \\ 2 & -3 & 6 & 5 \end{bmatrix} \quad (7.3.6)$$

The command **rref(M)** yields

```
>> rref(M)

ans =

1.0000      0      0    -5.4239
0    1.0000      0    0.0217
0      0    1.0000    2.6522
```

This is the answer given in Exercise 1.3.1 where this problem was addressed.

Example 7.3.4: For the system of equations (7.3.3), the augmented matrix is

$$M = (A|b) = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 2 & 4 \end{bmatrix} \quad (7.3.7)$$

The command **rref(M)** yields

```
ans =

1      0      2      8
0      1     -1     -5
0      0      0      0
```

This result is equivalent to the answer obtained earlier, equation (1.3.36).

In Section 4.3 we discussed the *Gram-Schmidt orthogonalization* process. The basic idea is that if one is given a set $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_K\}$ of K linearly independent vectors, one can construct an orthonormal set of linearly independent vectors $\{\mathbf{i}_1, \dots, \mathbf{i}_K\}$ such that

$\text{Span}(\mathbf{e}_1, \dots, \mathbf{e}_K) = \text{Span}(\mathbf{i}_1, \dots, \mathbf{i}_K)$. In the matrix context, one can select a basis for the underlying vector space and construct a matrix $A \in \mathcal{M}^{N \times K}$, where N is the dimension of the vector space. Each column of the $N \times K$ matrix A corresponds to the projection of one of the vectors in the set $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_K\}$ into basis of the vector space. Given the matrix $A \in \mathcal{M}^{N \times K}$, the Gram-Schmidt process constructs a $N \times K$ matrix Q from A whose columns are mutually orthogonal. As explained in Section 4.3, the key formula in the construction is equation (4.3.22), repeated,

$$\mathbf{i}_k = \frac{\mathbf{e}_k - \sum_{j=1}^{k-1} \langle \mathbf{e}_k, \mathbf{i}_j \rangle \mathbf{i}_j}{\left\| \mathbf{e}_k - \sum_{j=1}^{k-1} \langle \mathbf{e}_k, \mathbf{i}_j \rangle \mathbf{i}_j \right\|} \quad \text{for } k = 1, 2, \dots, K \quad (7.3.8)$$

Examples 4.3.1 and 4.3.3 illustrate how this equation can be used. It is not difficult to implement the calculation with the following function m-file.²

```
function Q = GmSchmidt(A)
% Gram-Schmidt orthonormalization method for column
% matrices
% Q=GmSchmidt(A)
% input:
% A an NxK matrix of real or complex numbers.
% A can be a symbolic matrix
% output:
% Q a NxK matrix of K orthonormal column vectors
% Q is a symbolic matrix if A is a symbolic matrix
if nargin~=1
    error('Only one matrix argument is allowed')
end
if size(A,2)<2
    error('At least two columns are required.');
end
[N,K]=size(A);
P(:,:,1)=eye(N);
D(:,1)=A(:,1);
Q(:,1)=D(:,1)/norm(D(:,1));
for i=2:K;
    P(:,:,i)=P(:,:,i-1)-Q(:,i-1)*Q(:,i-1)';
    D(:,i)=P(:,:,i)*A(:,i);
    Q(:,i)=D(:,i)/norm(D(:,i));
end
```

The script in **GmSchmidt.m** builds the results (7.3.8) by capitalizing on the matrix form of the tensor product definition in Section 6.7. That definition allows (7.3.8) to be written

$$\mathbf{i}_k = \frac{\mathbf{P}_k \mathbf{e}_k}{\|\mathbf{P}_k \mathbf{e}_k\|} \quad \text{for } k = 1, 2, \dots, K \quad (7.3.9)$$

where \mathbf{P}_k , for $k = 1, 2, \dots, K$, is the orthogonal projection

² The script for **GmSchmidt.m** utilizes the built in MATLAB function **norm**. This function simply performs the calculation given by equation (4.1.23).

$$\mathbf{P}_k = \mathbf{I} - \sum_{j=1}^{k-1} \mathbf{i}_j \otimes \mathbf{i}_j \quad (7.3.10)$$

The expressions (7.3.9) and (7.3.10) were developed in Exercise 6.7.5. Of course, the function m-file **Gmschmidt.m** can be used to work Exercises 4.3.1 through 4.3.5 as well as the more complicated problem, Exercise 4.15.2.

There is a numerical problem with an implementation of the Gram Schmidt process based upon (7.3.8). As briefly mentioned in Section 4.3, for some problems the round off error can cause the resulting vectors $\{\mathbf{i}_1, \dots, \mathbf{i}_k\}$ not to be orthogonal. The *modified* Gram Schmidt method is a modification that attempts to minimize the round off errors.³ This is achieved by writing the projection (7.3.10) in the form

$$\mathbf{P}_k = \mathbf{I} - \sum_{j=1}^{k-1} \mathbf{i}_j \otimes \mathbf{i}_j = (\mathbf{I} - \mathbf{i}_{k-1} \otimes \mathbf{i}_{k-1})(\mathbf{I} - \mathbf{i}_{k-2} \otimes \mathbf{i}_{k-2}) \cdots (\mathbf{I} - \mathbf{i}_1 \otimes \mathbf{i}_1) \quad (7.3.11)$$

The fact that the vectors $\{\mathbf{i}_1, \dots, \mathbf{i}_k\}$ are orthogonal and the identity (6.7.7) cause equation (7.3.11) to be valid. Given (7.3.11), we can write (7.3.9) as

$$\mathbf{i}_k = \frac{\mathbf{P}_k \mathbf{e}_k}{\|\mathbf{P}_k \mathbf{e}_k\|} = \frac{(\mathbf{I} - \mathbf{i}_{k-1} \otimes \mathbf{i}_{k-1})(\mathbf{I} - \mathbf{i}_{k-2} \otimes \mathbf{i}_{k-2}) \cdots (\mathbf{I} - \mathbf{i}_1 \otimes \mathbf{i}_1) \mathbf{e}_k}{\|(\mathbf{I} - \mathbf{i}_{k-1} \otimes \mathbf{i}_{k-1})(\mathbf{I} - \mathbf{i}_{k-2} \otimes \mathbf{i}_{k-2}) \cdots (\mathbf{I} - \mathbf{i}_1 \otimes \mathbf{i}_1) \mathbf{e}_k\|} \quad (7.3.12)$$

While (7.3.9), projects \mathbf{e}_k into the direction defined by the orthogonal projection \mathbf{P}_k , the equivalent formula (7.3.12) achieves the same result by a series of orthogonal projections starting with $\mathbf{I} - \mathbf{i}_1 \otimes \mathbf{i}_1$ applied to \mathbf{e}_k , followed by $\mathbf{I} - \mathbf{i}_2 \otimes \mathbf{i}_2$ applied to $(\mathbf{I} - \mathbf{i}_1 \otimes \mathbf{i}_1) \mathbf{e}_k$ and so forth. The MATLAB script that will implement the modified Gram Schmidt procedure is⁴

```
function Q = GmSchmidtModified(A)
    % Modified Gram-Schmidt orthonormalization method
    % for column matrices
    % Q=GmSchmidtModified(A)
    % input:
    % A an NxK matrix of real or complex numbers.
    % A can be a symbolic matrix
    % output:
    % Q a NxK matrix of K orthonormal column vectors
    % Q is a symbolic matrix if A is a symbolic matrix
```

³ A discussion of the modified Gram Schmidt process can be found in many references. The textbook, Strang, Gilbert, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, 1986, has a good discussion. Another good discussion can be found in the textbook, Golub, Gene H., and Charles F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1996.

⁴ A numerical example that compares the Gram Schmidt process to the modified Gram Schmidt process can be found at http://ocw.mit.edu/courses/mathematics/18-335j-introduction-to-numerical-methods-fall-2010/lecture-notes/MIT18_335JF10_lec10a_hand.pdf.

```

if nargin~=1
    error('Only one matrix argument is allowed')
end
if size(A,2)<2
    error('At least two columns are required.');
end
[N,K]=size(A);
P(:,:,1)=eye(N);
D(:,:,1)=A(:,:,1);
Q(:,:,1)=D(:,:,1)/norm(D(:,:,1));
for k=2:K
    P(:,:,k)=(eye(N)-Q(:,:,k-1)*Q(:,:,k-1)')*P(:,:,k-1);
    D(:,:,k)=P(:,:,k)*A(:,:,k);
    Q(:,:,k)=D(:,:,k)/norm(D(:,:,k));
end

```

As one might expect, MATLAB has a built in function that will implement the Gram Schmidt process. It is the function `qr` and has the syntax

$$[Q, R] = \text{qr}(A) \quad (7.3.13)$$

The function `qr` is referred to as the *orthogonal-triangular decomposition* by MATLAB. When A is an $N \times K$ matrix, the output is an $N \times K$ upper triangular matrix R and an $N \times N$ orthogonal matrix Q with the property

$$A = QR \quad (7.3.14)$$

If $N > K$, which is our case, the syntax

$$[Q, R] = \text{qr}(A, 0) \quad (7.3.15)$$

produces the $N \times K$ matrix Q and the $K \times K$ matrix R that provides the QR decomposition discussed in Exercise 4.14.4.

Example 7.3.5: In Example 4.3.3, we utilized the Gram Schmidt process to generate the orthogonal polynomials known as Legendre Polynomials. In that example it was mentioned that there are other sets of orthogonal polynomials that are important in applications. The ones that were mentioned are Chebyshev, Gegenbaur, Hermite, Jacobi and Leguerre polynomials. It is possible to utilize MATLAB and the Gram Schmidt process to generate these polynomials. For example, the first six Chebyshev polynomials are given by⁵

⁵ Information about the Russian mathematician Pafnuty Chebyshev can be found at http://en.wikipedia.org/wiki/Pafnuty_Chebyshev.

$$\begin{aligned}
 T_0(x) &= 1 \\
 T_1(x) &= x \\
 T_2(x) &= 2x^2 - 1 \\
 T_3(x) &= 4x^3 - 3x \\
 T_4(x) &= 8x^4 - 8x^2 + 1 \\
 T_5(x) &= 16x^5 - 20x^3 + 5x
 \end{aligned} \tag{7.3.16}$$

They obey a recursion relationship

$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x) \quad \text{for } k = 1, 2, 3, \dots \tag{7.3.17}$$

and the integral condition

$$\int_{x=-1}^{x=1} T_k(x)^2 \frac{dx}{\sqrt{1-x^2}} = \begin{cases} \pi & \text{for } k = 0 \\ \frac{\pi}{2} & \text{for } k \neq 0 \end{cases} \tag{7.3.18}$$

The inner product that is assigned to the set of polynomials \mathcal{P} on the interval $[-1, 1]$ is a special case of (4.1.17) given by

$$\langle p, q \rangle = \int_{-1}^1 w(x) p(x) q(x) dx \tag{7.3.19}$$

where, for the Chebyshev polynomials, the weighting function is

$$w(x) = \frac{1}{\sqrt{1-x^2}} \tag{7.3.20}$$

Given the inner product (7.3.19), equation (7.3.18) shows that the Chebyshev polynomials are not normalized to have unit magnitude. As with Example 4.3.3, we can start with the monomials

$$p_k(x) = x^{k-1} \quad \text{for } k = 1, 2, 3, 4, 5, 6 \tag{7.3.21}$$

and apply the Gram Schmidt process utilizing the inner product (7.3.19). The result is the orthonormal set $\{i_1, i_2, i_3, i_4, i_5, i_6\}$ where

$$\begin{aligned}
 i_1(x) &= \frac{1}{\sqrt{\pi}} \\
 i_2(x) &= \sqrt{\frac{2}{\pi}}x \\
 i_3(x) &= \sqrt{\frac{2}{\pi}}(2x^2 - 1) \\
 i_4(x) &= \sqrt{\frac{2}{\pi}}(4x^3 - 3x) \\
 i_5(x) &= \sqrt{\frac{2}{\pi}}(8x^4 - 8x^2 + 1) \\
 i_6(x) &= \sqrt{\frac{2}{\pi}}(16x^5 - 20x^3 + 5x)
 \end{aligned} \tag{7.3.22}$$

Other than being normalized to have magnitude of one, these polynomials and the Chebyshev polynomials defined by (7.3.16) are equivalent. The following script utilizes the symbolic features of MATLAB to generate the above results

```

clc
clear
syms x
%Define weighting function
w=1/sqrt(1-x^2)
K=6
a=-1
b=1
%Generate the K polynomials of max degree K-1
for k=1:K
    p(k)=x^(k-1)
end
%Introduce a square K-1 by K-1 matrix
I=sym(zeros(K-1)); %Preallocate
%Introduce two column matrices Kx1
d=sym(zeros(K,1)); %Preallocate
f=sym(zeros(K,1)); %Preallocate
%Calculating the first vector f(1)
f(1)=p(1)/(sqrt(int(w*p(1)*p(1),x,a,b)))
%Generate the other elements of the column matrix f
for q=2:K
    for j=1:q-1
        I(q-1,j)=int(w*p(q)*f(j),x,a,b)
    end
    d(q)=p(q)-I(q-1,:)*f(1:K-1)
    f(q)=d(q)/sqrt(int(w*d(q)*d(q),x,a,b))
end

```

```
f=simplify(f)
```

Exercises

7.3.1: Use the MATLAB **rref** command to try to solve the following system of equations

$$\begin{aligned} 4x - 4y - 8z &= 27 \\ 2y + 2z &= -6 \\ x - 2y - 3z &= 10 \end{aligned} \tag{7.3.23}$$

7.3.2: Use the MATLAB **rref** command to try to solve the following system of equations

$$\begin{aligned} x_1 + 2x_2 - 4x_3 + 3x_4 + 9x_5 &= 1 \\ 4x_1 + 5x_2 - 10x_3 + 6x_4 + 18x_5 &= 4 \\ 7x_1 + 8x_2 - 16x_3 &= 7 \end{aligned} \tag{7.3.24}$$

7.3.3: The first six Hermite polynomials are given by ⁶

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= 2x \\ H_2(x) &= 4x^2 - 2 \\ H_3(x) &= 8x^3 - 12x \\ H_4(x) &= 16x^4 - 48x^2 + 12 \\ H_5(x) &= 32x^5 - 160x^3 + 120x \end{aligned} \tag{7.3.25}$$

They obey a recursion relationship

$$H_{k+1}(x) = 2xH_k(x) - \frac{dH_k(x)}{dx} \quad \text{for } k = 0, 1, 2, 3, \dots \tag{7.3.26}$$

and the integral condition

$$\int_{x=-\infty}^{x=\infty} H_k(x)^2 e^{-x^2} dx = 2^k k! \sqrt{\pi} \tag{7.3.27}$$

The inner product that is assigned to the set of polynomials \mathcal{P} on the interval $[-1, 1]$ is a special case of (4.1.17), where, for the Hermite polynomials, the weighting function is

⁶ Information about the French mathematician Charles Hermite can be found at http://en.wikipedia.org/wiki/Charles_Hermite.

$$w(x) = e^{-x^2} \quad (7.3.28)$$

Given the inner product (4.1.17), equation (7.3.27) shows that the Hermite polynomials are also not normalized to have unit magnitude. As with Example 4.3.3, we can start with the monomials

$$p_k(x) = x^{k-1} \quad \text{for } k = 1, 2, 3, 4, 5, 6 \quad (7.3.29)$$

and apply the Gram Schmidt process utilizing the given inner product. Show that the result is the orthonormal set $\{i_1, i_2, i_3, i_4, i_5, i_6\}$ where

$$\begin{aligned} i_1(x) &= \frac{1}{\pi^{1/4}} \\ i_2(x) &= \frac{1}{2^{1/2}\pi^{1/4}}(2x) \\ i_3(x) &= \frac{1}{2^{3/2}\pi^{1/4}}(4x^2 - 2) \\ i_4(x) &= \frac{1}{4\sqrt{3}\pi^{1/4}}(8x^3 - 12x) \\ i_5(x) &= \frac{1}{2^{7/2}\sqrt{3}\pi^{1/4}}(16x^4 - 48x^2 + 12) \\ i_6(x) &= \frac{1}{16\sqrt{15}\pi^{1/4}}(32x^5 - 160x^3 + 120x) \end{aligned} \quad (7.3.30)$$

7.3.4: The first six Laguerre polynomials are given by ⁷

$$\begin{aligned} L_0(x) &= 1 \\ L_1(x) &= 1 - x \\ L_2(x) &= \frac{1}{2}(x^2 - 4x + 2) \\ L_3(x) &= \frac{1}{6}(-x^3 + 9x^2 - 18x + 6) \\ L_4(x) &= \frac{1}{24}(x^4 - 16x^3 + 72x^2 - 96x + 24) \\ L_5(x) &= \frac{1}{120}(-x^5 + 25x^4 - 200x^3 + 600x^2 - 600x + 120) \end{aligned} \quad (7.3.31)$$

They obey a recursion relationship

⁷ Information about the French mathematician Edmond Laguerre can be found at http://en.wikipedia.org/wiki/Edmond_Laguerre.

$$(k+1)L_{k+1}(x) = (2k+1-x)L_k(x) - kL_{k-1} \quad \text{for } k = 1, 2, 3, \dots \quad (7.3.32)$$

and the integral condition

$$\int_{x=0}^{x=\infty} L_k(x)^2 e^{-x} dx = 1 \quad (7.3.33)$$

The inner product that is assigned to the set of polynomials \mathcal{P}_∞ on the interval $[-1, 1]$ is a special case of (4.1.17), where, for the Laguerre polynomials, the weighting function is

$$w(x) = e^{-x} \quad (7.3.34)$$

Given the inner product (4.1.17), equation (7.3.27) shows that the Hermite polynomials are normalized to have unit magnitude. As a slight modification of Example 4.3.3, start with the monomials

$$p_k(x) = (-x)^{k-1} \quad \text{for } k = 1, 2, 3, 4, 5, 6 \quad (7.3.35)$$

and apply the Gram Schmidt process utilizing the given inner product. Show that the result is the orthonormal set $\{L_0, L_1, L_2, L_3, L_4, L_5\}$ where the elements are given by (7.3.31)

Section 7.4. Ill Conditioned Matrices

Given the information we have accumulated about the solution of matrix equations

$$\mathbf{Ax} = \mathbf{b} \quad (7.4.1)$$

in Chapter 1, Chapter 2, parts of Chapters 3 through 6 plus the MATLAB tools explained in Sections 7.2 and 7.3 one would get the impression that finding the solution to (7.4.1) is entirely routine. However, there are still a few issues that one can encounter. The numbers that make up the matrix A are often not known precisely. If, for example, they are the result of experiments the numbers observed will have intrinsic inaccuracies. If they are the result of calculations, there is the inevitable possibility of round off and other types of errors. The question arises whether or not the answers produced by MATLAB or any other solution procedure are significantly dependent upon small inaccuracies in the elements of A . The following simple example shows that for some matrices the dependence is strong.

Example 7.4.1: Consider the following two systems

$$\begin{bmatrix} 200 & -101 \\ -400 & 201 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 100 \\ -100 \end{bmatrix} \quad (7.4.2)$$

and

$$\begin{bmatrix} 201 & -101 \\ -400 & 201 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 100 \\ -100 \end{bmatrix} \quad (7.4.3)$$

The question is whether or not the small change in the 11 element makes a significant change in the answers. If we use the simple formula for the inverse given in Exercise 1.1.7, the two answers are

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \frac{1}{(200)(201) - (400)(101)} \begin{bmatrix} 201 & 101 \\ 400 & 200 \end{bmatrix} \begin{bmatrix} 100 \\ -100 \end{bmatrix} = -\frac{1}{200} \begin{bmatrix} 10000 \\ 20000 \end{bmatrix} = \begin{bmatrix} -50 \\ -100 \end{bmatrix} \quad (7.4.4)$$

and

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \frac{1}{(201)(201) - (400)(101)} \begin{bmatrix} 201 & 101 \\ 400 & 201 \end{bmatrix} \begin{bmatrix} 100 \\ -100 \end{bmatrix} = \frac{1}{19900} \begin{bmatrix} 10000 \\ 19900 \end{bmatrix} = \begin{bmatrix} 10000 \\ 19900 \end{bmatrix} \quad (7.4.5)$$

Clearly, the small change in one element of this particular matrix does produce a significantly different result. The particular matrix we start with in this example is known as an *ill conditioned matrix*. More formally, our definition is as follows:

Definition: An *ill-conditioned matrix* is one where *small changes* in the coefficient matrix of the system (7.4.1) produces *large changes* in the solution.

A similar question arises when one asked whether or not small changes in the matrix \mathbf{b} in (7.4.1) produces large changes in the solution. One can easily modify the above example by a small adjustment in \mathbf{b} and confirm that in this example the answer is certainly yes. Sometimes ill-conditioned systems are referred to when small changes in the matrix A and/or small changes in the matrix \mathbf{b} produce large changes in the answer.

The question of whether or not a square matrix is ill-condition is one that can be answered by utilizing the concept of a norm of a matrix. We first introduced this concept in Section 4.1 and, again, for linear transformations in Section 4.10. Since we are examining a matrix equation (7.4.1), the norm we have adopted is given by equation (4.1.24), repeated,

$$\|A\| = \sqrt{\langle A, A \rangle} = \sqrt{\text{tr}(AA^T)} = \left(\sum_{j=1}^N \sum_{k=1}^N A_{jk} \bar{A}_{jk} \right)^{1/2} = \left(\sum_{j=1}^N \sum_{k=1}^N |A_{jk}|^2 \right)^{1/2} \quad (7.4.6)$$

Also, the norm of the matrix $\mathbf{x} \in \mathcal{M}^{N \times 1}$ is given by (4.1.23), repeated,

$$\|\mathbf{x}\| = \left(\sum_{j=1}^N x_j \bar{x}_j \right)^{1/2} = \left(\sum_{j=1}^N |x_j|^2 \right)^{1/2} \quad (7.4.7)$$

As explained in Section 4.10, the norms (7.4.6) and (7.4.7) obey the inequality (4.10.4), repeated,

$$\|\mathbf{Ax}\| \leq \|A\| \|\mathbf{x}\| \quad (7.4.8)$$

Given these definitions, we shall show that

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A\| \|A^{-1}\| \frac{\|\Delta A\|}{\|A\|} \quad (7.4.9)$$

Relative Error in Solution Relative Error in A

Equation (7.4.9) gives a bound on the change in the column vector \mathbf{x} , normalized by the length of \mathbf{x} , resulting from a change in the coefficient matrix A . The coefficient $\|A\| \|A^{-1}\|$ amplifies or diminishes the error in A . In a sense, it determines how an error propagates to an error in \mathbf{x} . This coefficient, or one equivalent, is called the *condition number* for the matrix A . We shall say more about this coefficient in the following. First, however, the derivation of (7.4.9) will be given.

We begin the derivation with a solution of (7.4.1) which we shall write

$$\mathbf{x} = A^{-1}\mathbf{b} \quad (7.4.10)$$

We are interested in determining the change in the solution when the matrix A is replaced by a matrix $A + \Delta A$. The change in the solution will be denoted by $\Delta \mathbf{x}$ and, from (7.4.1), will be governed by

$$(A + \Delta A)(\mathbf{x} + \Delta \mathbf{x}) = \mathbf{b} \quad (7.4.11)$$

If we utilize (7.4.10), it follows from (7.4.11) that

$$A\Delta \mathbf{x} = -\Delta A\mathbf{x} \quad (7.4.12)$$

Because A is nonsingular, we can write (7.4.12) as

$$\Delta \mathbf{x} = -A^{-1}\Delta A\mathbf{x} \quad (7.4.13)$$

We can next use (7.4.8) twice and write the norm $\|\Delta \mathbf{x}\|$ as

$$\|\Delta \mathbf{x}\| = \|A^{-1}\Delta A\mathbf{x}\| \leq \|A^{-1}\| \|\Delta A\mathbf{x}\| \leq \|A^{-1}\| \|\Delta A\| \|\mathbf{x}\| \quad (7.4.14)$$

Equation (7.4.14) is easily rewritten in the dimensionless form (7.4.9).

As mentioned above, equation (7.4.9) places a bound on the relative error of the solution in terms of the relative error of the matrix A . The coefficient $\|A\| \|A^{-1}\|$ is the *condition number* and we shall give it the symbol defined by

$$\text{Cond}[A] = \|A\| \|A^{-1}\| \quad (7.4.15)$$

It has the property that

$$\text{Cond}[A] \geq 1 \quad (7.4.16)$$

For a *well-conditioned matrix*, this condition number is of the order of unity. Of course, an ill-conditioned matrix the condition number is large.

Example 7.4.2: The norm of the matrix

$$A = \begin{bmatrix} 200 & -101 \\ -400 & 201 \end{bmatrix} \quad (7.4.17)$$

that was introduced in Example 7.4.1 is easily shown to be

$$\|A\| = 500.60 \quad (7.4.18)$$

The inverse of A is the matrix

$$A^{-1} = -\frac{1}{200} \begin{bmatrix} 201 & 101 \\ 400 & 200 \end{bmatrix} \quad (7.4.19)$$

The norm of A^{-1} is easily shown to be

$$\|A^{-1}\| = 2.503 \quad (7.4.20)$$

which yields a condition number of

$$\text{Cond}[A] = \|A\| \|A^{-1}\| = 1253 \quad (7.4.21)$$

As one would expect, MATLAB has built in functions that calculate the condition number. It will also warn you, when a matrix inversion is attempted, if you are dealing with an ill conditioned matrix. Because there are several norms, all equivalent for finite dimensional vector spaces, it is not surprising that MATLAB will calculate the condition number utilizing a variety of norms. The norm we have used above goes by the name of *Frobenius norm*.⁸

Given a matrix $A \in \mathcal{M}^{M \times N}$, MATLAB will calculate the Frobenius norm by the command

$$\text{norm}(A, 'fro') = \|A\| \quad (7.4.22)$$

Another norm that is common is the so called *column sum* norm. It is defined by

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_i^m |A_{ij}| \quad (7.4.23)$$

What this symbolic equation actually means is that one forms, for each column, the sum $\sum_i^m |A_{ij}|$.

The norm is the maximum value of the various sums. In the case of a 3×3 matrix,

⁸ Information about the German mathematician, Ferdinand Georg Frobenius, can be found at http://en.wikipedia.org/wiki/Ferdinand_Georg_Frobenius.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$\rightarrow |a_{13}| + |a_{23}| + |a_{33}| \equiv A_3$
 $\rightarrow |a_{12}| + |a_{22}| + |a_{32}| \equiv A_2$
 $\rightarrow |a_{11}| + |a_{21}| + |a_{31}| = A_1$

$$\|A\|_1 = \max(A_1, A_2, A_3)$$

Given a matrix $A \in \mathbb{M}^{M \times N}$, MATLAB will calculate the column sum norm by the command

$$\text{norm}(A, 1) = \|A\|_1 \quad (7.4.24)$$

There are other definitions of norm that can be found in the references. For example, the *row sum norm* $\|A\|_\infty$ is defined by

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_j^n |A_{ij}| \quad (7.4.25)$$

MATLAB will calculate the row sum norm by the command

$$\text{norm}(A, \text{Inf}) = \|A\|_\infty \quad (7.4.26)$$

The *spectral norm* is defined by

$$\|A\|_2 = \sqrt{\lambda_{\max}} \quad (7.4.27)$$

where λ_{\max} is the largest eigenvalue of the Hermitian matrix $A^* A$. MATLAB will calculate the spectral norm by the command

$$\text{norm}(A, 2) = \|A\|_2 \quad (7.4.28)$$

As indicated above, one can establish that for finite dimensional vector spaces, the various norm definitions are equivalent in the sense of how they characterize the size of a matrix. In other words, even though they will give different numbers for the size, conclusions about the condition of the matrix are the same.

Within MATLAB, the command **cond** is used to calculate the condition number of a matrix. For the various norms listed above, MATLAB will calculate the condition number as follows:

For the Forbenius norm defined by (7.4.6)

$$\mathbf{cond}(\mathbf{A}, \text{'fro'}) = \|A\| \|A^{-1}\| \quad (7.4.29)$$

For the column sum norm defined by (7.4.23)

$$\mathbf{cond}(\mathbf{A}, 1) = \|A\|_1 \|A^{-1}\|_1 \quad (7.4.30)$$

For the row sum norm defined by (7.4.25)

$$\mathbf{cond}(\mathbf{A}, \text{inf}) = \|A\|_\infty \|A^{-1}\|_\infty \quad (7.4.31)$$

For the spectral norm defined by (7.4.27)

$$\mathbf{cond}(\mathbf{A}, 2) = \|A\|_2 \|A^{-1}\|_2 \quad (7.4.32)$$

As mentioned, for a well-conditioned matrix, the condition number is of the order of one. MATLAB also uses the command **rcond** that is called the *reciprocal condition number* and yields a number calculated from the column sum norm by the formula

$$\mathbf{rcond}(\mathbf{A}) = \frac{1}{\|\mathbf{A}\|_1 \|\mathbf{A}^{-1}\|_1} \quad (7.4.33)$$

Example 7.4.3: An example ill conditional matrix is the so called *Hilbert matrix*.⁹ This matrix is the *symmetric matrix* defined by

⁹ Information about the German mathematician, David Hilbert, can be found at http://en.wikipedia.org/wiki/David_Hilbert. He introduced the Hilbert matrix in 1894.

$$A = \left[\frac{1}{i+j-1} \right] = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdot & \cdot & \cdot & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdot & \cdot & \cdot & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdot & \cdot & \cdot & \frac{1}{n+2} \\ \cdot & \cdot & \cdot & & & & \\ \cdot & \cdot & \cdot & & & & \\ \cdot & \cdot & \cdot & & & & \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdot & \cdot & \cdot & \frac{1}{2n-1} \end{bmatrix} \quad (7.4.34)$$

Our objective is to calculate the condition numbers for a 10×10 Hilbert matrix. The following script will generate this matrix and calculate the condition numbers based upon the four norms discussed above

```

clc
clear
for n=[1:1:10]
    for m=[1:1:10]
        A(m,n)=[1/(m+n-1)]
    end
end
%Forbenius Norm
condfro=cond(A,'fro')
%Column Sum Norm
cond1=cond(A,1)
%Row Sum Norm
cond2=cond(A,inf)
%Spectral Norm
condspect=cond(A,2)

```

Yields the output

```
condfro =
```

```
1.6332e+013
```

```
cond1 =
```

```
3.5354e+013
```

```
cond2 =
3.5354e+013
```

```
condspect =
1.6025e+013
```

The symmetry of the Hilbert matrix is the reason the condition numbers based upon the column sum norm and the row sum norm are the same. In any case, the large numbers calculated for the four norms displays that the Hilbert matrix is ill conditioned.

Example 7.4.4: Another example of an ill conditional matrix is the Vandermonde matrix introduced in Section 1.10. This matrix is given by equation (1.10.33), repeated,

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & \cdot & \cdot & x_N \\ x_1^2 & x_2^2 & x_3^2 & \cdot & \cdot & x_N^2 \\ \cdot & & & \cdot & & \cdot \\ \cdot & & & & \cdot & \cdot \\ x_1^{N-1} & x_2^{N-1} & x_3^{N-1} & \cdot & \cdot & x_N^{N-1} \end{bmatrix} \quad (7.4.35)$$

As with Example 7.4.3, our objective is to calculate the condition numbers for a 10×10 Vandermonde matrix.¹⁰ The following script will generate this matrix for a given set of numbers $(x_1, x_2, x_3, \dots, x_N)$ and calculate the condition numbers based upon the four norms discussed above

```
clc
clear
x=[1,2,3,4,5,6,7,8,9,10]
A=[x.^0;x.^1;x.^2;x.^3;x.^4;x.^5;x.^6;x.^7;x.^8;x.^9]
%Forbenius Norm
condfro=cond(A,'fro')
%Column Sum Norm
cond1=cond(A,1)
%Row Sum Norm
cond2=cond(A,inf)
%Spectral Norm
condspect=cond(A,2)
```

Yields the output

¹⁰ MATLAB has a command **vander** that will also generate the Vandermonde matrix.

```
condfro =  
2.1068e+12  
  
cond1 =  
3.3064e+12  
  
cond2 =  
3.6366e+12  
  
condspect =  
2.1063e+12
```

The large numbers calculated for these four norms illustrate that the Vandermonde is also ill conditioned.

We motivated the discussion in this section, with the observation that the numbers that make up the matrix A are often not known precisely. We observed that often these numbers are the result of experiments the numbers observed will have intrinsic inaccuracies. We also observed that round off errors during the inversion process create similar inaccuracies. We shall return to a discussion of round off errors and other errors that arise from the way computers generate real numbers in Chapter 8.

Section 7.5. Additional Discussion of LU Decomposition

In Section 1.7, the concept of a *LU* decomposition was introduced. We discussed two cases. The first, which we called an *elementary LU decomposition* arose when one was given an $M \times N$ matrix A such that $N \geq M - 1$, with the property that a Gaussian Elimination method without partial pivoting (row switching) could be used to find an upper triangular $M \times N$ matrix U and a lower triangular nonsingular $M \times M$ matrix L with 1s down the diagonal such that

$$A = LU \quad (7.5.1)$$

The second, which we called a *generalized LU decomposition* arose when one was given an $M \times N$ matrix A such that $N \geq M - 1$, with the property that a Gaussian Elimination method *with* partial pivoting could be used to find an upper triangular $M \times N$ matrix U and a lower triangular nonsingular $M \times M$ matrix L with 1s down the diagonal and a $M \times M$ permutation matrix P such that

$$PA = LU \quad (7.5.2)$$

We shall first discuss the elementary *LU* decomposition. As explained, the computation method to obtain (7.5.1) works when A can be reduced to upper triangular form without using partial pivoting (i.e. row switching). In those cases where the decomposition exists, the requirement that the diagonal elements of L be 1 results in a unique decomposition. Because the foundation of the *LU* decomposition is Gaussian Elimination, we can create a function m-file to perform the decomposition, or, at least, tell us when the decomposition does not exist. One such m-file is **elemlu.m** which contains the script

```

function [L,U] = elemlu(A);
%elemlu: Performs LU Decomposition of MxN matrix.
%[L,U]=elemlu(A)
%A=MxN matrix with N greater than or equal to M-1
%Output:
% L=nonsingular lower triangular MxM matrix with 1s on diagonal
% U=upper triangular MxN matrix

[M,N] = size(A);
if N<M-1
    error('Number of columns cannot exceed Number of rows minus one')
end
L = eye(M); %Preallocates L as identity matrix
%Forward Elimination Process: Build zeros below A(k,k)
for k=1:N-1;
    for i=k+1:M;
        if A(k,k)==0
            A

```

```

        error('Cannot divide by zero. Decomposition fails.')
end
L(i,k)=A(i,k)/A(k,k);
A(i,k:N)=A(i,k:N)-L(i,k)*A(k,k:N);
end
end

for i=1:M
U(i,i:N)=A(i,i:N);
end

```

Example 7.5.1: In Example 1.7.1 we gave the LU decomposition for the matrix (1.7.4), repeated,

$$A = \begin{bmatrix} 2 & 4 & 2 \\ 1 & 5 & 2 \\ 4 & -1 & 9 \end{bmatrix} \quad (7.5.3)$$

The MATLAB script

```
>> A=[2,4,2;1,5,2;4,-1,9]
```

```
A =
```

```

2      4      2
1      5      2
4     -1      9

```

```
>> [L,U]=elemlu(A)
```

yields the output

```
L =
```

```

1.0000      0      0
0.5000    1.0000      0
2.0000   -3.0000    1.0000

```

```
U =
```

```

2      4      2
0      3      1
0      0      8

```

which is the result (1.7.5)

Example 7.5.2: In Example 1.7.2 we explained why the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 1 & -1 \end{bmatrix} \quad (7.5.4)$$

does not have a *LU* decomposition. This fact is reflected in the MATLAB calculation

```
>> A=[1,0,0;0,0,2;0,1,-1]
```

```
A =
```

1	0	0
0	0	2
0	1	-1

```
>> [L,U]=elemlu(A)
```

which produces the output

```
A =
```

1	0	0
0	0	2
0	1	-1

```
??? Error using ==> elemlu at 15
Cannot divide by zero. Decomposition fails.
```

The zero in the 22 slot of the matrix displayed is the source of the error.

Example 7.5.3: In Example 1.7.4, we constructed the *LU* decomposition of the matrix (1.7.39), repeated,

$$A = \begin{bmatrix} 1 & 2 & -4 & 3 & 9 \\ 4 & 5 & -10 & 6 & 18 \\ 7 & 8 & -16 & 0 & 0 \end{bmatrix} \quad (7.5.5)$$

If this matrix is entered into MATLAB and the command `[L,U]=elemlu(A)` is issued, the resulting output is

```
>> [L,U]=elemlu(A)
```

```
L =
```

1	0	0
---	---	---

$$\begin{matrix} 4 & 1 & 0 \\ 7 & 2 & 1 \end{matrix}$$

U =

$$\begin{matrix} 1 & 2 & -4 & 3 & 9 \\ 0 & -3 & 6 & -6 & -18 \\ 0 & 0 & 0 & -9 & -27 \end{matrix}$$

which is the result obtained in Section 1.7.

As explained, the generalized *LU* decomposition allows partial pivoting and results in the representation (7.5.2). While the elementary *LU* is unique when it exists, in Example 1.7.6 we illustrated the fact that the generalized *LU* decomposition is not necessarily unique. MATLAB constructs the decomposition (7.5.2) with the built in function **lu**. The syntax of this command is

$$[L, U, P] = \text{lu}(A) \quad (7.5.6)$$

It is instructive to illustrate this decomposition with our Examples 7.5.1 through 7.5.3 above. The results are summarized as follows:

For Example 7.5.1:

$$A = \begin{bmatrix} 2 & 4 & 2 \\ 1 & 5 & 2 \\ 4 & -1 & 9 \end{bmatrix} \quad (7.5.7)$$

the MATLAB command (7.5.6) yields

$$A = \begin{bmatrix} 2 & 4 & 2 \\ 1 & 5 & 2 \\ 4 & -1 & 9 \end{bmatrix} = P^T LU = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0.2500 & 1 & 0 \\ 0.5000 & 0.8571 & 1 \end{bmatrix} \begin{bmatrix} 4 & -1 & 9 \\ 0 & 5.2500 & -0.2500 \\ 0 & 0 & -2.2857 \end{bmatrix} \quad (7.5.8)$$

where we have used the property of permutations mentioned in Section 1.7 that $P^T P = PP^T = I$. The result (7.5.8) replaces the result of Example 7.5.1, namely,

$$A = \begin{bmatrix} 2 & 4 & 2 \\ 1 & 5 & 2 \\ 4 & -1 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ 2 & -3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 8 \end{bmatrix} \quad (7.5.9)$$

For Example 7.5.2:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 1 & -1 \end{bmatrix} \quad (7.5.10)$$

the MATLAB command (7.5.6) yields

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 1 & -1 \end{bmatrix} = P^T LU = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 2 \end{bmatrix} \quad (7.5.11)$$

which was obtained in Example 1.7.5. This example was one where the *LU* decomposition failed because a division by zero was encountered during the Gaussian elimination process. The partial

pivoting as implemented by $P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ avoided the problem.

For Example 7.5.3:

$$A = \begin{bmatrix} 1 & 2 & -4 & 3 & 9 \\ 4 & 5 & -10 & 6 & 18 \\ 7 & 8 & -16 & 0 & 0 \end{bmatrix} \quad (7.5.12)$$

the MATLAB command (7.5.6) yields

$$A = \begin{bmatrix} 1 & 2 & -4 & 3 & 9 \\ 4 & 5 & -10 & 6 & 18 \\ 7 & 8 & -16 & 0 & 0 \end{bmatrix} = P^T LU = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0.1429 & 1 & 0 \\ 0.5714 & 0.5000 & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & -16 & 0 & 0 \\ 0 & 0.8571 & -1.7143 & 3 & 9 \\ 0 & 0 & 0 & 4.5000 & 13.5000 \end{bmatrix} \quad (7.5.13)$$

rather than the results of Example 7.5.3, namely,

$$A = \begin{bmatrix} 1 & 2 & -4 & 3 & 9 \\ 4 & 5 & -10 & 6 & 18 \\ 7 & 8 & -16 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -4 & 3 & 9 \\ 0 & -3 & 6 & -6 & -18 \\ 0 & 0 & 0 & -9 & -27 \end{bmatrix} \quad (7.5.14)$$

Even though an elementary *LU* decomposition exists in this case, MATLAB reordered the rows of A such that the first column was sorted largest to smallest value before the elimination process was implemented. This convention as mentioned in Section 1.3. This type of reordering is

implemented in order to minimize round off errors. This concern does not show up in the elementary problems we are utilizing in our examples.

Example 7.5.4: This example is based upon the same one used in Example 1.7.6, namely the one defined by equation (1.7.70), repeated,

$$A = \begin{bmatrix} 0 & 5 & 5 \\ 2 & 9 & 0 \\ 6 & 8 & 8 \end{bmatrix} \quad (7.5.15)$$

the MATLAB command (7.5.6) yields

$$\begin{aligned} A &= \begin{bmatrix} 0 & 5 & 5 \\ 2 & 9 & 0 \\ 6 & 8 & 8 \end{bmatrix} = P^T LU \\ &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0.3333 & 1 & 0 \\ 0 & 0.7895 & 1 \end{bmatrix} \begin{bmatrix} 6 & 8 & 6 \\ 0 & 6.3333 & -2.667 \\ 0 & 0 & 7.1053 \end{bmatrix} \end{aligned} \quad (7.5.16)$$

which is the result obtained in equation (1.7.77) of Example 1.7.6. In Exercise 1.7.4, we indicated that

$$A = \begin{bmatrix} 0 & 5 & 5 \\ 2 & 9 & 0 \\ 6 & 8 & 8 \end{bmatrix} = P^T LU = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & -\frac{19}{5} & 1 \end{bmatrix} \begin{bmatrix} 2 & 9 & 0 \\ 0 & 5 & 5 \\ 0 & 0 & 27 \end{bmatrix} \quad (7.5.17)$$

is also a generalized *LU* decomposition of the same matrix. The MATLAB **lu** command chose the permutation that ordered the first column as described above.

The two implementations we have discussed have applied to the case where the $M \times N$ matrix A has the property $N \geq M - 1$. The MATLAB **lu** command does not have this restriction. For example, if we adopt an example that is a special case of equation (1.7.54), namely,

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix} \quad (7.5.18)$$

the MATLAB **lu** command applied to this matrix yields

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0.6667 & 0.3333 \\ 0.3333 & 0.6667 \end{bmatrix} \begin{bmatrix} 6 & 7 \\ 0 & 1 \end{bmatrix} \quad (7.5.19)$$

This result shows that in such cases, MATLAB relaxes the requirement that L be square. This possibility was mentioned at the end of our discussion of the elementary LU decomposition in Section 1.7.

In Section 1.7 we briefly discussed an advantage of utilizing the LU decomposition to solve systems of linear equations

$$Ax = b \quad (7.5.20)$$

In summary, we pointed out that if we have the decomposition, (7.5.1), then the system of linear equations $Ax = b$ can be written

$$LUx = b \quad (7.5.21)$$

Because L , is nonsingular, we can multiply on the left by L^{-1} and obtain

$$L^{-1}LUx = Ix = Ux = L^{-1}b \quad (7.5.22)$$

Thus, our problem is reduced to solving

$$Ux = L^{-1}b \quad (7.5.23)$$

Because U is an upper triangular matrix, (7.5.23) can be solved, for example, by back substitution or Gauss-Jordan elimination. The implementation of this part of the calculation is more computationally efficient when one works directly with A .

Exercises

7.5.1: Use MATLAB to perform the generalized LU decomposition for the matrix.

$$A = \begin{bmatrix} -23 & 26 & -42 & -32 & -90 \\ -2 & 1 & 0 & -3 & -4 \\ -17 & 19 & -28 & -22 & -63 \\ -13 & 14 & -24 & -16 & -52 \\ 18 & -20 & 32 & 23 & 69 \end{bmatrix} \quad (7.5.24)$$

The result that should be obtained is

$$A = \begin{bmatrix} -23 & 26 & -42 & -32 & -90 \\ -2 & 1 & 0 & -3 & -4 \\ -17 & 19 & -28 & -22 & -63 \\ -13 & 14 & -24 & -16 & -52 \\ 18 & -20 & 32 & 23 & 69 \end{bmatrix} = \begin{bmatrix} 1.000 & 0 & 0 & 0 & 0 \\ 0.0870 & 1.000 & 0 & 0 & 0 \\ 0.7391 & 0.1724 & 1.000 & 0 & 0 \\ 0.5652 & 0.5517 & -0.9429 & 1.000 & 0 \\ -0.7826 & -0.2759 & 0.0571 & -0.5789 & 1.000 \end{bmatrix} \times \begin{bmatrix} -23.000 & 26.000 & -42.000 & -32.000 & -90.000 \\ 0 & -1.2609 & 3.6522 & -0.2174 & 3.8261 \\ 0 & 0 & 2.4138 & 1.6897 & 2.8621 \\ 0 & 0 & 0 & 3.8000 & -0.5429 \\ 0 & 0 & 0 & 0 & -0.8571 \end{bmatrix} \quad (7.5.25)$$

Section 7.6. Additional Discussion of Eigenvalue Problems

In Chapter 5 and 6 we discussed eigenvalue problems. In this section, we shall discuss how MATLAB can be used to solve these problems. Recall that an eigenvalue problem is one where you are given a vector space \mathcal{V} and a linear transformation $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{V}$. The eigenvalue problem is the problem of finding a *nonzero* vector $\mathbf{v} \in \mathcal{V}$ that obeys

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \quad (7.6.1)$$

As explained in Chapter 5, the nonzero vector \mathbf{v} which satisfies (7.6.1) is called an *eigenvector* of \mathbf{A} . The scalar λ is called the *eigenvalue* of \mathbf{A} . As in Chapters 5 and 6, we are interested in finite dimensional vector spaces and shall use the notation $N = \dim \mathcal{V}$.

Given a basis $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_N\}$ for \mathcal{V} , the component version of (7.6.1) is the system of equations

$$\sum_{k=1}^N A^j_k v^k = \lambda v^j \quad \text{for } j = 1, 2, \dots, N \quad (7.6.2)$$

or, equivalently, the matrix equation

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \quad (7.6.3)$$

where

$$\mathbf{A} = M(\mathbf{A}, \mathbf{e}_j, \mathbf{e}_k) = \begin{bmatrix} A^1_1 & A^1_2 & \cdot & \cdot & \cdot & A^1_N \\ A^2_1 & A^2_2 & & & & A^2_N \\ A^3_1 & & A^3_3 & & & \cdot \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ A^N_1 & A^N_2 & \cdot & \cdot & \cdot & A^N_N \end{bmatrix} \quad (7.6.4)$$

and

$$\mathbf{v} = \begin{bmatrix} v^1 \\ v^2 \\ v^3 \\ \cdot \\ \cdot \\ v^N \end{bmatrix} \quad (7.6.5)$$

We showed in Section 5.2 that the eigenvalues of $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{V}$ are the roots of the *characteristic polynomial* defined by equation (5.2.6), repeated,

$$f(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I}) = \underbrace{(\lambda_1 - \lambda)(\lambda_2 - \lambda)(\lambda_3 - \lambda) \cdots (\lambda_N - \lambda)}_{N \text{ Factors}} \quad (7.6.6)$$

Equivalently, the characteristic polynomial can be written in terms of the matrix (7.6.4) by the formula

$$f(\lambda) = \det(A - \lambda I) = \underbrace{(\lambda_1 - \lambda)(\lambda_2 - \lambda)(\lambda_3 - \lambda) \cdots (\lambda_N - \lambda)}_{N \text{ Factors}} \quad (7.6.7)$$

Given these roots, which need not be distinct or real, one then utilizes (7.6.3) to determine the eigenvectors. The N set of equations (7.6.3) can be written as the single matrix equation

$$AT = TD \quad (7.6.8)$$

where

$$D = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdot & \cdot & 0 \\ 0 & \lambda_2 & 0 & \cdot & \cdot & 0 \\ 0 & 0 & \lambda_3 & 0 & \cdot & 0 \\ \cdot & \cdot & 0 & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \lambda_N \end{bmatrix} \quad (7.6.9)$$

and

$$T = \begin{bmatrix} v_{(1)}^1 & v_{(2)}^1 & v_{(3)}^1 & \cdot & \cdot & v_{(N)}^1 \\ v_{(1)}^2 & v_{(2)}^2 & v_{(3)}^2 & \cdot & \cdot & v_{(N)}^2 \\ v_{(1)}^3 & v_{(2)}^3 & v_{(3)}^3 & \cdot & \cdot & v_{(N)}^3 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ v_{(1)}^N & v_{(2)}^N & v_{(3)}^N & \cdot & \cdot & v_{(N)}^N \end{bmatrix} \quad (7.6.10)$$

where each column of (7.6.10) consists of the components of an eigenvector with respect to the basis $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_N\}$. In the case where the matrix (7.6.10) has rank N , T is the transition matrix for the basis transformation from $\{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_N\}$ to a basis of eigenvectors $\{v_{(1)}, v_{(2)}, \dots, v_{(N)}\}$.

MATLAB implements the solution of (7.6.8) with the syntax

$$[T, D] = \text{eig}(A) \quad (7.6.11)$$

The function **eig** accepts a matrix where the elements are numbers entered as symbols and, of course, it accepts a matrix where the elements are double precision numbers. It has certain normalization conventions that we shall illustrate in the examples below.

Example 7.6.1: Consider the case discussed in Example 5.3.1. In this case the matrix A is given by

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 1 \\ 4 & -4 & 5 \end{bmatrix} \quad (7.6.12)$$

The results we obtained in Section 5.3 for this problem are given by (5.3.21) and (5.3.22), respectively,

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix} \quad (7.6.13)$$

and

$$T = \begin{bmatrix} -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{4} \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ 1 & 1 & 1 \end{bmatrix} \quad (7.6.14)$$

The MATLAB script

```
clc
clear
A=sym([1,2,-1;1,0,1;4,-4,5])
[T,D]=eig(A)
```

yields the output

```
T =
```

```
[ -1/4, -1/2, -1/2]
[ 1/4, 1/2, 1/4]
[ 1, 1, 1]
```

(7.6.15)

```
D =
```

```
[ 3, 0, 0]
[ 0, 1, 0]
[ 0, 0, 2]
```

The first point to note is that MATLAB does not order the eigenvalues the same as with Example 5.3.1. Because of this, the columns of **T** are also not ordered the same. In addition, because the length of eigenvectors are not determined by the defining equation (7.6.3), the corresponding columns in the two solutions, equation (7.6.14) and (7.6.15)₁ can differ by a constant. In the case at hand, the corresponding columns are the same because when working example 5.3.1, we chose the normalization used by MATLAB. Finally, the script utilized above, defined the matrix (7.6.12) as a symbol. If we had, instead, utilized the script

```
clc
clear
A=[1,2,-1;1,0,1;4,-4,5]
[T,D]=eig(A)
```

The resulting MATLAB output is

```
T =
```

```
-0.2357 0.4364 0.4082
0.2357 -0.2182 -0.4082
0.9428 -0.8729 -0.8165
```

(7.6.16)

```
D =
```

```
3.0000 0 0
0 2.0000 0
0 0 1.0000
```

The eigenvalues, when the matrix elements are double precision numbers, are sorted in decreasing order. Also, the eigenvectors are normalized to have unit length.

Example 7.6.2: In this example, we shall utilize MATLAB to rework Example 5.3.2. In this case, the matrix A is given by

$$A = \begin{bmatrix} 1 & -2 & 2 \\ -2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix} \quad (7.6.17)$$

If this matrix is entered into MATLAB as a symbolic matrix, the results from script like that above are

```
T =
[ -1, -1, 1]
[ -1,  1, 0]
[  1,  0, 1]
```

(7.6.18)

```
D =
[ -3, 0, 0]
[  0, 3, 0]
[  0, 0, 3]
```

This example has two not three distinct eigenvalues. The first, $\lambda = -3$, has multiplicity one and its characteristic subspace has dimension one. The next eigenvalue, $\lambda = 3$, has multiplicity two and its characteristic subspace has dimension two. These features are illustrated by equations (5.3.33) and (5.3.38), respectively. As we did with Example 5.3.1, we can make arbitrary choices of the basis for the two dimensional characteristic subspace defined by (5.3.38). The results, while not unique, provide a basis of eigenvectors. The result (7.6.18)₁ reflects MATLAB's choice of these two vectors. Its algorithm for handling repeated eigenvalues is built into the function **eig**. The arbitrary choices made in Example 5.3.2 were made to replicate the result (7.6.18)₁. As explained in Section 5.3, problems for which the algebraic multiplicity and the geometric multiplicity are the same have the property that there is a basis consisting of eigenvectors. Example 5.3.2 is such an example. Example 5.3.3 is one where the algebraic multiplicity is not the same as the geometric multiplicity. If Example 5.3.3 is worked with MATLAB, one will find that it yields only a single eigenvector. This result reflects that this problem does not have a basis of eigenvectors.

There are other MATLAB functions that are useful when one is solving eigenvalue problems. A few of them are as follows:

poly

For an $N \times N$ matrix **A** of double precision values, **poly(A)** yields a $N + 1$ dimensional row vector whose elements are the coefficients of the characteristic polynomial written in the form $\det(\lambda I - A)$. The roots of the characteristic polynomial can be calculated from **roots(poly(A))**.

charpoly

For an $N \times N$ matrix **A**, **charpoly(A)** yields a $N + 1$ dimensional row vector whose elements are the coefficients of the *characteristic polynomial* written in the form $\det(\lambda I - A)$. If **A** is a symbolic matrix, **charpoly(A)** returns a symbolic vector. If **A** is a matrix of double precision values, then **charpoly(A)** returns double precision values. The syntax **charpoly(A,x)** returns the actual polynomial in the variable *x*.

minpoly

For an $N \times N$ matrix **A**, **minpoly(A)** yields a row vector whose elements are the coefficients of the *minimum polynomial* of **A**. If **A** is a symbolic matrix, **minpoly(A)** returns a symbolic vector. If **A** is a matrix of double precision values, then **minpoly(A)** returns double precision values.

svd

For an $M \times N$ matrix **A**, $[K, D, Q] = \text{svd}(A)$ yields the *singular value decomposition* of **A**. This decomposition is discussed in Section 6.8. The matrix **K** is an $M \times M$ unitary matrix, the matrix **D** is an $M \times K$ diagonal matrix and **Q** is an $N \times N$ unitary matrix. As shown by equation (6.8.56), these four matrices are related by the formula **A = K * D * Q'**.

expm

For an $N \times N$ matrix **A**, **expm(A)** yields the *exponential matrix* of **A**. The calculation of the exponential linear transformation was discussed in Section 6.4.

The above summary can be greatly expanded by utilizing MATLAB's help utility. In particular, each function has options not discussed above that can be useful. Also, whether or not these functions work for both symbolic matrices and double precision matrix is not fully discussed in the above. MATLAB help will explain when this feature is included as a property of the function

Exercises

7.6.1: Utilize MATLAB and work Example 5.3.5.

7.6.2: Utilize MATLAB and work Example 6.8.2.

Chapter 8

ERRORS THAT ARISE IN NUMERICAL ANALYSIS

In this Chapter, we shall look briefly at the kinds of numerical errors that arise in Numerical Analysis. Our principal discussion will be the category of errors that arise as a result of the way a computer represents a real number.

Numerical analysis techniques arise in a multitude of applications. One such application is when one utilizes a theoretical mathematical formulation to model a physical phenomenon. It is inevitable that such models are approximations of nature. One might approximate a nonlinear stress strain relationship by a linear one. This kind of approximation arises prior to any effort to solve the controlling governing equations. The process of solving the approximating governing equations will also often involve the utilization of an approximate procedure which, itself, will be an approximation of the exact solution. If, for example, the result of this solution is an analytical solution, additional approximations are introduced when one utilizes a computer to numerically evaluate the solution. Thus, as this simply discussion illustrates, the process of solving a physical problem can introduce approximations at every stage of the solution procedure.

As one might anticipate, an in-depth discussion of errors and their propagation is complication. Our goals are more modest. As indicated, our principal goal is to understand how MATLAB generates real numbers and to identify approximations associated with this generation.

We begin this Chapter with a discussion of Taylor's Theorem.¹ This theorem is fundamental to discussions of approximations in numerical analysis and is the origin of an important type of error. After this discussion, we shall discuss the *IEEE 64 bit floating-point number system*.

Section 8.1. Taylor's Theorem

We shall begin this discussion with a mention of the *Mean Value Theorem*. This theorem is stated and proven in every elementary Calculus course. In order to state the theorem, we need to be reminded of some standard notation used in mathematics and, as a result, in this work. The notation \mathbb{R} denotes the set of real numbers. The notation $[a,b]$ denotes a subset of \mathbb{R} defined by

¹ Information about the English mathematician Brook Taylor can be found at http://en.wikipedia.org/wiki/Brook_Taylor.

$$[a, b] = \underbrace{\{x \mid x \in \mathbb{R} \text{ and } a \leq x \leq b\}}_{\substack{\text{Reads: Set of real numbers} \\ \text{that obey } a \leq x \leq b}} \quad (8.1.1)$$

Likewise (a, b) is the set of real numbers defined by

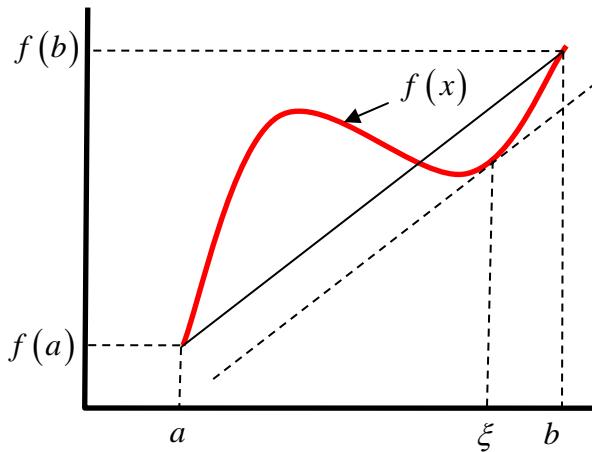
$$(a, b) = \underbrace{\{x \mid x \in \mathbb{R} \text{ and } a < x < b\}}_{\substack{\text{Reads: Set of real numbers} \\ \text{that obey } a < x < b}} \quad (8.1.2)$$

Given this notation, the *Mean Value Theorem* is as follows:

Theorem 8.1.1: Given a continuous function $f : [a, b] \rightarrow \mathbb{R}$ that is differentiable on (a, b) , then, there exists at least one point ξ between a and b such that

$$f'(\xi) = \frac{f(b) - f(a)}{b - a} \quad (8.1.3)$$

The following figure illustrates this theoretical result.



The Mean Value Theorem is a special case of *Taylor's Theorem*. Hopefully, the reader will recall this theorem from an earlier Calculus course. This theorem is a fundamental mathematical result which is used to express functions in terms of linear and higher order approximations. The formal statement is as follows:

Theorem 8.1.2: Given a continuous function $f : [a, b] \rightarrow \mathbb{R}$ that is differentiable of order $K + 1$ on (a, b) and a point $c \in (a, b)$, then at any point $x \in (a, b)$

$$f(x) = \sum_{k=0}^K \frac{f^k(c)}{k!} (x-c)^k + R_{K+1}(x) \quad (8.1.4)$$

where the *remainder* R_{K+1} is given by

$$R_{K+1}(x) = \frac{f^{(K+1)}(\xi)}{(K+1)!} (x-c)^{K+1} \quad \text{for some } \xi \text{ between } c \text{ and } x \quad (8.1.5)$$

and

$$f^k(c) = \left. \frac{d^k f(x)}{dx^k} \right|_{x=c} \quad (8.1.6)$$

Equation (8.1.4) is called the *Taylor's series* of the function $f(x)$ about the point $x = c$. If the function has derivatives of all orders, then it is customary to write (8.1.4) as

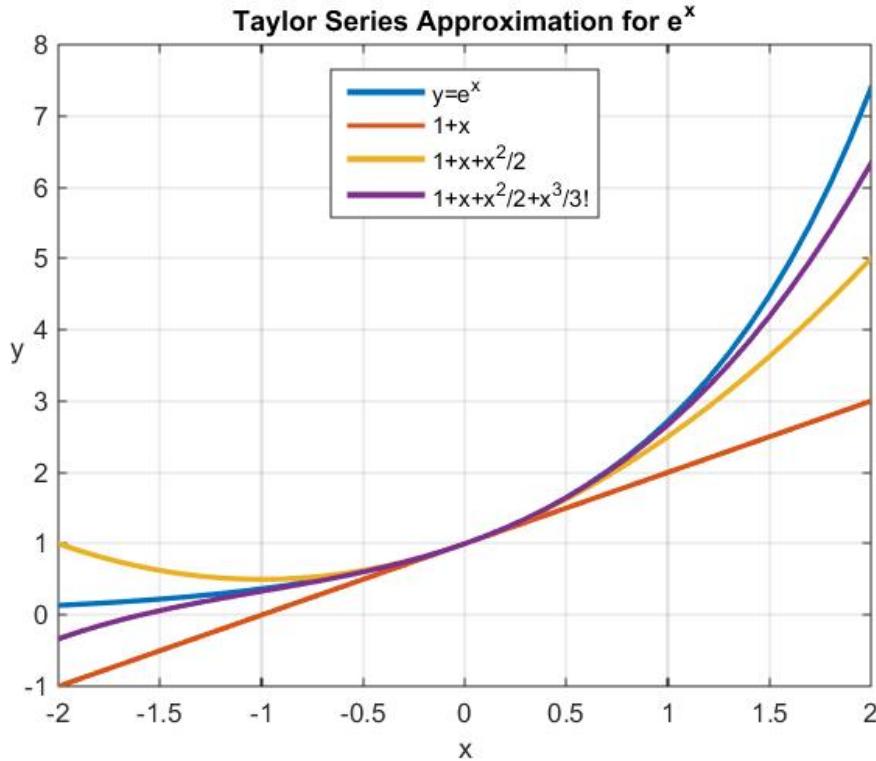
$$f(x) = \lim_{K \rightarrow \infty} \sum_{k=0}^K \frac{f^k(c)}{k!} (x-c)^k \equiv \sum_{k=0}^{\infty} \frac{f^k(c)}{k!} (x-c)^k \quad (8.1.7)$$

The following are example Taylor series approximations about the point $x = 0$. These formulas can be found in any Calculus book or online at a large number of sites. You will probably recall that such series are also called *Maclaurin series* because they illustrate (8.1.7) in the case $c = 0$.²

² Information about the Scottish mathematician Colin Maclaurin can be found at http://en.wikipedia.org/wiki/Colin_Maclaurin.

$$\begin{aligned}
 e^x &= \sum_{n=0}^{\infty} \frac{1}{n!} x^n = 1 + x + \frac{1}{2} x^2 + \dots \\
 \cos x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2} + \frac{x^4}{4!} + \dots \\
 \sin x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots \\
 \cosh x &= \sum_{n=0}^{\infty} \frac{1}{(2n)!} x^{2n} = 1 + \frac{x^2}{2} + \frac{x^4}{4!} + \dots \\
 \sinh x &= \sum_{n=0}^{\infty} \frac{1}{(2n+1)!} x^{2n+1} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots \\
 \ln(1-x) &= -\sum_{n=1}^{\infty} \frac{1}{n} x^n \quad \text{for } |x| < 1 \\
 \arcsin x &= \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!) (2n+1)} x^{2n+1} \quad \text{for } |x| < 1 \\
 \arctan x &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)} x^{2n+1} \quad \text{for } |x| < 1
 \end{aligned} \tag{8.1.8}$$

It is instructive to use MATLAB to plot the function e^x and the approximations obtained by retaining the first term, the first two terms and the first three terms. The result of this approximation is



Among other things, this figure displays the error associated with adopting approximations to the exact function. This kind of error is known as a *truncation error*.

The MATLAB script that produces the above figure is

```

clc
clear
x=[-2:.1:2]
y=exp(x)
plot(x,y,'LineWidth',2)
grid on
xlabel('x')
ylabel('y','Rotation',0)
title('Taylor Series Approximation for e^x')

y1=1+x
y2=1+x+x.^2/2
y3=1+x+x.^2/2+x.^3/factorial(3)
hold on
plot(x,y1,'LineWidth',2)
plot(x,y2,'LineWidth',2)
plot(x,y3,'LineWidth',2)
legend('y=e^x','1+x','1+x+x^2/2','1+x+x^2/2+x^3/3!',...
'Location','North')

```

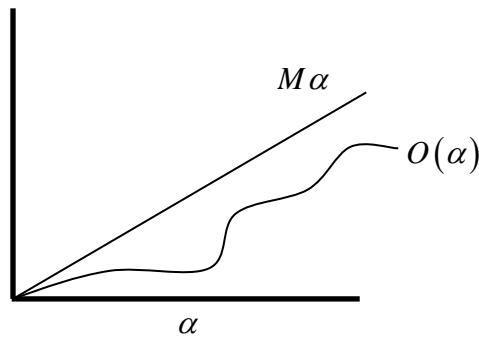
The remainder as defined by (8.1.5) is sometimes written in the short hand notation

$$R_{K+1} = O\left((x - c)^{K+1}\right) \quad (8.1.9)$$

and reads “of the order $K + 1$ ”. To qualify for this notation, it must obey

$$O\left((x - c)^{K+1}\right) \leq M |x - c|^{K+1} \quad (8.1.10)$$

for some real number M . The notation is intended to capture the idea that the quantity in question, in this case R_{K+1} goes to zero as $|x - c|^{K+1}$ goes to zero. The following figure might help understand the idea.



The definition (8.1.9) allows us to write the Taylor's series (8.1.4) in the more convenient form

$$f(x) = \sum_{k=0}^K \frac{f^k(c)}{k!} (x - c)^k + O\left((x - c)^{K+1}\right) \quad (8.1.11)$$

With this notation, Taylor's Theorem can be written in the equivalent forms

$$\begin{aligned}
f(x) &= f(c) + O(x - c) \\
f(x) &= f(c) + f'(c)(x - c) + O((x - c)^2) \\
f(x) &= f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 + O((x - c)^3) \\
&\quad \cdot \\
&\quad \cdot \\
f(x) &= f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 \\
&\quad + \frac{f'''(c)}{3!}(x - c)^3 + \cdots + \frac{f^{(K)}(c)}{K!}(x - c)^K + O((x - c)^{K+1})
\end{aligned} \tag{8.1.12}$$

If we write the equation

$$\begin{aligned}
f(x) &= f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 \\
&\quad + \frac{f'''(c)}{3!}(x - a)^3 + \cdots + \frac{f^{(K)}(a)}{K!}(x - a)^K + R_{K+1}
\end{aligned} \tag{8.1.13}$$

as

$$R_{K+1} = f(x) - \left(f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 \right. \\
\left. + \frac{f'''(c)}{3!}(x - c)^3 + \cdots + \frac{f^{(K)}(c)}{K!}(x - c)^K \right) \tag{8.1.14}$$

then R_{K+1} is a measure of the *truncation error* which results if the exact value, $f(x)$, is approximated by

$$\begin{aligned}
&f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 \\
&+ \frac{f'''(c)}{3!}(x - c)^3 + \cdots + \frac{f^{(K)}(c)}{K!}(x - c)^K
\end{aligned} \tag{8.1.15}$$

Section 8.2. Round Off and Truncation Errors

In simple terms a *round off error* arises when an irrational number, such as π , is represented by a finite number of digits. It is intrinsic to computers that they represent all numbers by a finite number of digits. A different kind of error is a *truncation error*. As indicated in Section 8.1, this kind of error arises when a function is represented by a convergent infinite series and, as an approximation; the series is truncated at a finite number of terms.

Example 8.2.1: Given a Taylor's series (Maclaurin series expansion) expansion of the hyperbolic sine function equation (8.1.8)₅, repeated,

$$\sinh x = \sum_{n=0}^{\infty} \frac{1}{(2n+1)!} x^{2n+1} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots \quad (8.2.1)$$

In this example, we shall establish an iterative approach to trying to determine the value of $\sinh x$ at $x = 0.6$. The expression we wish to evaluate is

$$\sinh 0.6 = \sum_{n=0}^{\infty} \frac{1}{(2n+1)!} (0.6)^{2n+1} \quad (8.2.2)$$

By retaining more and more terms of the expansion, we get closer to the correct value of $\sinh 0.6$. One way to measure the error associated with the choice of a finite number of terms is to define

$$y_M = \sum_{n=0}^M \frac{1}{(2n+1)!} (0.6)^{2n+1} \quad (8.2.3)$$

and to measure the error by how close are terms in the sequence (y_0, y_1, y_2, \dots) . In particular, we *define* the error to be the percentage

$$\varepsilon_{(M)} = \left| \frac{y_M - y_{M-1}}{y_{M-1}} \right| \times 100 \quad (8.2.4)$$

for $M = 1, 2, \dots$. The iteration we shall construct can be displayed by completing the table:

M	$\sinh 0.6 \approx y_M = \sum_{n=0}^M \frac{1}{(2n+1)!} (0.6)^{2n+1}$	$\varepsilon_{(M)}$
0		N/A
1		
2		
3		
4		
5		

The idea is to continue the iteration until $\varepsilon_{(M)}$ is smaller than a preassigned value. In this example, we shall continue the iteration until

$$\varepsilon_{(M)} < .05\% \quad (8.2.5)$$

Iteration 0: (For $M = 0$),

$$\sinh 0.6 \approx 0.6 \quad (8.2.6)$$

This result is obtained by writing the expansion for $\sinh 0.6$ as

$$\sinh 0.6 \approx y_0 = 0.6 \quad (8.2.7)$$

and simply dropping the last set of terms. The terms dropped represent the *truncation error* for this approximation. This error is displayed by writing (8.2.2) as

$$\sinh 0.6 = \underbrace{y_0}_{0.6} + \underbrace{\sum_{n=1}^{\infty} \frac{1}{(2n+1)!} (0.6)^{2n+1}}_{\text{Truncation Error}} \quad (8.2.8)$$

Iteration 1: (For $M = 1$)

$$\begin{aligned}\sinh 0.6 &= \underbrace{0.6 + \frac{1}{6}(0.6)^3}_{y_1} + \underbrace{\sum_{n=2}^{\infty} \frac{1}{(2n+1)!}(0.6)^{2n+1}}_{\text{Truncation Error}} \\ &= \underbrace{0.636}_{y_1} + \underbrace{\sum_{n=2}^{\infty} \frac{1}{(2n+1)!}(0.6)^{2n+1}}_{\text{Truncation Error}}\end{aligned}\quad (8.2.9)$$

The error for this iteration is

$$\varepsilon_{(1)} = \left| \frac{y_1 - y_0}{y_0} \right| \times 100 = \left(\frac{0.036}{0.6} \right) 100 = 6\% \quad (8.2.10)$$

Iteration 2: (For $M = 2$)

$$\begin{aligned}\sinh 0.6 &= \underbrace{0.6 + \frac{1}{6}(0.6)^3 + \frac{1}{120}(0.6)^5}_{y_2} + \underbrace{\sum_{n=3}^{\infty} \frac{1}{(2n+1)!}(0.6)^{2n+1}}_{\text{Truncation Error}} \\ &= \underbrace{0.636648}_{y_2} + \underbrace{\sum_{n=3}^{\infty} \frac{1}{(2n+1)!}(0.6)^{2n+1}}_{\text{Truncation Error}}\end{aligned}\quad (8.2.11)$$

The error for this iteration is

$$\varepsilon_{(2)} = \left| \frac{y_2 - y_1}{y_1} \right| \times 100 = \left(\frac{0.0648}{0.636} \right) 100 = .1019\% \quad (8.2.12)$$

Iteration 3: (For $M = 3$)

$$\begin{aligned}\sinh 0.6 &= \underbrace{0.6 + \frac{1}{6}(0.6)^3 + \frac{1}{120}(0.6)^5 + \frac{1}{5040}(0.6)^7}_{y_3} + \underbrace{\sum_{n=4}^{\infty} \frac{1}{(2n+1)!}(0.6)^{2n+1}}_{\text{Truncation Error}} \\ &= \underbrace{0.63665355}_{y_3} + \underbrace{\sum_{n=4}^{\infty} \frac{1}{(2n+1)!}(0.6)^{2n+1}}_{\text{Truncation Error}}\end{aligned}\quad (8.2.13)$$

The error for this iteration is

$$\varepsilon_{(3)} = \left| \frac{y_3 - y_2}{y_2} \right| \times 100 = \left(\frac{0.63665355 - 0.636648}{0.636648} \right) 100 = .00087\% \quad (8.2.14)$$

Thus, our error bound was reached in three iterations.

With these numbers, the table above becomes

M	$\sinh 0.6 \approx y_M = \sum_{n=0}^M \frac{1}{(2n+1)!} (0.6)^{2n+1}$	$\varepsilon_{(M)}$
0	0.6	N/A
1	0.636	6%
2	0.636648	.1019 %
3	0.63665355	.00087 %

The calculations of this example illustrated truncation errors generated by truncating the infinite series representation for $\sinh 0.6$. The numerical calculations in equations (8.2.11) through (8.2.14) involved *round off* errors. This kind of error is the difference between an approximation of a number used in a calculation and its exact value. For example, when we expressed, in equation (8.2.11), the exact value $0.6 + \frac{1}{6}(0.6)^3 + \frac{1}{120}(0.6)^5 + \frac{1}{5040}(0.6)^7$ by the number 0.63665355 a round off error was introduced.

Because we have MATLAB available, it is reasonable to carry out the above calculation utilizing its power. The m-file that carries out the essential elements of the above calculation is

```
%Example 8.2.1: Approximations for sinh(x) by Taylor Series
clc
clear
x=.6
n=5
%Display more digits
format long
%Calculate the first n+1 terms in the series
%Note: MATLAB begins its indexing at 1 not 0. Thus,
%the series representation of sinh(x) has been
%adjusted so that the summation begins at n=1.
for k=1:1:n+1
    f(k)=x^(2*k-1)/factorial(2*k-1);
    approx_sinh(k)=sum(f)
end
```

The numerical output from MATLAB is

```
approx_sinh =
```

```

Columns 1 through 2

0.600000000000000  0.636000000000000

Columns 3 through 4

0.636648000000000  0.636653554285714

Columns 5 through 6

0.636653582057143  0.636653582148031

```

An m-file that also calculates the relative error and presents the results in a table like that above is

```

%Example 8.2.1 with Table: Approximations for sinh(x) by
%Taylor %Series
clc
clear
x=.6;
n=5;
%Display more digits
format long
%Put a title line on a table. fprintf is discussed in
%MATLAB Help.
fprintf('\n\t\t\tn\tapprox_sinh(0.6)\terror(Percent) \n');
%Put a dividing line in table
fprintf('\t\t=====\\n');
%Calculate the first n+1 terms in the series
%Note: MATLAB begins its indexing at 1 not 0. Thus,
%the series representation of sinh(x) has been
%adjusted so that the summation begins at n=1.
for k=[1:1:n+1];
    if k==1
        f(k)=x;
        approx_sinh(k)=x;
        %Print the output for k-1=0
    fprintf(' \t\t%5.0f \t\t%5.9f \t\tN/A\\n',...
        k-1,approx_sinh(k))
    else
        f(k)=x^(2*k-1)/factorial(2*k-1);
        approx_sinh(k)=sum(f);
        error(k)=abs(((approx_sinh(k)-approx_sinh(k-1)) / ...
            approx_sinh(k-1))*100);
        %Print the output for k-1=1,2,3,4,5
    fprintf(' \t\t%5.0f \t\t%5.9f \t\t%5.4f\\n',...

```

```
k-1,approx_sinh(k),error(k))  
end  
end
```

There are probably more elegant ways to generate the table, but the above file will work. The resulting MATLAB output is the table ³

n	approx_sinh(0.6)	error(Percent)
0	0.600000000	N/A
1	0.636000000	6.0000
2	0.636648000	0.1019
3	0.636653554	0.0009
4	0.636653582	0.0000
5	0.636653582	0.0000

³ Good discussions of the MATLAB command `fprintf` can be found online. An example is <http://www.mathworks.com/help/matlab/ref/fprintf.html>.

Section 8.3 Computer Representation of Real Numbers, Round-Off Errors

One source of round off errors is the method used by digital computers to store and manipulate real numbers. In this section, we shall discuss certain aspects of how computers store numbers and how this method causes round off errors. One of the topics we shall discuss is the IEEE 64 bit floating-point number representation. It is this representation that is implemented in MATLAB.

As an introduction, it is instructive to recall the following:

- Computers store information utilizing a *binary* or *base 2* number system. Each location, or bit, represents either the number 1 or the number zero.
- The stored information is then converted to real numbers by some kind of prescribed rule.

Example 8.3.1: If we were to adopt a number system what uses two bits, then in the computer we have the following $2^2 = 4$ possibilities:

$[00]$
 $[01]$
 $[10]$
 $[11]$

While a number system with only four numbers is probably not very useful, we can convert these binary numbers to real numbers by, for example, by the rule

$$01 \quad \left. \begin{array}{c} \uparrow \\ \downarrow \end{array} \right\} \quad 0 \times \underbrace{2^1}_{\substack{\text{Slot 1} \\ \text{Value}}} + 1 \times \underbrace{2^0}_{\substack{\text{Slot 2} \\ \text{Value}}} = 0 \times 2 + 1 \times 1 = 1$$

Given this rule, our computer has the following four real numbers in its universe:

$$\begin{aligned}[00] &\rightarrow 0 \times 2^1 + 0 \times 2^0 = 0 \\ [01] &\rightarrow 0 \times 2^1 + 1 \times 2^0 = 0 \times 2 + 1 \times 1 = 1 \\ [10] &\rightarrow 1 \times 2^1 + 0 \times 2^0 = 1 \times 2 + 0 \times 1 = 2 \\ [11] &\rightarrow 1 \times 2^1 + 1 \times 2^0 = 1 \times 2 + 1 \times 1 = 3\end{aligned}$$

Different rules would produce a different set of four numbers. Real computers, as you would expect, utilize a lot more bits of information and they utilize more complicated rules to compute numbers. The simple example does illustrate that as long as computers use a finite number of bits to represent numbers, the universe of numbers they can generate is finite.

The real number system the computer system is attempting to model has an infinite number of elements. The set of real numbers have the known properties of a *field* discussed in Section 2.1. In particular,

- The product of any two real numbers is a real number.
- The sum of any two real numbers is a real number.

In addition, the set of real numbers have a well-defined definition of distance with the property that

- The distance between two real numbers can be made arbitrarily small.

The numbers generated by computer number systems do not have these characteristics. Thus, it is unavoidable that *computer number systems represent approximations to the real number system*. Our objective here is to gain an understanding of this approximation.

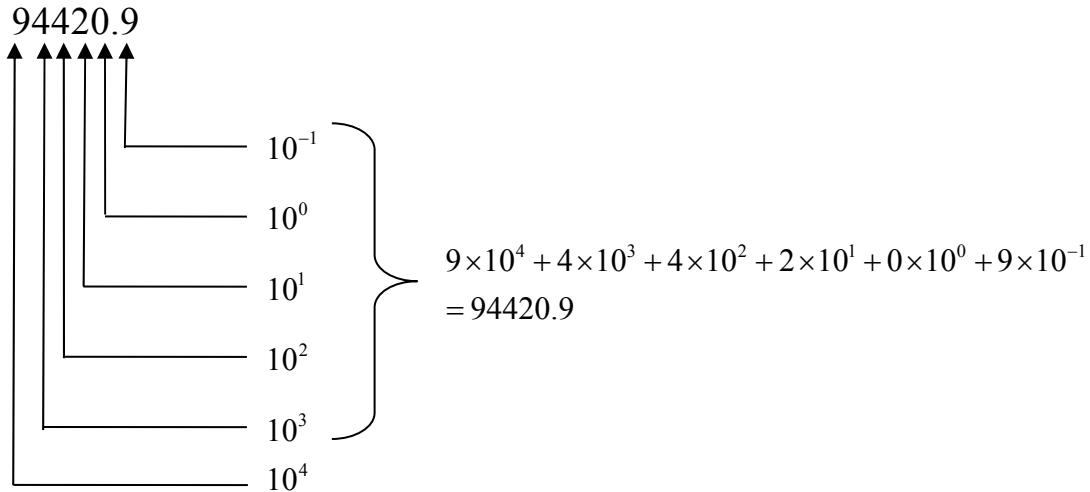
MATLAB and other computer programs utilize *floating point arithmetic*. As we shall see, this kind of arithmetic results in a huge subset of the real numbers that, in turn, represents a good approximation to the real numbers. None the less, the result is still an approximation. This fact leads to concepts such as

- *Round off error*
 - Arises when one represents a number with a fixed number of digits when the exact number does not have that representation. Examples are π , e , or, for example, $\sqrt{5}$. In addition, because computers use base 2 representations, they cannot precisely represent certain exact base 10 numbers. π and other irrational numbers by a finite number of digits.
 - There is more than one kind of round off error. One type might take $\frac{2}{3}$ and replace it by 0.6667. Another type might take $\frac{2}{3}$ and replace it by .6666. The second type of round off is sometimes called *chopping*.

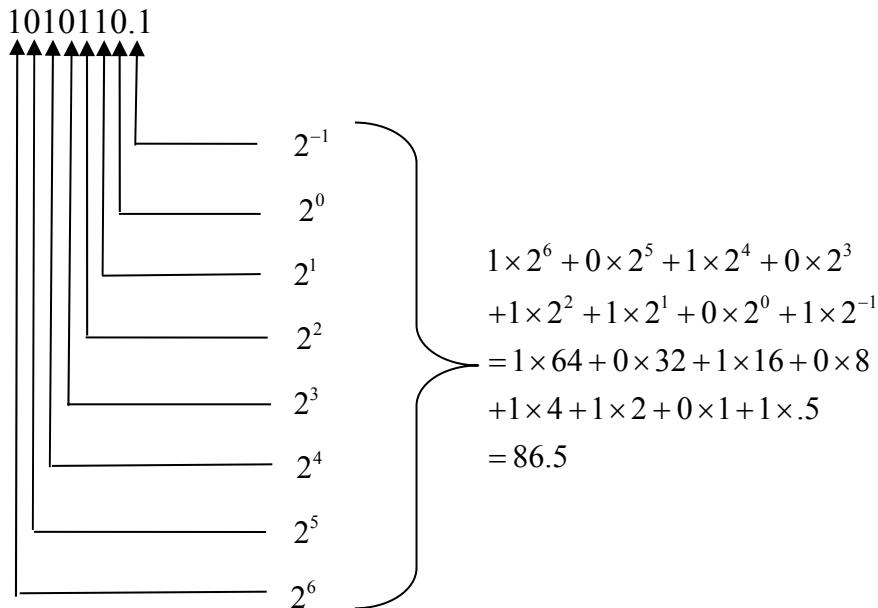
- *underflow and overflow*
 - Arises when trying to store in the computer a number too large or too small for its number system.

In order to introduce the floating point arithmetic utilized by computers it is helpful to briefly review number systems. A binary system, as indicated above, utilizes the two digits $\{0,1\}$. A base ten system uses the ten digits $\{0,1,2,3,4,5,6,7,8,9\}$. This set of ten digits builds positive numbers by a rule illustrated by the following example. The rule is characterized by a *positional notation*.

Example 8.3.2: In Base-10 the number 94,420.9 means



Example 8.3.3: In Base-2 the number 1010110.1 means

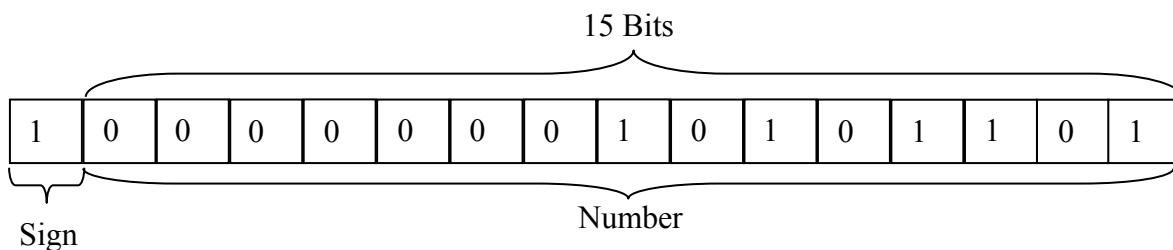


A scheme as illustrated in the last example will generate *positive numbers*. The modification which allows for negative as well as positive numbers is called the *signed magnitude method*. For this method, the first bit of a word is used to indicate the sign of the integer. The convention is

$0 \Rightarrow$ Positive Number

$1 \Rightarrow$ Negative Number

Example 8.3.4: For a computer with a *sixteen bit word*, the negative integer -173 is represented by

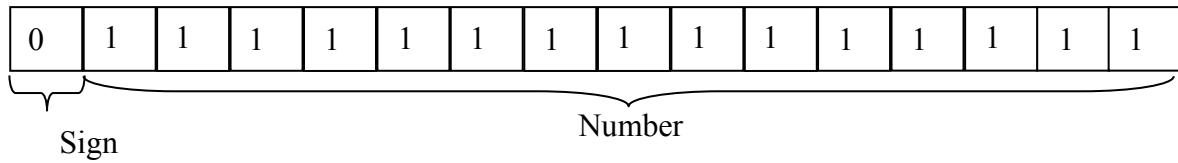


The conversion to base 10 is

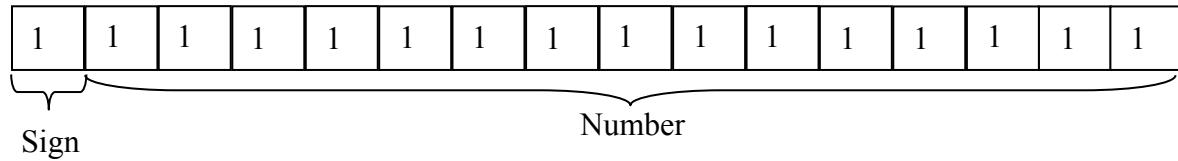
$$\begin{aligned}
 & -(0 \times 2^{14} + 0 \times 2^{13} + 0 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^9 + 0 \times 2^8 \\
 & + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \\
 & = -(128 + 32 + 8 + 4 + 1) = -173
 \end{aligned}$$

Example 8.3.5: Range of Integers. Determine the range of integers in a base 10 that can be represented on a 16-bit computer.

The first bit holds the sign and the remaining 15 bits have the values 1 or 0. The largest integer is the positive number



and the smallest integer is ⁴



In Base-10, the largest integer is

$$\begin{aligned}
 & 1 \times 2^{14} + 1 \times 2^{13} + 1 \times 2^{12} + 1 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 \\
 & + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
 & = 16384 + 8192 + 2048 + 1024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 \\
 & = 32767
 \end{aligned} \tag{8.3.1}$$

Likewise, the smallest integer is

⁴ A useful formula is $32767 = 2^{15} - 1$. This result is a special case of $2^n - 1 = 2^0 + 2^1 + 2^2 + \dots + 2^{n-1}$

$$\begin{aligned}
 & -(1 \times 2^{14} + 1 \times 2^{13} + 1 \times 2^{12} + 1 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 \\
 & + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) \\
 & = -(16384 + 8192 + 2048 + 1024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2) \\
 & = -32767
 \end{aligned} \tag{8.3.2}$$

With the exception of Examples 8.2.2 and 8.2.3, examples thus far have resulted in integer values when the binary representation is converted to a real number. The idea of floating point arithmetic arises when trying to relate a binary representation of a number to numbers that are not integers. A floating point number has the representation

$$\frac{m}{\text{Mantissa}} \times \frac{b}{\text{Base}}^e, \text{ where } e = \text{exponent} \tag{8.3.3}$$

The “Mantissa” is sometimes called the “Significant”

Example 8.3.6: (For Base-10 numbers.) The following two numbers can be written in the form (8.3.3) as follows

$$\begin{aligned}
 -165.78 &= -0.16578 \times 10^3 \\
 0.00004717 &= 0.4747 \times 10^{-4}
 \end{aligned} \tag{8.3.4}$$

The mantissa is usually *normalized* so that the leading digit, i.e. the first digit past the decimal point, is not zero.

Example 8.3.7: The number $\frac{1}{34} = 0.029411765\dots$, if stored in a floating point base 10 system that only allowed four decimal places could be stored as

$$0.0294 \times 10^0 \text{ or } 0.2941 \times 10^{-1}$$

The second form, the normalized form, is more accurate and uses the same storage capacity. This is the reason for the normalization.

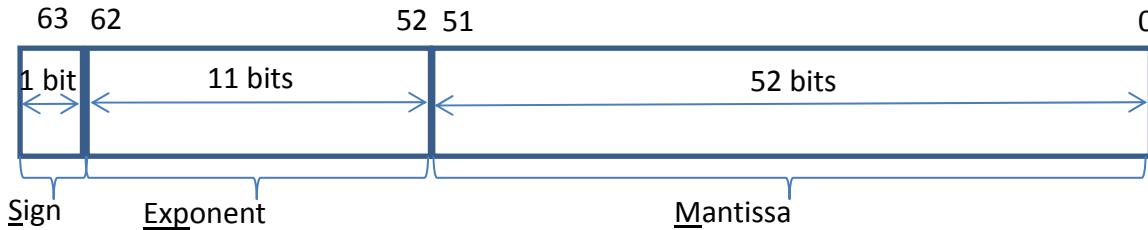
Because the mantissa only holds a finite number of significant figures, it is an inevitable source of *round off errors* of the *chopping* type.

Some features of floating point numbers that we shall discover are:

- a) There is a limited range of numbers that may be represented in a computer.
 - a. An attempt to utilize a number larger than the range results in *overflow error*.
 - b. An attempt to utilize a number smaller than the range results in *underflow error*.
- b) There are only a finite number of quantities that can be represented within the range.

- a. Other numbers are approximated through *rounding off*, i.e. *chopping*.

The *IEEE 64 bit floating-point number system* is virtually the standard for computer number systems.⁵ It is the number system used within MATLAB. The system, as the name indicates, starts with 64 bits as the binary representation of the floating point number. The problem is how to convert these 64 bits into a floating point number. There is not a unique way to make this conversion. The history of the digital computer is full of different choices. Fortunately, over time a standard has been adopted.⁶ The rules begin by representing the 64 bits in three components as follows



In this figure, S , Exp and M are binary numbers of size 1, 11 and 52 bits respectively.

The question is how to take the three pieces of information:

1.

$$S = b_{63} = \begin{cases} 0 \\ 1 \end{cases} \quad (8.3.5)$$

2.

$$Exp = \underbrace{b_{62}b_{61}b_{60}b_{59}b_{58}b_{57}b_{56}b_{55}b_{54}b_{53}b_{52}}_{11 \text{ bits}} \quad (8.3.6)$$

3.

$$M = \underbrace{b_{51}b_{50}b_{49}b_{48}b_{47}b_{46}b_{45}b_{44}b_{43}b_{42}b_{41}b_{40}b_{39}b_{38}b_{37}b_{36}b_{35} \cdots b_7b_6b_5b_4b_3b_2b_1b_0}_{52 \text{ bits}} \quad (8.3.7)$$

and calculate a real number which we will call v .

⁵ http://en.wikipedia.org/wiki/IEEE_floating_point. This standard was established in 1985 and went through a significant revision in 2008. An excellent discussion of this standard can be found in the article, Floating Points, by Cleve Moler. Dr. Moler is one of the founders of MATLAB and this article and others can be found at http://www.mathworks.com/company/newsletters/news_notes/clevescorner/.

⁶ The structure of the IEEE Standard is that the binary numbers are ordered consistent with the real numbers they produce. This means that if the 64 bit number is viewed as a 64 digit real number and ordered accordingly, the resulting output real numbers are ordered in the same fashion.

The rules are somewhat complicated. First, one adopts a convention on the sign of v and then calculates two real numbers. The steps are:

- a) If $S = 0$, v is a positive number. If $S = 1$, v is a negative number. More formally,

$$\begin{aligned} S = 0 &\Rightarrow v \geq 0 \\ S = 1 &\Rightarrow v \leq 0 \end{aligned} \quad (8.3.8)$$

The 64th bit is called the *sign bit*.

- b) The binary number Exp defined by (8.3.6) is converted to a real number E by the formula

$$E = b_{62} \times 2^{10} + b_{61} \times 2^9 + b_{60} \times 2^8 + b_{59} \times 2^7 + b_{58} \times 2^6 + b_{57} \times 2^5 + b_{56} \times 2^4 + b_{55} \times 2^3 + b_{54} \times 2^2 + b_{53} \times 2^1 + b_{52} \times 2^0 \quad (8.3.9)$$

Because we all have MATLAB handy, it is probably useful to recognize that (8.3.9) can be written as the matrix product

$$E = \left[2^{10} \ 2^9 \ 2^8 \ 2^7 \ 2^6 \ 2^5 \ 2^4 \ 2^3 \ 2^2 \ 2^1 \ 2^0 \right] \begin{bmatrix} b_{62} \\ b_{61} \\ b_{60} \\ b_{59} \\ b_{58} \\ b_{57} \\ b_{56} \\ b_{55} \\ b_{54} \\ b_{53} \\ b_{52} \end{bmatrix} \quad (8.3.10)$$

It follows from the definition (8.3.9) that E is a *positive integer*. The largest value of E , corresponds to the choice

$$Exp_{\max} = \underbrace{11111111111}_{11 \text{ bits}} \quad (8.3.11)$$

Given (8.3.11), it follows from (8.3.10) that ⁷

$$E_{\max} = 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 2^{11} - 1 = 2047 \quad (8.3.12)$$

It is also evident that the *smallest* value of E is *zero*, and it arises when $\text{Exp} = \underbrace{00000000000}_{11 \text{ bits}}$.

Therefore, the definition (8.3.9) yields

$$0 \leq E \leq 2047 \quad (8.3.13)$$

- c) The binary number M defined by (8.3.7) is converted to a real number f by the formula

$$f = b_{51} \frac{1}{2} + b_{50} \frac{1}{2^2} + b_{49} \frac{1}{2^3} + b_{48} \frac{1}{2^4} + b_{47} \frac{1}{2^5} + \dots + b_2 \frac{1}{2^{50}} + b_1 \frac{1}{2^{51}} + b_0 \frac{1}{2^{52}} \quad (8.3.14)$$

The matrix version of (8.3.14) is

$$f = \left[\begin{array}{ccccccccc} \frac{1}{2} & \frac{1}{2^2} & \frac{1}{2^3} & \frac{1}{2^4} & \frac{1}{2^5} & \cdot & \cdot & \cdot & \frac{1}{2^{50}} & \frac{1}{2^{51}} & \frac{1}{2^{52}} \end{array} \right] \begin{bmatrix} b_{51} \\ b_{50} \\ b_{49} \\ b_{48} \\ b_{47} \\ \vdots \\ \vdots \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} \quad (8.3.15)$$

Note that (8.3.14) yields

$$0 \leq f < 1 \quad (8.3.16)$$

⁷ We have used the geometric series identity

$$2^n - 1 = 2^0 + 2^1 + 2^2 + \dots + 2^{n-1}$$

Another important conclusion that comes from the formula (8.3.14) is that $2^{52} f$ is a *positive integer*. It follows from (8.3.14) that the *largest* value of f arises when

$$M = \underbrace{1111111111 \cdots 1111111111}_{52 \text{ bits}} \quad (8.3.17)$$

and, from (8.3.14), the corresponding value of f is

$$f_{\max} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots + \frac{1}{2^{50}} + \frac{1}{2^{51}} + \frac{1}{2^{52}} \quad (8.3.18)$$

This expression can be manipulated as follows:

$$\begin{aligned} f_{\max} &= \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots + \frac{1}{2^{50}} + \frac{1}{2^{51}} + \frac{1}{2^{52}} \\ &= \frac{1}{2^{52}} (2^0 + 2^1 + 2^2 + \cdots + 2^{47} + 2^{48} + 2^{49} + 2^{50} + 2^{51}) \\ &= \frac{1}{2^{52}} (2^{52} - 1) = 1 - \frac{1}{2^{52}} \end{aligned} \quad (8.3.19)$$

It follows from (8.3.14) that the *smallest* value of f arises when

$$M = \underbrace{0000000000 \cdots 0000000000}_{52 \text{ bits}} \quad (8.3.20)$$

and the resulting value of f is zero. It is useful to record the *smallest nonzero* value of f . It follows from (8.3.14) that this value is

$$f_{\min} = 2^{-52} \quad (8.3.21)$$

In summary, the real number f has the following properties:

- a) Its smallest value is zero
- b) Its smallest nonzero value is $f_{\min} = 2^{-52}$
- c) Its largest value is $f_{\max} = 1 - \frac{1}{2^{52}}$

Given

1. The sign of v as determined by (8.3.8),

2. The value E from (8.3.10),
and
3. The value f from (8.3.15),

the number v is calculated utilizing the following table:

Cases	E	f	v
1	0	0	$\Rightarrow v = (-1)^s 0 = \pm 0 = 0$
2	0	$\neq 0$	$\Rightarrow v = (-1)^s f 2^{-1022}$
3	$E_{\max} = 2047$	0	$\Rightarrow v = (-1)^s \text{inf} = \pm \text{inf}$
4	$E_{\max} = 2047$	$\neq 0$	$\Rightarrow v = NaN$
5	$0 < E < 2047$		$\Rightarrow v = (-1)^s (1 + f) 2^{E-1023}$

Case 5 in this table reflects what is called *exponential biasing*. In the case being discussed, the biasing is $2^{\text{Number of bits in } Exp-1} - 1 = 2^{11-1} - 1 = 1023$. The advantage of biasing is that it makes unnecessary the use of an additional exponential bit to allow for negative exponents.

The list of rules that produced this table tells us how to take a 64 bit binary number and calculate a real number. In the trivial example earlier, where we started with a 2 bit binary number, we obtained a short list of real numbers. Thus, the relationship was not one to one. We can always find a real number not in the list

$$\begin{aligned}[00] &\rightarrow 0 \\ [01] &\rightarrow 1 \\ [10] &\rightarrow 2 \\ [11] &\rightarrow 3\end{aligned}$$

Given this example, it should not be a surprise to find that the full set of 64 bit binary numbers *does not* generate the full set of real numbers. There are some “gaps” in the output. In other words, there are real numbers that cannot be expressed in one of the forms shown in the last column of the above table. Equivalently, the function defined by the above table whose domain is the set of 64 bit binary numbers is not a one to one correspondence with the real numbers.

Example 8.3.9: A simple example of a real number that is representable as a 64 bit binary number in the format of the above table is

$$\nu = -3.5 \quad (8.3.22)$$

Because (8.3.22) is a negative number, it follows from the above table that

$$S = 0 \quad (8.3.23)$$

It also follows for a number whose magnitude is finite and larger than one that option 5 in the above table is the possible representation of ν . Therefore, we are seeking an f defined by (8.3.14) and an E defined by (8.3.9) that obeys

$$3.5 = (1 + f)2^{E-1023} \quad (8.3.24)$$

Since

$$3.5 = \left(1 + \frac{3}{4}\right)2 \quad (8.3.25)$$

it follows from (8.3.24) that

$$E = 1024 = 2^{10} = \begin{bmatrix} 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (8.3.26)$$

which from (8.3.6)

$$Exp = 10000000000 \quad (8.3.27)$$

From (8.3.14) and (8.3.25), we can determine the other parts of the 64 bit representation of (8.3.22) if we can satisfy

$$f = \frac{3}{4} = \frac{1}{2} + \frac{1}{4} = b_{51} \frac{1}{2} + b_{50} \frac{1}{2^2} + b_{49} \frac{1}{2^3} + b_{48} \frac{1}{2^4} + b_{47} \frac{1}{2^5} \cdots + b_2 \frac{1}{2^{50}} + b_1 \frac{1}{2^{51}} + b_0 \frac{1}{2^{52}} \quad (8.3.28)$$

Therefore, (8.3.28) and (8.3.7) yield

$$M = \underbrace{1100000000000000 \cdots 00000000}_{52 \text{ bits}} \quad (8.3.29)$$

Equations (8.3.23), (8.3.27) and (8.3.29) yield the 64 bit representation of (8.3.22).

It was mentioned above that $2^{52} f$ is a *positive integer*. This fact and the formulas in the above table show that not every real number can be produced from the set of rules listed. Consider the following example of a real number that cannot be converted to a 64 bit binary number.⁸

Example 8.3.10: Consider the real number

$$\nu = .1 = \frac{1}{10} \quad (8.3.30)$$

The object is to work backwards and try to determine the associated 64 bit binary number. Because ν is positive, then

$$S = 0 \quad (8.3.31)$$

Because of the size of ν , the only case that fits the above table is the last row and, as such, we need to write ν in the form

$$\nu = \frac{1}{10} = (1 + f) 2^{E-1023} \quad (8.3.32)$$

We shall show that there is *no* 52bit M and *no* 11 bit Exp that will obey this equation. We shall show that it takes an *infinite sequence* of binary numbers to generate this real number.

Basically, our task is to write the factor $\frac{1}{10}$ in the form of the right hand side of (8.3.32).

When this is done, if successful, we can identify f and E . The first step is to write

⁸ Example 8.2.8 is a standard one that is used to illustrate that not every real number can be generated from computers utilizing the IEEE 64 bit floating-point number system. See, for example, Floating points by Cleve Moler, http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf.

$$\frac{1}{10} = \underbrace{\frac{1}{2^4}}_{\substack{\text{Smallest} \\ n \text{ that makes} \\ 2^n \text{ larger} \\ \text{than } 10}} - \frac{2^4}{10} = \frac{1}{2^4} \frac{16}{10} = \frac{1}{2^4} (1 + .6) = (1 + .6) 2^{-4} \quad (8.3.33)$$

The right hand side of (8.3.33) is of the form of (8.3.32) with $f = .6$ and $E = 1019$.

The next step is to *try* to express the factor $.6$ in (8.3.33) in the form (8.3.14), i.e., in the form

$$f = .6 = b_{51} \frac{1}{2} + b_{50} \frac{1}{2^2} + b_{49} \frac{1}{2^3} + b_{48} \frac{1}{2^4} + b_{47} \frac{1}{2^5} \cdots + b_2 \frac{1}{2^{50}} + b_1 \frac{1}{2^{51}} + b_0 \frac{1}{2^{52}} \quad (8.3.34)$$

It is this step that will fail as the following calculation illustrate:

$$\begin{aligned} f = .6 &= .5 + .1 = \frac{1}{2} + .1 = \frac{1}{2} + .065 + .035 = \frac{1}{2} + \frac{1}{2^4} + .035 \\ &= \frac{1}{2} + \frac{1}{2^4} + .03125 + .00375 = \frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} + .00375 \\ &= 1 \frac{1}{2} + 0 \frac{1}{2^2} + 0 \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + .00375 \end{aligned} \quad (8.3.35)$$

If you continue this process, you get the repeating sequence

$$f = .6 = 1 \frac{1}{2^1} + 0 \frac{1}{2^2} + 0 \frac{1}{2^3} + 1 \frac{1}{2^4} + 1 \frac{1}{2^5} + 0 \frac{1}{2^6} + 0 \frac{1}{2^7} + 1 \frac{1}{2^8} + 1 \frac{1}{2^9} + 0 \frac{1}{2^{10}} + 0 \frac{1}{2^{11}} + \cdots \quad (8.3.36)$$

Thus, this f *does not* produce a 52 bit M that obeys the definition (8.3.14). Instead, one needs an infinite number of bits to generate a binary number of the form

$$10011 \underbrace{0011}_{\substack{\text{Repeating} \\ \text{Sequence}}} 00110011 \cdots \quad (8.3.37)$$

Therefore, it takes an *infinite* number of terms in this series and, corresponding, a binary number with an infinite number of digits to cause the right hand side of (8.3.32) to equal exactly the left.

The above example illustrates a source of *round off errors* created by the limits of a finite number of bits.

Example 8.3.11: Find the smallest nonzero positive floating point number.

We have two cases in the above table to consider, Case 2 and Case 5. In Case 2, we need calculate the number by the formula $(-1)^S f 2^{-1022}$. The smallest positive nonzero value predicted by this number corresponds to the case $f_{\min} = 2^{-52}$ (see (8.3.21)) which would imply

$$\nu_{\min} = f_{\min} 2^{-1022} = 2^{-52} 2^{-1022} = 2^{-1074} \quad (8.3.38)$$

For Case 5, we calculate ν by the formula $(-1)^S (1+f) 2^{E-1023}$. This case requires that $0 < E < 2047$. Therefore, the smallest positive value would be

$$\nu_{\min} = (1+0) 2^{1-1023} = 2^{-1022} \quad (8.3.39)$$

Thus, the strict application of our definitions yields (8.3.38) as the smallest nonzero positive floating point number.

In MATLAB, the command **realmin** produces a number different from (8.3.38).⁹ The actual **realmin** output is

$$\text{realmin} = 2.2251 \times (10)^{-308} \quad (8.3.40)$$

Another way to write this output is

$$\text{realmin} = 2^{-1022} \quad (8.3.41)$$

which is the same as (8.3.39).

The above two smallest positive numbers, equations (8.3.38) and (8.3.39) involve an aspect of the IEEE standard that is optional and somewhat controversial. Many, but not all, machines allow the Case 2 in the above Table. The fact that Case 2 is allowed means we have floating point numbers in the interval between 2^{-1074} and 2^{-1022} . For these machines, a number that is entered smaller than 2^{-1074} is set to zero. For those machines that do not utilize Case 2, a number that is entered smaller than 2^{-1022} is set to zero.

It is readily shown that MATLAB with the input

⁹ MATLAB Help explains that **realmin** yields the smallest positive normalized floating point number. The qualifier normalized has to do with the mantissa not having zero in its first digit past the decimal point.

```
>> 2^(-1074)
```

yields the output

```
ans =
```

```
4.9407e-324
```

The input

```
>> 2^(-1075)
```

yields the output

```
ans =
```

```
0
```

These results illustrate that MATLAB *does utilize* Case 2 when it constructs real numbers from the 64 bit representation.

Example 8.3.12: Find the largest floating point number produced by Case 2.

From the table, this number is going to be given by

$$\nu_2|_{\max} = f_{\max} 2^{-1022} = \left(1 - \frac{1}{2^{52}}\right) \frac{1}{2^{1022}} = \frac{1}{2^{1022}} - \frac{1}{2^{1074}} \quad (8.3.42)$$

where the result (8.3.19) has been used.

Example 8.3.13: Find the *largest* floating point number.

From the table, the largest floating point number will arise from Case 5 and the result must be calculated from $\nu = (1 + f) 2^{E-1023}$. In order to calculate this number, we use

$f = f_{\max} = \left(1 - \frac{1}{2^{52}}\right)$ and $E = 2046$, the largest E allowed in Case 5. Therefore

$$(1 + f_{\max}) 2^{2046-1023} = \left(2 - \frac{1}{2^{52}}\right) 2^{1023} \quad (8.3.43)$$

MATLAB produces this result with the command **realmax**. The MATLAB output from this command

```
>> realmax
```

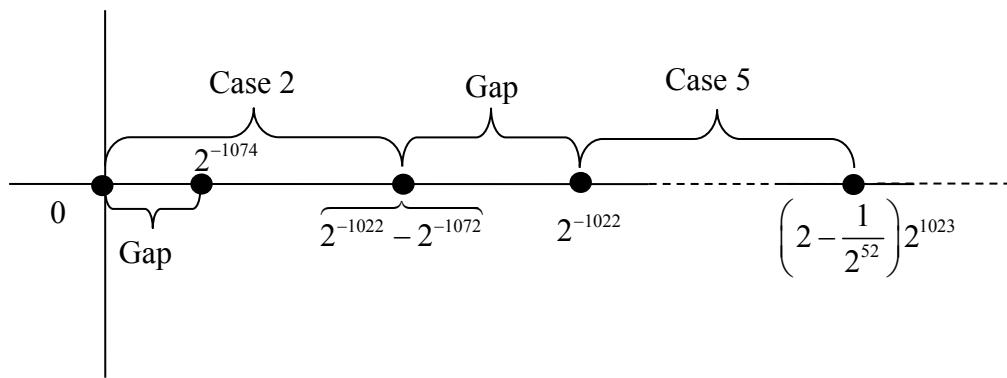
is

```
ans =
```

```
1.7977e+308
```

The same result is obtained if you enter $\left(2 - \frac{1}{2^{52}}\right)2^{1023}$. If you enter a larger number, such as $\left(2 - \frac{1}{2^{52}}\right)2^{1024}$, MATLAB yields **inf**. This is MATLAB's way of telling you that you have created an *overflow*.

The results (8.3.38), (8.3.42), (8.3.39) and (8.3.43) are illustrated by the following figure:



for positive floating point numbers. A simple calculation with MATLAB shows that

$$2^{-1022} - 2^{-1072} = 2.225073858507199(10)^{-308} \quad (8.3.44)$$

and

$$2^{-1022} = 2.225073858507201(10^{-308}) \quad (8.3.45)$$

Thus, the gap shown above between $2^{-1022} - 2^{-1072}$ and 2^{-1022} is tiny.

The *machine epsilon* is the *largest* number ϵ that obeys

$$(1 + \epsilon) - 1 = 0 \quad (8.3.46)$$

In MATLAB, the machine epsilon is produced by the command **eps**. This command yields

$$\epsilon = 2.22044604925031 \times (10)^{-16} = 2^{-52} \quad (8.3.47)$$

When any calculation tries to produce a smaller value than $2^{-1074} = 2^{-52}2^{-1022} = \epsilon \times \text{realmin}$, it is said to *underflow*.

Summary of Kinds of Computing Errors Discussed (A Partial List)

- Truncation Error: Arises when series representations of functions are approximated by a finite number of terms.
- Round off Error: Caused by representing and storing numbers with a finite number of bits.
 - Computers can only store a finite number of digits.
 - Example: Arises when π and other irrational numbers are represented by a finite number of digits. (Chopping type of round off)
- Overflow/Underflow

There are other kinds of errors that arise in numerical analysis. Examples are

- Loss of Significance: Caused by subtracting two numbers that are almost equal in value.
- Error Propagation: Caused by multiplying and dividing numbers that contain errors.
 - The idea is that if variables x and y have errors, and we perform addition, multiplication, and, possibly, even compute $f(x, y)$ for some function f , then the *errors propagate*.

Example 8.3.14: The following example is one that illustrates round off and loss of significance errors.¹⁰ The object is to find the solution of the system of equations

$$\begin{aligned} x &= 4 / 3 \\ y &= x - 1 \\ z &= y + y + y \\ w &= 1 - z \end{aligned} \quad (8.3.48)$$

The obvious solution for w is $w = 0$. However, if you utilize MATLAB, the result turns out to be $2^{-52} = \epsilon$. The round off when calculating y , combined with the loss of significant figures in

¹⁰ Floating points, by Cleve Moler,
http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf.

the last step, when calculating w , is the source of the error. The m-file that carries out the above calculation is

```
clc
clear
format long
% The format command above causes the output
% to be displayed with 14 to 15 digits
% after the decimal point.
A=[1,0,0,0;-1,1,0,0;-3,1,0,0;0,1,1]
b=[4/3;-1;0;1]
A\b
```

The MATLAB output from the above script is

```
A =
1 0 0 0
-1 1 0 0
0 -3 1 0
0 0 1 1

b =
1.33333333333333
-1
0
1

ans =
1.33333333333333
0.33333333333333
1
2.22044604925031e-016
```

It is often possible to structure the sequence of calculations in a problem so as to decrease the magnitude of round off errors and to lower the possibility of overflow and underflow errors. The basic idea is to make the intermediate answers as close to unity as possible in consecutive multiplication and division processes. For example, write

- $\frac{(xy)}{z}$ when x and y in the multiplication are very different in magnitude

- $x\left(\frac{y}{z}\right)$ when y and z in the division are close in magnitude
- $\left(\frac{x}{z}\right)y$ when x and z in the division are close in magnitude

Exercises

8.3.1: Utilize the IEEE 64-bit floating point standard and convert the following 64 bit number to a real number:

$$\begin{aligned} S &= 1 \\ Exp &= 10000000001 \\ M &= 1100001100 \end{aligned} \tag{8.3.49}$$

The answer is -7.0469 .

8.3.2: Utilize the IEEE 64-bit floating point standard and convert the following 64 bit number to a real number:

$$\begin{aligned} S &= 1 \\ Exp &= 10000000000 \\ M &= 1100 \end{aligned} \tag{8.3.50}$$

The answer is -3.5 .

Chapter 9

ROOTS OF NONLINEAR EQUATIONS

At the risk of oversimplification, it is useful to think of a large class of mathematical problems as having their solution in the solution of the following three types of equations:

- a) Systems of nonlinear real valued algebraic equations

$$\left\{ \begin{array}{l} f_1(x_1, x_2, \dots, x_n) = 0 \\ f_2(x_1, x_2, \dots, x_n) = 0 \\ \cdot \\ \cdot \\ f_m(x_1, x_2, \dots, x_n) = 0 \end{array} \right. \quad m \text{ equations} \quad n \text{ unknowns} \quad (9.1.1)$$

which one would normally write in a vector notation as

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad (9.1.2)$$

- b) Systems of nonlinear ordinary differential equations ¹

$$\left\{ \begin{array}{l} \frac{dx_1}{dt} = f_1(t, x_1, x_2, \dots, x_n) \\ \frac{dx_2}{dt} = f_2(t, x_1, x_2, \dots, x_n) \\ \cdot \\ \cdot \\ \frac{dx_m}{dt} = f_m(t, x_1, x_2, \dots, x_n) \end{array} \right. \quad m \text{ equations} \quad n \text{ unknowns} \quad (9.1.3)$$

which one would normally write in a vector notation as

¹ Initial conditions are also required.

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x}) \quad (9.1.4)$$

c) Systems of nonlinear partial differential equations.

Depending upon the problem and depending upon the details of the above functions, one can use analytical methods to generate exact or good approximate solutions. In the more complicated cases, one resorts to numerical methods. In real life, it is a fact that one usually approaches problems with a mix of analytical solutions and numerical solutions. This dual approach, when possible, is a part of the confidence building that one needs to go through in order to have confidence in an answer.

Our *first interest* is in those circumstances where the physical problem is governed by (9.1.1) or, equivalently, by (9.1.2). Later, in Chapter 12, we shall look at situations where the physical problem is governed by ordinary differential equations. Thus, we have decided to study physical problems governed by equation (9.1.2) and the answer to these problems is found by *finding the roots* of

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad (9.1.5)$$

We shall begin the study of equations like (9.1.5) by looking at the special case of one nonlinear equation in *one scalar variable*. Thus, we are initially interested in schemes for finding the roots of the equation

$$f(x) = 0 \quad (9.1.6)$$

To be more precise, we are interested in finding the *real roots* of the equation (9.1.6) in some given interval $[a, b]$ of the real axis. The qualifier *real* when we are talking about the roots of (9.1.6) is important to note at this point in the discussion. An equation of the type (9.1.6) can have *real and complex* numbers for its roots. Until we restrict $f(x)$ to be a polynomial, we will not attempt to find its complex roots.

Roughly speaking, there are two methods that are used to find the roots of the real valued function $f(x)$.

- *Bracketing method:* This method involves first establishing that the root lies within some interval and the use of an iteration scheme to find the root.
- *Open method:* An open method utilizes formulas that only require a *single* starting value or *two* starting values that do not necessarily bracket the root.

The two bracketing methods we shall discuss are

- Bisection method
- False position method

Before we consider these two methods, we shall first, In Section 9.1 discuss the MATLAB's built in root calculation tool, **fzero**. In Section 9.4, we shall look at the open method known as the Newton-Raphson method.

All of these methods represent different ways of finding the roots of (9.1.6).

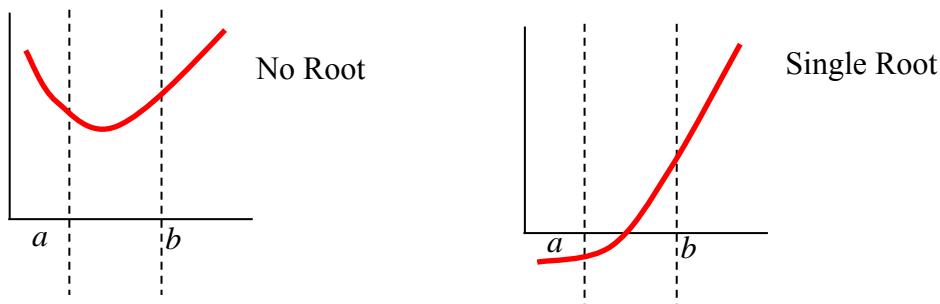
Section 9.1. Use of Graphics to Locate the Real Roots of a Nonlinear Equation

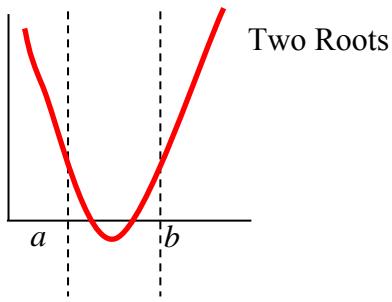
In this section, we shall explain the importance of first plotting the function whose real zeros we seek. The use of graphical representations of the function is of fundamental importance to the implementation of the methods mentioned above for finding the real roots of real valued functions.

Because we are now dealing with one dimensional nonlinear problems, we can make extensive use of *graphical representations* of the function $f(x)$ as we attempt to find its zeros. Graphical representations, properly used, provide good initial estimates of the roots. They do not provide great numerical accuracy. The numerical schemes we shall discuss provide this accuracy.

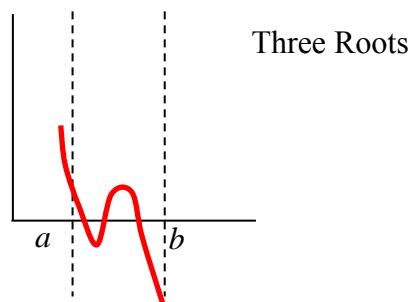
Graphical representations can be slightly treacherous. If the wrong scale is adopted, for example, one might misinterpret the results. However, with proper use they are helpful. The following sketches help to remind us how to interpret a graph of a *continuous* function

$$y = f(x) \quad \text{for } x \in [a, b] \quad (9.1.7)$$





Two Roots



Three Roots

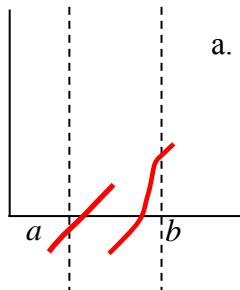
A couple of *almost* general statements one can make from these sketches are as follows:

- If $f(a)$ and $f(b)$ are of *opposite* sign, then the number of roots between a and b is an *odd* number.
- If $f(a)$ and $f(b)$ have the *same* sign, then the number of roots between a and b is an *even* number.

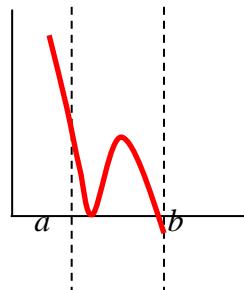
The two exceptions to these two observations occur when

- (9.1.7) is *not* continuous.
- The roots of (9.1.7) are *not* distinct.

The following sketches illustrate these two exceptions



a. Two Roots

b. Two Roots:
one of the two
roots has a
multiplicity of
two.

Example 9.1.1: You are given the transcendental function

$$y = f(x) = \cos(8x) + \frac{99}{100} \sin(5x) \quad (9.1.8)$$

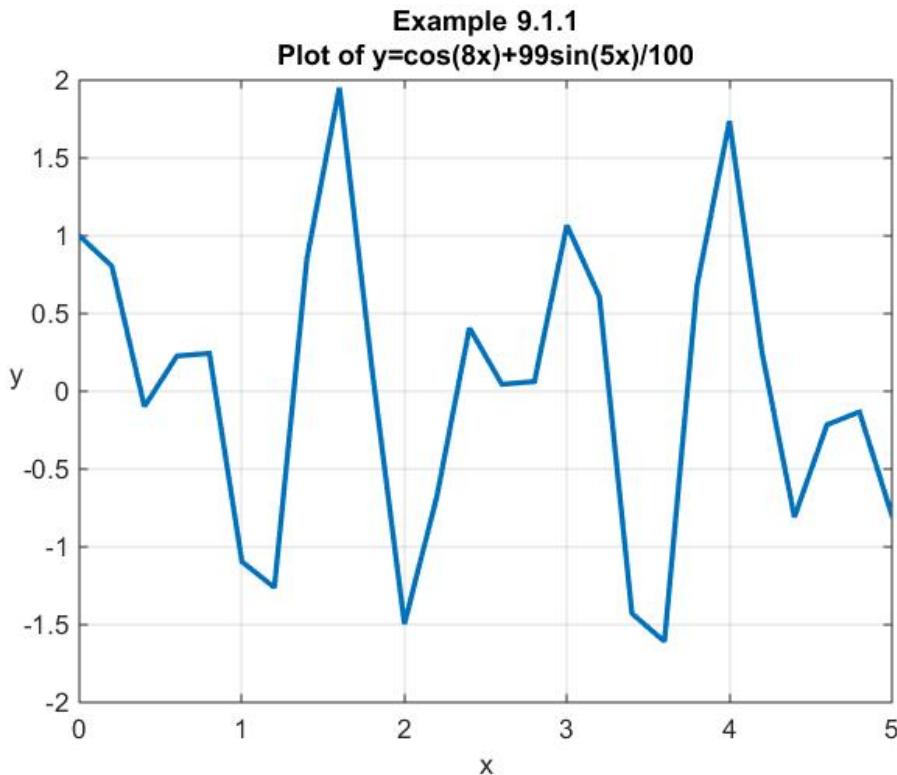
and you are interested in learning about the roots for $x \in [0, 5]$. The first plot we shall generate will use the script

```

clc
clear
% Define the range and 0.2 data intervals for the x vector
x = [0:0.2:5];
% Calculate the value f(x) for given x vector
y = cos(8*x)+99*sin(5*x)/100;
plot(x,y,'LineWidth',2);
grid on;
title({'Example 9.1.1','Plot of y=cos(8x)+99sin(5x)/100'});
xlabel('x');
ylabel('y','Rotation',0)

```

The graph generated is



The graph suggests that there are nine roots in the interval $x \in [0, 5]$. However, the example was structured to lead you to an *incorrect conclusion*. We chose to evaluate the function defined by (9.1.8) at intervals of .2. The grid is *not* sufficiently refined, and the computer has produced a figure with jagged connection elements. A more refined figure arises from the script

```

clc
clear
% Define the range and 0.05 data intervals for the x vector

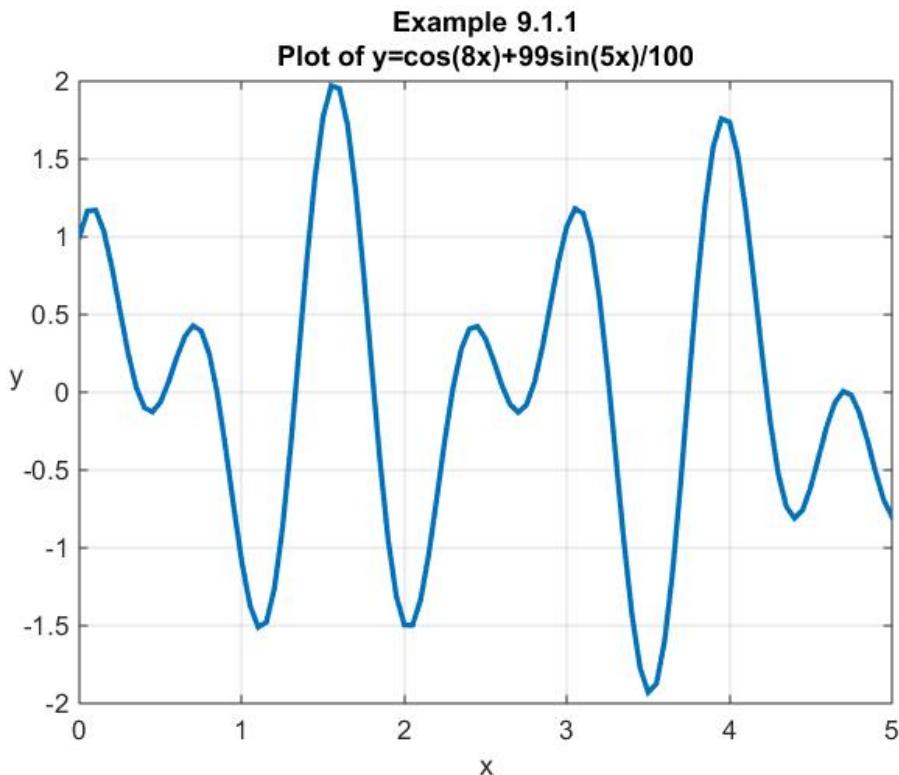
```

```

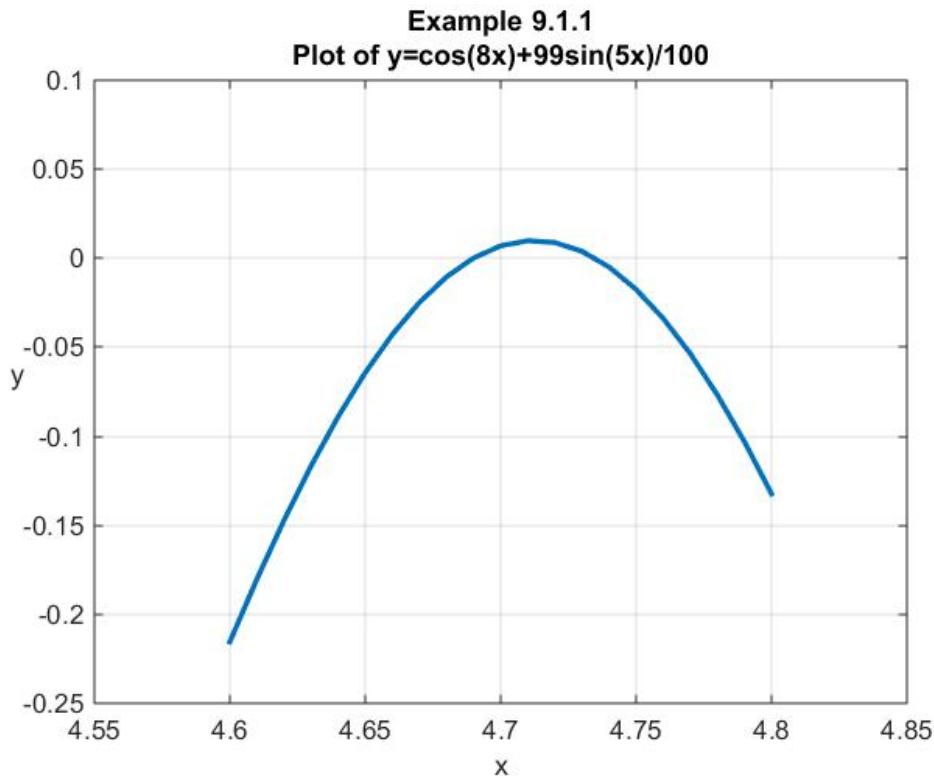
x = [0:0.05:5];
% Calculate the value f(x) for given x vector
y =cos(8*x)+99*sin(5*x)/100;
plot(x,y,'LineWidth',2);
grid on;
title({'Example 9.1.1','Plot of y=cos(8x)+99sin(5x)/100'});
xlabel('x');
ylabel('y','Rotation',0);

```

The graph generated is



Instead of the nine roots shown in first figure, this figure shows the possibility twelve roots with what appears to be a double root near $x = 4.7$. The possibility of a double root can be investigated further by utilizing the zoom feature of MATLAB plots. However, for our purposes here, we can achieve the same result by utilization of the same script above but with the range to be $x \in [4.6, 4.7]$ and an interval of 0.01. In this case the following figure is obtained:



This zoomed in look at the curve near $x = 4.7$ shows that what first appeared not to be a root and next appeared to be a double root is, in fact, two distinct roots very close together.

The moral to the story illustrated by Example 9.1.1 is that computer graphics are a practical method of understanding in rough terms the location of the roots before one actually tries to calculate their values. However, one must be cautious about how the graphical information is utilized. In the applications you usually want greater numerical accuracy than can be read from the graph. This greater accuracy is obtained from the methods we shall study next. As we shall see, these schemes adopted depend in a fundamental way on knowledge about the roots that a properly utilized graph will provide.

Exercises:

9.1.1: Use a graphical approach and analyze and estimate the roots of

$$y = x^3 - 4.1000x^2 + 3.6025x - 0.9075 \quad (9.1.9)$$

Section 9.2. MATLAB's **fzero** Command

As one would expect, MATLAB has a command that will deliver the real roots of nonlinear equations. For those not especially interested in how MATLAB implements this calculation, it is sufficient to simply introduce the command and certain elements of its syntax. The command in question is **fzero**. This command will find the root of a *continuous function* of one variable. In its simplest form, its syntax is

$$\mathbf{x=fzero(fun,x0)} \quad (9.2.1)$$

This command tries to find a zero of **fun** near **x0**.

Summary Information About **fzero**:

- **fun** is a function handle for either a function m-file, an anonymous function or an inline function.²
- If **x0** is a scalar, the value **x** returned by **fzero** is near a point where **fun** changes sign, or **NaN** if the search fails. In this case, the search terminates when the search interval is expanded until an **Inf**, **NaN**, or complex value is found.
- If **x0** is a vector of length two, **fzero** assumes **x0** is an interval where the sign of **fun(x0(1))** differs from the sign of **fun(x0(2))**. An error occurs if this is not true. Calling **fzero** with such an interval guarantees that **fzero** returns a value near a point where **fun** changes sign.
 - An interval **x0** must be finite; it cannot contain **±Inf**.

A check of the MATLAB **Help** facility will provide significant additional information about this command. It is also helpful to execute in the MATLAB command window

```
>> edit fzero
```

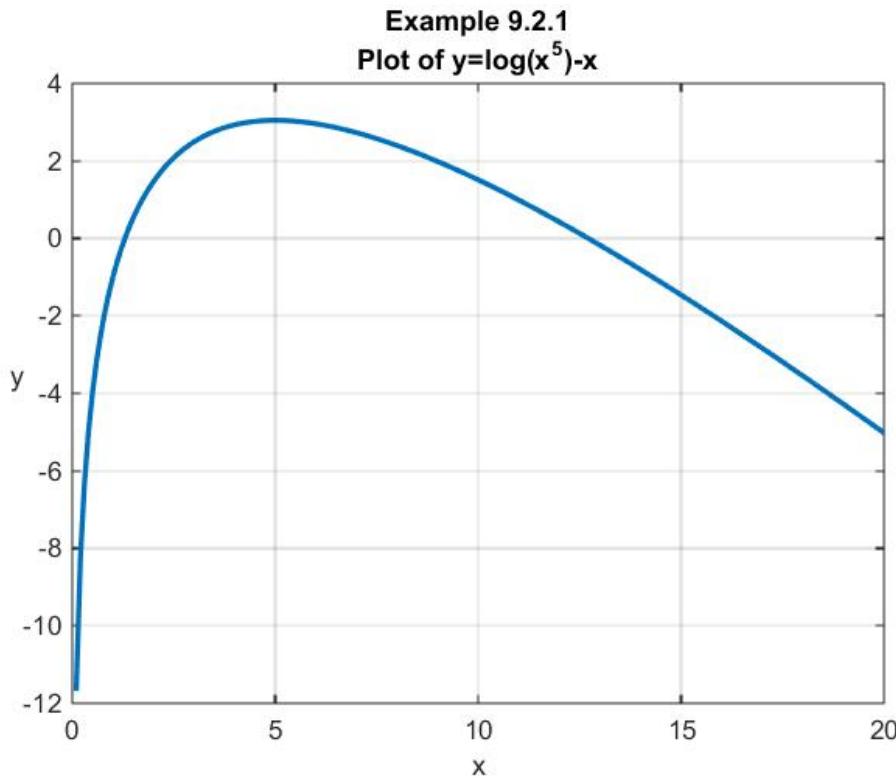
The result opens the m-file **fzero.m** and displays the structure of the numerical method implemented. The method is an enhanced version of the bisection method to be discussed in the next section.

Example 9.2.1: As an example of the use of **fzero**, find the root of the function

$$y = \ln(x^5) - x \quad (9.2.2)$$

in the interval $(0, 5]$. The plot of (9.3.2) in the interval $(0, 20]$ is

² Inline functions, anonymous functions and function m-files are discussed in Appendix A as well as online. In Example 9.2.1 we will illustrate how an inline function and an anonymous function are used with **fzero**.



The graph illustrates that the function has real roots near $x = 1$ and $x = 13$. We are interested in utilizing **fzero** near $x = 1$. The answer is produced by the commands

```
>> format long
>> x=fzero('log(x^5)-x',1.0)
```

where the command **format long** has been introduced in order to increase the number of significant figures. In addition, the function (9.2.2) has been introduced in the format of an *inline function*. This kind of MATLAB function is briefly mentioned in Appendix A. The output from the above is

```
x =
1.295855509095369
```

If the function (9.2.2) is entered as an *anonymous function*, the root is found by entering into MATLAB

```
>> format long
>> x=fzero(@(x)(log(x^5)-x),1.0)
```

MATLAB's documentation indicates that the inline function format will be removed in a future release. The anonymous function is the recommended format. Anonymous functions are also briefly discussed in Appendix A.

The command **fzero** has built in options which allow information about the iterations towards the root to be displayed in MATLAB's command window. For example, if we enter

```
>> options=optimset('Display','iter');
>> f=@(x)(log(x^5)-x)
>> fzero(f,1,options)
```

The output in the command window is

```
Search for an interval around 1 containing a sign change:
Func-count      a          f(a)          b          f(b)          Procedure
    1            1            -1            1            -1  initial interval
    3        0.971716      -1.11518      1.02828      -0.888826  search
    5            0.96        -1.16411        1.04        -0.843896  search
    7        0.943431      -1.23459      1.05657      -0.781436  search
    9            0.92        -1.33691        1.08        -0.695195  search
   11        0.886863      -1.48719      1.11314      -0.577226  search
   13            0.84        -1.71177        1.16        -0.4179  search
   15        0.773726      -2.05641      1.22627      -0.206372  search
   17            0.68        -2.60831        1.32        0.0681587  search

Search for a zero in the interval [0.68, 1.32]:
Func-count      x          f(x)          Procedure
    17            1.32        0.0681587  initial
    18            1.3037      0.0223371  interpolation
    19        1.29582      -8.92705e-005  interpolation
    20        1.29586      3.64826e-007  interpolation
    21        1.29586      5.93392e-012  interpolation
    22        1.29586            0  interpolation

Zero found in the interval [0.68, 1.32]

ans =
1.295855509095369
```

The first block of output shows a series of calculations where **fzero** searched for an interval $[a,b]$ that obeyed the requirement that the given function change sign. After the interval $[a,b]=[0.68,1.32]$ was established, the second block of output shows the iterations leading to the root.

Example 9.2.2: Utilize **fzero** to find the nonzero positive root in the interval $[0,4]$ for the equation

$$f(x) = \tan(\pi - x) - x = 0 \quad (9.2.3)$$

If we simply pick a couple of points and use syntax like above, the results from various efforts are

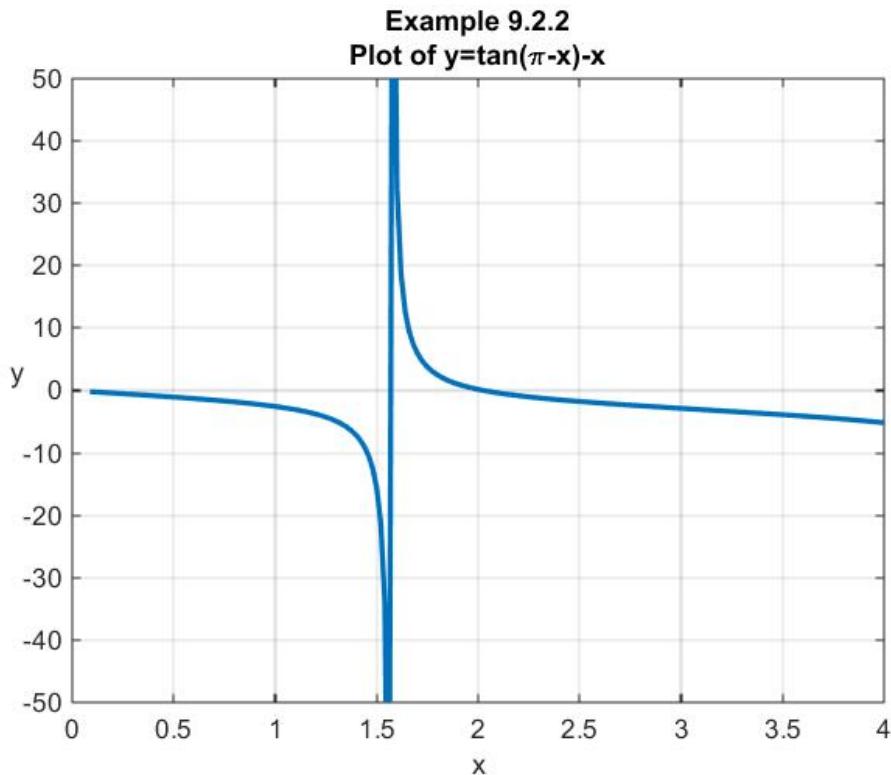
```
fzero('tan(pi-x)-x',1) => ans = 1.5708
```

```
fzero('tan(pi-x)-x',2) => ans = 2.0288
```

```
fzero('tan(pi-x)-x',[1,2])⇒ ans = 1.5708
```

```
fzero('tan(pi-x)-x',[1,4])⇒ Error using fzero (line 274)
The function values at the interval endpoints must differ in
sign.
```

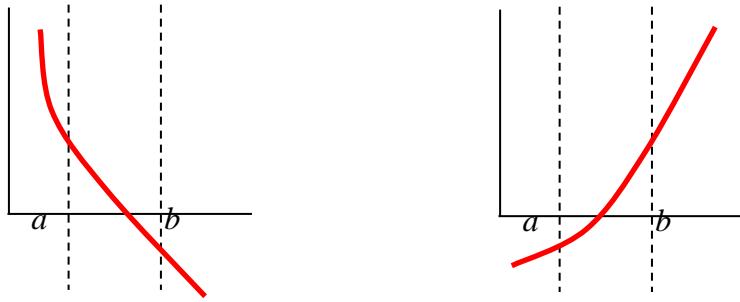
Without further information, the two different answers combined with the error message do not provide any confidence that the output of MATLAB is providing correct information. The problem is that the function has not been graphed. We simply do not know enough about the function to be confident that **fzero** has been applied correctly. The first problem with the function is that it is *not defined* at $x = \frac{\pi}{2}$. The method utilized by **fzero** breaks down when the function is *not continuous* in the neighborhood of the root. A plot of the function $y = \tan(\pi - x) - x$ looks like



The figure reveals that the root in question is the one at $x = 2.0288$ and the point $x = 1.5708$ is simply not a root. The fact that the interval $[1, 2]$ bracketed the discontinuity caused the result predicted by **fzero**, i.e. $x = 1.5708$ to be incorrect. The choice of interval $[1, 4]$ correctly produced the above error message.

Section 9.3. Bracketing Methods

In this Section, we shall take a more detailed look at the techniques that provide the foundation for finding the real roots of nonlinear equations of a real variable. As indicated in the introduction to this chapter, one important method of finding the real roots of a real valued nonlinear equation is a *bracketing method*. This method involves first establishing that the root lies within some interval and the use of an iteration scheme to find the root. The first of these that we shall discuss is known as the *bisection method*. This method is a systematic method of finding the distinct roots of *continuous* functions. The basic idea is that near a root of a real valued continuous function one of the following graphs is correct:

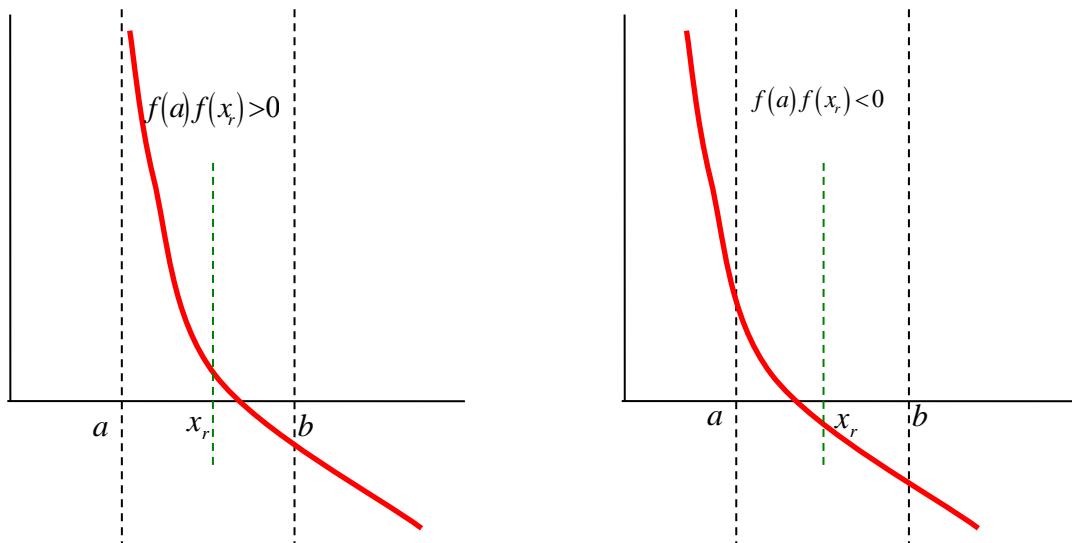


The analytical condition which reflects *both* of these graphs is that

$$f(a)f(b) < 0 \quad (9.3.1)$$

Bisection Method

By dividing the interval into smaller and smaller segments, one gets closer to the actual root. The *bisection method* is one where *the interval $[a, b]$ is always divided into half*. At each iteration, the test (9.3.1) is applied to determine which half of the divided interval contains the root. When this new interval is identified, it is divided in half and so forth. The following figure suggests the geometric arrangement.



The solution algorithm consists of the following sequence of steps:

Step 1: Choose lower $a = x_L$ and upper $b = x_U$ points in the neighborhood of the root such that $f(x_L)f(x_U) < 0$.

Step 2: Estimate that the root occurs at $x_r = \frac{x_L + x_U}{2}$

Step 3: You next need to determine which subinterval contains the actual root. Make the following evaluations to determine the correct subinterval:

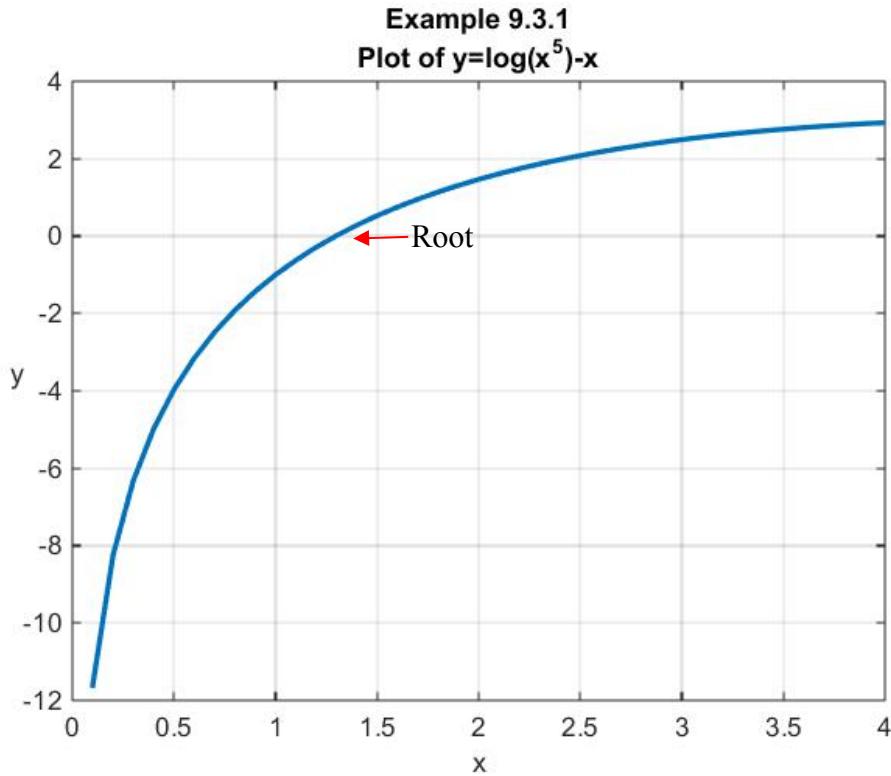
- If $f(x_U)f(x_r) < 0$, the root is contained in the left or lower subinterval. The next step is to set the new x_U to be the current x_r and return to Step 2.
- If $f(x_U)f(x_r) > 0$, the root is not contained in the left or lower subinterval. It lies in the right or upper subinterval. The next step is to set the new x_U to the current x_r and to return to Step 2.
- If $f(x_U)f(x_r) = 0$ (or is sufficiently close to zero by some prescribed rule), the root equals x_r .

Example 9.3.1: As an example of the bisection method, you are asked to find the real root of the function (9.2.2), repeated,

$$y = \ln(x^5) - x \quad (9.3.2)$$

in the interval $(0,5]$.³ The plot of (9.3.2) in the interval $(0,20]$ is shown as a part of our discussion in Example 9.2.1.

We are interested in utilizing the bisection method to find the root near $x=1$. In order to get a feel for the function, we first plot $f(x) = \ln(x^5) - x$ for $x \in [0,1,4]$. The result is



From this figure we see that a good place to start the iteration process towards the root is to start with the initial bracket $[a,b] = [1.0,1.5]$. The iteration process towards the root is as follows:

$$\text{Step 1: } (x_{L(1)}, x_{U(1)}) = (1.0, 1.5) \Rightarrow (f(x_{L(1)}), f(x_{U(1)})) = (-1.000, 0.5273) \Rightarrow f(x_{L(1)})f(x_{U(1)}) < 0$$

The first estimate of the root is to assume it is at the bisection of the initial interval. Therefore,

$$x_{r(1)} = \frac{1}{2}(1.0 + 1.5) = 1.25 \quad (9.3.3)$$

The value of the function at $x_{r(1)}$ is

³ The notation $(0,5]$ identifies the interval $\{x | 0 < x \leq 5\}$. The interval $(0,20]$ is defined in a similar fashion.

$$f(x_{r(1)}) = f(1.25) = -0.1343 \quad (9.3.4)$$

and, as a result,

$$f(x_{L(1)})f(x_{r(1)}) > 0 \quad (9.3.5)$$

Conclusion: Because $f(x_{r(1)})$ is negative, as is $f(x_{L(1)})$, the root lies in the *new interval*
 $(x_{L(2)}, x_{U(2)}) = (1.25, 1.5), (f(x_{L(2)}), f(x_{U(2)})) = (-0.1343, 0.5273) \Rightarrow f(x_{L(2)})f(x_{U(2)}) < 0$

Step 2: Second Estimate of Root:

$$x_{r(2)} = \frac{1}{2}(1.25 + 1.5) = 1.3750 \quad (9.3.6)$$

The value of function at this value of x is

$$f(x_{r(2)}) = f(1.3750) = 0.2173 \quad (9.3.7)$$

which implies that

$$f(x_{L(2)})f(x_{r(2)}) < 0 \quad (9.3.8)$$

Conclusion: Root lies in *new interval*

$$(x_{L(3)}, x_{U(3)}) = (1.25, 1.3750), (f(x_{L(3)}), f(x_{U(3)})) = (-0.1343, 0.2173) \Rightarrow f(x_{L(3)})f(x_{U(3)}) < 0$$

Step 3: Third Estimate of Root:

$$x_{r(3)} = \frac{1}{2}(1.25 + 1.3750) = 1.3125 \quad (9.3.9)$$

Value of function at this x value:

$$f(x_{r(3)}) = f(1.3125) = 0.0472 \quad (9.3.10)$$

which implies that

$$\Rightarrow f(x_{L(3)})f(x_{r(3)}) < 0 \quad (9.3.11)$$

Conclusion: Root lies in *new* interval

$$(x_{L(4)}, x_{U(4)}) = (1.25, 1.3125) \Rightarrow (f(x_{L(4)}), f(x_{U(4)})) = (-0.1343, 0.0472) \Rightarrow f(x_{L(4)})f(x_{U(4)}) < 0$$

Step 4: Fourth Estimate of Root:

$$x_{r(4)} = \frac{1}{2}(1.25 + 1.3125) = 1.2813 \quad (9.3.12)$$

Value of the function at this value of x is

$$f(x_{r(4)}) = f(1.2813) = -0.0421 \quad (9.3.13)$$

which implies that

$$f(x_{L(4)})f(x_{r(4)}) > 0 \quad (9.3.14)$$

Conclusion: Root lies in *new* interval

$$(x_{L(5)}, x_{U(5)}) = (1.2813, 1.3125), (f(x_{L(5)}), f(x_{U(5)})) = (-0.0421, 0.0472) \Rightarrow f(x_{L(5)})f(x_{U(5)}) < 0$$

Step 5: Fifth Estimate of Root:

$$x_{r(5)} = \frac{1}{2}(1.2813 + 1.3125) = 1.2969 \quad (9.3.15)$$

The value of function at this value of x is:

$$f(x_{r(5)}) = f(1.2969) = 0.0030 \quad (9.3.16)$$

which implies that

$$f(x_{L(5)})f(x_{r(5)}) < 0 \quad (9.3.17)$$

Conclusion: Root lies in *new* interval

$$(x_{L(6)}, x_{U(6)}) = (1.2813, 1.2969), (f(x_{L(6)}), f(x_{U(6)})) = (-0.0421, 0.0030) \Rightarrow f(x_{L(6)})f(x_{U(6)}) < 0$$

Depending upon the accuracy desired in the final answer, one could stop the iteration by accepting (9.3.15) as the answer. One more iteration yields the results

Step 6: Sixth Estimate of Root:

$$x_{r(6)} = \frac{1}{2}(1.2813 + 1.2969) = 1.2891 \quad (9.3.18)$$

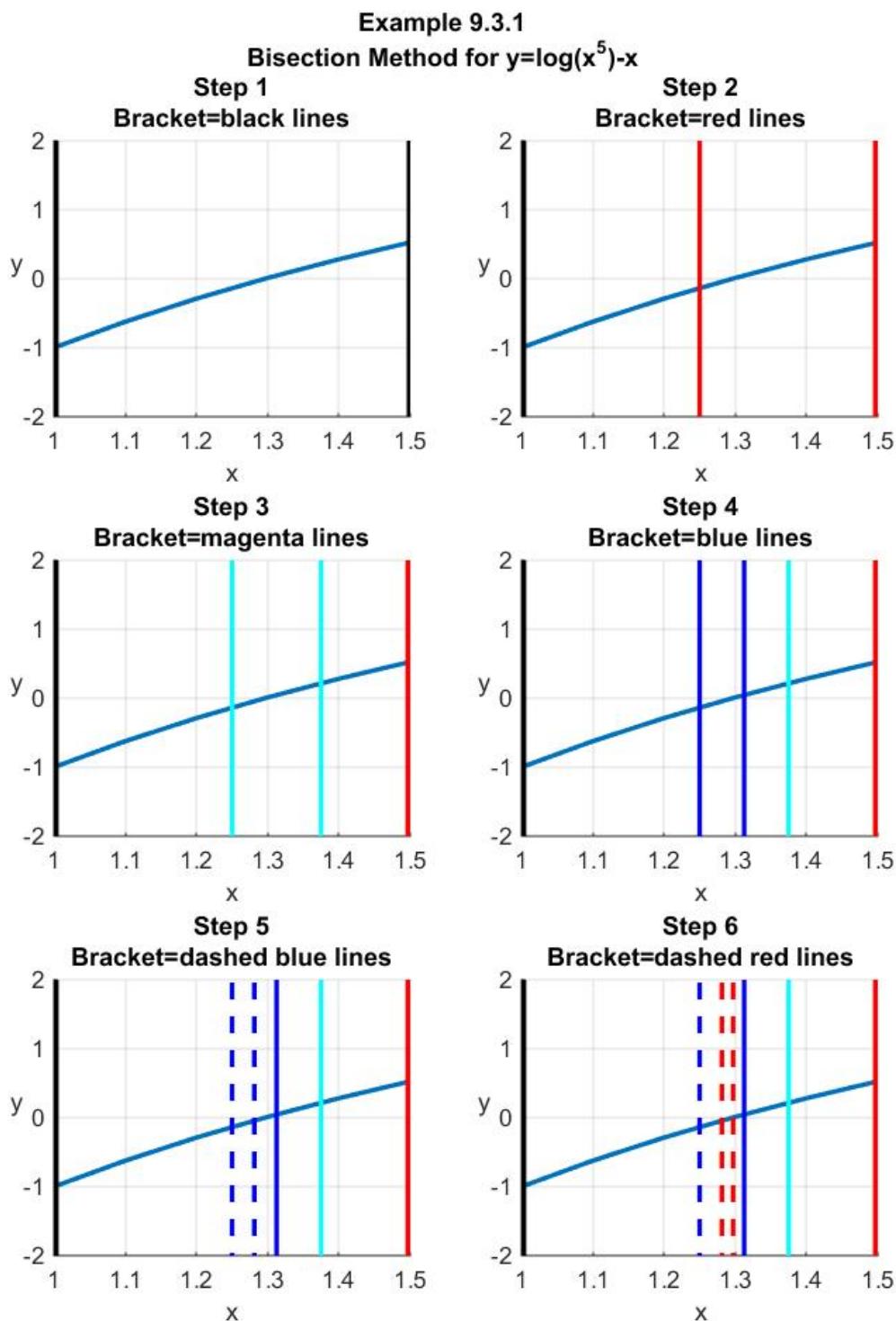
The value of function at this value of x is

$$f(x_{r(6)}) = f(1.2891) = -0.0194 \quad (9.3.19)$$

Conclusion: Root lies in

$$(x_{L(7)}, x_{U(7)}) = (1.2891, 1.2969) \Rightarrow (f(x_{L(7)}), f(x_{U(7)})) = (-0.0194, 0.0030) \Rightarrow f(x_{L(7)})f(x_{U(7)}) < 0$$

The progression of the six bisections above is displayed in the following set of figures:



It is possible to implement the above iteration with a spreadsheet. A thirteen iteration process produces the following spreadsheet:

Bisection Method Example: Example 9.3.1

Step	x _L	x _U	x _r	f=ln(x^5)-x				
				f(x _L)	f(x _U)	f(x _r)	f(x _L)*f(x _r)	ε _a
1	1.0000	1.5000	1.25000000	-1.0000	0.5273	-0.1343	0.134282	
2	1.2500	1.5000	1.37500000	-0.1343	0.5273	0.2173	-0.029175	9.0909%
3	1.2500	1.3750	1.31250000	-0.1343	0.2173	0.0472	-0.006334	4.7619%
4	1.2500	1.3125	1.28125000	-0.1343	0.0472	-0.0421	0.005649	2.4390%
5	1.2813	1.3125	1.29687500	-0.0421	0.0472	0.0029	-0.000123	1.2048%
6	1.2813	1.2969	1.28906250	-0.0421	0.0029	-0.0195	0.000820	0.6061%
7	1.2891	1.2969	1.29296875	-0.0195	0.0029	-0.0083	0.000161	0.3021%
8	1.2930	1.2969	1.29492188	-0.0083	0.0029	-0.0027	0.000022	0.1508%
9	1.2949	1.2969	1.29589844	-0.0027	0.0029	0.0001	0.000000	0.0754%
10	1.2949	1.2959	1.29541016	-0.0027	0.0001	-0.0013	0.000003	0.0377%
11	1.2954	1.2959	1.29565430	-0.0013	0.0001	-0.0006	0.000001	0.0188%
12	1.2957	1.2959	1.29577637	-0.0006	0.0001	-0.0002	0.000000	0.0094%
13	1.2958	1.2959	1.29583740	-0.0002	0.0001	-0.0001	0.000000	0.0047%

In the above spreadsheet, ε_a , is the *relative error* defined by

$$\varepsilon_{a(n)} = 100 \left| \frac{x_{r(n)} - x_{r(n-1)}}{x_{r(n)}} \right| \quad \text{for } n = 2, 3, \dots \quad (9.3.20)$$

and it measures how rapidly the iteration converges to a common value.

While spreadsheets will provide the roots, MATLAB is an ideal tool to carry out the kinds of iterative steps illustrated above. A primitive m-file to carry out these steps for the function (9.3.2) is as follows:

```

clc
clear
f=inline('log(x^5)-x');
%The function can be entered an alternate
% way by use of the idea of an anonymous
%function. The syntax for this case is
% f=@(x)(log(x^4)-.7)
%Set the initial bracket
xl=1;
xu=1.5;
n=1;
while n < 100
    xr=(xl+xu)/2;
    if (f(xl)*f(xr))<0

```

```

    xu=xr;
end
if (f(xl)*f(xr))>0
    xl=xr;
end
n=n+1;
end
format long
root=xr

```

This m-file produces the result

```

root =
1.295855509095369

```

which is the same as produced by **fzero**. The m-file is primitive in that it simply specifies a maximum number of iterations, 100 in this case, and does not concern itself whether or not an accurate result is obtained. A slightly improved version of the above but with the same limitation is

```

clc
clear
f=inline('log(x^5)-x');
%The function can be entered an alternate
% way by use of the idea of an anonymous
%function. The syntax is f=@(x)(log(x^4)-.7)
%Set the initial bracket

xl=1;
xu=1.5;
n=1;
while n < 100
    xr=(xl+xu)/2;
    if (f(xl)*f(xr))<0
        xu=xr;
    elseif (f(xl)*f(xr))>0
        xl=xr;
    end
    n=n+1;
end
format long
root=xr

```

This version uses the **if-elseif-end** structure briefly mentioned in Appendix A. An m-file that is even more robust is

```

clc
clear all
%For purposes of illustration, the function
%is entered %as an anonymous function.
%The syntax is f=@(x)(log(x^4)-.7)

f=@(x)(log(x^5)-x)
xl=1;
xu=1.5;
n=0;
xr=xl;
e=1;
e_s=.01;
while e>e_s;
    xrold=xr;
    xr=(xl+xu)/2;
    n=n+1;
    if xr~=0;
        e=abs((xr-xrold)/xr)*100;
    end
    if f(xl)*f(xr)<0 ;
        xu=xr;
    elseif f(xl)*f(xr)>0 ;
        xl=xr;
    end
end
root=xr

```

This file prescribes a *stopping criteria* based on whether or not the *relative error* **e** is larger than the prescribed *stop value* **e_s=.01**. In other words, it concerns itself with the accuracy of the answer and does not prescribe in advance the number of iterations.

The *iterative relative error* in the root is defined as a percentage by

$$\varepsilon_{a(n)} = \left| \frac{x_{r(n)} - x_{r(n-1)}}{x_{r(n)}} \right| \times 100\% \quad \text{for } n = 1, 2, \dots \quad (9.3.21)$$

The stopping criteria, introduced in the above script, occurs when $\varepsilon_{a(n)}$ reached a prescribed lower value.

An online search will produce a variety of implementations of the bisection method or varying sophistication. One that implements a lot of features is the function m-file⁴

⁴ This m-file is essentially the same as one published on page 127 of the textbook, Applied Numerical Methods with MATLAB, Second Edition, by Steven C. Chapra, McGraw Hill, 2008.

```

function [root,ea,iter] = bisect(func,xl,xu,es,maxit,varargin)
%SOURCE:Page 127 of Applied Numerical Methods with MATLAB
%by Steven C. Chapra
%bisect: root location zeros
% [root,ea,iter]=bisect(func,xl,xu,es,maxit,p1,p2,...):
% uses bisect method to find the root of func
%input:
% func=name of function
% xl,xu=lower and upper guesses
% es=desired relative error (default=.0001%)
% maxit=maximum allowable iterations (default=50)
% p1,p2,...=additional parameters used by func
%output:
% root=real root
% ea=appropriate relative error(%)
% iter=number of iterations
if nargin<3,error('at least 3 input arguments required'),end
test=func(xl,varargin{:})*func(xu,varargin{:});
if test>0,error('no sign change'),end
if nargin<4|isempty(es),es=0.0001;end
if nargin<5|isempty(maxit),maxit=50;end
iter=0;xr=xl;
while (1)
    xrold=xr;
    xr=(xl+xu)/2;
    iter=iter+1;
    if xr~=0,ea=abs((xr-xrold)/xr)*100;end
    test=func(xl,varargin{:})*func(xr,varargin{:});
    if test<0
        xu=xr;
    elseif test>0
        xl=xr;
    else
        ea=0;
    end
    if ea<=es|iter>=maxit,break,end
end
root=xr;

```

As explained in Appendix A where function m-files are discussed, this function m-file passes the function **func** along with the lower **xl** and upper **xu** guesses. In addition, an optional stopping criteria **es** and maximum iteration number **maxit** can be entered. The function first checks whether there are sufficient arguments, and if the initial guesses bracket a sign change. If not, an error message is displayed and the calculation is terminated. It also assigns default values if **maxit** and **es** are not supplied. Then a **while...break** loop is employed to implement the bisection algorithm. The

iteration proceeds until the approximate error falls below **es** or the iterations exceed **maxit**. The root of the function $y = \ln(x^5) - x$, that was introduced in Example 9.2.1, can be solved utilizing **bisect.m** with the script

```
clc
clear
format long
f=@(x)(log(x^5)-x)
[root,ea,iter]=bisect(f,.5,2)
```

The output from this script is

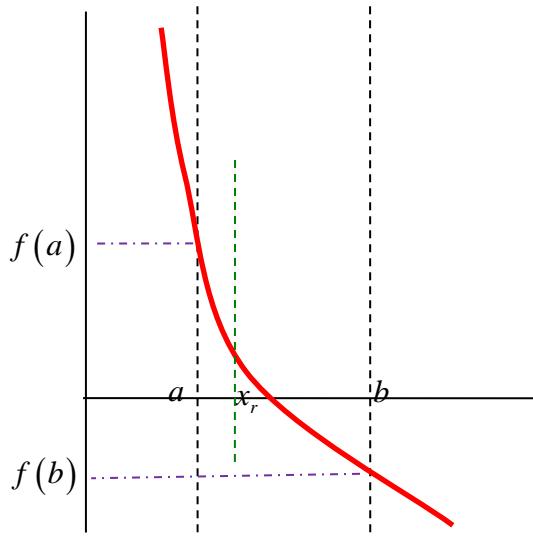
```
root =
1.295855760574341

ea =
5.519562894775238e-005

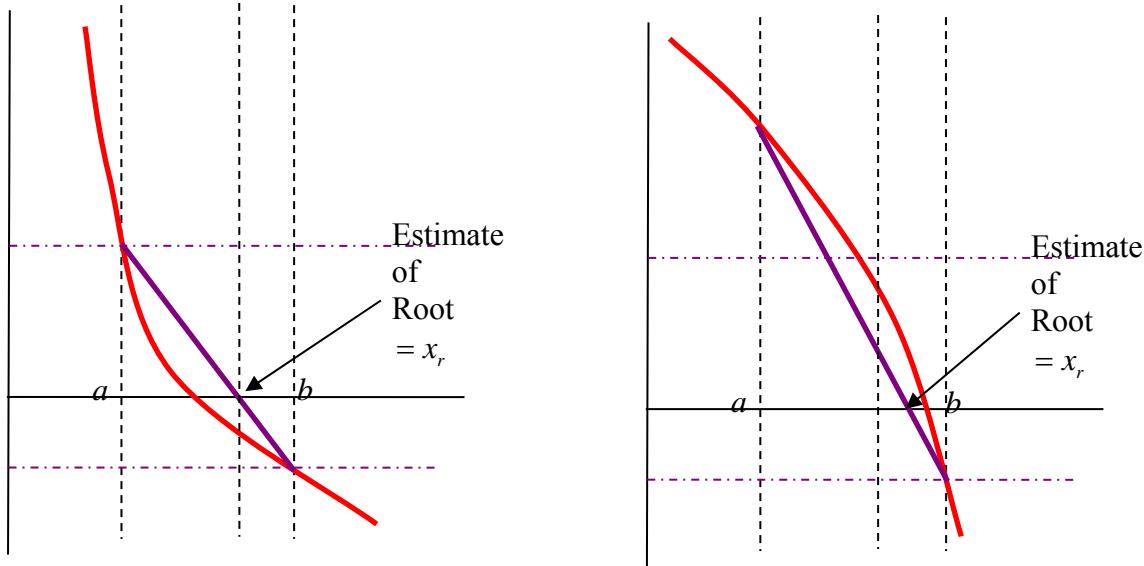
iter =
21
```

The False Position Method

The second bracketing method we wish to discuss is the False Position Method mentioned in the introduction to this chapter. The basic idea is best explained by an examination of the following figure:



With the Bisection Method, we estimated the root by estimating the value x_r . The estimates that formed the basis of the iteration method were calculated by averaging values of x lying on both sides of the root. The *False Position Method* (or *Interpolation Method*) tries to find the root by *interpolating* between the two values $f(a)$ and $f(b)$. Geometrically, the method projects a straight line as shown in the following figures:



The false position method estimates the root to be the point where the straight line connecting $f(a)$ and $f(b)$ passes through $y = 0$. The equation that determines this point is obtained by equating the slopes as follows:

$$\underbrace{\frac{f(b) - f(a)}{b - a}}_{\text{Slope of line from } f(b) \text{ to } f(a)} = \underbrace{\frac{f(b) - 0}{b - x_r}}_{\text{Slope of line from } f(b) \text{ to } f(x_r) = 0} \quad (9.3.22)$$

If this equation is solved for the value x_r , the result is

$$x_r = b - \frac{f(b)}{f(b) - f(a)}(b - a) = \frac{af(b) - bf(a)}{f(b) - f(a)} \quad (9.3.23)$$

This equation gives an *interpolated approximation* for the root. By an iterative process, one attempts to obtain an accurate approximation to the actual root. This iterative scheme is as follows:

Step 1: Choose lower $x_{L(1)}$ and upper $x_{U(1)}$ in the neighborhood of the root such that

$$f(x_{L(1)})f(x_{U(1)}) < 0.$$

Step 2: Estimate that the root occurs at

$$x_{r(1)} = \frac{x_{L(1)}f(x_{U(1)}) - x_{U(1)}f(x_{L(1)})}{f(x_{U(1)}) - f(x_{L(1)})} \quad (9.3.24)$$

in the interval $[x_{L(1)}, x_{U(1)}]$.

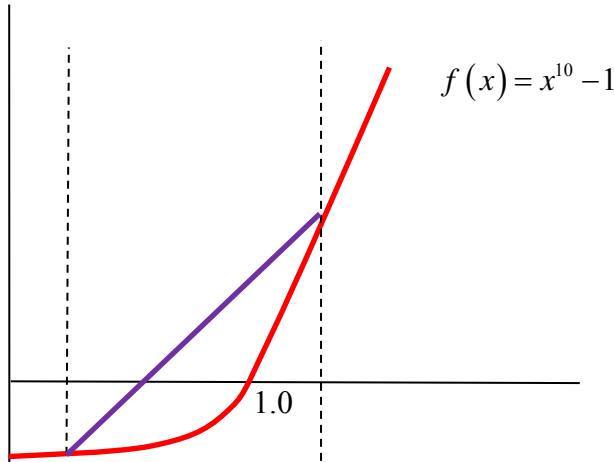
Step 3: You next need to determine which subinterval, $[x_{L(1)}, x_{r(1)}]$ or $[x_{r(1)}, x_{U(1)}]$ contains the actual root. Make the following evaluations to determine which subinterval contains the root:

- a. If $f(x_{L(1)})f(x_{r(1)}) < 0$, the root is contained in the left or lower subinterval. The next step is to set $x_{L(2)} = x_{L(1)}$ and $x_{U(2)} = x_{r(1)}$ and return to Step 2 applied to the interval $[x_{L(2)}, x_{U(2)}]$.
- b. If $f(x_{L(1)})f(x_{r(1)}) > 0$, the root is *not* contained in the left or lower subinterval. It lies in the right or upper subinterval. The next step is to set $x_{L(2)} = x_{r(1)}$ and $x_{U(2)} = x_{U(1)}$ and to return to Step 2 applied to the interval $[x_{L(2)}, x_{U(2)}]$.
- c. If, after n iterations $f(x_{L(n)})f(x_{r(n)}) = 0$ (or is sufficiently close to zero by some prescribed rule), the root equals $x_{r(n)}$.

The above is the *same iterative procedure* used for the bisection method except that the location of $x_{r(n)}$ is determined by a different procedure.

It turns out that the error for the False Position Method often, but not always, decreases faster than for Bisection Method. In the bisection case, the interval between $x_{r(n)}$ and $x_{L(n)}$ grew smaller through the iteration. In the false position case, one of the initial guesses may stay fixed though out the computation and the other guess converges on the root.

A premise of the False Position Method is that a straight line between x_L and x_U is a good approximation for the actual curve. A consequence is that if $f(x_L)$ is closer to zero than $f(x_U)$, then the root is closer to x_L than x_U . A counter example is a curve like the following:



The purple line is such a poor approximation for the red line in the neighborhood of the root $x_r = 1$, the bisection method converges faster than the false position method. We shall see an example below that confirms this fact in the case of the function $y = f(x) = x^{12} - 1$.

As with the Bisection Method, MATLAB can implement the False Position Method without difficulty. It is interesting to simply take the file **bisect.m** and modify it for the false position method. The following is that modification:

```

function [root,ea,iter] = falsepos(func,xl,xu,es,maxit,varargin)
%SOURCE: Simple modification of bisect.m for the False
%Position Method.
%falsepos: root location zeros
%    [root,ea,iter]=falsepos(func,xl,xu,es,maxit,p1,p2,...):
%        uses False Position Method to find the root of func
%input:
%    func=name of function
%    xl,xu=lower and upper guesses
%    es=desired relative error (default=.0001%)
%    maxit=maximum allowable iterations (default=50)
%    p1,p2,...=additional parameters used by func
%output:
%    root=real root
%    ea=appropriate relative error(%)
%    iter=number of iterations
if nargin<3,error('at least 3 input arguments required'),end
test=func(xl,varargin{:})*func(xu,varargin{:});
if test>0,error('no sign change'),end
if nargin<4|isempty(es),es=0.0001;end
if nargin<5|isempty(maxit),maxit=50;end

```

```

iter=0;xr=xl;
while (1)
    xrold=xr;
%Comment out the next line
%    xr=(xl+xu)/2;
%Insert the next line
    xr=(xl*func(xu,varargin{:})-
xu*func(xl,varargin{:}))/(func(xu,varargin{:})-
func(xl,varargin{:}));
    iter=iter+1;
    if xr~=0,ea=abs((xr-xrold)/xr)*100;end
    test=func(xl,varargin{:})*func(xr,varargin{:});
    if test<0
        xu=xr;
    elseif test>0
        xl=xr;
    else
        ea=0;
    end
    if ea<=es|iter>=maxit,break,end
end
root=xr;

```

Example 9.3.2: As an example that uses the above function m-file, consider the problem of finding the finite positive roots of the function

$$f(x) = e^{-x}(3.2\sin x - .5\cos x) \quad (9.3.25)$$

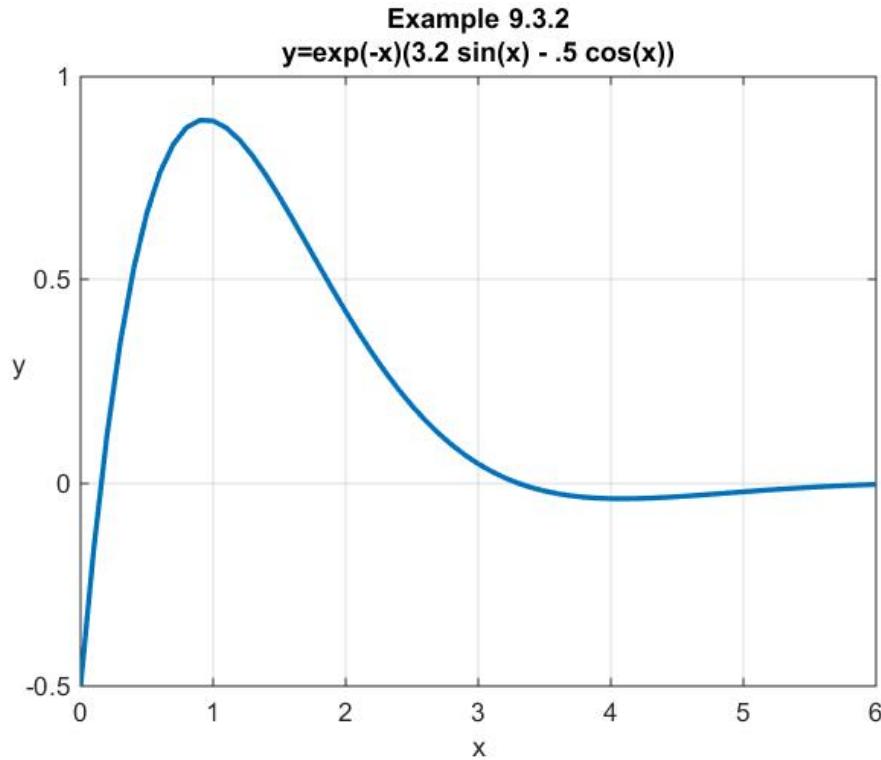
The script

```

clc
clear
x=[0:.1:6]
y=exp(-x).*(3.2*sin(x)-.5*cos(x))
plot(x,y,'LineWidth',2)
grid on
xlabel('x')
ylabel('y','Rotation',0)
title({'Exercise 9.3.2','y=exp(-x)(3.2 sin(x) - .5 cos(x))'})

```

produces the plot



This figure shows that there is a root in the interval $[0,1]$ and one in the interval $[3,4]$. The MATLAB script that utilizes the function **falsepos.m** to calculate the root in $[0,1]$ is

```

clc
clear
%Set the format long in order to display more digits
format long
%Define func as an anonymous function
func=@(x)(exp(-x).*(3.2*sin(x)-.5*cos(x)))
%Set starting bracket
xl=0
xu=1
[root,ea,iter] = falsepos(func,xl,xu)

```

The answer for **x** that this routine yields is

```

root =
0.154996746852896

```

```

ea =

```

```
1.951129133876905e-005
```

```
iter =
```

```
10
```

In a similar fashion, the root in the interval $[3,4]$ yields

```
root =
```

```
3.296590395688960
```

```
ea =
```

```
9.317296850808173e-005
```

```
iter =
```

```
10
```

Example 9.3.3: It was mentioned above that there are circumstances where the Bisection Method converges faster than the False Position Method. The function $y = x^{12} - 1$ was asserted to be an example. In this example, we shall use both **bisect** and **falsepos** to calculate the root in the interval $[0,2]$. Obviously, the two real roots of this function are $x = \pm 1$. There is no need to plot this function because we already know how to bracket its positive real root. The MATLAB script that utilizes the two function m-files **bisect.m** and **falsepos.m** to calculate the root in $[0,2]$ is

```
clc
clear
func=@(x)(x^12-1)
xl=0
xu=2
format long
[root,ea,iter] = falsepos(func,xl,xu)
[root,ea,iter] = bisect(func,xl,xu)
```

The answer for **x** that this routine yields is

For **falsepos.m**

```
root = 0.024268599922666
```

```
ea = 1.988059022401048
```

```
iter = 50
```

and

For **bisect.m**

```
root = 1
ea = 0
iter = 1
```

Therefore, for this example, **falsepos** did not find the answer in the allowed 50 iterations while **bisect.m** found it in 1 iteration. It is instructive to allow additional iterations in **falsepos.m** in order to determine when an acceptable answer is obtained. If we allow for a maximum number of iterations of 100 iterations by changing the script above to

```
clc
clear
func=@(x)(x^12-1)
xl=0
xu=2
format long
[root,ea,iter] = falsepos(func,xl,xu,[],100)
[root,ea,iter] = bisect(func,xl,xu,[],100)
```

The results are

For **bisect.m**

```
root = 0.048242717374229
ea = 0.987961771182121
iter = 100
```

and

For **bisect.m**

```
root = 1
ea = 0
iter = 1
```

Thus, after 100 iterations, the False Position Method is still not giving a good answer. If the maximum number of iterations is set at 4000, the result begins to get close to the known answer. The conclusion is evident. Namely, that for some functions the False Position Method converges very slowly.

The syntax in the last example requires an explanation. The functions **bisect** and **falspos** require a minimum of three arguments. These are the function, **func**, and the two points x_L and x_U . The functions contain the option to prescribe **es**, **maxit** and parameters that are required to define **func**. In the case where the default value is used for **es** and a value of **maxit** is used that is not the default, one utilizes a placeholder [] in the slot for **es**. This use of a placeholder is illustrated in the above script.

It is interesting to utilize **fzero** to calculate the root of $y = x^{12} - 1$. In particular, it is interesting to know how many iterations it would require to achieve an accurate answer for the root. If we utilize the various options associated with the function m-file **fzero.m** explained in MATLAB HELP, the following script

```
clc
clear
func=@(x)(x^12-1)
xL=0
xU=2
format long
[root,fval,exitflag,options]=fzero(func,[xU,xL])
```

Yields the output

```
root = 1.000000000000000
fval = -2.664535259100376e-015
exitflag = 1
options =
    intervaliterations: 0
        iterations: 7
        funcCount: 9
    algorithm: 'bisection, interpolation'
        message: [1x33 char]
```

In addition to the root, the output gives **fval**, the value of the function at the root, **exitflag**, which in this case tells you that **fzero** converged to a solution and **options**. The options provide a variety of information. In particular, it tells us that the calculation involved only 7 iterations.

Exercises:

9.3.1: Use the bisection method to find the roots of

$$y = \sin x + \cos(1 + x^2) - 1 \quad (9.3.26)$$

in the interval $x \in [1, 3]$.

9.3.2: Use the bisection method to find the roots of

$$y = \cos x + 10 \sin(1 + x^2) - 1 \quad (9.3.27)$$

in the interval $x \in [1, 3]$.

9.3.3: Use the bisection method to find the roots of

$$y = 2 \sin(\sqrt{x}) - x \quad (9.3.28)$$

in the interval $x \in [1.5, 2.5]$.

9.3.4: Use the false position method to find the roots of

$$y = e^{-x} (3.2 \sin x - .5 \cos x) \quad (9.3.29)$$

in the interval $x \in [0, 4]$.

9.3.5: Use the false position method to find the roots of

$$y = 5 \sin^2 x - 8 \cos^5 x \quad (9.3.30)$$

in the interval $x \in [0, 10]$.

9.3.6: Use either the bisection or the false position method to find the annual interest on a \$63,821.59 car loan to be paid in sixty equal monthly payments of \$1,165.31. Recall that loan amount, annual interest, payment amount and annual interest are related by the formula

$$\text{Payment} = \text{Loan Amount} \frac{\frac{x}{12} \left(1 + \frac{x}{12}\right)^n}{\left(1 + \frac{x}{12}\right)^n - 1} \quad (9.3.31)$$

where x is the annual interest and n is the number of payments.

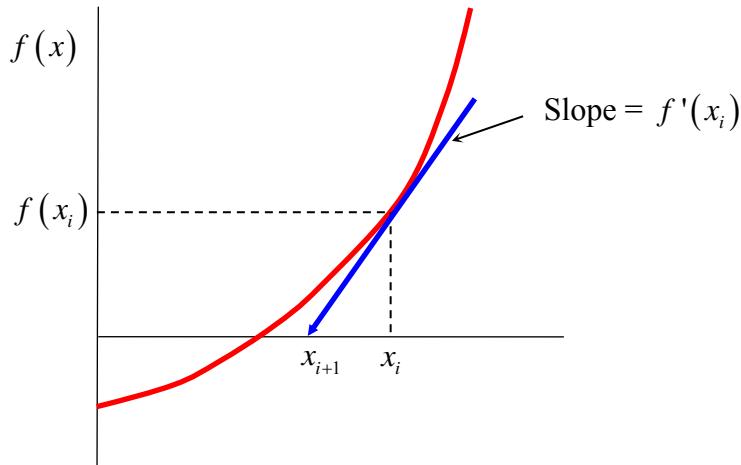
9.3.7: Use either the bisection or the false position method to find the nonzero positive roots for the equation

$$f(\theta) = 1320 \cos^2 \theta - 2640 \sin \theta \cos \theta + 311.6960 \quad (9.3.32)$$

in the interval $[0, 90^\circ]$.

9.4 The Newton-Raphson Method

The *Newton-Raphson Method* is probably the most widely used of all root location formulations. It is named after Sir Isaac Newton and Joseph Raphson⁵. The Newton-Raphson Method is an *open method*. The basis of this method for finding the roots of $f(x) = 0$ is the following graphic:



The iteration is based upon the idea of selecting a point x_i , extending the tangent from the point $(x_i, f(x_i))$ back to where it intersects the x axis, and adopting this intersection as the *next estimate* of the root x_{i+1} .

The geometry in the above figure shows that the point x_{i+1} obeys

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}} \quad (9.4.1)$$

which can be solved, if $f'(x_i) \neq 0$, for x_{i+1} to obtain

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (9.4.2)$$

⁵ Information about Isaac Newton can be found at http://en.wikipedia.org/wiki/Isaac_Newton. Information about Joseph Raphson can be found at http://en.wikipedia.org/wiki/Joseph_Raphson.

Example 9.4.1: You are again given the function given by (9.2.2), repeated,

$$f(x) = \ln(x^5) - x \quad (9.4.3)$$

The plot of this function is given by the first figure of Section 9.2. From our discussions in Sections 9.2 and 9.3, a good estimate of the first root is

$$x_r = 1.295855509095369 \quad (9.4.4)$$

So as to illustrate the Newton-Raphson method, we shall *start* the iteration at $x_1 = 2$. It is elementary to create the following spreadsheet to display the steps in the iteration

f=ln(x^5)-x				
i	x_i	$f(x_i)$	$f'(x_i)$	ε_a
1	2	1.465736	1.5	
2	1.022843	-0.90991	3.888337	95.53%
3	1.256854	-0.1138	2.978187	18.62%
4	1.295064	-0.00226	2.860814	2.95%
5	1.295855	-9.3E-07	2.858456	0.06%
6	1.295856	-1.6E-13	2.858455	0.00%
7	1.295856	0	2.858455	0.00%
8	1.295856	0	2.858455	

A good answer is quickly reached in four iterations.

The convergence rate of the Newton-Raphson is better than the methods we have discussed thus far. It is a theoretical result that

$$(x_r - x_{i+1}) = -\frac{f''(x_i)}{2f'(x_i)}(x_r - x_i)^2 + O((x_r - x_i)^3) \quad (9.4.5)$$

The derivation of (9.4.5) begins with Taylor's Theorem in the form (8.1.11). If we expand the function f about the i^{th} iteration point x_i , it follows from (8.1.11) that

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(x_i)(x - x_i)^2 + O((x - x_i)^3) \quad (9.4.6)$$

If we next evaluate (9.4.6) at the root $x = x_r$, then (9.4.6) yields

$$x_r - \left(x_i - \frac{f(x_i)}{f'(x_i)} \right) = -\frac{1}{2} \frac{f''(x_i)}{f'(x_i)} (x_r - x_i)^2 + O((x_r - x_i)^3) \quad (9.4.7)$$

when $f(x_r) = 0$ and $f'(x_i) \neq 0$ have been used. If we next use (9.4.2), equation (9.4.7) reduces to the result (9.4.5). The importance of (9.4.5) is that the error at the $i+1$ iteration is roughly the square of the error of the previous iteration. This means that the number of correct decimal places approximately doubles with each iteration. This feature is called *quadratic convergence*.

Roughly speaking, one can see from (9.4.5) that convergence problems will arise when $f'(x_i)$ is small, i.e., the curve is too flat, or when $f''(x_i)$, the rate of change of the slope is large. Another way to discuss convergence problems is simply to list some more or less practical issues:⁶

- a) If an inflection point, i.e. where $f''(x) = 0$, occurs somewhere near the root the iteration will not converge.
- b) The iterations can oscillate around a local minimum or maximum, i.e. where $f'(x) = 0$.
- c) An initial guess next to one root can jump to a different root.

The form of the iteration expression

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (9.4.8)$$

suggests that difficulties will arise if the iteration takes one through or near a point of zero slope.

The conclusion is that, unlike the bracketing methods, in advance of the calculation there is *no guarantee of convergence* of the Newton-Raphson method. Its convergence depends on the nature of the function and on the accuracy of the initial guess. There is no fool proof remedy of this problem. Usually, a graph of the function and an initial guess close to the roots is sufficient.

Example 9.4.1: An example that illustrates some of the problems that can arise with the Newton-Raphson method is the problem of finding the roots of

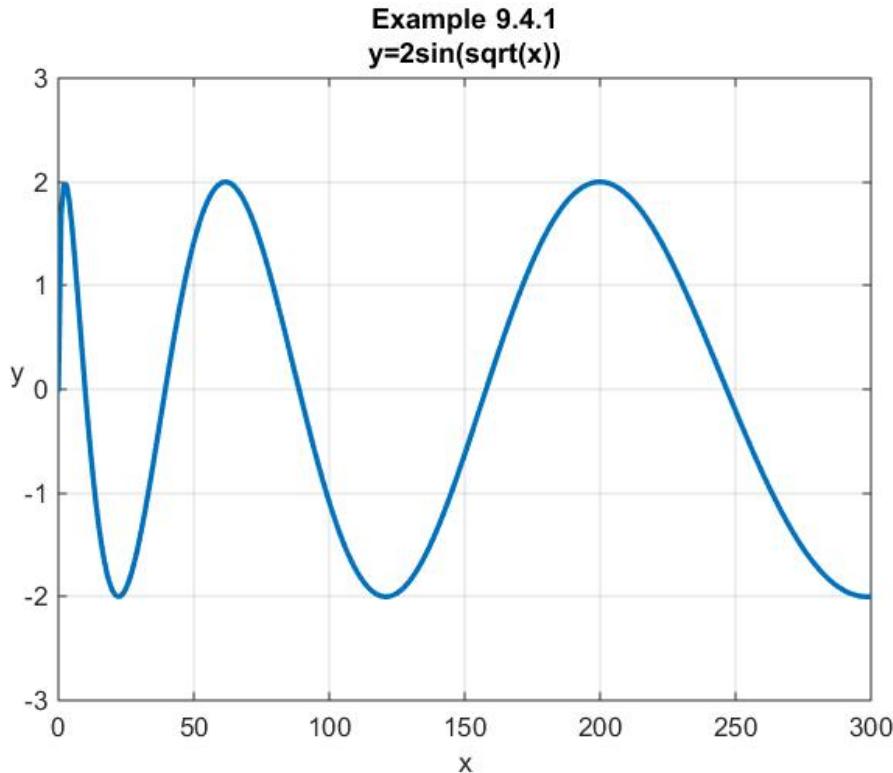
$$f(x) = 2 \sin(\sqrt{x}) \quad (9.4.9)$$

The derivative of this function is

⁶ Another problem with the Newton-Raphson method is that it requires the calculation of the derivative $f'(x)$ at every step of the iteration. A method similar to the Newton-Raphson method but one that avoids the calculation of the derivative goes by the name *Secant Method*.

$$f'(x) = \frac{\cos(\sqrt{x})}{\sqrt{x}} \quad (9.4.10)$$

Equation (9.4.9) has a root at $x = 0$. A plot for $x \in (0, 300)$ shows some of the other roots. This plot is



This plot tells us that between $0 < x < 300$ we have the following *approximate roots*: $(10, 40, 80, 160, 240)$. It is desired to use the Newton-Raphson method to try to find good approximations for these roots. We shall illustrate the points we wish to make by trying to calculate the root near $x = 10$. It is helpful to note that MATLAB's **fzero** command gives

$$x_r = 9.8696 \quad (9.4.11)$$

We shall try to find this root by utilizing the Newton-Rapson method but with *two different starting points*.

Case 1: Start iteration at $x_1 = 10$

The following table shows the iteration to the root in this case $x_1 = 10$.

$$f(x) = 2\sin(\sqrt{x})$$

i	x_i	$f(x_i)$	$f'(x_i)$	ε_a
1	10	-0.04137	-0.31616	
2	9.869158	0.000142	-0.31832	1.33%
3	9.869604	1.61E-09	-0.31831	0.00%
4	9.869604	2.45E-16	-0.31831	0.00%
5	9.869604	2.45E-16	-0.31831	0.00%
6	9.869604	2.45E-16	-0.31831	0.00%
7	9.869604	2.45E-16	-0.31831	0.00%
8	9.869604	2.45E-16	-0.31831	0.00%
9	9.869604	2.45E-16	-0.31831	0.00%
10	9.869604	2.45E-16	-0.31831	0.00%

In this case, a good answer was obtained in one iteration.

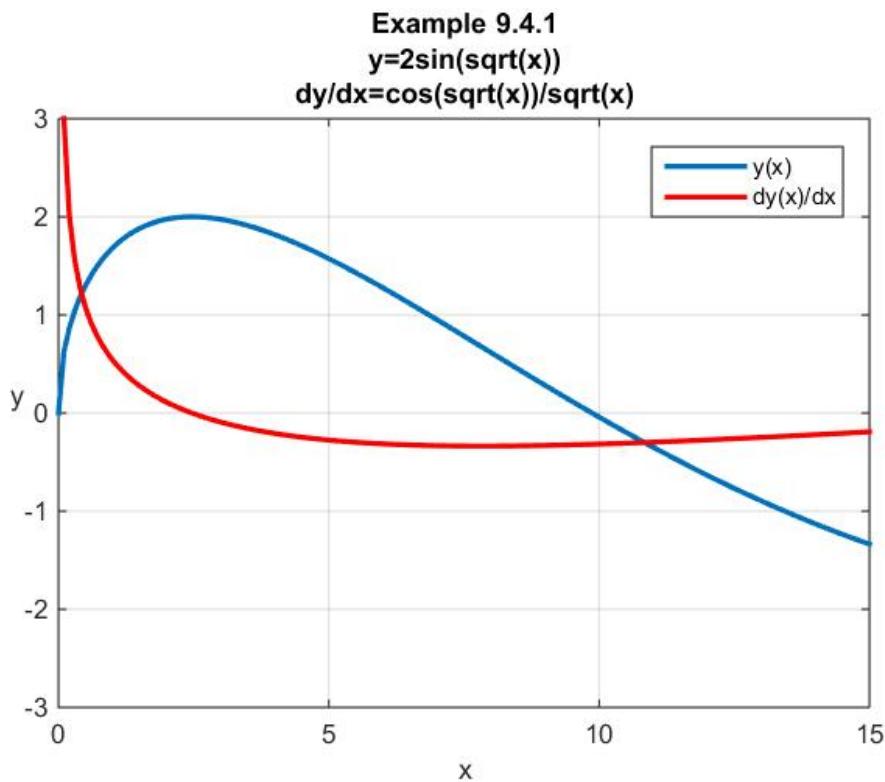
Case 2: Start iteration at $x_1 = 2.5$

For $x_1 = 2.5$ the method jumped past the root near $x = 250$. It then converged back to the root near 80. The table below shows the iteration to this root.

$$f(x) = 2\sin(\sqrt{x})$$

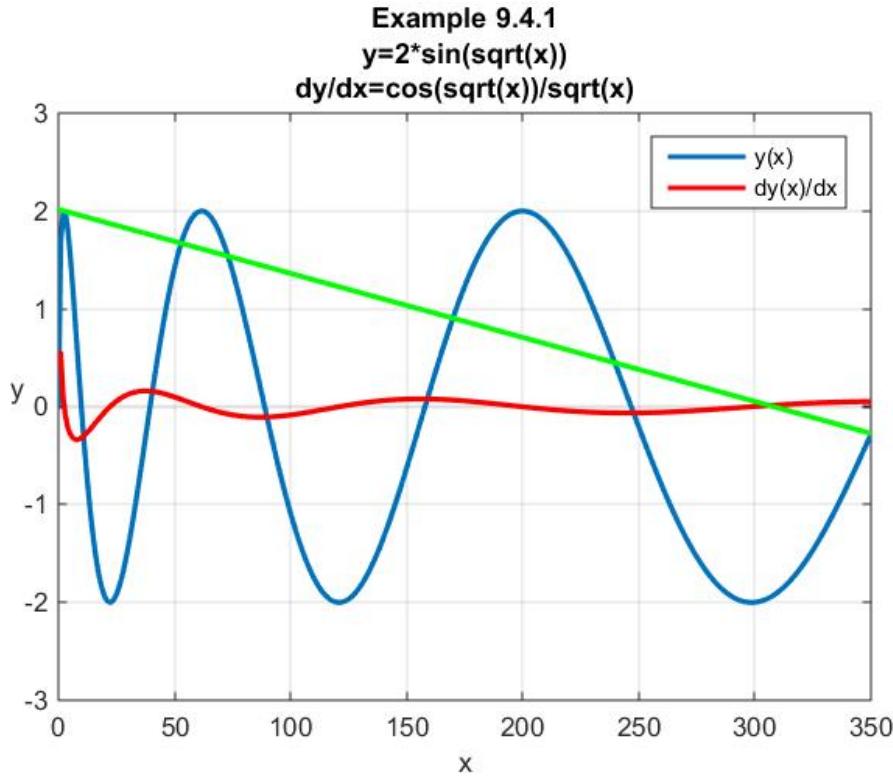
i	x_i	$f(x_i)$	$f'(x_i)$	ε_a
1	2.5	1.999893	-0.00654	
2	308.2446	-1.92314	0.015639	99.19%
3	431.2184	1.881849	-0.01631	28.52%
4	546.6235	-1.96698	-0.00774	21.11%
5	292.4938	-1.969	-0.01026	86.88%
6	100.5021	-1.12977	-0.08231	191.03%
7	86.77638	0.218352	-0.10671	15.82%
8	88.82264	0.000403	-0.10611	2.30%
9	88.82644	4.3E-09	-0.1061	0.00%
10	88.82644	7.35E-16	-0.1061	0.00%
11	88.82644	7.35E-16	-0.1061	0.00%
12	88.82644	7.35E-16	-0.1061	0.00%
13	88.82644	7.35E-16	-0.1061	0.00%
14	88.82644	7.35E-16	-0.1061	0.00%
15	88.82644	7.35E-16	-0.1061	0.00%
16	88.82644	7.35E-16	-0.1061	0.00%
17	88.82644	7.35E-16	-0.1061	0.00%

If we focus on the above plot in the interval $(0, 15)$ and superimpose a plot of (9.4.10), it is possible to see the cause of this jump.



The red line plots the *slope* of the function $f(x) = 2 \sin(\sqrt{x})$. The analytical formula for the slope is given by equation (9.4.10). The slope vanishes at $\sqrt{x} = \frac{\pi}{2}$, i.e. at $x = 2.4676$. This point is very close to the starting point $x_1 = 2.5$. The small value of the slope, from (9.4.2), causes the estimate of the root in the iteration process to be far from the starting point x_1 .

The plot above, but on a different scale, is



The green line displays the slope of the blue line at the point $x_1 = 2.5$. The small slope, in turn, caused the *jump* to the point near $x = 308$. As the above table shows, subsequent iterations gave the root $x_r = 88.6264$.

A Google search will produce a variety of implementations of the Newton-Raphson method or varying sophistication. One that implements a lot of features is the function m-file **newraph.m**.⁷ The script that defines this file is

```
function [root,ea,iter]=newraph(func,dfunc,xr,es,maxit,varargin)
% newraph: Newton-Raphson root location zeros
% [root,ea,iter]=newraph(func,dfunc,xr,es,maxit,p1,p2,...):
%     uses Newton-Raphson method to find the root of func
%input:
%   fun=name of function
%   dfun=name of derivative of function
%   xr=initial guess
%   es=desired relative error (default=0.0001%)
%   maxit=maximum allowable iterations (default=50)
%   p1,p2,...=additional parameters used by function
%output:
%   root=real root
%   ea=approximate relative error (%)
```

⁷ This m-file is the one published on page 149 of the textbook, Applied Numerical Methods with MATLAB, Second Edition, by Steven C. Chapra, McGraw Hill, 2008.

```
% iter=number of iterations

if nargin<3,error('at least 3 input arguments required'),end
if nargin<4|isempty(es),es=0.0001;end
if nargin<5|isempty(maxit),maxit=50;end
iter=0;
while (1)
    xrold=xr;
    %The next line is different from that in the reference.
    %The change seems to be necessary when one is passing
    %parameters through varargin as above
    xr=xr-func(xr,varargin{:})/dfunc(xr,varargin{:});
    iter=iter+1;
    if xr~ =0,ea=abs((xr-xrold)/xr)*100;end
    if ea<=es|iter>=maxit,break,end
end
root=xr;
```

Example 9.4.2: In order to illustrate the function file **newtrap.m**, we shall use it to find the first root of the function defined by equation (9.4.9). This calculation replaces the iteration displayed in the spreadsheet above. The script that achieves this calculation is

```
clc
clear
f=@(x)(2*sin(sqrt(x)))
df=@(x)(cos(sqrt(x))/sqrt(x))
xr=9
[root,ea,iter]=newtrap(f,df,xr)
```

The MATLAB output for this example is

```
root =
9.8696

ea =
5.2454e-005

iter =
3
```

which is consistent with the result in the first table above. If the example is modified by changing the initial point from 9 to 2.5, the result turns out to be

```
root =
88.8264
```

```
ea =  
4.5647e-008
```

```
iter =  
9
```

This result shows again how the method can iterate to the wrong root if the initial point is selected incorrectly.

Exercises

9.4.1: Use the Newton-Raphson method to find the root of

$$y = 2x^3 - 11.7x^2 + 17.7x - 5 \quad (9.4.12)$$

near the point $x = 3.5$.

9.4.2: Use the Newton-Raphson method to find the root of

$$y = \cos x + 10 \sin(1 + x^2) - 1 \quad (9.4.13)$$

near the point $x = 2.4$.

9.5 Systems of Nonlinear Equations

We introduced the discussions in Sections 9.2 through 9.4 by explaining that we were given a function $f : (c, d) \rightarrow \mathbb{R}$, i.e. a *real valued* function of the *real numbers*. Our objective during that discussion was to find the zeros of this function, i.e. the values of x with the property that $f(x) = 0$. In this section, we shall look briefly how the Newton-Raphson method *generalizes* when we are given a set of n functions $\{f_1, f_2, \dots, f_n\}$ of n real numbers (x_1, x_2, \dots, x_n) , we seek the solutions of the n equations

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0 \\ f_2(x_1, x_2, \dots, x_n) &= 0 \\ \cdot & \\ \cdot & \\ \cdot & \\ f_n(x_1, x_2, \dots, x_n) &= 0 \end{aligned} \tag{9.5.1}$$

As in the introduction to this Chapter, we shall write this system as the vector equation

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{9.5.2}$$

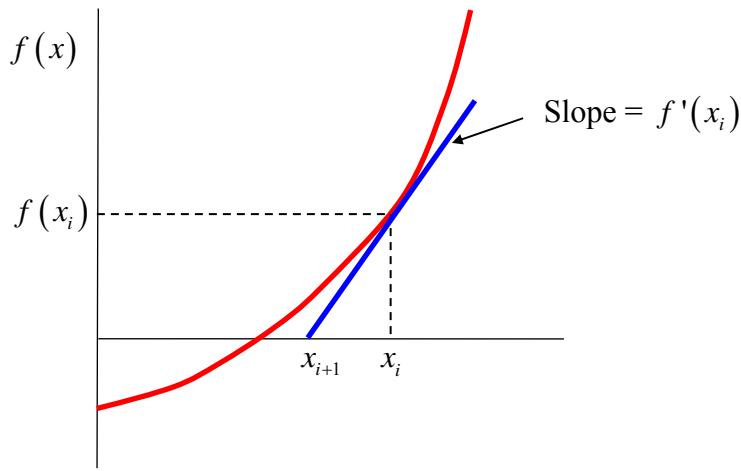
The Newton-Raphson method for vector valued functions like (9.5.2) is a straight forward generalization of the one dimensional version discussed in Section 9.4. The formalism does require that we use some of our experience in multidimensional calculus. We begin by being given a vector valued function of a vector \mathbf{f} and we write

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) \tag{9.5.3}$$

and we are interested in finding those n dimensional vectors \mathbf{x} which satisfy

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{9.5.4}$$

In the one dimensional case, we built our argument around the ability to draw the figure



Because of the vector nature of this problem, a simple two dimensional figure cannot be used. We can, however, use the multidimensional version of Taylor's Theorem and write

$$\mathbf{f}(\mathbf{x}^{(j+1)}) = \mathbf{f}(\mathbf{x}^{(j)}) + (\text{grad } \mathbf{f}(\mathbf{x}^{(j)}))(\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}) + O\left(\|\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}\|^2\right) \quad (9.5.5)$$

where $\mathbf{x}^{(j+1)}$ and $\mathbf{x}^{(j)}$ are two vectors in the iteration we are trying to relate. The notation in the last equation is a problem and can stand in your way to understanding. The vectors that appear are *not* a problem, but the quantity $\text{grad } \mathbf{f}(\mathbf{x}^{(j)})$ needs to be defined utilizing notation that we have introduced thus far. The best way, at this point, is to explain that it is an $n \times n$ matrix. The explicit form of the matrix is

$$\text{grad } \mathbf{f} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \dots & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \dots & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & & & \ddots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \dots & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \quad (9.5.6)$$

The combination of terms $(\text{grad } \mathbf{f}(\mathbf{x}^{(j)}))(\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)})$, which appears in Taylor's Theorem is just the matrix produce of the matrix $\text{grad } \mathbf{f}(\mathbf{x}^{(j)})$ and the column vector $\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}$. Therefore, the component version of the product $(\text{grad } \mathbf{f}(\mathbf{x}^{(j)}))(\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)})$ is

$$\begin{aligned}
 (\text{grad } \mathbf{f})(\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}) &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \begin{bmatrix} x_1^{(j+1)} - x_1^{(j)} \\ x_2^{(j+1)} - x_2^{(j)} \\ \vdots \\ x_n^{(j+1)} - x_n^{(j)} \end{bmatrix} \\
 &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x_1^{(j+1)} - x_1^{(j)}) + \frac{\partial f_1}{\partial x_2}(x_2^{(j+1)} - x_2^{(j)}) + \dots + \frac{\partial f_1}{\partial x_n}(x_n^{(j+1)} - x_n^{(j)}) \\ \frac{\partial f_2}{\partial x_1}(x_1^{(j+1)} - x_1^{(j)}) + \frac{\partial f_2}{\partial x_2}(x_2^{(j+1)} - x_2^{(j)}) + \dots + \frac{\partial f_2}{\partial x_n}(x_n^{(j+1)} - x_n^{(j)}) \\ \vdots \\ \frac{\partial f_n}{\partial x_1}(x_1^{(j+1)} - x_1^{(j)}) + \frac{\partial f_n}{\partial x_2}(x_2^{(j+1)} - x_2^{(j)}) + \dots + \frac{\partial f_n}{\partial x_n}(x_n^{(j+1)} - x_n^{(j)}) \end{bmatrix} \quad (9.5.7)
 \end{aligned}$$

Returning to (9.5.5), repeated,

$$\mathbf{f}(\mathbf{x}^{(j+1)}) = \mathbf{f}(\mathbf{x}^{(j)}) + (\text{grad } \mathbf{f}(\mathbf{x}^{(j)}))(\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}) + O\left(\|\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}\|^2\right) \quad (9.5.8)$$

the Newton-Raphson iteration arises by our agreeing to calculate the point $\mathbf{x}^{(j+1)}$ by the formula

$$\mathbf{f}(\mathbf{x}^{(j)}) + (\text{grad } \mathbf{f}(\mathbf{x}^{(j)}))(\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}) = \mathbf{0} \quad (9.5.9)$$

When we were discussing the one dimensional iteration condition, i.e., (9.4.8), repeated,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (9.5.10)$$

arose by our forcing the relationship

$$f(x_i) + f'(x_i)(x_{i+1} - x_i) = 0 \quad (9.5.11)$$

Thus, the condition (9.5.9) achieves in n dimensions what (9.5.11) achieves in one dimension. Just as the one dimensional relationship is only useful (i.e. actually determines the point x_{i+1}) when $f'(x_i)$ is nonzero, the n dimensional relationship is only useful (i.e. actually determines the point $\mathbf{x}^{(j+1)}$) when the matrix $\text{grad } \mathbf{f}(\mathbf{x}^{(j+1)})$ is nonsingular. In this case, the point $\mathbf{x}^{(j+1)}$ is determined by

$$\mathbf{x}^{(j+1)} = \mathbf{x}^{(j)} - \left(\text{grad } \mathbf{f}(\mathbf{x}^{(j)}) \right)^{-1} \mathbf{f}(\mathbf{x}^{(j)}) \quad (9.5.12)$$

This formula, which plays for n dimensions what the formula $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ plays for one dimension, defines the basic iteration scheme one uses in order to find the roots to the vector equation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

The formalism becomes a little more transparent if we look at the case $n = 2$. We can specialize to this case by writing

$$\mathbf{x} = (x_1, x_2) \quad (9.5.13)$$

and

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} \quad (9.5.14)$$

It then follows from (9.5.6) that

$$\text{grad } \mathbf{f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} \quad (9.5.15)$$

In this special case, we can be more explicit about the calculation of the inverse $(\text{grad } \mathbf{f}(\mathbf{x}))^{-1}$. It follows from (1.10.51) that the formula for this inverse is

$$(\text{grad } \mathbf{f}(\mathbf{x}))^{-1} = \frac{1}{J} \begin{bmatrix} \frac{\partial f_2}{\partial x_2} & -\frac{\partial f_1}{\partial x_2} \\ -\frac{\partial f_2}{\partial x_1} & \frac{\partial f_1}{\partial x_1} \end{bmatrix} \quad (9.5.16)$$

where the symbol J is the *Jacobian* determinant defined by

$$J = \begin{vmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{vmatrix} = \frac{\partial f_1}{\partial x_1} \frac{\partial f_2}{\partial x_2} - \frac{\partial f_1}{\partial x_2} \frac{\partial f_2}{\partial x_1} \quad (9.5.17)$$

If we combine the above formulas, the iteration condition $\mathbf{x}^{(j+1)} = \mathbf{x}^{(j)} - (\text{grad } \mathbf{f}(\mathbf{x}^{(j)}))^{-1} \mathbf{f}(\mathbf{x}^{(j)})$ reduces in two dimensions to the matrix equation

$$\begin{bmatrix} x_1^{(j+1)} \\ x_2^{(j+1)} \end{bmatrix} = \begin{bmatrix} x_1^{(j)} \\ x_2^{(j)} \end{bmatrix} - \frac{1}{J(x_1^{(j)}, x_2^{(j)})} \begin{bmatrix} \frac{\partial f_2(x_1^{(j)}, x_2^{(j)})}{\partial x_2} & -\frac{\partial f_1(x_1^{(j)}, x_2^{(j)})}{\partial x_2} \\ -\frac{\partial f_2(x_1^{(j)}, x_2^{(j)})}{\partial x_1} & \frac{\partial f_1(x_1^{(j)}, x_2^{(j)})}{\partial x_1} \end{bmatrix} \begin{bmatrix} f_1(x_1^{(j)}, x_2^{(j)}) \\ f_2(x_1^{(j)}, x_2^{(j)}) \end{bmatrix} \quad (9.5.18)$$

Example 9.5.1: You are asked to find the roots of the two nonlinear equations⁸

$$x_2 = \cos(x_1) \quad (9.5.19)$$

and

⁸ This example is special in that the two functions (9.5.19) and (9.5.20) can be combined to yield $\cos x_1 = \frac{\sin(x_1)}{\sqrt{x_1}}$.

The values of x_1 that satisfy this equation are the zeros of $f(x_1) = \cos x_1 - \frac{\sin(x_1)}{\sqrt{x_1}}$ and, as such can be calculated

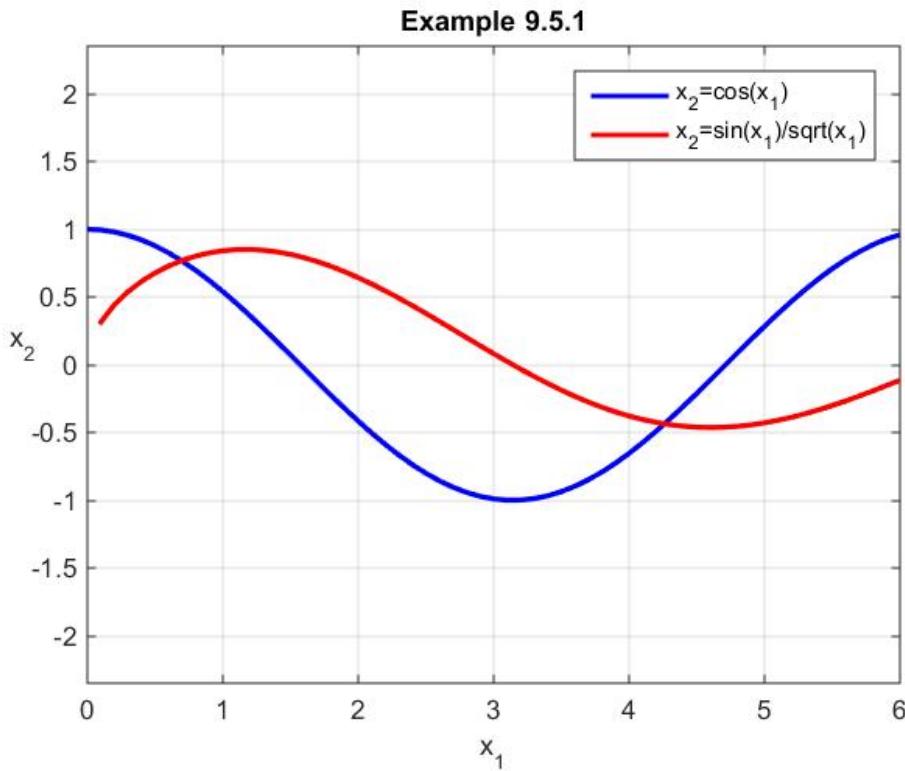
by the methods introduced in Sections 9.2 through 9.4. In spite of the simplicity of Example 9.5.1, it does illustrate the Newton-Raphson method for systems of nonlinear equations.

$$x_2 = \frac{\sin(x_1)}{\sqrt{x_1}} \quad (9.5.20)$$

in the region $x \in (0, 6)$. The roots we seek are the points where the curves (9.5.19) and (9.5.20) intersect. The following MatLab script yields the plot of these two curves superimposed on the same set of axes.

```
clc
clear
x=[0:.1:6]
y=cos(x);
z=sin(x)./sqrt(x);
plot(x,y,'LineWidth',2,'Color','b');
hold on;
plot(x,z,'LineWidth',2,'Color','r')
xlabel('x_1');
ylabel('x_2');
legend('x_2=cos(x_1)', 'x_2=sin(x_1)/sqrt(x_1)')
title('Example 9.3.1')
grid on;
axis equal;
```

The resulting plot is as follows



Therefore, there are two roots. One is approximately at the point $(x_1, x_2) = (.75, .75)$ and the other at approximately $(x_1, x_2) = (4.5, -.4)$. These values will be used to suggest the starting place for the two dimensional Newton-Raphson iteration. We shall identify the functions $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$ in the iteration formula by

$$f_1(x_1, x_2) = x_2 - \cos(x_1) \quad (9.5.21)$$

and

$$f_2(x_1, x_2) = x_2 - \frac{\sin(x_1)}{\sqrt{x_1}} \quad (9.5.22)$$

The derivatives that appear in the iteration formula are given by

$$\begin{aligned}\frac{\partial f_1}{\partial x_1} &= \sin(x_1) & \frac{\partial f_1}{\partial x_2} &= 1 \\ \frac{\partial f_2}{\partial x_1} &= -\frac{1}{x_1^2} \cos(x_1) + \frac{1}{2x_1^2} \sin(x_1) & \frac{\partial f_2}{\partial x_2} &= 1\end{aligned}\quad (9.5.23)$$

If we substitute into the formula for J the result is

$$J = \frac{\partial f_1}{\partial x_1} \frac{\partial f_2}{\partial x_2} - \frac{\partial f_1}{\partial x_2} \frac{\partial f_2}{\partial x_1} = \sin(x_1) + \frac{1}{x_1^2} \cos(x_1) - \frac{1}{2x_1^2} \sin(x_1) \quad (9.5.24)$$

Just as the one dimensional Newton-Raphson method has problems when the derivative $f'(x)$ vanishes, one can anticipate problems if J vanishes or becomes small during the iteration.

The last figure above displays a root near $(x_1, x_2) = (4.5, -4)$. In order to illustrate the calculation, we shall start the iteration at

$$\begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \end{bmatrix} \quad (9.5.25)$$

and attempt to find the root near that point. The iteration formula we must utilize is (9.5.18), repeated,

$$\begin{bmatrix} x_1^{(j+1)} \\ x_2^{(j+1)} \end{bmatrix} = \begin{bmatrix} x_1^{(j)} \\ x_2^{(j)} \end{bmatrix} - \frac{1}{J(x_1^{(j)}, x_2^{(j)})} \begin{bmatrix} \frac{\partial f_2(x_1^{(j)}, x_2^{(j)})}{\partial x_2} & -\frac{\partial f_1(x_1^{(j)}, x_2^{(j)})}{\partial x_2} \\ -\frac{\partial f_2(x_1^{(j)}, x_2^{(j)})}{\partial x_1} & \frac{\partial f_1(x_1^{(j)}, x_2^{(j)})}{\partial x_1} \end{bmatrix} \begin{bmatrix} f_1(x_1^{(j)}, x_2^{(j)}) \\ f_2(x_1^{(j)}, x_2^{(j)}) \end{bmatrix} \quad (9.5.26)$$

If we implement the iteration through a spreadsheet, the following table is obtained:

Root Near (4,0)

n	x_1^0	x_2^0	f_1^0	f_2^0	$D_{x1}f_1^J$	$D_{x2}f_1^J$	$D_{x1}f_2^J$	$D_{x2}f_2^J$	J
1	4	0	0.653644	0.378401	-0.7568	1	0.279522	1	-1.03632
2	4.265595	-0.45264	-0.02056	-0.01599	-0.90184	1	0.158021	1	-1.05986
3	4.261275	-0.43597	-4E-06	-4.4E-06	-0.89996	1	0.160041	1	-1.06
4	4.261276	-0.43597	-2E-14	-2.1E-14	-0.89996	1	0.160041	1	-1.06
5	4.261276	-0.43597	0	0	-0.89996	1	0.160041	1	-1.06
6	4.261276	-0.43597	0	0	-0.89996	1	0.160041	1	-1.06
7	4.261276	-0.43597	0	0	-0.89996	1	0.160041	1	-1.06

Therefore, the root near $(4,0)$ is approximated by $(4.261276, -0.43597)$ in six iterations. The figure above shows that the second root we seek is near $(x_1, x_2) = (4.5, -4)$. We shall calculate this root by starting the calculation at

$$\begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (9.5.27)$$

The iteration in this case yields the following table

Root Near (1,0)

n	x_1^0	x_2^0	f_1^0	f_2^0	$D_{x1}f_1^J$	$D_{x2}f_1^J$	$D_{x1}f_2^J$	$D_{x2}f_2^J$	J
1	1	0	-0.5403	-0.84147	0.841471	1	-0.11957	1	0.961038
2	0.686621	0.804001	0.030609	0.038967	0.633928	1	-0.37624	1	1.01017
3	0.694895	0.768148	2.64E-05	3.1E-05	0.640305	1	-0.36876	1	1.009066
4	0.694899	0.768118	7.88E-12	9.25E-12	0.640308	1	-0.36876	1	1.009065
5	0.694899	0.768118	0	0	0.640308	1	-0.36876	1	1.009065
6	0.694899	0.768118	0	0	0.640308	1	-0.36876	1	1.009065
7	0.694899	0.768118	0	0	0.640308	1	-0.36876	1	1.009065

Therefore, the root near $(1,0)$ is approximated by $(0.694899, 0.768118)$ in six iterations.

As illustrated in the above example the Newton-Raphson method for systems of nonlinear equations can be implemented by utilizing a spreadsheet to display the iteration. However, in order to enlarge our collection of MATLAB tools, we shall try to find the root by utilizing the MATLAB command **fsolve**. This command is MATLAB's tool to solve systems of nonlinear equations. The syntax of this command can be complicated and is best learned from using MATLAB's Help utility. For our limited purposes, the syntax is simply

$$\mathbf{x} = \mathbf{fsolve}(\mathbf{fun}, \mathbf{x0}) \quad (9.5.28)$$

The command as written in (9.5.28) starts at **x0** and tries to solve the equations described in **fun**. Our Example 9.5.1 is solved by the MATLAB script

```

clc
clear
%Enter the two equations as an autonomous function with
%two components
f=@(x)([x(2)-cos(x(1)),x(2)-sin(x(1))/sqrt(x(1))]);
%For the root near [1,0]
x1=fsolve(f,[1,0])
%For the root near [4,0]
x2=fsolve(f,[4,0])

```

yields the output

Equation solved.

`fsolve` completed because the vector of function values is near zero as measured by the default value of the function tolerance, and the problem appears regular as measured by the gradient.

<stopping criteria details>

`x1 =`

0.6949 0.7681

Equation solved.

`fsolve` completed because the vector of function values is near zero as measured by the default value of the function tolerance, and the problem appears regular as measured by the gradient.

<stopping criteria details>

`x2 =`

4.2613 -0.4360

These answers are consistent with the spreadsheet based iteration schemes above.

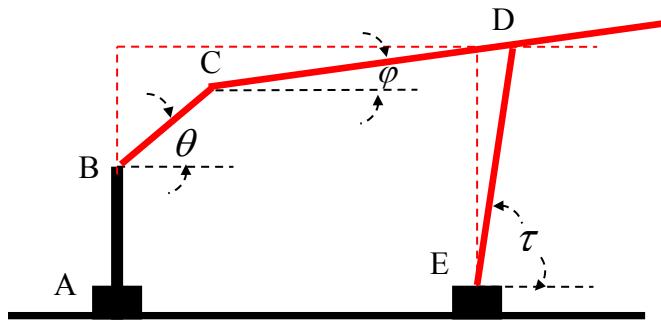
Example 9.5.2: In this example, we shall to use `fsolve` and find the solutions of the two nonlinear equations

$$3\cos\varphi - 2\cos\tau = 3 - \cos\theta \quad (9.5.29)$$

and

$$3\sin\varphi - 2\sin\tau = -(1 + \sin\theta) \quad (9.5.30)$$

for the angles φ and τ for *given* values of the angle θ . The origin of these two equations is the kinematic problem of determining the motion of a linkage like that shown in the following figure:



The bar BC spins about the point B and the bars CD and DE, that are hinged at C and D and E, respectively, move in a way determined by equations (9.5.29) and (9.5.30). While not essential for our discussion, the lengths in the figure that produce the coefficients of (9.5.29) and (9.5.30) are AB=BC=.5m, CD=1m and DE =1m.

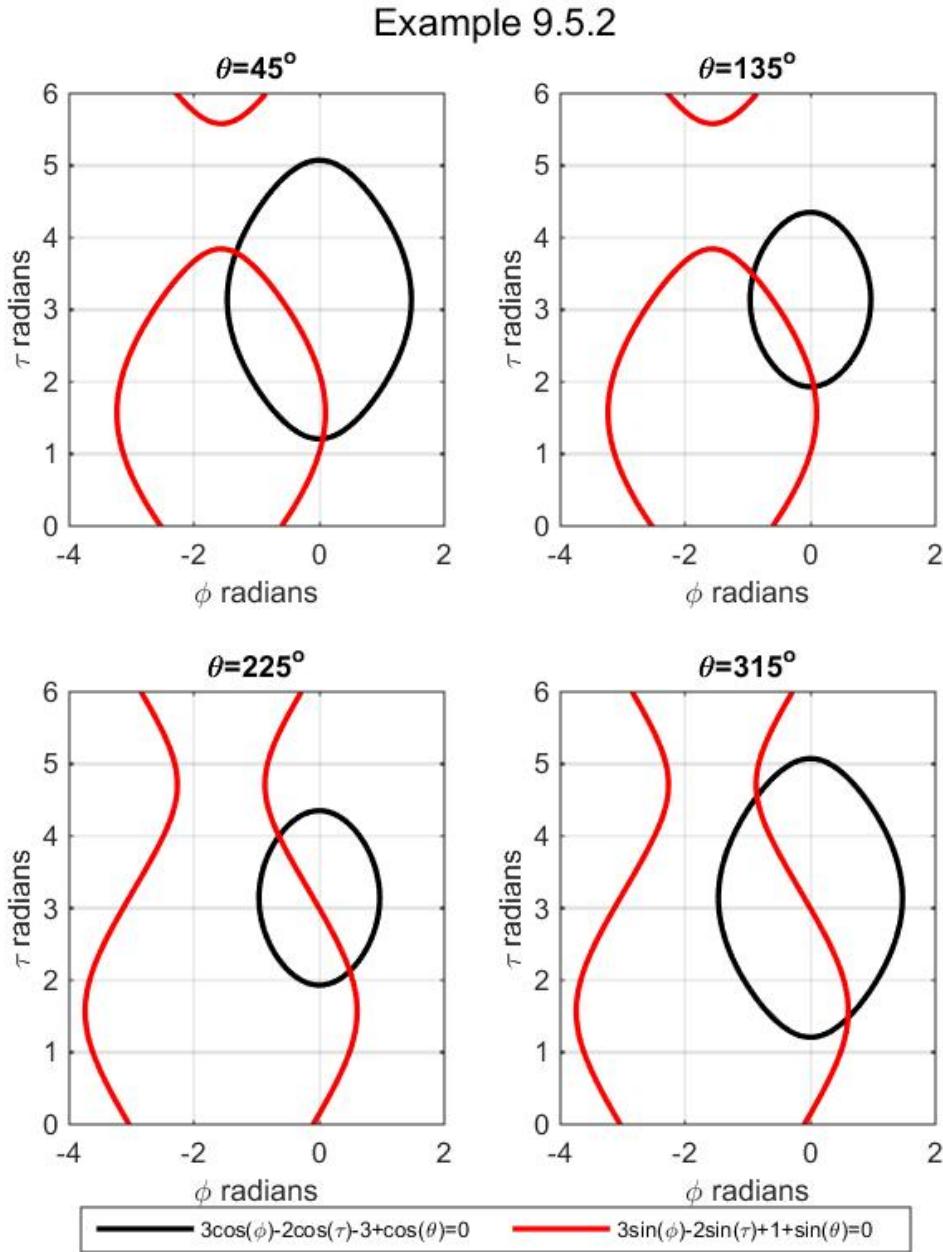
In this example, we shall calculate the values of the angles φ and τ for θ ranging from 0 to 360° in increments of 30° . As with all problems involving the determination of roots of nonlinear equations, it is helpful to plot the equations in order to obtain an estimate of the roots. In this case, we need to plot the two equations

$$f_1(\varphi, \tau) = 3\cos\varphi - 2\cos\tau - 3 + \cos\theta = 0 \quad (9.5.31)$$

and

$$f_2(\varphi, \tau) = 3\sin\varphi - 2\sin\tau + (1 + \sin\theta) = 0 \quad (9.5.32)$$

for various values of θ . The intersections of these two curves represent the roots we seek. In order to get a feel for the location of the roots, the plots of (9.5.31) and (9.5.32) for four values of θ are⁹



These figures suggest two roots for each given value of θ . For the four choices of θ , used in the figure, the two sets of roots are, roughly speaking, in the intervals $(\phi, \tau) \subset [0, 1] \times [1, 2]$ and $(\phi, \tau) \subset [-2, 0] \times [3, 5]$.¹⁰

⁹ The MATLAB plot command used to create the above figure is **ezplot**. This command is useful when the function to be plotted is given implicitly. Also, as with earlier examples, the **subplot** structure explained in the appendix was used to create the four plot display.

For the first set of roots, the simple script

```

clc
clear
phi=zeros(13,1); % Preallocate
tau=zeros(13,1); % Preallocate
for n=1:13
    theta=(n-1)*pi/6
%Enter the two equations as an autonomous function with
%two components
f=@(x)([3*cos(x(1))-2*cos(x(2))-3+cos(theta),...
    3*sin(x(1))-2*sin(x(2))+1+sin(theta)]);
%For the root near [0,1]
x1=fsolve(f,[0,1])
phi(n)=x1(1)
tau(n)=x1(2)
end

phidegrees=phi*180/pi
taudegrees=tau*180/pi
thetadegrees=30*[0:12]'10
answer=[thetadegrees,phidegrees,taudegrees]

```

yields the output, in degrees,

```

answer =

```

0	15.2453	63.4349
30.0000	5.9393	64.8518
60.0000	1.3500	75.5471
90.0000	0.0000	90.0000
120.0000	1.3419	104.5019
150.0000	5.6564	116.1241
180.0000	13.1787	122.6499
210.0000	22.9215	123.4668
240.0000	32.5544	119.0569
270.0000	38.9424	109.4712
300.0000	38.3317	94.2064
330.0000	28.6107	75.5312
360.0000	15.2453	63.4349

An identical calculation yields

¹⁰ The notation $[0,1] \times [1,2]$ designates the *Cartesian product* of two sets. This concept was introduced in Section 2.1.

answer =

0	-68.3754	243.4349
30.0000	-76.1471	224.9404
60.0000	-74.8261	210.9768
90.0000	-67.3801	202.6199
120.0000	-57.4705	199.3695
150.0000	-48.0684	201.4639
180.0000	-41.2512	209.2776
210.0000	-37.6601	221.7947
240.0000	-36.9386	236.5588
270.0000	-38.9424	250.5288
300.0000	-44.4667	259.6585
330.0000	-54.9843	258.0952
360.0000	-68.3754	243.4349

for the second set of roots. In the above two tables, the first column corresponds to the input angle, θ , while the second and third columns correspond to φ and τ , respectively. These tables display the answers in degrees.

Exercises

9.5.1: Utilize **fsolve** to find the solution of the simultaneous nonlinear equations

$$\begin{aligned}x^2 + y^2 - 5 &= 0 \\y + 1 - x^2 &= 0\end{aligned}\tag{9.5.33}$$

9.6 Polynomials

In Section 2.1, we gave as an example of a vector space the set \mathcal{P}_N of all polynomials p of degree equal to or less than N . Elements of \mathcal{P}_N were defined, for all $x \in \mathbb{C}$, by

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_Nx^N \quad (9.6.1)$$

where $a_0, a_1, a_2, \dots, a_N \in \mathbb{R}$. The zero polynomial in \mathcal{P}_N was given the symbol 0 and was defined by

$$0(x) = 0 + 0x + 0x^2 + \cdots + 0x^N \quad (9.6.2)$$

The vector space operations of addition and scalar multiplication were defined in the usual way, i.e., by

$$(p_1 + p_2)(x) = p_1(x) + p_2(x) \text{ for all } x \in \mathbb{R} \quad (9.6.3)$$

and

$$(\lambda p)(x) = \lambda p(x) \quad \text{for all } x \in \mathbb{R} \quad (9.6.4)$$

It was pointed out in Example 2.5.4 that \mathcal{P}_N is a finite dimensional vector space of dimension $N+1$.

A common problem in the applications is to find the roots of polynomials. MATLAB has several built in functions that will calculate these roots as well as perform other important polynomial related manipulations. In this section, we shall discuss utilizing MATLAB to calculate the roots of a polynomial. In addition, we shall discuss several of the other polynomial related functions available in MATLAB.

MATLAB exploits the isomorphism between the $N+1$ dimensional vector space \mathcal{P}_N and the $N+1$ dimensional vector space $\mathbb{M}^{1 \times (N+1)}$ in a manner to be discussed next. If $p \in \mathcal{P}_N$, then it has the representation (9.6.1). Given, (9.6.1), we define a row vector in $P \in \mathbb{M}^{1 \times (N+1)}$ by the formula

$$P = [a_N, a_{N-1}, \dots, a_1, a_0] \quad (9.6.5)$$

As a practical matter when converting from polynomials to row vectors, it is often convenient to write the polynomial in \mathcal{P}_N as by the contention used by MATLAB, namely,

$$p = p_1x^N + p_2x^{N-1} + \cdots + p_{N-2}x^3 + p_{N-1}x^2 + p_Nx + p_{N+1} \quad (9.6.6)$$

which allows the associated column vector $P \in \mathcal{M}^{1 \times (N+1)}$ to be written

$$P = [p_1, p_2, p_3, \dots, p_{N+1}] \quad (9.6.7)$$

Often when we wish to emphasize the dependence of the a polynomial on the argument x , we will write (9.6.6) as

$$f(x) = p_1x^N + p_2x^{N-1} + \cdots + p_{N-2}x^3 + p_{N-1}x^2 + p_Nx + p_{N+1} \quad (9.6.8)$$

Example 9.6.1: The polynomial

$$f(x) = x^{12} - 1 \quad (9.6.9)$$

is one we used in Section 9.3. This polynomial defines the 1×13 row vector

$$P = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1] \quad (9.6.10)$$

Example 9.6.2: The polynomial

$$f(x) = x^6 - 4x^4 + 9x^2 - x + 19 \quad (9.6.11)$$

defines the 1×7 row vector

$$P = [1, 0, -4, 0, 9, -1, 19] \quad (9.6.12)$$

The following table gives a *partial list* of several MATLAB commands for manipulating polynomials. It would be advisable to read more information about these command by utilizing the MATLAB HELP function.

roots(P)	Computes the roots of a polynomial specified by P . Output is a column vector. Unlike the methods we have discussed thus far, this command yields all of the roots, real and complex.
polyval(P,x)	Evaluates a polynomial specified by P at the point x
Polyvalm(P,A)	Evaluates a polynomial specified by P at the <i>square</i> matrix A
poly(r)	Computes the coefficients of a polynomial whose roots are given by the row vector r . Output is a row vector.
conv(P1,P2)	Computes the product of the polynomials defined by P1 and P2 .
[Q,R]=deconv(P1,P2)	Computes the result of dividing a numerator polynomial defined by P1 by a denominator polynomial defined by P2 . The output is the quotient polynomial defined by Q and the remainder polynomial defined by R .
K=Polyder(P)	Derivative of a polynomial P . The output is the polynomial represented by the row matrix K

A polynomial $p \in \mathcal{P}_N$ can have roots that are complex numbers. It will have N roots, not necessarily distinct. Because the polynomials in this discussion have real coefficients, it is elementary to establish that

$$\begin{aligned} & \overline{(p_1x^N + p_2x^{N-1} + \cdots + p_{N-2}x^3 + p_{N-1}x^2 + p_Nx + p_{N+1})} \\ &= p_1\bar{x}^N + p_2\bar{x}^{N-1} + \cdots + p_{N-2}\bar{x}^3 + p_{N-1}\bar{x}^2 + p_N\bar{x} + p_{N+1} \end{aligned} \quad (9.6.13)$$

Thus, if a polynomial has a complex root, its complex conjugate is also a root.

Example 9.6.3: Find the roots of (9.6.9). In Example 9.3.2, we used the function (9.6.9) to illustrate an example where the false position method was less efficient than the bisection method. In any case, the row matrix that defines the polynomial is given by (9.6.10)

The MATLAB syntax

```
clc
clear
P=[1,0,0,0,0,0,0,0,0,0,0,-1]
roots(P)
```

yields the twelve roots

```
ans =
-1.0000
-0.8660 + 0.5000i
-0.8660 - 0.5000i
-0.5000 + 0.8660i
-0.5000 - 0.8660i
-0.0000 + 1.0000i
-0.0000 - 1.0000i
0.5000 + 0.8660i
0.5000 - 0.8660i
1.0000
0.8660 + 0.5000i
0.8660 - 0.5000i
```

Example 9.6.4: Find the roots of

$$f(x) = x^6 - 4x^4 + 9x^2 - x + 19 \quad (9.6.14)$$

where, from, (9.6.12),

$$P = [1, 0, -4, 0, 9, -1, 19] \quad (9.6.15)$$

The MATLAB syntax

```
clc
clear
P=[1,0,-4,0,9,-1,19]
x=roots(P)
```

yields the output

```
x =
-1.8195 + 0.8242i
-1.8195 - 0.8242i
1.7981 + 0.7961i
```

```
1.7981 - 0.7961i
0.0214 + 1.1095i
0.0214 - 1.1095i
```

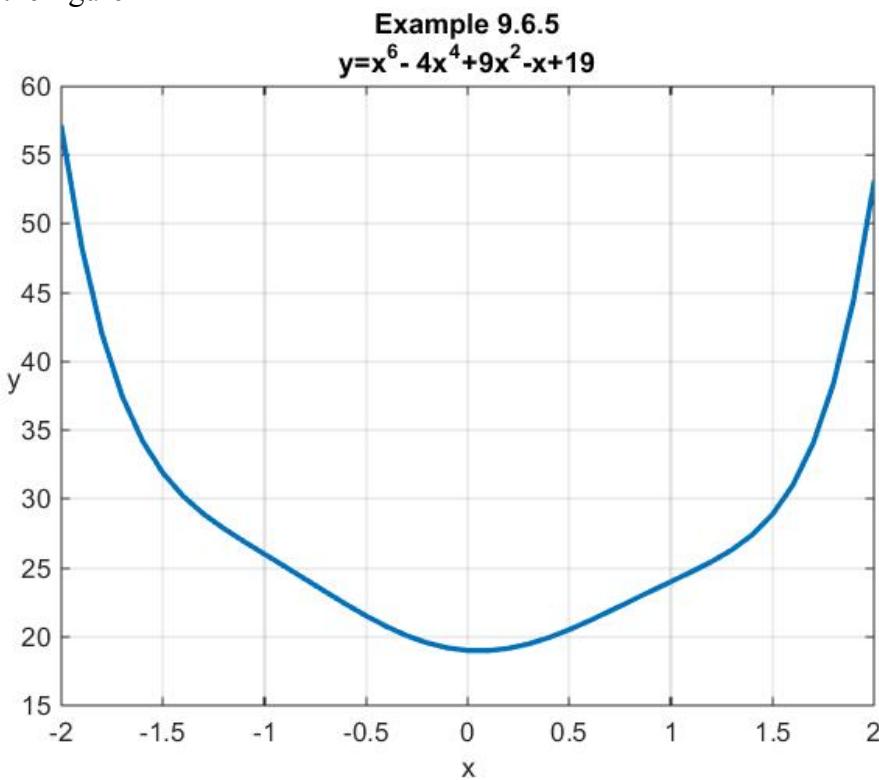
for the column vector of roots. Note that the roots, as is typical, are real and complex. As explained above with (9.6.13), these examples illustrate the fact that the the complex roots of polynomials always occur in complex conjugate pairs.

If we wanted to *plot* the polynomial (9.6.14), we would use the **polyval** command.

Example 9.6.5: As an illustration of **polyval**, in this example we shall plot the polynomial (9.6.14) over the interval $[-10,10]$. This plot is achieved by the MATLAB syntax

```
clc
clear
P=[1,0,-4,0,9,-1,19]
x=[-2:.1:2]
y=polyval(P,x)
plot(x,y,'LineWidth',2)
grid on
xlabel('x'),ylabel('y','Rotation',0)
title({'Exercise 9.6.5','y=x^6- 4x^4+9x^2-x+19'})
```

The output is the figure



Exercises

9.6.1: Utilize MATLAB to find the five roots of

$$f(x) = x^5 + 19x^3 - 122x^2 + 296x + 192 \quad (9.6.16)$$

Chapter 10

REGRESSION

In this chapter, we shall cover again certain of the aspects of regression problems. One purpose is to add a small amount of depth to that which has already been covered in Chapter 4. Unlike Chapter 4, we shall make use of special features within MATLAB to solve regression problems.

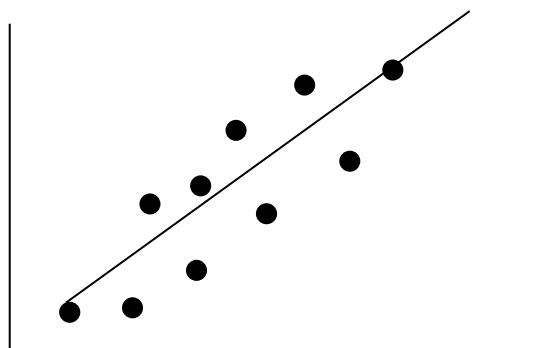
Section 10.1. Least Squares Problems and Overdetermined Systems

In this section, we shall return to a discussion of regression problems that were first discussed in Sections 4.13, 4.14 and 4.15. In particular, we shall utilize the tools within MATLAB to solve least squares problems. As explained in Section 4.14, regression problems are approximate solutions to an algebraic problem. The basic idea utilized to solve regression problems is to formulate the approximation in the form of a least squares problem. As in Section 4.14, this problem is formulated as follows. If we are given a linear transformation $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{U}$ and a vector $\mathbf{b} \in \mathcal{U}$, the *consistency theorem* for linear systems says that the system

$$\mathbf{Ax} = \mathbf{b} \tag{10.1.1}$$

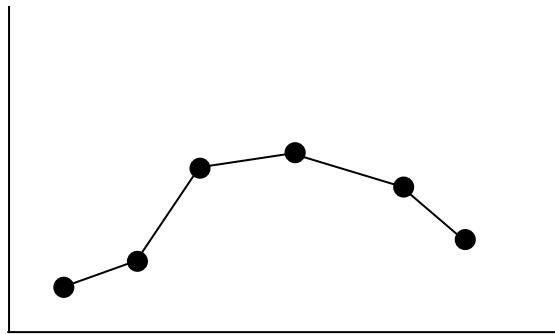
has a solution if and only if $\mathbf{b} \in R(\mathbf{A})$. This theorem was discussed in Sections 1.8 and 2.7 for matrix systems. It was mentioned again in Sections 3.3 and 4.14. As explained in Section 4.14, for *regression* problems we encounter the problem of finding an *approximate* solution of (10.1.1) when $\mathbf{b} \notin R(\mathbf{A})$.

For a *regression problem* the data exhibits a significant degree of error or scatter as shown in the following figure.



- a. The objective is to derive a single curve that represents the general *trend* of the data.
- b. The derivation makes no effort to find a curve that intersects the given points.
- c. The curve is designed to follow the pattern of points taken as a group.
- d. The approach is to try to pass a curve through the data that *minimizes error* in some fashion.

In Chapter 4 we explained an *interpolation problem*. For this kind of problem the data is known to be precise as shown in the following figure.



- a. The approach is to fit a curve or series of curves that *pass directly through* each of the points.
- b. The objective is to use the constructed curve to obtain an estimation of values between well-known discrete points

while regression problems exploit the least squared approximation, interpolation problems are framed as finding the solution to equations that, while approximate representations of fact, indeed have solutions. We shall discuss interpolation in Chapter 11. Our focus in this Chapter is regression.

As illustrated in Section 4.15, we arrive at an equation of the form (10.1.1) when we attempt to use a polynomial to produce the curve fit in the regression problem described above. We shall continue to write the polynomial as

$$y(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_s x^s \quad (10.1.2)$$

Viewed as a member of a vector space, the polynomial (10.1.2) is in the vector space \mathcal{P}_s introduced in Section 2.1. Our objective is to utilize the least squares procedure to determine the $S + 1$ unknown coefficients in (10.1.2). The curve fit examples we shall consider will be built

upon the assumption that we have a data set of K distinct points, x_1, x_2, \dots, x_K , where $K > S + 1$. The data set is displayed in the table

y_1	y_2	y_3	.	.	.	y_K
x_1	x_2	x_3	.	.	.	x_K

We can evaluate the polynomial (10.1.2) at each data pair and obtain the system of K equations for the $S + 1$ unknowns

$$\begin{aligned}
 a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^3 + \cdots + a_S x_1^S &= y_1 \\
 a_0 + a_1 x_2 + a_2 x_2^2 + a_3 x_2^3 + \cdots + a_S x_2^S &= y_2 \\
 a_0 + a_1 x_3 + a_2 x_3^2 + a_3 x_3^3 + \cdots + a_S x_3^S &= y_3 \\
 a_0 + a_1 x_4 + a_2 x_4^2 + a_3 x_4^3 + \cdots + a_S x_4^S &= y_4 \\
 &\vdots \\
 &\vdots \\
 &\vdots \\
 a_0 + a_1 x_K + a_2 x_K^2 + a_3 x_K^3 + \cdots + a_S x_K^S &= y_K
 \end{aligned} \tag{10.1.3}$$

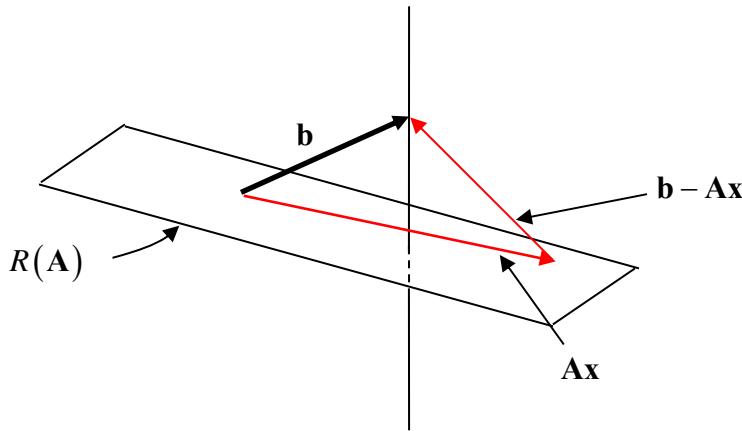
This result can be written as the matrix equation

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^S \\ 1 & x_2 & x_2^2 & \cdots & x_2^S \\ 1 & x_3 & x_3^2 & \cdots & x_3^S \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_K & x_K^2 & \cdots & x_K^S \end{bmatrix}}_{K \times (S+1)} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix}}_{(S+1) \times 1} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_K \end{bmatrix}}_{K \times 1} \tag{10.1.4}$$

which is an equation of the form (10.1.1).

Therefore, as in Sections 4.14 and 4.15, we are confronted with a linear transformation $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{U}$ and a vector $\mathbf{b} \in \mathcal{U}$, where \mathcal{V} and \mathcal{U} are real inner product spaces. We are also have the situation where the number of equations in (10.1.4) is greater than the number of unknowns. Over determined systems such as (10.1.4) have the property that $\mathbf{b} \notin R(\mathbf{A})$.

Therefore, the system $\mathbf{Ax} = \mathbf{b}$ does not have a solution. At this point in our general discussion, $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{U}$ is not onto and we do not require that it be one to one. In Section 4.14, we illustrated our problem in the three dimensional case by the following figure



The plane shown is the image space of the linear transformation. The given vector \mathbf{b} is not in the image space. The vector $\mathbf{b} - \mathbf{Ax}$ in some sense measures the inconsistency in the system. We defined the *residual* to be the vector

$$\mathbf{r}(\mathbf{x}) = \mathbf{b} - \mathbf{Ax} \quad (10.1.5)$$

and we measured the error by the length or norm squared of the residual (10.1.5). The length squared of this residual is

$$\|\mathbf{r}(\mathbf{x})\|^2 = \langle \mathbf{b} - \mathbf{Ax}, \mathbf{b} - \mathbf{Ax} \rangle \quad (10.1.6)$$

where the inner product is the one for \mathcal{V} . Our problem is to find the $\mathbf{x} \in \mathcal{V}$, which makes the squared residual, $\|\mathbf{r}(\mathbf{x})\|^2$, a minimum.

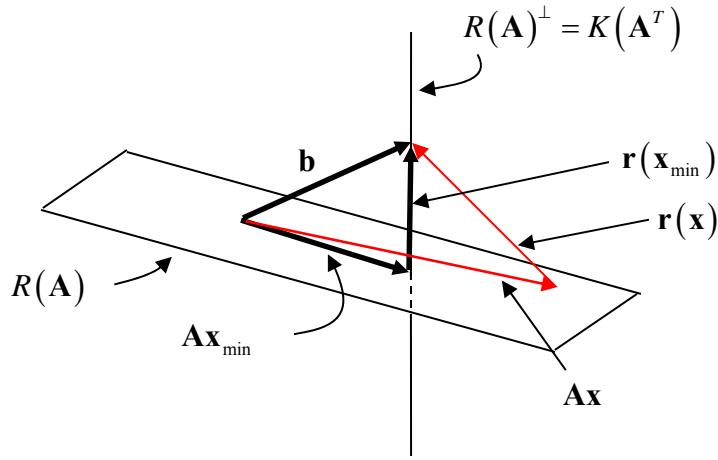
As shown in Section 4.14, the result of this minimization is that \mathbf{x} must obey the *normal equation*

$$\mathbf{A}^T \mathbf{A} \mathbf{x} = \mathbf{A}^T \mathbf{b} \quad (10.1.7)$$

A consequence of the result (10.1.7) is that

$$\mathbf{A} \mathbf{x} - \mathbf{b} \in K(\mathbf{A}^T) = R(\mathbf{A})^\perp \quad (10.1.8)$$

where (4.12.8) has been used. Repeating an observation made in Section 4.14, if we adopt the notation \mathbf{x}_{\min} for any solution to the normal equation (10.1.7), and display how it arises as a special choice of all possible vectors $\mathbf{x} \in \mathcal{V}$. The following figure is the result.



As the figure suggests, a solution \mathbf{x}_{\min} is the one that produces a residual vector $\mathbf{r}(\mathbf{x}_{\min}) = \mathbf{b} - \mathbf{Ax}_{\min}$ orthogonal to the image space $R(\mathbf{A})$. In other words, $\mathbf{r}(\mathbf{x}_{\min})$ is the projection of $\mathbf{r}(\mathbf{x})$ into the orthogonal compliment of $R(\mathbf{A})$. In Section 4.14, we established analytically the inequality

$$\|\mathbf{b} - \mathbf{Ax}\| \geq \|\mathbf{b} - \mathbf{Ax}_{\min}\| \quad (10.1.9)$$

Continuing our summary of material from Section 4.14, we established that

1. If \mathbf{b} is, in fact, in $R(\mathbf{A})$, then (10.1.7) written in the form

$$\mathbf{A}^T (\mathbf{b} - \mathbf{Ax}) = \mathbf{0} \quad (10.1.10)$$

shows that $\mathbf{b} - \mathbf{Ax}$, which is a vector in $R(\mathbf{A})$, is also in $K(\mathbf{A}^T)$. Because from (10.1.8), $K(\mathbf{A}^T) = R(\mathbf{A})^\perp$, it is necessarily true in this case that $R(\mathbf{A}) = R(\mathbf{A})^\perp$ and thus

$$\mathbf{Ax} = \mathbf{b} \quad (10.1.11)$$

The conclusion is that when $\mathbf{b} \in R(\mathbf{A})$, the normal equation (10.1.7) and (10.1.11) have the same solution.

2. The linear transformation $\mathbf{A}^T \mathbf{A} : \mathcal{V} \rightarrow \mathcal{V}$ is symmetric.

3. The kernel of $\mathbf{A}^T \mathbf{A} : \mathcal{V} \rightarrow \mathcal{V}$ equals the kernel of $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{U}$. In equation form this assertion is

$$K(\mathbf{A}^T \mathbf{A}) = K(\mathbf{A}) \quad (10.1.12)$$

4. The rank of the symmetric linear transformation $\mathbf{A}^T \mathbf{A} : \mathcal{V} \rightarrow \mathcal{V}$ and the linear transformation $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{U}$ are the same. In other words

$$\dim R(\mathbf{A}^T \mathbf{A}) = \dim R(\mathbf{A}) \quad (10.1.13)$$

5. The solution of the normal equation (10.1.7) is unique if and only if $K(\mathbf{A}) = \{\mathbf{0}\}$.
6. In the case where the solution of the normal equation (10.1.7) is not unique, the residuals for the various solutions are the same.

In the special case where $K(\mathbf{A}) = \{\mathbf{0}\}$, i.e., when $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{U}$ is one to one, we know from (3.3.12) that the rank of \mathbf{A} and $\mathbf{A}^T \mathbf{A}$ are both equal to $\dim \mathcal{V}$. As a result, the symmetric linear transformation $\mathbf{A}^T \mathbf{A} : \mathcal{V} \rightarrow \mathcal{V}$ is nonsingular even though the linear transformation $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{U}$ is not. The unique solution of the normal equation (10.1.7) is then

$$\mathbf{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b} \quad (10.1.14)$$

In general, when one is given a linear transformation $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{U}$ with the property that $\dim R(\mathbf{A}) = \dim \mathcal{V}$, the combination $(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T$ is known as the *left pseudo inverse* of \mathbf{A} . It is the left pseudo inverse that gives the solution to the least squared problem. If the linear transformation \mathbf{A} is nonsingular, it is evident that $(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T = \mathbf{A}^{-1} (\mathbf{A}^T)^{-1} \mathbf{A}^T = \mathbf{A}^{-1}$.

In those cases where $\mathbf{A}^T \mathbf{A}$ is nonsingular, the calculation of the left pseudo inverse, $(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T$, is easily calculated by the MATLAB script `(inv(A'*A))*A'` and the solution (10.1.14) by the script `(inv(A'*A))*A'*b`. It is important to know that MATLAB produces this sequence of calculation with the simplified script `A\b`.

Section 10.2. Linear Regression

Linear regression is the special regression process where (10.1.2) reduces to the first order polynomial

$$y(x) = a_0 + a_1 x \quad (10.2.1)$$

As in Section 10.1, we are given a data set of K distinct points

y_1	y_2	y_3	.	.	.	y_K
x_1	x_2	x_3	.	.	.	x_K

For our purposes, the number of data points is regarded as large. In particular, the number of data points is assumed to be larger than $S+1=2$, the number of unknowns in (10.2.1) above. The system of K equations for 2 unknowns a_0 and a_1 is the following special case of (10.1.4)

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_K \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 1 & x_K \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad (10.2.2)$$

The solution of this over determined problem is given by the solution of the normal equation (10.1.7), where

$$A = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 1 & x_K \end{bmatrix} \quad (10.2.3)$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_K \end{bmatrix} \quad (10.2.4)$$

and

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \quad (10.2.5)$$

Since we have assumed that the data points, x_1, x_2, \dots, x_N , are distinct, it is not difficult to show that the kernel of the matrix (10.2.3) only contains the zero vector in $\mathcal{M}^{2 \times 1}$. Thus, A is one to one and, from our discussion in Section 10.1, the matrix $A^T A$ is nonsingular and, as a result, the solution of (10.2.2) is

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = (A^T A)^{-1} A^T \mathbf{y} \quad (10.2.6)$$

If (10.2.3) and (10.2.4) are substituted into (10.2.6), the result is equation (4.15.27), repeated,

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \frac{1}{K \sum_{i=1}^K x_i^2 - \left(\sum_{i=1}^K x_i \right)^2} \begin{bmatrix} \sum_{i=1}^K x_i^2 & -\sum_{i=1}^K x_i \\ -\sum_{i=1}^K x_i & K \end{bmatrix} \begin{bmatrix} \sum_{i=1}^K y_i \\ \sum_{i=1}^K x_i y_i \end{bmatrix} \quad (10.2.7)$$

Given that we intend to utilize MATLAB to perform our calculations, we do not need to capitalize on the special result (10.2.7).

Example 10.1: Apply linear regression to the seven pairs of data points in the following table.

y	57	43	37	30	23	18	5
x	5	8	31	40	51	63	78

The MATLAB script ¹

```

clc
clear
x=[5,8,31,40,51,63,78]';
y=[57,43,37,30,23,18,5]';
A=[x.^0,x.^1];
%The direct solution of A'*A*a=A'*y is
a=inv(A'*A)*A'*y
%Plot of above results by the script
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',11)
xlabel('x')
ylabel('y','Rotation',0)
grid on
hold
xvalues=[0:5:80];
yvalues=a(1)+a(2)*xvalues;
plot(xvalues,yvalues,'LineWidth',2)
title('Example 10.1')
legend('data points', 'Linear Regression')

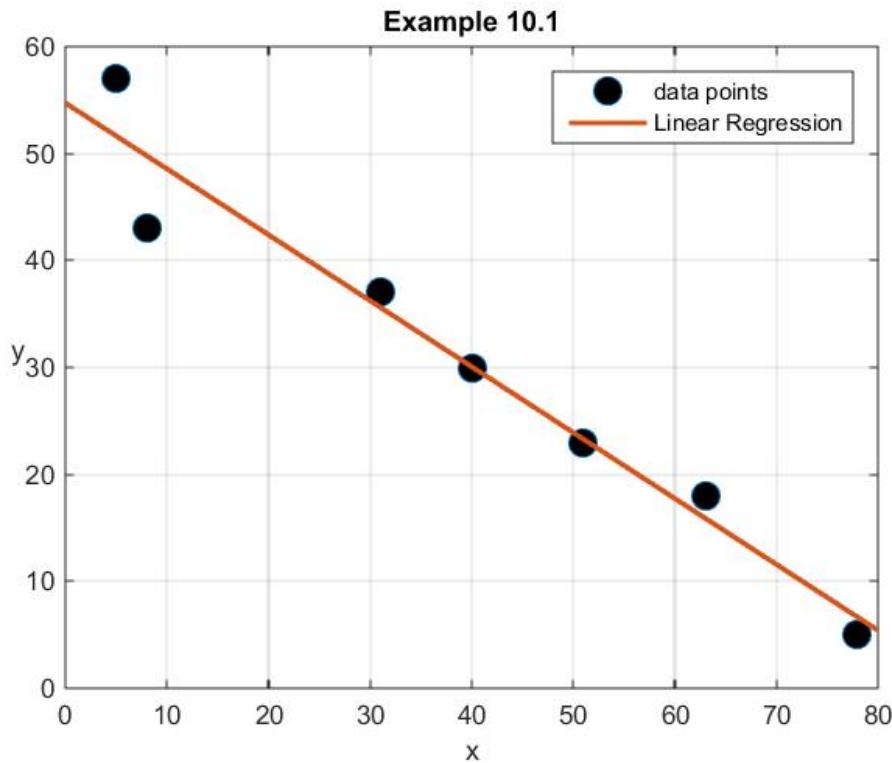
```

produces the values

$$\mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 54.73 \\ -0.6163 \end{bmatrix} \quad (10.2.8)$$

and the plot

¹ The MATLAB script **A=vander(x)** will also produce the matrix A.



Given (10.2.8), the linear regression from (10.2.1) is given by

$$y = a_0 + a_1 x = 54.73 - 0.6163x \quad (10.2.9)$$

As pointed out in the last section, the construction of the solution (10.2.6) is simplified if we utilize the MATLAB syntax `A\y`. This simplification is achieved if we replace the line, `a=inv(A'*A)*A'*y`, with `a=A\y`.

If a linear regression has been performed, it is useful to have a measure of the error associated with the use of (10.2.9). Next, we shall show how one such measure is defined. We begin by the assumption we have the data set shown in the following table.

y_1	y_2	y_3	.	.	.	y_K
x_1	x_2	x_3	.	.	.	x_K

The *arithmetic mean* of the y 's is defined by

$$\bar{y} = \frac{1}{K} \sum_{i=1}^K y_i \quad (10.2.10)$$

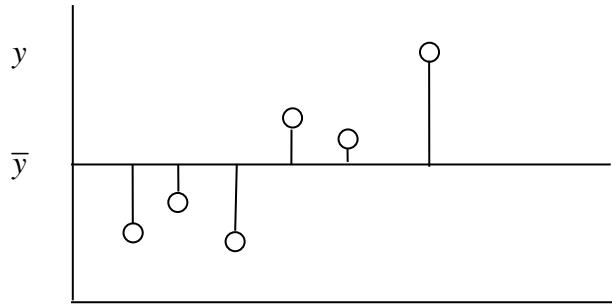
The MATLAB method of calculating the mean of the values of a vector \mathbf{y} is

$$\bar{y} = \text{mean}(\mathbf{y}). \quad (10.2.11)$$

The departures of the given data from the mean value, or the *spread*, is measured by

$$S_{\text{spread}} = \sum_{i=1}^K (y_i - \bar{y})^2 \quad (10.2.12)$$

The following figure shows these distances



The MATLAB method of calculating the spread of a vector \mathbf{y} is

$$(\mathbf{y}-\text{mean}(\mathbf{y}))' * (\mathbf{y}-\text{mean}(\mathbf{y})) \quad (10.2.13)$$

The *standard deviation* above the mean is defined by

$$s_y = \sqrt{\frac{1}{(K-1)} \sum_{i=1}^K (y_i - \bar{y})^2} = \sqrt{\frac{S_{\text{spread}}}{K-1}} \quad (10.2.14)$$

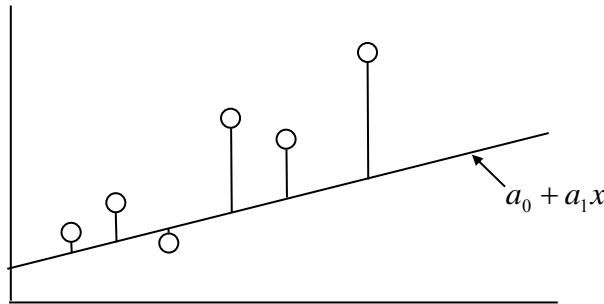
The square of the standard deviation is known as the *variance*. The standard deviation measures the spread of the data about the mean.

The MATLAB commands that calculate the standard deviation and the variance are, respectively, `std(y)` and `var(y)`.

The *departure* from the straight line calculated in the regression is the magnitude of the residual evaluated at the solution (10.2.6). It follows from (10.1.5) and (10.2.1), that the *departure* is calculated from

$$S_{\text{departure}} = \mathbf{r}^T \mathbf{r} = (\mathbf{y} - A\mathbf{a})^T (\mathbf{y} - A\mathbf{a}) = \sum_{i=1}^K (y_i - a_0 - a_1 x_i)^2 \quad (10.2.15)$$

where a_0 and a_1 are calculated from (10.2.6). Equations (10.2.3) and (10.2.4) have also been used to obtain the result (10.2.15)₃. The distances $y_i - a_0 - a_1 x_i$ are the distances in the y direction from the straight line calculated from the regression. The next figure shows these quantities



A measure of the departure of the data about the regression line is the quantity $s_{y/x}$ defined by

$$s_{y/x} = \sqrt{\frac{S_{\text{departure}}}{K - 2}} \quad (10.2.16)$$

This quantity is called the *standard error* of the regression. A measure of the *quality* of the regression is the ratio

$$r = \sqrt{\frac{S_{\text{spread}} - S_{\text{departure}}}{S_{\text{spread}}}} \quad (10.2.17)$$

where S_r is given by (10.2.12) and S_s is given by (10.2.15). The quantity r is called the *correlation coefficient*. It has the following properties

Perfect Fit: $S_{\text{departure}} = 0 \Rightarrow r = 1$

Worst Fit: $S_{\text{departure}} = S_{\text{spread}} \Rightarrow r = 0$

Example 10.2.2: If we adopt the data utilized in Example 10.2.1, the following MATLAB script

```
clc
clear
x=[5,8,31,40,51,63,78]';
y=[57,43,37,30,23,18,5]';
A=[x.^0,x.^1];
%The direct solution of A'*A*a=A'*b is
a=A\y
S_spread=(y-mean(y))'*(y-mean(y))
sy=sqrt(S_spread/(size(y,1)-1))
S_departure=(y-A*a)'*(y-A*a)
r=sqrt((S_spread-S_departure)/S_spread)
```

produces the results

$$\text{Spread: } S_{\text{spread}} = 1763.7$$

$$\text{Standard Deviation: } s_y = 17.1450$$

$$\text{Departure from Straight Line: } S_{\text{departure}} = 84.0078$$

$$\text{Correlation Coefficient: } r = 0.9758$$

The fact that the correlation coefficient is near unity for this example suggests that the regression is an accurate representation of the data.

Exercises

10.2.1: Apply linear regression to the six pairs of data points in the following table.

x	25	100	200	300	400	500
y	1.11	4.03	6.16	14.62	15.54	20.90

The result of this calculation is

$$y = -0.3098 + 0.0421x \quad (10.2.18)$$

Section 10.3. Linearization of Nonlinear Relationships

The simple linear regression discussed in Section 10.2 can be generalized in a number of ways. One of these is to change the independent and the dependent variables in such a fashion that one can transform a *nonlinear relationship* into a *linear relationship*. The formal step is to replace the linear relationship

$$y = a_0 + a_1 x \quad (10.3.1)$$

by relationships of the type

$$\begin{aligned} y &= a_0 e^{a_1 x} && \text{(exponential function)} \\ y &= a_0 x^{a_1} && \text{(power function)} \\ y &= \frac{1}{a_0 + a_1 \frac{1}{x}} && \text{(saturation growth-rate function)} \\ y &= a_0 + a_1 \ln(x) && \text{(logarithmic function)} \\ y &= \frac{1}{a_0 + a_1 x} && \text{(reciprocal function)} \end{aligned} \quad (10.3.2)$$

Equations (10.3.2) give a partial list of functions that are *nonlinear* in their dependence on the two parameters a_0 and a_1 . When we select these parameters so as to best fit in some sense a set of data, we are dealing with an example of *nonlinear regression*.

The special nature of the functions in the list (10.3.2) is such that we can utilize our result (10.2.6) to fit the above functions. The key to the use of these results is to *change variables* in such a way that we fit functions *equivalent* to each of those listed in (10.3.2).

The change of variables just mentioned involves writing the list of functions in (10.3.2) as

$$\begin{aligned}
 & \underbrace{\ln(y) = a_1 x + \ln(a_0)}_{\text{Linear in } \ln(y) \text{ and } x} && \text{(exponential function)} \\
 & \underbrace{\ln(y) = a_1 \ln(x) + \ln(a_0)}_{\text{Linear in } \ln(y) \text{ and } \ln(x)} && \text{(power function)} \\
 & \underbrace{\frac{1}{y} = a_0 + a_1 \frac{1}{x}}_{\text{Linear in } \frac{1}{y} \text{ and } \frac{1}{x}} && \text{(saturation growth-rate function)} \\
 & \underbrace{y = a_0 + a_1 \ln(x)}_{\text{Linear in } y \text{ and } \ln(x)} && \text{(logarithmic function)} \\
 & \underbrace{\frac{1}{y} = a_0 + a_1 x}_{\text{Linear in } \frac{1}{y} \text{ and } x} && \text{(reciprocal function)} \\
 & && (10.3.3)
 \end{aligned}$$

Example 10.3.1: In this example we shall adopt the data utilized in Example 4.15.1, namely,

x	5	10	15	20	25	30	35	40	45	50
y	17	24	31	33	37	37	40	40	42	41

In Section 4.15 we performed a regression of this data set by utilizing a third degree polynomial.² In this case, we shall attempt to fit the data to the power function (10.3.2)₂, i.e. to the function

$$y = a_0 x^{a_1} \quad (10.3.4)$$

The equivalent linear regression problem is to fit the data to the expression

$$\ln(y) = a_1 \ln(x) + \ln(a_0) \quad (10.3.5)$$

The following MATLAB script yields

```

clc
clear
x=[5,10,15,20,25,30,35,40,45,50]';
y=[17,24,31,33,37,37,40,40,42,41]';
%Convert data to its logarithm.
Logx=log(x)
Logy=log(y)
A=[Logx.^0,Logx.^1];
%The solution of A\Logy is
c=A\Logy

```

² We shall briefly return to a discussion of polynomial regression in Section 10.5.

$$\mathbf{c} = \begin{bmatrix} \ln a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 2.2979 \\ 0.3851 \end{bmatrix} \quad (10.3.6)$$

It follows then from (10.3.6) that

$$\ln(a_0) = 2.2979 \Rightarrow a_0 = 9.9529 \quad (10.3.7)$$

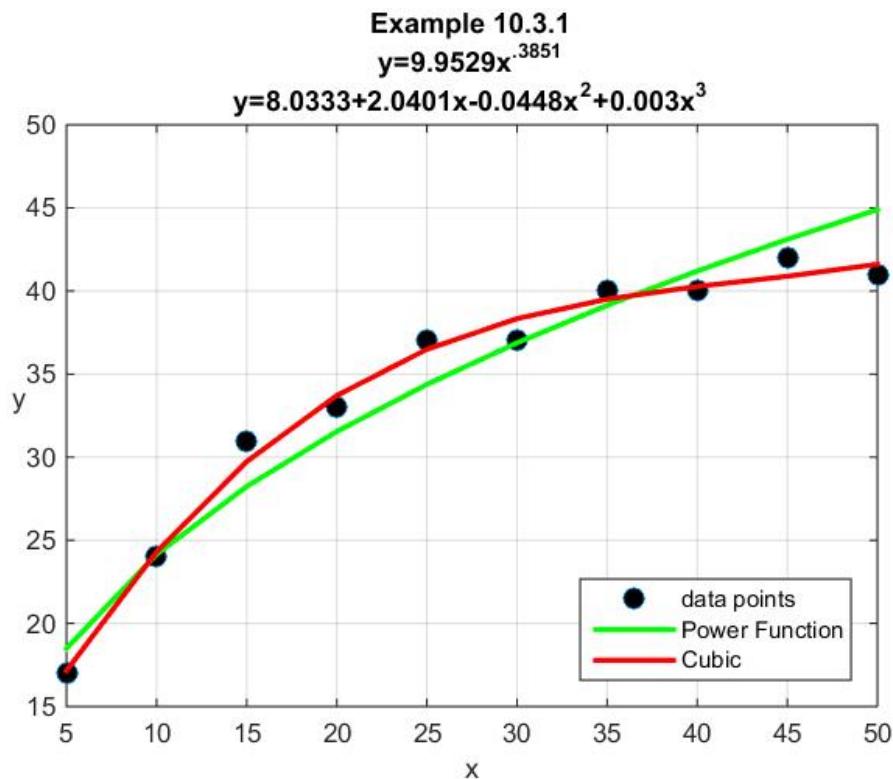
and

$$a_1 = 0.3851 \quad (10.3.8)$$

Therefore, equation (10.3.4) becomes

$$y = 9.9529x^{0.3851} \quad (10.3.9)$$

If the above data, the cubic regression (4.15.16) and the nonlinear regression (10.3.9) are plotted, the result is



The above plot is generated by the script

```

clc
clear
x=[5,10,15,20,25,30,35,40,45,50]';
y=[17,24,31,33,37,37,40,40,42,41]';
%Convert data to its logarithm.
Logx=log(x)
Logy=log(y)
%For the Power Function
A=[Logx.^0,Logx.^1];
%For the cubic
B=[x.^0,x.^1,x.^2,x.^3]
%The solution of A\Logy for the Power Function is
c=A\Logy
%The solution of B\y for the cubic is
b=B\y

%Plot of above results
%First, plot the data points
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',8)
xlabel('x')
ylabel('y','Rotation',0)
axis([5,50,15,50])
grid on
hold
%Next, plot the Power Function
xvalues=[5:5:50];
yvalues=exp(c(1))*xvalues.^c(2);
plot(xvalues,yvalues,'g','LineWidth',2)
%Next, plot the cubic
Yvalues=b(1)+b(2)*xvalues+b(3)*xvalues.^2+b(4)*xvalues.^3
plot(xvalues,Yvalues,'r','LineWidth',2)
title({'Example 10.3.1',...
'y=9.9529x^{.3851}',...
'y=8.0333+2.0401x-0.0448x^2+0.003x^3'})
legend('data points', 'Power Function','Cubic',...
'Location','southeast')

```

Exercises

10.3.1: Given the data table

x	1.3	1.8	3.0	4.5	6.0	8.0	9.0
y	1.0700	1.1300	1.2200	1.2750	1.3350	1.3500	1.3600

Fit this data to the saturation growth-rate function (10.3.2)₃, repeated,

$$y = \frac{1}{a_0 + a_1 \frac{1}{x}} \quad (10.3.10)$$

The result of this calculation is

$$y = \frac{1}{0.6230 + 0.4528 \frac{1}{x}} \quad (10.3.11)$$

10.3.2: Given the data table

x	.5	1.25	2	2.7	3
y	3	4	3	2	1

Fit this data to the function

$$y = Axe^{Bx} \quad (10.3.12)$$

The result of this calculation is

$$y = 11.7683xe^{-1.0969x} \quad (10.3.13)$$

Section 10.4. MATLAB Tools for Linear Regression

In this short section, we shall introduce the MATLAB built in function **polyfit**. This function provides an alternate way to perform the regression calculation illustrated in Sections 10.2 and 10.4.

Example 10.4.1: Consider the data set implied by the following linear regression problem

$$\mathbf{x} = \begin{bmatrix} 0 \\ 2 \\ 4 \\ 6 \\ 9 \\ 11 \\ 12 \\ 15 \\ 17 \\ 19 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 5 \\ 6 \\ 7 \\ 6 \\ 9 \\ 8 \\ 7 \\ 10 \\ 12 \\ 12 \end{bmatrix} \quad (10.4.1)$$

Our previous examples showed that the regression can be implemented if we adopt the script

```
clc,clear
x=[0,2,4,6,9,11,12,15,17,19]'
y=[5,6,7,6,9,8,7,10,12,12]'
A=[x.^0,x.^1]
c=inv(A'*A)*A'*y
```

In this case, the output is

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 4.8515 \\ 0.3525 \end{bmatrix} \quad (10.4.2)$$

As explained at the end of Section 10.1, the same answer is obtained if the last line of the above script is replaced by

$$\mathbf{c} = \mathbf{A} \setminus \mathbf{y} \quad (10.4.3)$$

Also, there is yet another method of calculating the coefficients in the linear regression. This method involves use of one of the special MATLAB polynomial commands to generate the answer. The particular command is called **polyfit**. The command has a number of applications that we shall explore. For the moment, we shall illustrate it by reworking the above example.

Example 10.4.2: Same example, except use **polyfit**.

The script

```
clc
clear all
x=[0;2;4;6;9;11;12;15;17;19];
y=[5;6;7;6;9;8;7;10;12;12]
P=polyfit(x,y,1)
```

Produces the output

```
P =
    0.3525    4.8515
```

Note that the output is in the MATLAB format explained in Section 9.6, where the *first entry is the coefficient of the power of the highest term* in the polynomial.

The **polyfit** command is especially convenient when using one of the nonlinear relationships given in equation (10.3.2) is adopted. The following table shows how **polyfit** can be used for these kinds of problems.

function	polyfit	Output
$y = a_0 + a_1 x$	P=polyfit(x,y,1)	P=[a1,a0]
$y = a_0 e^{a_1 x}$	P=polyfit(x,log(y),1)	P=[a1,log(a0)]
$y = a_0 x^{a_1}$	P=polyfit(log(x),log(y),1)	P=[a1,log(a0)]
$y = \frac{1}{a_0 + a_1 \frac{1}{x}}$	P=polyfit(1./x,1./y,1)	P=[a1,a0]
$y = a_0 + a_1 \ln(x)$	P=polyfit(log(x),y,1)	P=[a1,a0]

$y = \frac{1}{a_0 + a_1 x}$	P=polyfit(x,1./y,1)	P=[a1,a0]
-----------------------------	----------------------------	------------------

Section 10.5. Polynomial Regression

Rather than a regression based upon a straight line relationship (10.2.1), we can formulate one based upon a *polynomial* of degree N as follows:³

$$y(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_Nx^N \quad (10.5.1)$$

The regression we shall formulate will be built upon the assumption that we have a data set of K points where $K > N + 1$, the number of unknown coefficients in the polynomial. The data set is written

y_1	y_2	y_3	.	.	.	y_K
x_1	x_2	x_3	.	.	.	x_K

As in Section 4.15, we can evaluate the polynomial (10.5.1) at each data pair and obtain the system of equations

$$\begin{aligned} a_0 + a_1x_1 + a_2x_1^2 + a_3x_1^3 + \cdots + a_Nx_1^N &= y_1 \\ a_0 + a_1x_2 + a_2x_2^2 + a_3x_2^3 + \cdots + a_Nx_2^N &= y_2 \\ a_0 + a_1x_3 + a_2x_3^2 + a_3x_3^3 + \cdots + a_Nx_3^N &= y_3 \\ a_0 + a_1x_4 + a_2x_4^2 + a_3x_4^3 + \cdots + a_Nx_4^N &= y_4 \\ . \\ . \\ . \\ a_0 + a_1x_K + a_2x_K^2 + a_3x_K^3 + \cdots + a_Nx_K^N &= y_K \end{aligned} \quad (10.5.2)$$

This result can be written as the matrix equation

³ Recall from our discussion in Section 9.6, MATLAB writes polynomials such as (10.5.1) in the form

$$y(x) = p_1x^N + p_2x^{N-1} + \cdots + p_Nx + p_{N+1}$$

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdot & \cdot & x_1^N \\ 1 & x_2 & x_2^2 & \cdot & \cdot & x_2^N \\ 1 & x_3 & x_3^2 & \cdot & \cdot & x_3^N \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & \cdot & & & \cdot \\ 1 & x_K & x_K^2 & \cdot & \cdot & x_K^N \end{bmatrix}}_{K \times (N+1)} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdot \\ \cdot \\ \cdot \\ a_N \end{bmatrix}}_{(N+1) \times 1} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ \cdot \\ \cdot \\ y_K \end{bmatrix}}_{K \times 1} \quad (10.5.3)$$

or, equivalently, as the matrix equation

$$\underbrace{\begin{bmatrix} x_1^N & \cdot & \cdot & x_1^2 & x_1 & 1 \\ x_2^N & \cdot & \cdot & x_2^2 & x_2 & 1 \\ x_3^N & \cdot & \cdot & x_3^2 & x_3 & 1 \\ \cdot & \cdot & & \cdot & 1 & \\ \cdot & \cdot & & \cdot & 1 & \\ \cdot & \cdot & & \cdot & 1 & \\ x_K^N & \cdot & \cdot & x_K^2 & x_K & 1 \end{bmatrix}}_{K \times (N+1)} \underbrace{\begin{bmatrix} a_N \\ \cdot \\ a_2 \\ a_1 \\ a_0 \end{bmatrix}}_{(N+1) \times 1} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ \cdot \\ \cdot \\ y_K \end{bmatrix}}_{K \times 1} \quad (10.5.4)$$

The advantage of (10.5.4) over (10.5.3) is that the solution of (10.5.4) orders the powers of the polynomial in the same order as adopted by the MATLAB convention for polynomials.

We shall use a formalism virtually identical to that used earlier with (10.2.2). As an over determined system of K equations in $N+1$ unknowns, (10.5.3) or, equivalently, (10.5.4) is usually *inconsistent* and, as such, *does not* have a solution. As with the linear regression problem, we can define the residuals as a column vector \mathbf{r} defined by

$$\mathbf{r} = \mathbf{y} - A\mathbf{c} \quad (10.5.5)$$

where

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ y_K \end{bmatrix} \quad (10.5.6)$$

$$\mathbf{c} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ \vdots \\ a_N \end{bmatrix} \quad (10.5.7)$$

and

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdot & \cdot & x_1^N \\ 1 & x_2 & x_2^2 & \cdot & \cdot & x_2^N \\ 1 & x_3 & x_3^2 & \cdot & \cdot & x_3^N \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & \cdot & & & \cdot \\ 1 & x_K & x_K^2 & \cdot & \cdot & x_K^N \end{bmatrix} \quad (10.5.8)$$

or, equivalently,

$$A = \begin{bmatrix} x_1^N & \cdot & \cdot & x_1^2 & x_1 & 1 \\ x_2^N & \cdot & \cdot & x_2^2 & x_2 & 1 \\ x_3^N & \cdot & \cdot & x_3^2 & x_3 & 1 \\ \cdot & \cdot & & \cdot & 1 & \\ \cdot & \cdot & & \cdot & 1 & \\ \cdot & \cdot & & \cdot & 1 & \\ x_K^N & \cdot & \cdot & x_K^2 & x_K & 1 \end{bmatrix} \quad (10.5.9)$$

and

$$\mathbf{c} = \begin{bmatrix} a_N \\ \vdots \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \quad (10.5.10)$$

Example 10.5.1: Fit a cubic polynomial to the data set

x	10	20	30	40	50	60	70	80	90	100
y	11	24	35	33	53	65	73	89	111	151

Therefore, you are asked to determine coefficients

$$\mathbf{c} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (10.5.11)$$

in the cubic equation

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 \quad (10.5.12)$$

The matrix A in the normal equation is

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \\ 1 & x_5 & x_5^2 & x_5^3 \\ 1 & x_6 & x_6^2 & x_6^3 \\ 1 & x_7 & x_7^2 & x_7^3 \\ 1 & x_8 & x_8^2 & x_8^3 \\ 1 & x_9 & x_9^2 & x_9^3 \\ 1 & x_{10} & x_{10}^2 & x_{10}^3 \end{bmatrix} = \begin{bmatrix} 1 & 10 & 100 & 1000 \\ 1 & 20 & 400 & 8000 \\ 1 & 30 & 900 & 27000 \\ 1 & 40 & 1600 & 64000 \\ 1 & 50 & 2500 & 125000 \\ 1 & 60 & 3600 & 216000 \\ 1 & 70 & 4900 & 343000 \\ 1 & 80 & 6400 & 512000 \\ 1 & 90 & 8100 & 729000 \\ 1 & 100 & 10000 & 1000000 \end{bmatrix} \quad (10.5.13)$$

The matrix \mathbf{y} in the normal equation is

$$\mathbf{y} = \begin{bmatrix} 11 \\ 24 \\ 35 \\ 33 \\ 53 \\ 65 \\ 73 \\ 89 \\ 111 \\ 151 \end{bmatrix} \quad (10.5.14)$$

The MatLab solution of $A^T A \mathbf{c} = A^T \mathbf{y}$ is

$$\mathbf{c} = \begin{bmatrix} -7.000 \\ 2.0615 \\ -0.0321 \\ 0.0003 \end{bmatrix} \quad (10.5.15)$$

Therefore, the least square approximation to the data is

$$y = -7.000 + 2.0615x - 0.0321x^2 + 0.0003x^3 \quad (10.5.16)$$

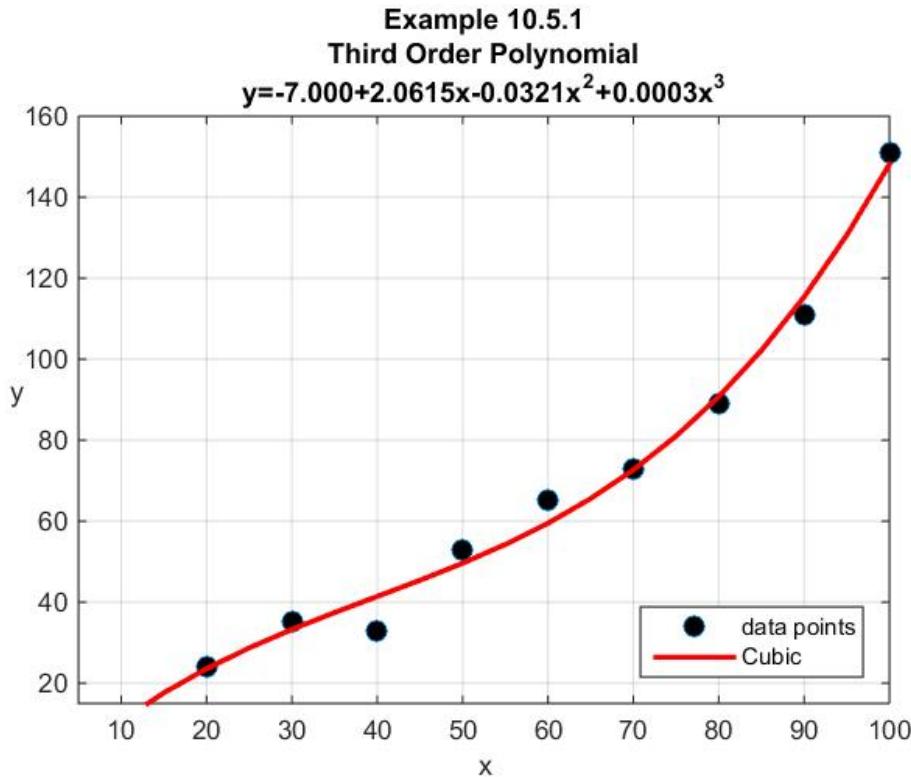
The MATLAB script that produces the above results is

```

clc
clear
x=[10,20,30,40,50,60,70,80,90,100]%
y=[11,24,35,33,53,65,73,89,111,151]%
A=[x.^0,x.^1,x.^2,x.^3]
c=A\y

```

The plot of the data and the polynomial (10.5.16) is as follows:



In Section 10.4, we utilized the **polyfit** command in the case of a linear regression. **polyfit** can also be used to find the solution of the problem just solved. When one wishes to apply least squared regression using a *polynomial* of degree N , the command syntax is

$$\text{polyfit}(\mathbf{x}, \mathbf{y}, \mathbf{N}) \quad (10.5.17)$$

where **x** and **y** are the data points and **N** is the order of the polynomial to be fitted to the data. The output of this command is a row vector

$$(p_1, p_2, p_3, \dots, p_N, p_{N+1}) \quad (10.5.18)$$

whose elements are the coefficients of a polynomial written in the form

$$y(x) = p_1x^N + p_2x^{N-1} + \cdots + p_Nx + p_{N+1} \quad (10.5.19)$$

As explained in Section 9.6 and repeated two times earlier in this Chapter, the MATLAB scheme for labeling the coefficients of polynomials and that based upon (10.5.1) are different. The connections are

$$\begin{aligned} a_0 &\rightarrow p_{N+1} \\ a_1 &\rightarrow p_N \\ a_2 &\rightarrow p_{N-1} \\ &\cdot \\ &\cdot \\ &\cdot \\ a_{N-1} &\rightarrow p_2 \\ a_N &\rightarrow p_1 \end{aligned} \quad (10.5.20)$$

In Section 10.2, we introduced a few statistical definitions that are useful when trying to make a judgment of the quality of a particular regression. It is useful to restate these definitions as they apply to polynomial regressions. For a polynomial regression of degree N for a data table of K points, a summary of these definitions is as follows:

Arithmetic Mean:

$$\bar{y} = \frac{1}{K} \sum_{i=1}^K y_i \quad (10.5.21)$$

Spread:

$$S_{spread} = \sum_{i=1}^K (y_i - \bar{y})^2 \quad (10.5.22)$$

Standard Deviation

$$s_y = \sqrt{\frac{S_{spread}}{K-1}} \quad (10.5.23)$$

As earlier, the square of the standard deviation is known as the *variance*. The standard deviation measures the spread of the data about the mean.

Histogram: A histogram is a graphical representation of the distribution of the data. It is constructed by sorting the measurements into intervals, sometimes called *bins*. The histogram is a plot of the frequency of occurrence resulting from grouping the data into intervals and plotting the frequency of occurrence against these intervals.

Departure: The departure from the polynomial curve is defined as the least squared error that was minimized to obtain the normal equation (4.14.20). From (10.5.5), it is given by

$$\begin{aligned} S_{\text{departure}} &= \mathbf{r}^T \mathbf{r} = (\mathbf{y} - \mathbf{A}\mathbf{c})^T (\mathbf{y} - \mathbf{A}\mathbf{c}) \\ &= \sum_{i=1}^K (y_i - a_0 - a_1 x_i + a_2 x_i^2 + a_3 x_i^3 + \dots + a_N x_i^N)^2 \end{aligned} \quad (10.5.24)$$

Standard Error: A measure of the spread of the data about the regression line is the quantity $s_{y/x}$ defined by

$$s_{y/x} = \sqrt{\frac{S_{\text{departure}}}{K - (N + 1)}} \quad (10.5.25)$$

Correlation Coefficient: As in Section 10.2, the correlation coefficient is a measure of the quality of the regression defined by the ratio

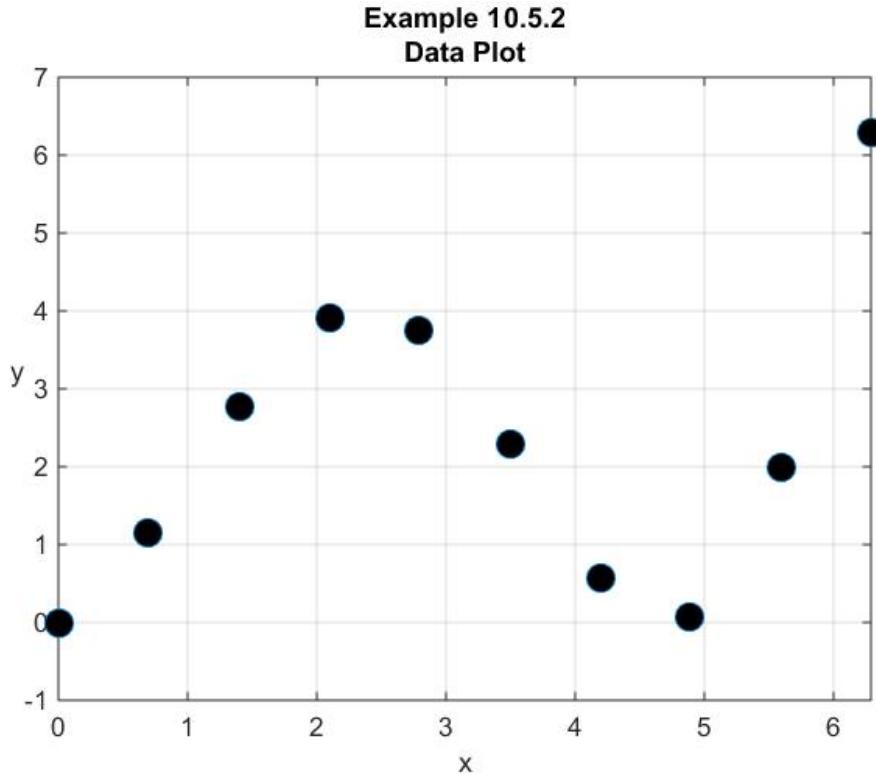
$$r = \sqrt{\frac{S_{\text{spread}} - S_{\text{departure}}}{S_{\text{spread}}}} \quad (10.5.26)$$

The MATLAB script that was given in Section 10.2 applies here for those quantities that do not depend upon the degree of the polynomial.

Example 10.5.2: The following example begins with the data

x	0	0.6981	1.3983	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
y	0	1.1469	2.7713	3.9082	3.7476	2.2968	0.5612	0.0742	1.9951	6.2832

which is easily shown to plot as



This particular data set does not have a lot of scatter. For this reason, our example is not all together typical of regression problems. It has been selected in order to illustrate the gains associated with various types of polynomial regressions.

The spread and the standard deviation for this data set is calculated by the script

```
S_spread=(y-mean(y))'*(y-mean(y))
%or (norm(y-mean(y)))^2
%Calculation of Standard Deviation
s_y=sqrt(S_spread/(length(x)-1))
%or std(y)
```

The resulting numerical values are

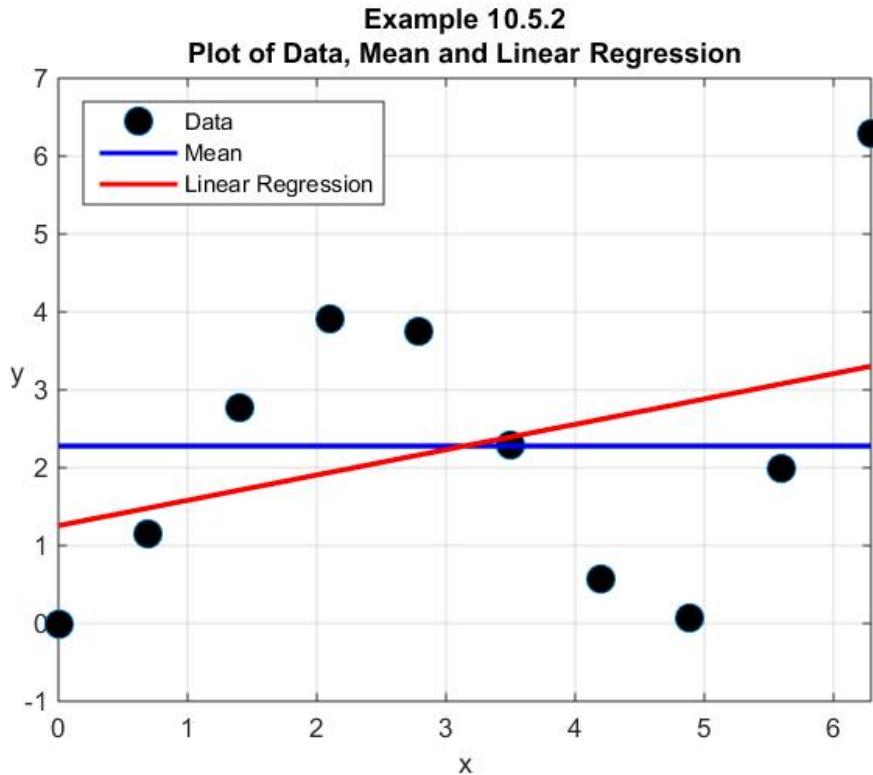
$$S_{spread} = 35.4554 \quad (10.5.27)$$

and

$$s_y = 1.9848 \quad (10.5.28)$$

These numbers reflect the large spread in the data set.

Next, we shall perform polynomial regressions on this data and attempt to reach some conclusions about the quality of the regressions both from the graphics and from calculation of the correlation coefficients. The first step will be to apply a *linear regression* to the above data and superimpose the straight line on the above plot. The result is

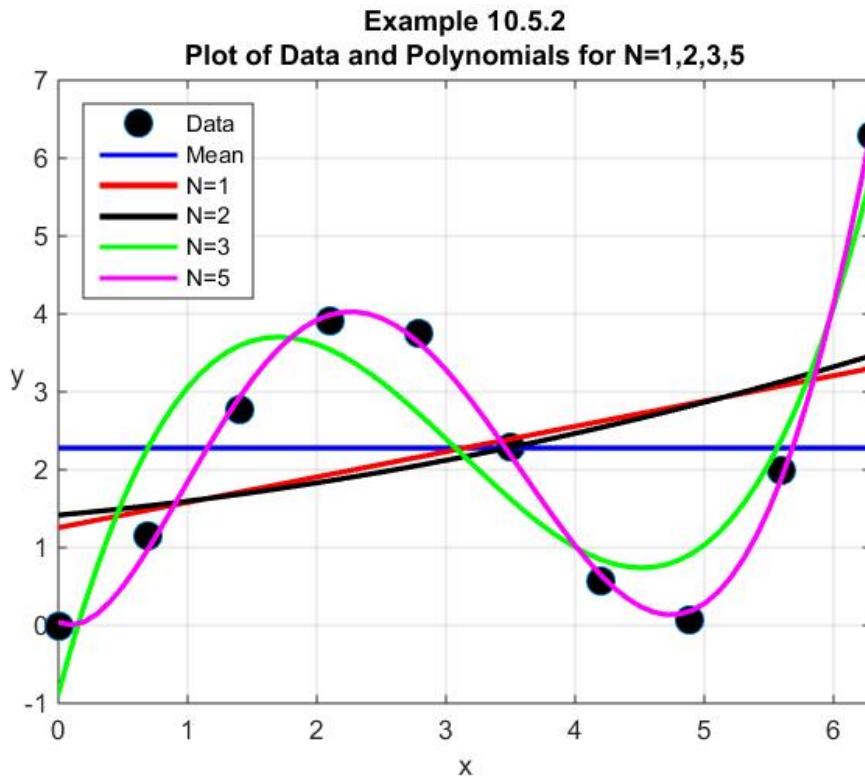


As one would expect, a linear regression is going to yield a poor representation of this particular data set. The correlation coefficient for the linear regression turns out to be

$$r|_{N=1} = 0.3468 \quad (10.5.29)$$

The small value of the correlation coefficient is reflected in the poor relationship displayed in the above figure.

The next step is to perform polynomial regressions for $N = 2, 3$ and 5 . The result is the plot



The correlation coefficients for the curves shown are

$$\begin{aligned}
 r|_{N=1} &= 0.3468 \\
 r|_{N=2} &= 0.3506 \\
 r|_{N=3} &= 0.9174 \\
 r|_{N=5} &= 0.9983
 \end{aligned} \tag{10.5.30}$$

These numbers and the above figure reflect essentially no improvement of the quadratic ($N = 2$) regression relative to the linear. They also reflect the general improvement associated with the $N = 3$ and $N = 5$ cases.

Exercises:

10.5.1: You are given the following data table:

x	1.4	2.6	3.1	3.9	5	7.1	9.5	11.9	14.1	15	16.5	17.2
y	3.8	2.6	4.1	5.2	6.2	6.9	7.2	6.7	5.8	3.8	3.8	2.8

Apply least square regression procedures to fit a polynomial of order two to this data. Include in your solution a plot of your derived polynomial and the given data.

10.5.2: You are given the following data table:

x	5	10	15	20	25	30	35	40	45	50
y	17	24	31	33	37	37	40	40	42	41

Apply least square regression procedures to fit a polynomial of order three to this data. Include in your solution a plot of your derived polynomial and the given data.

Section 10.6. More General Types of Regression

In addition to curve fitting data that can be written in the form

$$y = f(x) \quad (10.6.1)$$

one can imagine several more complicated circumstances.

In cases where the data sets depends upon two, three or more independent variables, the curve fit is actually a *surface fit*. Formally, we are given a data set that depends, for example, on p independent variables. If the dependent variable is denoted by y , and the independent variables by x_1, x_2, \dots, x_p , i.e., then (10.6.1) is replaced by

$$y = f(x_1, x_2, \dots, x_p) \quad (10.6.2)$$

Thus the idea of a curve fit goes over to fitting a p -dimensional surface as an approximation to the data. The idea of fitting a straight line to a set of data goes over to the idea of fitting a p -dimensional plane

$$y = f(x_1, x_2, \dots, x_p) = a_0 + a_1 x_1 + a_2 x_2 + \dots + a_p x_p \quad (10.6.3)$$

Rather than devising a scheme to calculate two coefficients as we did with linear regressions, we now need one to calculate the $p+1$ coefficients $a_0, a_1, a_2, \dots, a_p$. Fortunately, the formalities are not unlike what we have already been using. First, we are given K data points, where $K > p+1$, in a table as follows:

$y_{(1)}$	$y_{(2)}$	$y_{(3)}$.	.	.	$y_{(K)}$
$x_{(1)1}$	$x_{(2)1}$	$x_{(3)1}$.	.	.	$x_{(K)1}$
$x_{(1)2}$	$x_{(2)2}$	$x_{(3)2}$.	.	.	$x_{(K)2}$
.	.	.				.
.	.	.				.
.	.	.				.
$x_{(1)p}$	$x_{(2)p}$	$x_{(3)p}$				$x_{(K)p}$

As before, we introduce matrix representations of the data as follows:

$$\mathbf{y} = \begin{bmatrix} y_{(1)} \\ y_{(2)} \\ y_{(3)} \\ \vdots \\ \vdots \\ y_{(K)} \end{bmatrix}, \quad \mathbf{x}_j = \begin{bmatrix} x_{(1)j} \\ x_{(2)j} \\ x_{(3)j} \\ \vdots \\ \vdots \\ x_{(K)j} \end{bmatrix} \quad \text{for } j = 1, 2, \dots, p \quad (10.6.4)$$

We next represent the unknown coefficients by the matrix

$$\mathbf{c} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ \vdots \\ a_p \end{bmatrix} \quad (10.6.5)$$

In order to make the manipulations look like what we have done before, define the matrix A by

$$A = \begin{bmatrix} 1 & x_{(1)1} & x_{(1)2} & x_{(1)3} & \cdot & \cdot & \cdot & x_{(1)p} \\ 1 & x_{(2)1} & x_{(2)2} & x_{(2)3} & \cdot & \cdot & \cdot & x_{(2)p} \\ 1 & x_{(3)1} & x_{(3)2} & x_{(3)3} & \cdot & \cdot & \cdot & x_{(3)p} \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot \\ 1 & x_{(K)1} & x_{(K)2} & x_{(K)3} & \cdot & \cdot & \cdot & x_{(K)p} \end{bmatrix} \quad (10.6.6)$$

If we force the data to fit the linear expression (10.6.3), i.e., as

$$y = a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_p x_p \quad (10.6.7)$$

then, as before, we get a system of equations

$$\underset{K \times (p+1)}{A} \underset{(p+1) \times 1}{\mathbf{c}} = \underset{K \times 1}{\mathbf{y}} \quad (10.6.8)$$

This system, because $K > p + 1$, we again encounter an over determined system that has no solution. Following the same combination of steps used in Sections 4.14, 10.1 and 10.5, we define the *residual* \mathbf{r} by the matrix equation

$$\mathbf{r} = \mathbf{y} - A\mathbf{c} \quad (10.6.9)$$

and, as we have done before, force

$$S_r = \mathbf{r}^T \mathbf{r} = (\mathbf{y} - A\mathbf{c})^T (\mathbf{y} - A\mathbf{c}) \quad (10.6.10)$$

to be a *minimum*. The result is, again, that \mathbf{c} must be a solution of the normal equation

$$A^T A \mathbf{c} = A^T \mathbf{y} \quad (10.6.11)$$

The formal similarity of this result and our previous results for curve fitting straight lines and polynomials is both good news and bad. It is good because it aids in presenting the solution to several different types of problems in a common framework. It is bad because you must recognize that each different case involves different definitions of the matrices \mathbf{y} , \mathbf{x} , thus A , and the unknown matrix \mathbf{c} .

Example 10.6.1: Perform a multiple linear regression based upon the table

y ₁	x ₁	x ₂
28.76	0	0
31.85	1	1
26.93	1	2
32.85	2	2
34.82	3	4
33.75	3	5
29.72	4	5
32.55	5	6
31.86	6	7
35.85	6	9
30.87	7	10
35.65	7	11
30.37	8	13

25.9	9	14
27.45	10	15

In order to fix the notation, we shall write the linear regression in the form (10.6.7)

$$y = a_0 + a_1 x_1 + a_2 x_2 \quad (10.6.12)$$

and our problem is to determine the three coefficients a_0, a_1 and a_2 . The first step involves relating the entries in the above table to those used in the definitions (10.6.4). There are fifteen values of the quantity, y , displayed in the first column of the above table. Thus $K = 15$. It follows from the definitions (10.6.4), (10.6.5) and (10.6.6) and the table of data that

$$\mathbf{y} = \begin{bmatrix} 28.76 \\ 31.85 \\ 26.93 \\ 32.85 \\ 34.82 \\ 33.75 \\ 29.72 \\ 32.55 \\ 31.86 \\ 35.85 \\ 30.87 \\ 35.65 \\ 30.37 \\ 25.90 \\ 27.45 \end{bmatrix} \quad (10.6.13)$$

$$\mathbf{x}_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 2 \\ 3 \\ 3 \\ 4 \\ 5 \\ 6 \\ 6 \\ 7 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix} \quad (10.6.14)$$

and

$$\mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 2 \\ 4 \\ 5 \\ 5 \\ 6 \\ 7 \\ 9 \\ 10 \\ 11 \\ 13 \\ 14 \\ 15 \end{bmatrix} \quad (10.6.15)$$

Our next task is to form equation (10.6.11) and calculate the matrix

$$\mathbf{c} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \quad (10.6.16)$$

The following MATLAB script will do that calculation for (10.6.16).

```
clc
clear
y=[28.76,31.85,26.93,32.85,34.82,33.75,29.72, ...
    32.55,31.86,35.85,30.87,35.65,30.37,25.90,27.45]';
x1=[0,1,1,2,3,3,4,5,6,6,7,7,8,9,10]';
x2=[0,1,2,2,4,5,5,6,7,9,10,11,13,14,15]';
A=[ones(15,1),x1,x2]
c=A\y
```

The output from this script is

```
c =
31.5867
0.9038
-0.6701
```

Therefore, (10.6.12) reduces to

$$y = 31.5867 + 0.9038x_1 - 0.6701x_2 \quad (10.6.17)$$

If the script above is replaced by

```
clc
clear
y=[28.76,31.85,26.93,32.85,34.82,33.75,29.72, ...
    32.55,31.86,35.85,30.87,35.65,30.37,25.90,27.45]';
x1=[0,1,1,2,3,3,4,5,6,6,7,7,8,9,10]';
x2=[0,1,2,2,4,5,5,6,7,9,10,11,13,14,15]';
A=[ones(15,1),x1,x2]
c=A\y

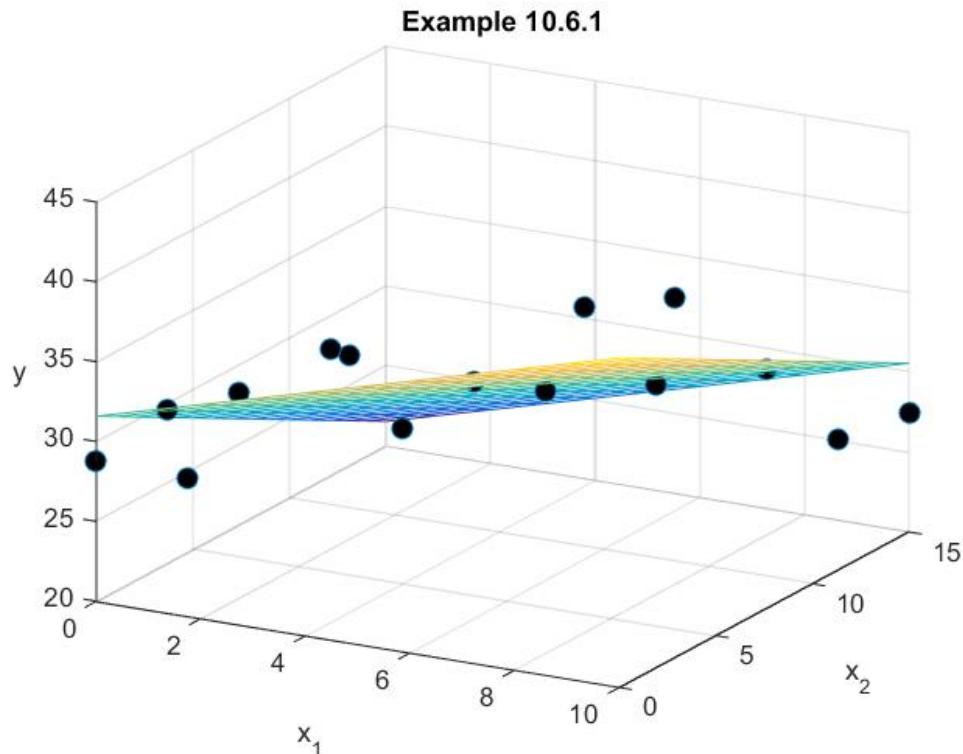
%Plot above results by the script
plot3(x1,x2,y,'o','MarkerFaceColor','k','MarkerSize',8)
xlabel('x_1'), ylabel('x_2'), zlabel('y','Rotation',0)
grid on
```

```

hold
x1values=[0:1:15];x2values=[0:1:15]
n=size(x1values);m=size(x2values)
[x1,x2]=meshgrid(x1values,x2values);
yvalues=c(1)+c(2)*x1+c(3)*x2
mesh(x1,x2,yvalues)
view(29,24)
title('Example 10.6.1')

```

the above problem is solved again and the output is displayed in the plot



A generalization of (10.6.3) arises when we are given a set of $p+1$ functions $\{z_0, z_1, \dots, z_p\}$, which are called *basis functions*. The idea is to fit, through a least squared approximation, a data set through the representation

$$y = a_0 z_0 + a_1 z_1 + a_2 z_2 + \cdots + a_p z_p \quad (10.6.18)$$

Equation (10.6.18) reduces to *linear regression* as discussed in Section 10.2 if we make the choices

$$p = 1, z_0 = 1, z_1 = x \quad (10.6.19)$$

Likewise, equation (10.6.18) reduces to *polynomial regression* as discussed in Section 10.5 if we make the choices

$$z_0 = x^0 = 1, z_1 = x, z_2 = x^2, \dots, z_p = x^p \quad (10.6.20)$$

The *multilinear regression* discussed earlier in this section results from (10.6.18) if we make the choices

$$z_0 = 1, z_1 = x_1, z_2 = x_2, \dots, z_p = x_p \quad (10.6.21)$$

The representation (10.6.18) is *linear* in the sense that it is *linear in the unknowns* $\{a_0, a_1, a_2, \dots, a_p\}$. Therefore, the nonlinear regressions discussed earlier, for example when $y = a_0 x^{a_1}$, does not fit the scheme being discussed.

As we have now done several times, assume we are given a data table

$y_{(1)}$	$y_{(2)}$	$y_{(3)}$.	.	.	$y_{(K)}$
$z_{(1)0}$	$z_{(2)0}$	$z_{(3)0}$.	.	.	$z_{(K)0}$
$z_{(1)1}$	$z_{(2)1}$	$z_{(3)1}$.	.	.	$z_{(K)1}$
$z_{(1)2}$	$z_{(2)2}$	$z_{(3)2}$.	.	.	$z_{(K)2}$
.	.	.				.
.	.	.				.
.	.	.				.
$z_{(1)p}$	$z_{(2)p}$	$z_{(3)p}$				$z_{(K)p}$

where

$$z_{(k)j} = k^{\text{th}} \text{ value of } j^{\text{th}} \text{ basis function, for } k = 1, \dots, K \text{ and } j = 0, 1, \dots, p \quad (10.6.22)$$

Of course, we are again assuming that $K > p + 1$. We define the matrices \mathbf{y} and \mathbf{z}_j for $j = 0, 1, \dots, p$ by the formulas

$$\mathbf{y} = \begin{bmatrix} y_{(1)} \\ y_{(2)} \\ y_{(3)} \\ \vdots \\ \vdots \\ y_{(K)} \end{bmatrix}, \quad \mathbf{z}_j = \begin{bmatrix} z_{(1)j} \\ z_{(2)j} \\ z_{(3)j} \\ \vdots \\ \vdots \\ z_{(K)j} \end{bmatrix} \quad \text{for } j = 0, 1, 2, \dots, p \quad (10.6.23)$$

We next represent the unknown coefficients by the matrix

$$\mathbf{c} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ \vdots \\ a_p \end{bmatrix} \quad (10.6.24)$$

In order to make the manipulations look like what we have done before, define the matrix A by

$$A = \begin{bmatrix} z_{(1)0} & z_{(1)1} & z_{(1)2} & z_{(1)3} & \cdot & \cdot & \cdot & z_{(1)p} \\ z_{(2)0} & z_{(2)1} & z_{(2)2} & z_{(2)3} & \cdot & \cdot & \cdot & z_{(2)p} \\ z_{(3)0} & z_{(3)1} & z_{(3)2} & z_{(3)3} & \cdot & \cdot & \cdot & z_{(3)p} \\ \vdots & \vdots & \vdots & \vdots & & & & \vdots \\ \vdots & \vdots & \vdots & \vdots & & & & \vdots \\ z_{(K)0} & z_{(K)1} & z_{(K)2} & z_{(K)3} & \cdot & \cdot & \cdot & z_{(K)p} \end{bmatrix} \quad (10.6.25)$$

Following the usual approach, we define the residual \mathbf{r} by the matrix equation

$$\mathbf{r} = \mathbf{y} - A\mathbf{c} \quad (10.6.26)$$

and, as done before, force

$$S_r = f(\mathbf{c}) \equiv \mathbf{r}^T \mathbf{r} = (\mathbf{y} - A\mathbf{c})^T (\mathbf{y} - A\mathbf{c}) \quad (10.6.27)$$

to be a minimum. The result is, again, that \mathbf{c} must be a solution of the normal equation

$$A^T A \mathbf{c} = A^T \mathbf{y} \quad (10.6.28)$$

Example 10.6.2: You are given the following table of data that represents the average monthly temperature in Fahrenheit in Norman, Oklahoma for each month for two years. The explicit table is

Month	Jan 2011	Feb 2011	Mar 2011	Apr 2011	May 2011	Jun 2011	Jul 2011	Aug 2011	Sept 2011	Oct 2011	Nov 2011	Dec 2011
Temp °F	35.65	41.33	53.94	65.23	68.98	84.74	90.45	89.18	72.01	62.61	50.62	40.94

Month	Jan 2012	Feb 2012	Mar 2012	Apr 2012	May 2012	Jun 2012	Jul 2012	Aug 2012	Sept 2012	Oct 2012	Nov 2011	Dec 2012
Temp °F	43.16	44.93	59.97	65.19	72.94	79.18	87.07	83.01	75.01	60.06	53.59	43.12

The goal is to obtain a formula that will estimate the temperature in each month in the twenty four month period based upon the trends displayed in the data. For simplicity, each day is assumed to be thirty days long and the average occurs at the middle of the month. The data in the above table displays the seasonal oscillations one would expect from the weather. In order to capture this oscillation, the following special case of (10.6.18) shall be adopted

$$y = a_0 z_0 + a_1 z_1 + a_2 z_2 = a_0 + a_1 \cos(\omega_0 t) + a_2 \sin(\omega_0 t) \quad (10.6.29)$$

where

$$\omega_0 = \frac{\pi}{180} \quad (10.6.30)$$

is the frequency and t is the number of days.

$$y = \text{Temperature}$$

$$z_0 = 1$$

$$z_1 = \cos(\omega_0 t)$$

$$z_2 = \sin(\omega_0 t)$$

From the data, the matrices $\mathbf{y}, \mathbf{z}_0, \mathbf{z}_1$ and \mathbf{z}_2 in (10.6.23) are given by

$$\begin{aligned}
 \mathbf{y} = & \begin{bmatrix} 35.65 \\ 41.33 \\ 53.94 \\ 65.29 \\ 68.98 \\ 84.74 \\ 90.94 \\ 89.18 \\ 72.01 \\ 62.61 \\ 50.62 \\ 40.94 \\ 43.16 \\ 44.93 \\ 59.97 \\ 65.19 \\ 72.94 \\ 79.18 \\ 87.07 \\ 83.01 \\ 75.01 \\ 60.06 \\ 53.59 \\ 43.12 \end{bmatrix}, \quad \mathbf{z}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{z}_1 = \begin{bmatrix} \cos\left(\frac{\pi}{180}15\right) \\ \cos\left(\frac{\pi}{180}45\right) \\ \cos\left(\frac{\pi}{180}75\right) \\ \cos\left(\frac{\pi}{180}105\right) \\ \cos\left(\frac{\pi}{180}135\right) \\ \cos\left(\frac{\pi}{180}165\right) \\ \cos\left(\frac{\pi}{180}225\right) \\ \cos\left(\frac{\pi}{180}255\right) \\ \cos\left(\frac{\pi}{180}285\right) \\ \cos\left(\frac{\pi}{180}315\right) \\ \cos\left(\frac{\pi}{180}345\right) \\ \cos\left(\frac{\pi}{180}375\right) \\ \cos\left(\frac{\pi}{180}405\right) \\ \cos\left(\frac{\pi}{180}435\right) \\ \cos\left(\frac{\pi}{180}465\right) \\ \cos\left(\frac{\pi}{180}495\right) \\ \cos\left(\frac{\pi}{180}525\right) \\ \cos\left(\frac{\pi}{180}555\right) \\ \cos\left(\frac{\pi}{180}585\right) \\ \cos\left(\frac{\pi}{180}615\right) \\ \cos\left(\frac{\pi}{180}645\right) \\ \cos\left(\frac{\pi}{180}675\right) \\ \cos\left(\frac{\pi}{180}705\right) \end{bmatrix}, \quad \mathbf{z}_2 = \begin{bmatrix} \sin\left(\frac{\pi}{180}15\right) \\ \sin\left(\frac{\pi}{180}45\right) \\ \sin\left(\frac{\pi}{180}75\right) \\ \sin\left(\frac{\pi}{180}105\right) \\ \sin\left(\frac{\pi}{180}135\right) \\ \sin\left(\frac{\pi}{180}165\right) \\ \sin\left(\frac{\pi}{180}225\right) \\ \sin\left(\frac{\pi}{180}255\right) \\ \sin\left(\frac{\pi}{180}285\right) \\ \sin\left(\frac{\pi}{180}315\right) \\ \sin\left(\frac{\pi}{180}345\right) \\ \sin\left(\frac{\pi}{180}375\right) \\ \sin\left(\frac{\pi}{180}405\right) \\ \sin\left(\frac{\pi}{180}435\right) \\ \sin\left(\frac{\pi}{180}465\right) \\ \sin\left(\frac{\pi}{180}495\right) \\ \sin\left(\frac{\pi}{180}525\right) \\ \sin\left(\frac{\pi}{180}555\right) \\ \sin\left(\frac{\pi}{180}585\right) \\ \sin\left(\frac{\pi}{180}615\right) \\ \sin\left(\frac{\pi}{180}645\right) \\ \sin\left(\frac{\pi}{180}675\right) \\ \sin\left(\frac{\pi}{180}705\right) \end{bmatrix}
 \end{aligned} \tag{10.6.31}$$

The MATLAB script

```

clc
clear
z0=ones(24,1)
days=[15,45,75,105,135,165,195,225,255,285,315,345,....
      375,405,435,465,495,525,555,585,615,645,675,705]'
```

$$z1=\cos(\pi*days/180)$$

$$z2=\sin(\pi*days/180)$$

$$y=[35.67,41.33,53.96,65.23,68.98,84.74,90.45,89.18,....$$

$$72.02,62.61,50.62,40.94,43.15,44.93,59.97,....$$

$$65.19,72.94,79.18,87.07,83.01,75.01,60.06,....$$

$$53.59,49.12]'$$

$$A=[z0,z1,z2]$$

%The direct solution of $A' * A * c = A' * y$ is

$$c=inv(A'*A)*A'*y$$

%Or capitalizing on the special properties of

%the left division

$$c=A\y$$

produces the output

```

c =

```

$$63.7063$$

$$-22.0566$$

$$-5.4177$$

Therefore, from (10.6.29) the temperature is given by

$$y = 63.7063 - 22.0566 \cos\left(\frac{\pi}{180}t\right) - 5.4177 \sin\left(\frac{\pi}{180}t\right) \quad (10.6.32)$$

By the sequence of rearrangements of (10.6.32)

$$\begin{aligned}
 y &= 63.7063 - 22.0566 \cos\left(\frac{\pi}{180}t\right) - 5.4177 \sin\left(\frac{\pi}{180}t\right) \\
 &= 63.7063 + \sqrt{(22.0566)^2 + (5.4177)^2} \\
 &\quad \times \begin{cases} -\frac{22.0566}{\sqrt{(22.0566)^2 + (5.4177)^2}} \cos\left(\frac{\pi}{180}t\right) \\ -\frac{5.4177}{\sqrt{(22.0566)^2 + (5.4177)^2}} \sin\left(\frac{\pi}{180}t\right) \end{cases} \\
 &= 63.7063 + 22.7122 \cos\left(\frac{\pi}{180}t + \theta\right) \\
 &= 63.7063 + 22.7122 \cos\left(\frac{\pi}{180}t + 2.9077\right) \\
 &= 63.7063 + 22.7122 \cos\left(\frac{\pi}{180}(t + 193.8001)\right)
 \end{aligned} \tag{10.6.33}$$

Equation (10.6.33) displays a temperature oscillation about a mean of 63.7063 with an amplitude of 22.7122. The factor $(t + 193.8001)$ displays a phase shift that shows that the maximum temperature occurs approximately 164 days prior to the start of the year.

If we modify the script above to

```

clc
clear
z0=ones(24,1)
days=[15,45,75,105,135,165,195,225,255,285,315,345,....
      375,405,435,465,495,525,555,585,615,645,675,705]';
z1=cos(2*pi*days/360)
z2=sin(2*pi*days/360)
y=[35.67,41.33,53.96,65.23,68.98,84.74,90.45,89.18,....
    72.02,62.61,50.62,40.94,43.15,44.93,59.97,....
    65.19,72.94,79.18,87.07,83.01,75.01,60.06,....
    53.59,49.12]'
A=[z0,z1,z2]
%The direct solution of A'*A*c=A'*y is
c=inv(A'*A)*A'*y
%Or capitalizing on the special properties of
%the left division
c=A\y
plot(days,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('Day'),ylabel('Temperature')

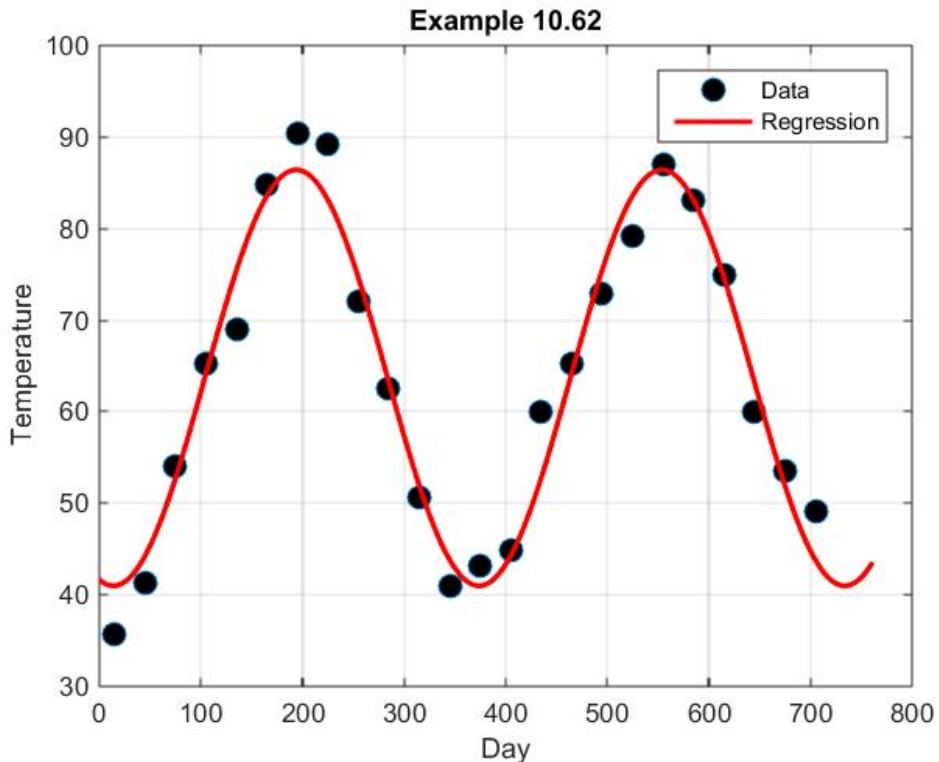
```

```

grid on
hold
xvalues=[0:1:760];
yvalues=c(1)+c(2)*cos(2*pi*xvalues/360)+...
    c(3)*sin(2*pi*xvalues/360)
plot(xvalues,yvalues,'r','LineWidth',2)
legend('Data','Regression')
title('Exercise 10.62')

```

generates the above answer and the plot



Exercises:

10.6.1: You are given the following two dimensional table of data:

y	14	21	11	12	23	23	14	6	11
x_1	0	0	1	2	0	1	2	2	1
x_2	0	2	2	4	4	6	6	2	1

Utilize the methods discussed in this section and fit this data to a surface of the form

$$y = a_0 + a_1 x_1 + a_2 x_2 \quad (10.6.34)$$

10.6.2: You are given information on a function $f(x)$ in the form of the following table:

x	100.00	120.00	140.00	160.00	180.00
$f(x)$	14.9340	14.5302	14.1646	14.8336	14.5333

Use the information in this table and fit the data to an equation of the form

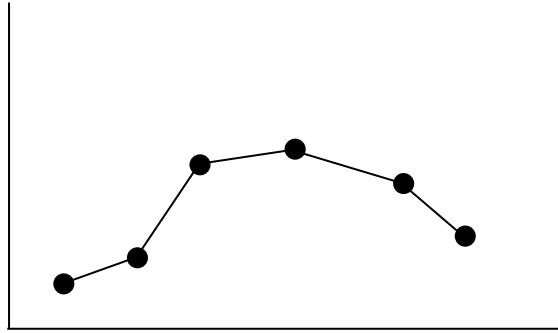
$$y = f(x) = a_0 + a_1 x + \frac{a_2}{x} + a_3 \log x \quad (10.6.35)$$

Note that (10.6.35) is an equation of the form (10.6.18).

Chapter 11

INTERPOLATION

In Chapter 4 and at the beginning of Section 10.1, we mentioned two kinds of curve fits. The first case was when the data exhibits scatter and the objective is to capture its general trend. This problem caused us to study least square regression. The second case was when the data is known to be precise and we wish to estimate values between given data points. In other words, the derived curve is forced to pass through every data point. We used the figure



to illustrate this case, and we identified the problem as one of *data interpolation*. This Chapter is concerned with certain aspects of how one actually implements the interpolation in a systematic fashion.

Section 11.1. Linear Interpolation

We shall first consider the simplest kind of interpolation. We shall begin by assuming we are given the two point data set

y_1	y_2
x_1	x_2

where $x_2 \neq x_1$ and without loss of generality $x_2 > x_1$. A *linear interpolation* for this data is an equation (first order polynomial) of the form

$$f_1(x) = a_0 + a_1 x \quad (11.1.1)$$

We can calculate the coefficients in this first order polynomial by use of the data to derive the following two equations and for the two unknowns:

$$\begin{aligned} y_1 &= a_0 + a_1 x_1 \\ y_2 &= a_0 + a_1 x_2 \end{aligned} \Rightarrow \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \quad (11.1.2)$$

If the inversion formula (1.10.66) is used, the solution of these two equations for the coefficients a_1 and a_2 is

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \frac{1}{x_2 - x_1} \begin{bmatrix} x_2 & -x_1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \quad (11.1.3)$$

Therefore,

$$a_0 = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1} \quad (11.1.4)$$

and

$$a_1 = \frac{y_2 - y_1}{x_2 - x_1} \quad (11.1.5)$$

These results allow us to write the linear interpolation formula (11.1.1) as

$$\begin{aligned} f_1(x) &= a_0 + a_1 x = [1 \ x] \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = [1 \ x] \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \\ &= \frac{1}{x_2 - x_1} [1 \ x] \begin{bmatrix} x_2 & -x_1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \\ &= \frac{1}{x_2 - x_1} [1 \ x] \begin{bmatrix} x_2 y_1 - x_1 y_2 \\ y_1 - y_2 \end{bmatrix} \end{aligned} \quad (11.1.6)$$

If the matrix multiplication in (11.1.6) are implemented, the values at the points intermediate to the points x_1 and x_2 are given by

$$f_1(x) = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1} + \frac{y_2 - y_1}{x_2 - x_1} x \quad (11.1.7)$$

There are two *rearrangements* of (11.1.7) that illustrate forms for the interpolated values that have numerical advantages in more complicated cases. They are

$$f_1(x) = y_1 + \frac{y_2 - y_1}{x_2 - x_1} (x - x_1) \quad (11.1.8)$$

and

$$f_1(x) = \frac{x - x_2}{x_1 - x_2} y_1 + \frac{x - x_1}{x_2 - x_1} y_2 \quad (11.1.9)$$

Equations (11.1.7), (11.1.8) and (11.1.9) are simply different ways to write the *same* linear interpolation between the points x_1 and x_2 .

- Equation (11.1.7) is a simply polynomial representation of the interpolation.
- Equation (11.1.8) is known a *Newton* polynomial.
 - It is a polynomial where the independent variable is $(x - x_1)$, the distance from the given x_1 .
- The form (11.1.9) is known as a *Lagrange* polynomial. Its distinguishing feature is its special dependence on the values of the data, i.e., on y_1 and y_2 .

A fundamentally important point is that for complicated interpolations, i.e., those involving more data points, equations in the forms (11.1.8) and (11.1.9) have computational advantages over the equivalent analytical expression (11.1.7).

Section 11.2. Polynomial Interpolation

The analytical simplicity of the straight line interpolation discussed in Section 11.1 is lost if one has additional data pairs. If we are given the $N + 1$ data set $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$, where $x_1 < x_2 < \dots < x_{N+1}$, we could attempt to interpolate between these points with a polynomial of degree N as follows:

$$f_N(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots + a_{N-1}x^{N-1} + a_Nx^N \quad (11.2.1)$$

The polynomial (11.2.1) is called the *interpolating polynomial* and the points x_1, x_2, \dots, x_{N+1} are the *interpolation points* or *interpolation nodes*. As we have explained several times, the MATLAB convention for polynomials would write (11.2.1) as, (10.5.19), repeated,

$$f_N(x) = p_1x^N + p_2x^{N-1} + \dots + p_Nx + p_{N+1} \quad (11.2.2)$$

In some cases we will adopt the convention (11.2.2) in order to capitalize on special tools within MATLAB.

Our objective in this Section is to show that given the data set $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$, we can construct a unique polynomial in the form of (11.2.1) or its equivalent that obeys $f_N(x_j) = y_j$ for $j = 1, 2, \dots, N + 1$.

In Section (2.1), we introduced the vector space \mathcal{P}_N of polynomials of degree less than or equal to N . In Section 2.5, we showed that \mathcal{P}_N is a finite dimensional vector space of dimension $N + 1$. These elementary ideas are useful as we manipulate the polynomial (11.2.1) or, equivalently, (11.2.2) into different forms. Viewed as an element in \mathcal{P}_N , the polynomial f_N in the formula (11.2.1) is a representation of $f_N \in \mathcal{P}_N$ with respect to the basis $\{1, x, x^2, \dots, x^N\}$. For an arbitrary basis $\{q_1, q_2, \dots, q_{N+1}\}$ of \mathcal{P}_N , a polynomial $f_N \in \mathcal{P}_N$ has the kind of representation discussed in Section 2.6, namely,

$$f_N(x) = \sum_{j=1}^{N+1} c_j q_j(x) \quad (11.2.3)$$

Examples 2.6.2 and 2.6.3 gave three examples of bases for the four dimensional vector space of third order polynomials \mathcal{P}_3 . The first example is a special case of (11.2.1) and corresponds to what is called a *monomial basis*. This basis is

$$\begin{aligned}
 m_1(x) &= 1 \\
 m_2(x) &= x \\
 m_3(x) &= x^2 \\
 m_4(x) &= x^3
 \end{aligned} \tag{11.2.4}$$

The next example corresponds to a *Newton basis*. The four dimensional version of this basis is

$$\begin{aligned}
 n_1(x) &= 1 \\
 n_2(x) &= x - a \\
 n_3(x) &= (x - a)(x - b) \\
 n_4(x) &= (x - a)(x - b)(x - c)
 \end{aligned} \tag{11.2.5}$$

where a, b, c are real numbers, and the third example corresponds to a *Lagrange basis*. The four dimensional version of a Lagrange basis is

$$\begin{aligned}
 l_1(x) &= \frac{(x - b)(x - c)(x - d)}{(a - b)(a - c)(a - d)} \\
 l_2(x) &= \frac{(x - a)(x - c)(x - d)}{(b - a)(b - c)(b - d)} \\
 l_3(x) &= \frac{(x - a)(x - b)(x - d)}{(c - a)(c - b)(c - d)} \\
 l_4(x) &= \frac{(x - a)(x - b)(x - c)}{(d - a)(d - b)(d - c)}
 \end{aligned} \tag{11.2.6}$$

where a, b, c, d are distinct real numbers.

Our immediate goal in this Section is to utilize the data set above to calculate the coefficients in (11.2.3) or its special case (11.2.1). The result of this calculation is a particular polynomial or, in the language of vectors, an element of the vector space \mathcal{P}_N . Our calculation scheme will establish the existence and uniqueness of the polynomial in \mathcal{P}_N that obeys

$$f_N(x_j) = y_j \text{ for } j = 1, 2, \dots, N + 1.$$

The calculation scheme for the polynomial f_N involves evaluating the polynomial (11.2.3) at the $N + 1$ data points $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$. The result can be written

$$\begin{aligned}
c^1 q_1(x_1) + c^2 q_2(x_1) + c^3 q_3(x_1) + \cdots + c^{N+1} q_{N+1}(x_1) &= y_1 \\
c^1 q_1(x_2) + c^2 q_2(x_2) + c^3 q_3(x_2) + \cdots + c^{N+1} q_{N+1}(x_2) &= y_2 \\
c^1 q_1(x_3) + c^2 q_2(x_3) + c^3 q_3(x_3) + \cdots + c^{N+1} q_{N+1}(x_3) &= y_3 \\
c^1 q_1(x_4) + c^2 q_2(x_4) + c^3 q_3(x_4) + \cdots + c^{N+1} q_{N+1}(x_4) &= y_4 \\
&\vdots \\
&\vdots \\
c^1 q_1(x_{N+1}) + c^2 q_2(x_{N+1}) + c^3 q_3(x_{N+1}) + \cdots + c^{N+1} q_{N+1}(x_{N+1}) &= y_{N+1}
\end{aligned} \tag{11.2.7}$$

The question of whether or not the polynomial exists and is unique comes down to the question of whether or not equation (11.2.7) has a unique solution for the coefficients $\{c^1, c^2, c^3, \dots, c^{N+1}\}$.

As we shall see in examples later in this section, the choice of the basis $\{q_1, q_2, \dots, q_{N+1}\}$ will influence how easily (11.2.7) can be solved. The question of whether or not (11.2.7) has a unique solution is answered if we can establish that the matrix

$$A = \begin{bmatrix} q_1(x_1) & q_2(x_1) & q_3(x_1) & \cdot & \cdot & \cdot & q_N(x_1) & q_{N+1}(x_1) \\ q_1(x_2) & q_2(x_2) & q_3(x_2) & \cdot & \cdot & \cdot & q_N(x_2) & q_{N+1}(x_2) \\ q_1(x_3) & q_2(x_3) & q_3(x_3) & \cdot & \cdot & \cdot & q_N(x_3) & q_{N+1}(x_3) \\ q_1(x_4) & q_2(x_4) & q_3(x_4) & \cdot & \cdot & \cdot & q_N(x_4) & q_{N+1}(x_4) \\ \cdot & \cdot & \cdot & \cdot & & & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & & & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & & & \cdot & \cdot \\ q_1(x_{N+1}) & q_2(x_{N+1}) & q_3(x_{N+1}) & \cdot & \cdot & \cdot & q_N(x_{N+1}) & q_{N+1}(x_{N+1}) \end{bmatrix} \tag{11.2.8}$$

is nonsingular. The proof that A is nonsingular involves two steps. The first step involves establishing it is nonsingular for the *monomial* basis choice

$$q_j(x) = x^{j-1} \quad \text{for } j = 1, 2, \dots, N+1 \tag{11.2.9}$$

and then transforming, by a basis change, the result to the case of an arbitrary basis. Equation (11.2.9) generalizes (11.2.4) for the case where $\dim \mathcal{P}_N = N+1$. With the choice, (11.2.9), equation (11.2.8) reduces to

$$A_M = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdot & \cdot & \cdot & x_1^{N-1} & x_1^N \\ 1 & x_2 & x_2^2 & \cdot & \cdot & \cdot & x_2^{N-1} & x_2^N \\ 1 & x_3 & x_3^2 & \cdot & \cdot & \cdot & x_3^{N-1} & x_3^N \\ 1 & x_4 & x_4^2 & \cdot & \cdot & \cdot & x_4^{N-1} & x_4^N \\ \cdot & \cdot & \cdot & & & & & \\ \cdot & \cdot & \cdot & & & & & \\ \cdot & \cdot & \cdot & & & & & \\ 1 & x_{N+1} & x_{N+1}^2 & \cdot & \cdot & \cdot & x_{N+1}^{N-1} & x_{N+1}^N \end{bmatrix} \quad (11.2.10)$$

The $(N+1) \times (N+1)$ matrix (11.2.10) is the transpose of the Vandermonde matrix introduced in Section 1.10. Because of the relationship (1.10.21), the determinant of (11.2.10) is given by (1.10.33). In terms of the notation used in (11.2.10), the determinant is

$$\det A_M = \prod_{\substack{i,j=1 \\ i>j}}^{N+1} (x_i - x_j) \quad (11.2.11)$$

Because we have required that $x_1 < x_2 < \dots < x_{N+1}$, it follows from (11.2.11) that the matrix A given in (11.2.10) has a nonzero determinant and, thus, is nonsingular. This fact insures that (11.2.7) has a solution for the basis (11.2.9). From our discussions in Sections 1.11 and 2.7, the solution is unique. If we now implement a change of basis, expressed in the form (2.6.3),

$$\hat{q}_j(x) = \sum_{k=1}^{N+1} T_j^k q_k(x) \quad (11.2.12)$$

it is readily established that

$$\begin{bmatrix}
\hat{q}_1(x_1) & \hat{q}_2(x_1) & \hat{q}_3(x_1) & \cdots & \hat{q}_N(x_1) & \hat{q}_{N+1}(x_1) \\
\hat{q}_1(x_2) & \hat{q}_2(x_2) & \hat{q}_3(x_2) & \cdots & \hat{q}_N(x_2) & \hat{q}_{N+1}(x_2) \\
\hat{q}_1(x_3) & \hat{q}_2(x_3) & \hat{q}_3(x_3) & \cdots & \hat{q}_N(x_3) & \hat{q}_{N+1}(x_3) \\
\hat{q}_1(x_4) & \hat{q}_2(x_4) & \hat{q}_3(x_4) & \cdots & \hat{q}_N(x_4) & \hat{q}_{N+1}(x_4) \\
\vdots & \vdots & \vdots & & & \\
\hat{q}_1(x_{N+1}) & \hat{q}_2(x_{N+1}) & \hat{q}_3(x_{N+1}) & \cdots & \hat{q}_N(x_{N+1}) & \hat{q}_{N+1}(x_{N+1})
\end{bmatrix} = \\
\begin{bmatrix}
1 & x_1 & x_1^2 & \cdots & \cdots & x_1^{N-1} & x_1^N \\
1 & x_2 & x_2^2 & \cdots & \cdots & x_2^{N-1} & x_2^N \\
1 & x_3 & x_3^2 & \cdots & \cdots & x_3^{N-1} & x_3^N \\
1 & x_4 & x_4^2 & \cdots & \cdots & x_4^{N-1} & x_4^N \\
\vdots & \vdots & \vdots & & & & \\
\vdots & \vdots & \vdots & & & & \\
1 & x_{N+1} & x_{N+1}^2 & \cdots & \cdots & x_{N+1}^{N-1} & x_{N+1}^N
\end{bmatrix} T \quad (11.2.13)$$

where T is the transition matrix, as defined by (2.6.9)₁, associated with the change of basis $(1, x, x^2, \dots, x^N) \rightarrow (\hat{q}_1(x), \hat{q}_2(x), \hat{q}_3(x), \dots, \hat{q}_{N+1}(x))$. Two important choices of the basis $(\hat{q}_1(x), \hat{q}_2(x), \hat{q}_3(x), \dots, \hat{q}_{N+1}(x))$ are illustrated in the case $N = 3$ by (11.2.5) and (11.2.6). Because of the identity, (1.10.67), it follows from (11.2.13) that

$$\det \begin{bmatrix}
\hat{q}_1(x_1) & \hat{q}_2(x_1) & \hat{q}_3(x_1) & \cdots & \cdots & \hat{q}_N(x_1) & \hat{q}_{N+1}(x_1) \\
\hat{q}_1(x_2) & \hat{q}_2(x_2) & \hat{q}_3(x_2) & \cdots & \cdots & \hat{q}_N(x_2) & \hat{q}_{N+1}(x_2) \\
\hat{q}_1(x_3) & \hat{q}_2(x_3) & \hat{q}_3(x_3) & \cdots & \cdots & \hat{q}_N(x_3) & \hat{q}_{N+1}(x_3) \\
\hat{q}_1(x_4) & \hat{q}_2(x_4) & \hat{q}_3(x_4) & \cdots & \cdots & \hat{q}_N(x_4) & \hat{q}_{N+1}(x_4) \\
\vdots & \vdots & \vdots & & & \vdots & \vdots \\
\vdots & \vdots & \vdots & & & \vdots & \vdots \\
\hat{q}_1(x_{N+1}) & \hat{q}_2(x_{N+1}) & \hat{q}_3(x_{N+1}) & \cdots & \cdots & \hat{q}_N(x_{N+1}) & \hat{q}_{N+1}(x_{N+1})
\end{bmatrix} \neq 0 \quad (11.2.14)$$

and the coefficient matrix for the system (11.2.7) is nonsingular for an arbitrary basis of \mathcal{P}_{N+1} .

Given that the matrix (11.2.8) is nonsingular, the polynomial that interpolates the given data is, from (11.2.3),

$$f_N(x) = \sum_{j=1}^{N+1} c_j q_j(x) = [y_1 \quad y_2 \quad y_3 \quad \dots \quad y_N \quad y_{N+1}] A^{-T} \begin{bmatrix} q_1(x) \\ q_2(x) \\ q_3(x) \\ \vdots \\ \vdots \\ q_N(x) \\ q_{N+1}(x) \end{bmatrix} \quad (11.2.15)$$

The result (11.2.15) gives the interpolating polynomial for any choice of basis $\{q_1, q_2, \dots, q_{N+1}\}$. At this point in the discussion the choice of basis for \mathcal{P}_{N+1} seems completely arbitrary. To an extent, the arbitrariness is true. However, we shall see there are numerical advantages of certain polynomials over others. The examples below as well as the discussion in Section 11.9 will illustrate certain of these advantages.

The following examples illustrate three calculations for the creation of a polynomial that interpolates our data for the case $N + 1 = 4$. We shall also indicate how these results are generalized for polynomials of higher order.

Example 11.2.1: For the case where we are given the data set $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$, we can use (11.2.4) and define a *monomial* basis for \mathcal{P}_3

$$\begin{aligned} m_1(x) &= 1 \\ m_2(x) &= x \\ m_3(x) &= x^2 \\ m_4(x) &= x^3 \end{aligned} \quad (11.2.16)$$

For this choice of basis, the matrix (11.2.8) reduces to

$$A_M = \begin{bmatrix} m_1(x_1) & m_2(x_1) & m_3(x_1) & m_4(x_1) \\ m_1(x_2) & m_2(x_2) & m_3(x_2) & m_4(x_2) \\ m_1(x_3) & m_2(x_3) & m_3(x_3) & m_4(x_3) \\ m_1(x_4) & m_2(x_4) & m_3(x_4) & m_4(x_4) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \end{bmatrix} \quad (11.2.17)$$

which, or course, is a special case of (11.2.10). As the transpose of a 4×4 Vandermonde matrix, from (1.10.61) its inverse is given by

$$A_M^{-1} = \frac{\text{adj } A_M}{\det A_M} \quad (11.2.18)$$

where the determinant is given by (11.2.11) and the adjugate of A is given by the definition in Section 1.10. This definition was repeated in Section 7.1. These two formulas can be shown to combine to yield

$$A_M^{-1} = \begin{bmatrix} -\frac{x_2x_3x_4}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & -\frac{x_1x_3x_4}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & -\frac{x_1x_2x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & -\frac{x_1x_2x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ \frac{x_3x_4+x_2x_4+x_2x_3}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & \frac{x_1x_4+x_3x_4+x_1x_3}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & \frac{x_1x_2+x_1x_4+x_2x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & \frac{x_1x_2+x_1x_3+x_2x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ -\frac{x_2+x_3+x_4}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & -\frac{x_1+x_3+x_4}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & -\frac{x_1+x_2+x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & -\frac{x_1+x_2+x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ \frac{1}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & \frac{1}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & \frac{1}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & \frac{1}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \end{bmatrix} \quad (11.2.19)$$

Equation (11.2.19) can be derived by utilizing the symbolic manipulator features of MATLAB. Example 7.1.5 performs the same calculation except the matrix has dimension 3 and it is transposed. Given (11.2.19), then the solution of (11.2.7) is

$$\begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} = \begin{bmatrix} -\frac{x_2x_3x_4}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & -\frac{x_1x_3x_4}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & -\frac{x_1x_2x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & -\frac{x_1x_2x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ \frac{x_3x_4+x_2x_4+x_2x_3}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & \frac{x_1x_4+x_3x_4+x_1x_3}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & \frac{x_1x_2+x_1x_4+x_2x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & \frac{x_1x_2+x_1x_3+x_2x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ -\frac{x_2+x_3+x_4}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & -\frac{x_1+x_3+x_4}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & -\frac{x_1+x_2+x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & -\frac{x_1+x_2+x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ \frac{1}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & \frac{1}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & \frac{1}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & \frac{1}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \quad (11.2.20)$$

Of course, the complexity of (11.2.20) makes it less than useful. As one would anticipate when interpolations are implemented utilizing the basis (11.2.16), a computational tool such as MATLAB is essential to have available. There are other problems with this choice of basis that we shall discuss in Section 11.9. Having made this point, the following example illustrates this type of interpolation.

Example 11.2.2: As a simple illustration of an interpolation based upon the formulas of Example 11.2.1, consider the following data set for $N + 1 = 4$

x	-2	-1	1	2
y	-5	0	1	6

If these values are used in the above table, we obtain from (11.2.17)

$$A_M = \begin{bmatrix} 1 & -2 & 4 & -8 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \quad (11.2.21)$$

The inverse of the matrix in (11.2.21) is

$$A_M^{-1} = \frac{1}{12} \begin{bmatrix} -2 & 8 & 8 & -2 \\ 1 & -8 & 8 & -1 \\ 2 & -2 & 2 & 2 \\ -1 & 2 & -2 & 1 \end{bmatrix} \quad (11.2.22)$$

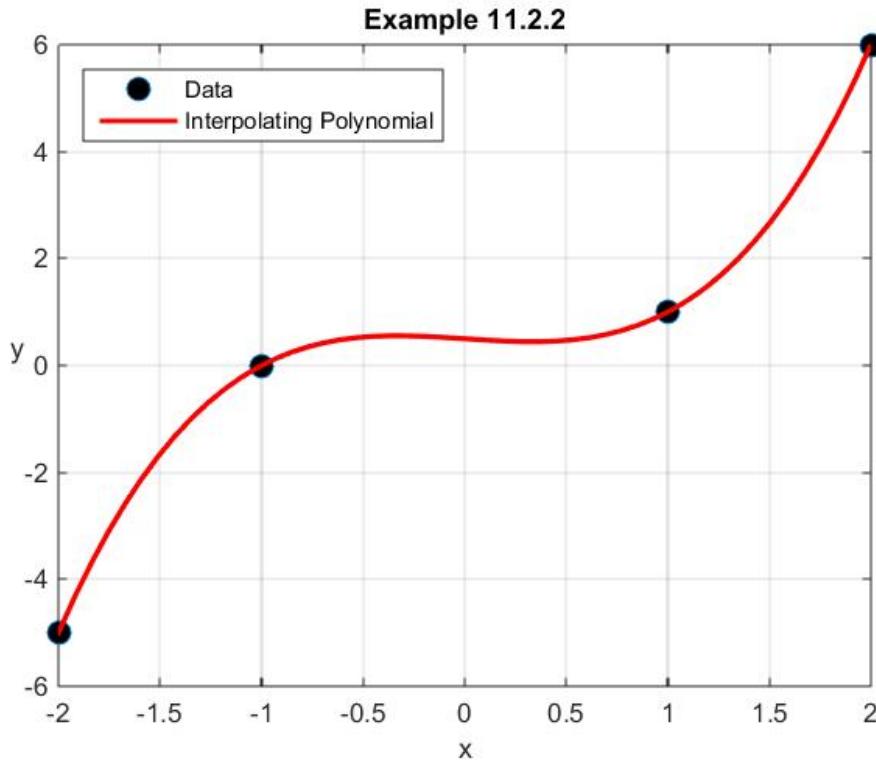
It follows by (11.2.20) and (11.2.22) that

$$\begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} = \frac{1}{12} \begin{bmatrix} -2 & 8 & 8 & -2 \\ 1 & -8 & 8 & -1 \\ 2 & -2 & 2 & 2 \\ -1 & 2 & -2 & 1 \end{bmatrix} \begin{bmatrix} -5 \\ 0 \\ 1 \\ 6 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{4} \\ 0 \\ \frac{3}{4} \end{bmatrix} \quad (11.2.23)$$

and the polynomial that interpolates the above data is, from (11.2.3) and (11.2.16),

$$f_3(x) = \frac{1}{2} - \frac{1}{4}x + \frac{3}{4}x^3 \quad (11.2.24)$$

The data and the interpolating polynomial are displayed on the following figure.



The following MATLAB script will generate the result (11.2.23) and create the above figure¹

```

clc
clear
x=sym([-2,-1,1,2])
y=sym([-5,0,1,6])
A=[x.^0;x.^1;x.^2;x.^3]'
```

`c=A\y'`

```

plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
grid on
hold on
%Construct the polynomial and plot it at 50 points in the
%interval (-2,2)
xvalues=linspace(-2,2,50);
%Use polyval to assign values at 50 points.
%Order entries in c to fit MATLAB format. Call it p
p=zeros(1,4)'; %Preallocate
p(:)=c(4:-1:1)
yvalues=polyval(p,xvalues)
plot(xvalues,yvalues,'r','LineWidth',2)
```

¹ Because of the simplicity of the numbers in the data table, the data were entered as symbols in the MATLAB script. This resulted in the answers in (11.2.23) being given in terms of rational numbers.

```
legend('Data','Interpolating Polynomial',...  
      'Location','NorthWest')  
title('Example 11.2.2')
```

In the above script, the MATLAB function **polyval** was used to calculate the numerical values of the polynomial at the points in the interval $(0, 2)$.² This function was introduced in Section 9.6. These values can also be calculated from (11.2.3) and the replacement of the three lines of script

```
p=zeros(1,4)'; %Preallocate
p(:)=c(4:-1:1)
yvalues=polyval(p,xvalues)
```

by

```
c=double(c)
yvalues=c'*[xvalues.^0;xvalues.^1;xvalues.^2;xvalues.^3]
```

The first line converts the symbolic form of c to its double precision form. The second line is simply a direct substitution into the polynomial (11.2.24). A computationally more efficient way than this direct substitution into the polynomial is to adopt what is known as the *Horner Method* and write the polynomial (11.2.1) as³

$$f_N(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_{N-1}x^{N-1} + a_Nx^N$$

$$= a_0 + x \left(a_1 + x \left(a_2 + x \left(a_3 + \cdots + \underbrace{\left(a_{N-3} + x \left(a_{N-2} + x \left(\underbrace{a_{N-1} + a_N x}_{1} \right)} \right) \cdots \right)}_{2} \right) \right) \right) \underbrace{\cdots}_{3} \underbrace{\cdots}_{N-3} \underbrace{\cdots}_{N-2} \underbrace{\cdots}_{N-1} \quad (11.2.25)$$

It is possible to show that the evaluation of the monomial form of the polynomial $f_N(x)$,

equation (11.2.25)₁, requires $\frac{N(N+1)}{2}$ multiplications and N additions. The evaluation of the

² An interesting discussion of how **polyval** works can be found at <http://blogs.mathworks.com/loren/2009/07/08/a-brief-history-of-polyval/#4>.

³ Information about the British mathematician William George Horner can be found at http://en.wikipedia.org/wiki/William_George_Horner.

polynomial written in the form (11.2.25)₂ requires N multiplications and N additions.⁴ The MATLAB script

```
c=double(c)
yvalues=c(4)
for j=3:-1:1
    yvalues=c(j)+yvalues.*xvalues
end
```

can be used to replace the script above if one wants to use the Horner method.

Example 11.2.3: In this example, we continue with the data set

$\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$. We shall adopt the polynomials in (11.2.5) and define a Newton basis for \mathcal{P}_3 by the polynomials

$$\begin{aligned} n_1(x) &= 1 \\ n_2(x) &= x - x_1 \\ n_3(x) &= (x - x_1)(x - x_2) \\ n_4(x) &= (x - x_1)(x - x_2)(x - x_3) \end{aligned} \tag{11.2.26}$$

With the choice (11.2.26), the matrix (11.2.8) reduces to

$$A_N = \begin{bmatrix} n_1(x_1) & n_2(x_1) & n_3(x_1) & n_4(x_1) \\ n_1(x_2) & n_2(x_2) & n_3(x_2) & n_4(x_2) \\ n_1(x_3) & n_2(x_3) & n_3(x_3) & n_4(x_3) \\ n_1(x_4) & n_2(x_4) & n_3(x_4) & n_4(x_4) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & x_2 - x_1 & 0 & 0 \\ 1 & x_3 - x_1 & (x_3 - x_1)(x_3 - x_2) & 0 \\ 1 & x_4 - x_1 & (x_4 - x_1)(x_4 - x_2) & (x_4 - x_1)(x_4 - x_2)(x_4 - x_3) \end{bmatrix} \tag{11.2.27}$$

The inverse of (11.2.27) can be constructed utilizing the symbolic manipulator in MATLAB. Example 7.1.5 can be modified to construct the inverse. The fact that (11.2.27) is a lower triangular matrix also makes the analytical determination of the inverse rather easy.⁵ In any case, it is easily shown that

⁴ See, for example, http://en.wikipedia.org/wiki/Horner%27s_method.

⁵ For example, the method described in Section 1.6 could be utilized where one starts with the augmented matrix $(A_N | I)$ and performs forward elimination row operations until the result $(I | A_N^{-1})$ is obtained. The lower diagonal form of A_N makes the row operations straight forward to implement.

$$A_N^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{(x_2 - x_1)} & \frac{1}{(x_2 - x_1)} & 0 & 0 \\ \frac{1}{(x_2 - x_1)(x_3 - x_1)} & -\frac{1}{(x_2 - x_1)(x_3 - x_2)} & \frac{1}{(x_3 - x_1)(x_3 - x_2)} & 0 \\ -\frac{1}{(x_2 - x_1)(x_3 - x_1)(x_4 - x_1)} & \frac{1}{(x_2 - x_1)(x_3 - x_2)(x_4 - x_2)} & -\frac{1}{(x_3 - x_1)(x_3 - x_2)(x_4 - x_3)} & \frac{1}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \end{bmatrix} \quad (11.2.28)$$

While not obvious at this point, the solution of (11.2.7) in this case can be arranged such that it depends on the points y_1, y_2, y_3 and y_4 in the combination $y_1, y_2 - y_1, y_3 - y_2$ and $y_4 - y_3$. We shall force this feature by the introduction of the identity

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 - y_1 \\ y_3 - y_2 \\ y_4 - y_3 \end{bmatrix} \quad (11.2.29)$$

Given (11.2.28), it is not difficult to show that

$$A_N^{-1} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{x_2 - x_1} & 0 & 0 \\ 0 & -\frac{1}{(x_3 - x_1)(x_2 - x_1)} & \frac{1}{(x_3 - x_1)(x_3 - x_2)} & 0 \\ 0 & \frac{1}{(x_4 - x_1)(x_3 - x_1)(x_2 - x_1)} & \left(\frac{1}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \right) & \frac{1}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \end{bmatrix} \quad (11.2.30)$$

The (4,3) element of (11.2.30) can be rearranged and the result (11.2.30) written

$$A_N^{-1} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{x_2 - x_1} & 0 & 0 \\ 0 & -\frac{1}{(x_3 - x_1)(x_2 - x_1)} & \frac{1}{(x_3 - x_1)(x_3 - x_2)} & 0 \\ 0 & \frac{1}{(x_4 - x_1)(x_3 - x_1)(x_2 - x_1)} & -\frac{1}{(x_4 - x_1)(x_3 - x_2)} \left(\frac{1}{x_4 - x_2} + \frac{1}{x_3 - x_1} \right) & \frac{1}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \end{bmatrix} \quad (11.2.31)$$

Given (11.2.31) and (11.2.29), the solution of (11.2.7) in this case is

$$\begin{aligned} \begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{x_2 - x_1} & 0 & 0 \\ 0 & -\frac{1}{(x_3 - x_1)(x_2 - x_1)} & \frac{1}{(x_3 - x_1)(x_3 - x_2)} & 0 \\ 0 & \frac{1}{(x_4 - x_1)(x_3 - x_1)(x_2 - x_1)} & -\frac{1}{(x_4 - x_1)(x_3 - x_2)} \left(\frac{1}{x_4 - x_2} + \frac{1}{x_3 - x_1} \right) & \frac{1}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 - y_1 \\ y_3 - y_2 \\ y_4 - y_3 \end{bmatrix} \\ &= \begin{bmatrix} y_1 \\ \frac{y_2 - y_1}{x_2 - x_1} \\ \frac{1}{x_3 - x_1} \left(\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1} \right) \\ \left(\frac{1}{x_4 - x_1} \right) \left(\frac{1}{x_4 - x_2} \left(\frac{y_4 - y_3}{x_4 - x_3} - \frac{y_3 - y_2}{x_3 - x_2} \right) - \frac{1}{x_3 - x_1} \left(\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1} \right) \right) \end{bmatrix} \quad (11.2.32) \end{aligned}$$

The final result (11.2.32) yields the coefficients of the polynomial (11.2.3) for the case of a Newton polynomial basis. Therefore,

$$\begin{aligned} f_3(x) &= c^1 n_1(x) + c^2 n_2(x) + c^3 n_3(x) + c^4 n_4(x) \\ &= c^1 + c^2 (x - x_1) + c^3 (x - x_1)(x - x_2) + c^4 (x - x_1)(x - x_2)(x - x_3) \end{aligned} \quad (11.2.33)$$

Unlike the results in Example 11.2.1, the simplicity of the result (11.2.32) has computational advantages. In the following section, MATLAB will be used to implement interpolation utilizing the Newton basis.

An important feature of Newton polynomials is illustrated by (11.2.33). The polynomial coefficient c^1 is determined by y_1 , the coefficient c^2 by $\{(x_1, y_1), (x_2, y_2)\}$, the coefficient c^3 by $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ and the coefficient c^4 by $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$. As a result, if, for example, a second order interpolating polynomial is calculated by the data set $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$, the interpolating polynomial is

$$f_3(x) = c^1 n_1(x) + c^2 n_2(x) + c^3 n_3(x) = c^1 + c^2 (x - x_1) + c^3 (x - x_1)(x - x_2) \quad (11.2.34)$$

The addition of the additional pair (x_4, y_4) to the data set produces the interpolating polynomial

$$f_4(x) = f_3(x) + c^4 (x - x_1)(x - x_2)(x - x_3) \quad (11.2.35)$$

without a recalculation of the second order interpolating polynomial (11.2.34).

There are numerical advantages of interpolations with Newton polynomials vs. interpolations with monomial polynomials that will be illustrated in Section 11.9.

The generalization of (11.2.32) and (11.2.33) for Newton polynomials of higher order appears at first look to be complicated. Actually, it is not too difficult. The key to the generalization is to recognize the answer (11.2.32) in terms of *divided differences*. It is customary in the discussion of Newton interpolation to introduce a special notation and write (11.2.32) as

$$\begin{aligned} \begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} &= \begin{bmatrix} y_1 \\ \frac{y_2 - y_1}{x_2 - x_1} \\ \frac{1}{x_3 - x_1} \left(\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1} \right) \\ \left(\frac{1}{x_4 - x_1} \right) \left(\frac{1}{x_4 - x_2} \left(\frac{y_4 - y_3}{x_4 - x_3} - \frac{y_3 - y_2}{x_3 - x_2} \right) - \frac{1}{x_3 - x_1} \left(\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1} \right) \right) \end{bmatrix} \quad (11.2.36) \\ &= \begin{bmatrix} y_1 \\ f[x_2, x_1] \\ f[x_3, x_2, x_1] \\ f[x_4, x_3, x_2, x_1] \end{bmatrix} \end{aligned}$$

where

$$c^2 = f[x_2, x_1] \equiv \frac{y_2 - y_1}{x_2 - x_1} \quad (11.2.37)$$

$$c^3 = f[x_3, x_2, x_1] = \frac{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1} \equiv \frac{f[x_3, x_2] - f[x_2, x_1]}{x_3 - x_1} \quad (11.2.38)$$

and

$$\begin{aligned} c^4 &= f[x_4, x_3, x_2, x_1] = \frac{\left(\frac{y_4 - y_3}{x_4 - x_3} - \frac{y_3 - y_2}{x_3 - x_2} \right) - \left(\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1} \right)}{x_4 - x_1} \\ &= \frac{f[x_4, x_3, x_2] - f[x_3, x_2, x_1]}{x_4 - x_1} \end{aligned} \quad (11.2.39)$$

The forms of (11.2.37), (11.2.38) and (11.2.39) reveal an iteration scheme that allow for the calculation of the Newton polynomial coefficients for polynomials of arbitrary order. Equation (11.2.37) defines the *first divided difference*. Equation (11.2.38) defines the *second divided difference* in terms of the first. Therefore, for a Newton polynomial in \mathcal{P}_N

$$c^1 = y_1 \quad (11.2.40)$$

$$c^2 = f[x_2, x_1] \quad (11.2.41)$$

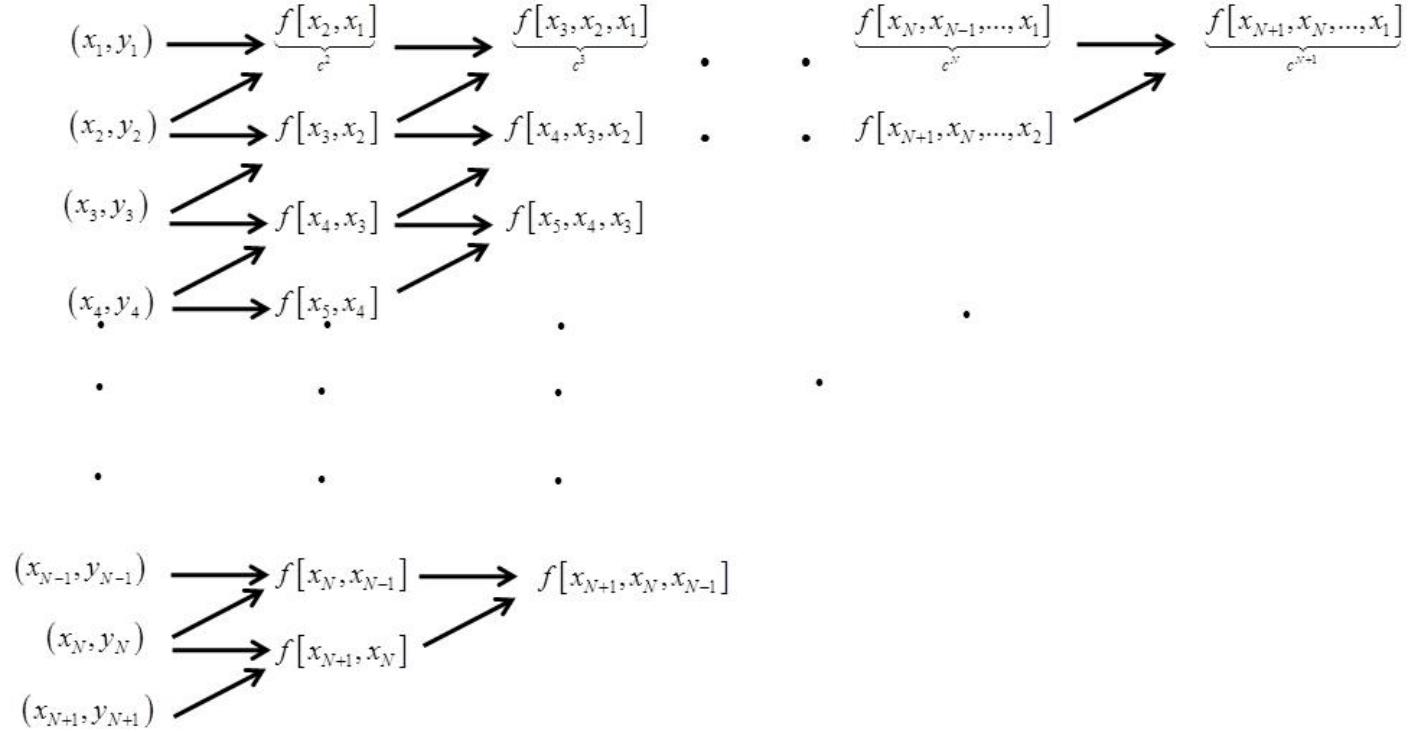
$$c^3 = f[x_3, x_2, x_1] = \frac{f[x_3, x_2] - f[x_2, x_1]}{x_3 - x_1} \quad (11.2.42)$$

$$c^4 = f[x_4, x_3, x_2, x_1] = \frac{f[x_4, x_3, x_2] - f[x_3, x_2, x_1]}{x_4 - x_1} \quad (11.2.43)$$

.

$$c^{N+1} = f[x_{N+1}, x_N, \dots, x_1] = \frac{f[x_{N+1}, x_N, \dots, x_2] - f[x_N, x_{N-1}, \dots, x_1]}{x_{N+1} - x_1} \quad (11.2.44)$$

The actual calculation of these coefficients is facilitated by constructing the following *divided difference table*:



This table, in effect, constructs the formulas one would use to calculate the coefficients in the Newton interpolation by hand. After the matrix represented by the above table, the coefficient c^1 is given by (11.2.40) and the coefficients $c^2, c^3, \dots, c^N, c^{N+1}$ are read off from the first row of the above table.

Given the information in this table, the Newton interpolation formula (11.2.3) becomes

$$\begin{aligned}
 f_N(x) = & y_1 + (x - x_1) f[x_2, x_1] \\
 & + (x - x_1)(x - x_2) f[x_3, x_2, x_1] \\
 & + (x - x_1)(x - x_2)(x - x_3) f[x_4, x_3, x_2, x_1] \\
 & + \dots \\
 & + (x - x_1)(x - x_2)(x - x_3) \cdots (x - x_{N-1})(x - x_N) f[x_{N+1}, x_N, \dots, x_2, x_1]
 \end{aligned} \tag{11.2.45}$$

When evaluating the polynomial (11.2.45) at particular values of x , there are computational benefits of adopting a modification of the *Horner method* illustrated by (11.2.25). This method is implemented by replacing (11.2.45) with

$$f_N(x) = y_1 + (x - x_1) \left(f[x_2, x_1] + (x - x_2) \left(f[x_3, x_2, x_1] + (x - x_3) \left(f[x_4, x_3, x_2, x_1] + \dots + \dots (x - x_{N-1}) ((x - x_N) f[x_{N+1}, x_N, \dots, x_2, x_1]) \right) \right) \right) \quad (11.2.46)$$

Example 11.2.4: As a simple illustration of the interpolation based upon the formulas of Example 11.2.3, consider the following data set for $N + 1 = 4$

x	-1	0	1	2
y	5	3	-1	-5

With this data, the divided difference table takes the form

$x_1 = -1$	$c^1 = y_1 = 5$	$c^2 = f[x_2, x_1] = \frac{3 - 5}{0 - (-1)} = -2$	$c^3 = f[x_3, x_2, x_1] = \frac{-4 - (-2)}{1 - (-1)} = -1$	$c^4 = f[x_4, x_3, x_2, x_1] = \frac{0 - (-1)}{2 - (-1)} = \frac{1}{3}$
$x_2 = 0$	$y_2 = 3$	$f[x_3, x_2] = \frac{-1 - 3}{1 - (0)} = -4$	$f[x_4, x_3, x_2] = \frac{-4 - (-4)}{2 - 0} = 0$	
$x_3 = 1$	$y_3 = -1$	$f[x_4, x_3] = \frac{-5 - (-1)}{2 - 1} = -4$		
$x_4 = 2$	$y_4 = -5$			

Therefore, from (11.2.41) through (11.2.44),

$$\begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} = \begin{bmatrix} 5 \\ -2 \\ -1 \\ \frac{1}{3} \end{bmatrix} \quad (11.2.47)$$

and the polynomial, from (11.2.45), is given by

$$f_3(x) = 5 - (x + 1)2 - (x + 1)(x) + \frac{1}{3}(x + 1)(x)(x - 1) \quad (11.2.48)$$

If the polynomial (11.2.48) is expanded into its monomial form, the result is

$$f_3(x) = 3 - \frac{10}{3}x - x^2 + \frac{1}{3}x^3 \quad (11.2.49)$$

Because the form (11.2.49) is related to (11.2.48) by a basis change, the polynomial coefficients in (11.2.49) are related to the coefficients in (11.2.47) by the basis change formula (2.6.31), which with the transition matrix (2.6.21), repeated in this example's notation,

$$T = \begin{bmatrix} T_k^j \end{bmatrix} = \begin{bmatrix} T_1^1 & T_2^1 & T_3^1 & T_4^1 \\ T_1^2 & T_2^2 & T_3^2 & T_4^2 \\ T_1^3 & T_2^3 & T_3^3 & T_4^3 \\ T_1^4 & T_2^4 & T_3^4 & T_4^4 \end{bmatrix} = \begin{bmatrix} 1 & -x_1 & x_1x_2 & -x_1x_2x_3 \\ 0 & 1 & -(x_1+x_2) & x_1x_2 + x_1x_3 + x_2x_3 \\ 0 & 0 & 1 & -(x_1+x_2+x_3) \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11.2.50)$$

yields

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & -x_1 & x_1x_2 & -x_1x_2x_3 \\ 0 & 1 & -(x_1+x_2) & x_1x_2 + x_1x_3 + x_2x_3 \\ 0 & 0 & 1 & -(x_1+x_2+x_3) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ -2 \\ -1 \\ \frac{1}{3} \end{bmatrix} = \begin{bmatrix} \frac{3}{10} \\ -\frac{3}{2} \\ -1 \\ \frac{1}{3} \end{bmatrix} \quad (11.2.51)$$

where the values x_1, x_2, x_3 have been taken from the data table and the matrix (11.2.47) has been used.

In Section 11.3, we shall utilize MATLAB to perform the calculations illustrated by Example 11.2.4. It will prove helpful in Section 11.3 if we rewrite (11.2.51)₁ in a form that illustrates and supports the MATLAB script that will be used. The first step is to recall the result (2.7.12) where it was recognized that a matrix is a linear function of its columns. This fact allows us to write

$$\begin{bmatrix} 1 & -x_1 & x_1x_2 & -x_1x_2x_3 \\ 0 & 1 & -(x_1+x_2) & x_1x_2 + x_1x_3 + x_2x_3 \\ 0 & 0 & 1 & -(x_1+x_2+x_3) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} c^1 + \begin{bmatrix} -x_1 \\ 1 \\ 0 \\ 0 \end{bmatrix} c^2 + \begin{bmatrix} x_1x_2 \\ -(x_1+x_2) \\ 1 \\ 0 \end{bmatrix} c^3 + \begin{bmatrix} -x_1x_2x_3 \\ x_1x_2 + x_1x_3 + x_2x_3 \\ -(x_1+x_2+x_3) \\ 1 \end{bmatrix} c^4 \quad (11.2.52)$$

The column matrix $\begin{bmatrix} -x_1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ in (11.2.52) can be written in terms of the column matrix $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, also in (11.2.52), by the formula

$$\begin{bmatrix} -x_1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (11.2.53)$$

In a similar fashion the column matrix $\begin{bmatrix} x_1 x_2 \\ -(x_1 + x_2) \\ 1 \\ 0 \end{bmatrix}$ in (11.2.52) can be written in terms of the one in (11.2.53) by the formula

$$\begin{bmatrix} x_1 x_2 \\ -(x_1 + x_2) \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -x_1 \\ 1 \\ 0 \end{bmatrix} - x_2 \begin{bmatrix} -x_1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (11.2.54)$$

and the column matrix $\begin{bmatrix} -x_1 x_2 x_3 \\ x_1 x_2 + x_1 x_3 + x_2 x_3 \\ -(x_1 + x_2 + x_3) \\ 1 \end{bmatrix}$ in (11.2.52) can be written in terms of the one in (11.2.54) by the formula

$$\begin{bmatrix} -x_1 x_2 x_3 \\ x_1 x_2 + x_1 x_3 + x_2 x_3 \\ -(x_1 + x_2 + x_3) \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ x_1 x_2 \\ -(x_1 + x_2) \\ 1 \end{bmatrix} - x_3 \begin{bmatrix} x_1 x_2 \\ -(x_1 + x_2) \\ 1 \\ 0 \end{bmatrix} \quad (11.2.55)$$

Equations (11.2.53), (11.2.54) and (11.2.55) allow (11.2.52) to be written

$$\begin{aligned}
 & \begin{bmatrix} 1 & -x_1 & x_1 x_2 & -x_1 x_2 x_3 \\ 0 & 1 & -(x_1 + x_2) & x_1 x_2 + x_1 x_3 + x_2 x_3 \\ 0 & 0 & 1 & -(x_1 + x_2 + x_3) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} c^1 + \left(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right) c^2 \\
 & \quad + \left(\begin{bmatrix} 0 \\ -x_1 \\ 1 \\ 0 \end{bmatrix} - x_2 \begin{bmatrix} -x_1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \right) c^3 \\
 & \quad + \left(\begin{bmatrix} 0 \\ x_1 x_2 \\ -(x_1 + x_2) \\ 1 \end{bmatrix} - x_3 \begin{bmatrix} x_1 x_2 \\ -(x_1 + x_2) \\ 1 \\ 0 \end{bmatrix} \right) c^4
 \end{aligned} \tag{11.2.56}$$

The relationship between the various matrices in (11.2.56) will allow us, in Section 11.3, to establish a simple **for-end** loop that will create the set of equations (11.2.51) for the simple example in this section as well as for Newton interpolations involving large data sets.

Example 11.2.5: For the case where we are again given the data set

$\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$, we can use (11.2.6) and define a *Lagrange* basis for \mathcal{P}_3

$$\begin{aligned}
 l_1(x) &= \frac{(x - x_2)(x - x_3)(x - x_4)}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} \\
 l_2(x) &= \frac{(x - x_1)(x - x_3)(x - x_4)}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} \\
 l_3(x) &= \frac{(x - x_1)(x - x_2)(x - x_4)}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_4)} \\
 l_4(x) &= \frac{(x - x_1)(x - x_2)(x - x_3)}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)}
 \end{aligned} \tag{11.2.57}$$

The choice (11.2.57) arises when one is representing the polynomial as a *Lagrange* polynomial. For this choice of basis, the matrix (11.2.8) reduces to

$$A_L = \begin{bmatrix} l_1(x_1) & l_2(x_1) & l_3(x_1) & l_4(x_1) \\ l_1(x_2) & l_2(x_2) & l_3(x_2) & l_4(x_2) \\ l_1(x_3) & l_2(x_3) & l_3(x_3) & l_4(x_3) \\ l_1(x_4) & l_2(x_4) & l_3(x_4) & l_4(x_4) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11.2.58}$$

The fact that (11.2.58) is the identity matrix, the inverse A_L^{-1} is, trivially, given by

$$A_L^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11.2.59)$$

Given (11.2.59), the solution of (11.2.7) in this case is

$$\begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \quad (11.2.60)$$

The final result (11.2.60) yields the important result that the coefficients of the polynomial (11.2.3) for the case of a Lagrange polynomial are simply the data points y_1, y_2, y_3 and y_4 . This fact is the main advantage of Lagrange interpolation polynomials. The polynomial can simply be written down without the need to solve a system of equations as is the case of monomial polynomials and Newton polynomials. As a result of this special feature,

$$\begin{aligned} f_3(x) &= c^1 l_1(x) + c^2 l_2(x) + c^3 l_3(x) + c^4 l_4(x) \\ &= \frac{(x - x_2)(x - x_3)(x - x_4)}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} y_1 + \frac{(x - x_1)(x - x_3)(x - x_4)}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} y_2 \\ &\quad + \frac{(x - x_1)(x - x_2)(x - x_4)}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_4)} y_3 + \frac{(x - x_1)(x - x_2)(x - x_3)}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} y_4 \end{aligned} \quad (11.2.61)$$

Of course the result (11.2.61) easily generalizes to polynomials in \mathcal{P}_N . A downside to Lagrange polynomial interpolation is the complexity of the answer. This fact makes computations more difficult, especially differentiation and integration.

Example 11.2.6: It is helpful to rework Example 11.2.4 with a Lagrange basis. The data table is again

x	-1	0	1	2
y	5	3	-1	-5

Given this table, equation (11.2.61) reduces to

$$\begin{aligned}
 f_3(x) &= \frac{(x-0)(x-1)(x-2)}{(-1-0)(-1-1)(-1-2)} 5 + \frac{(x-(-1))(x-1)(x-2)}{(0-(-1))(0-1)(0-2)} 3 \\
 &\quad - \frac{(x-(-1))(x-0)(x-2)}{(1-(-1))(1-0)(1-2)} - \frac{(x-(-1))(x-0)(x-1)}{(2-(-1))(2-0)(2-1)} 5 \\
 &= -\frac{5}{6}(x)(x-1)(x-2) + \frac{3}{2}(x+1)(x-1)(x-2) + \frac{1}{2}(x+1)(x)(x-2) - \frac{1}{6}(x+1)(x)(x-1)
 \end{aligned} \tag{11.2.62}$$

If this polynomial is expended, the result is again (11.2.49), a result that can be obtained by the same kind of basis change calculation used in equation (11.2.51)

Unlike the results in Example 11.2.1 and like Example 11.2.3, the simplicity of the result (11.2.60) does allow the construction of the polynomial (11.2.61) without a computer being essential. However, as explained with Example 11.2.3, it is sensible to implement interpolations with Lagrange polynomials by use of MATLAB. These implementations will be discussed in Section 11.5.

Exercises:

11.2.1: Derive a quadratic equation that interpolates the data

x	4	5	7
y	-5	-40	10

Work the exercise with a monomial basis, a Newton basis and a Lagrange basis. The resulting polynomial should turn out to be

$$y(x) = 20x^2 - 215x + 535 \tag{11.2.63}$$

Section 11.3. Monomial Interpolation with MATLAB

The results obtained in Section 11.2 establish three analytically equivalent ways to perform a polynomial interpolation. One can work entirely with the monomial basis or one can adopt the Newton or Lagrange basis, do the interpolation, and transform the results by a change of basis back to the monomial one. The monomial basis is analytically the most complicated and the Lagrange basis is analytically the simplest. There are other polynomial bases that could be adopted but for simplicity we shall continue our discussion of these three.

Given a computational tool such as MATLAB the matrix inversions associated with the monomial basis is not difficult. However, a complicating issue is that for a monomial basis, the matrix to invert is the transposed Vandermonde matrix. As illustrated in Example 7.4.4, this matrix is ill-conditioned. As discussed in Section 7.4, solutions based upon this matrix will be sensitive to errors in the data and round off errors. Small errors in the data table can produce large errors in the solution (11.2.7) for the polynomial coefficients. This problem makes interpolations by use of monomial polynomials limited in their use. We shall see in Section 11.9 that use of a Newton basis or a Lagrange basis provides a numerically superior approach to monomial interpolation. We shall also see in Section 11.9 how the data can be preconditioned in a way that avoids numerical problems that are inherent with monomial interpolation.

In spite of the ill conditioned matrix (11.2.10), in this section we shall formulate a MATLAB implementation of interpolation utilizing the monomial basis. Our approach will be to formulate a function m-file whose input will generate the appropriate polynomial coefficients. Function m-files were mentioned in Appendix A. They were utilized in Sections 7.1, 7.5, 9.2, 9.3 and 9.4. For interpolation problems based upon the monomial basis, the following script defines the function m-file **monomial.m**.

```

function [c_m,yvalues]=monomial(x,y,xvalues)
%monomial: Monomial polynomial interpolation
%N=degree of polynomial
%N+1=number of data pairs
%input:
% x = row matrix of independent variable
% values = [x1, x2, ..., xN+1]
% y = row matrix of dependent variable
% values = [y1, y2, ..., yN+1]
% xvalues=row matrix of points where the
% interpolated values are to be calculated.
% The last argument can be omitted.
%output:
% c_m = polynomial coefficients, a row matrix of
% dimension N+1 ordered with increasing powers
% of the variable
% yvalues=values of the interpolated polynomial
% at points xvalues
N=length(x)-1;

```

```

if length(y)~=N+1, error('x and y must be of the same
length'); end
%Build the transposed Vandermonde Matrix
A=zeros(N+1,N+1); %Preallocate
for k=1:N+1
    A(:,k)=x'.^(k-1);
end
c_m=(A\y)';
if nargin==3
    %reorder the polynomial coefficients in order to use
    %polyval
    p=zeros(1,N+1);
    p(:)=c_m((N+1):-1:1);
    yvalues=polyval(p,xvalues);
end

```

The notes within the script provide information about how to utilize the file. One begins with the data table organized into two $N + 1$ dimensional row vectors **x** and **y**. Optionally, one can also prescribe a row matrix **xvalues**. These are the values of the independent variable x where the value of the interpolated polynomial will be calculated. The argument **xvalues** can be omitted from the function m-file. When executed, the function file produces an $N + 1$ dimensional row vector **c_m**. The elements of this matrix are ordered according to the convention reflected in equation (11.2.1). Therefore,

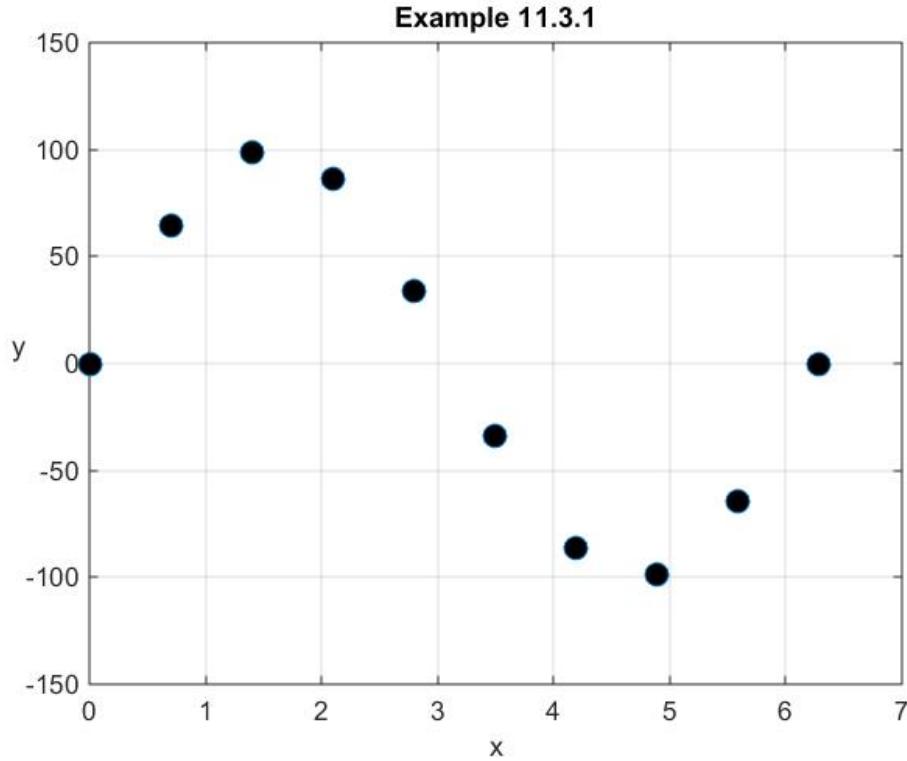
$$\mathbf{c_m} = [a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots \quad a_{N-1} \quad a_N] \quad (11.3.1)$$

If the argument **xvalues** was included as an input, the output includes a row matrix **yvalues** of the corresponding values of the interpolated polynomial. The option of including the third argument or not is accommodated by the **if-end** construct discussed in Appendix A and use of the MATLAB command **nargin**. This command returns the *number of arguments* of the function **monomial**. The script **if nargin==3** will determine if the number of argument equal to 3 is a true statement. If it is true, the next commands in the **if-end** construct will be executed. If false, the command jumps to the **end** and the script associated with the second output is skipped. Another feature of the above script is that the output (11.3.1) for the polynomial coefficients has been reordered in order that the MATLAB function **polyval** can be used to determine the value of the interpolated polynomial at the given points. This function was discussed briefly in Section 9.6 and was utilized in Section 11.2. The same set of script was used in conjunction with Example 11.2.2.

Example 11.3.1: You are given the data table

x	0	0.6981	1.3963	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
y	0	64.28	98.48	86.60	34.20	-34.20	86.60	98.48	64.28	0

consisting of 10 data pairs. The objective is to adopt the monomial basis and use the formalism above to find the polynomial of degree 9 that passes through each of these points.⁶ MATLAB is readily used to create the following plot of this set of data.



We can implement the calculation (11.2.7) by the script

```

clc
clear
%Construct the data table
N=9;
x=linspace(0,2*pi,N+1);
y=100*sin(x);
%Assign the 50 equally spaced points x where the
%resulting polynomial will be evaluated
xvalues=linspace(0,2*pi,50);
%Calculate the row matrix of polynomial coefficients and
%the yvalues
[c_m,yvalues]=monomial(x,y,xvalues)

%Plot the data points

```

⁶This data table was generated by partitioning the interval $[0, 2\pi]$ into 9 equal intervals and assigning values at each of the ten points making up the intervals by the formula $y = 100 \sin(x)$.

```

plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([0,7,-150,150])
grid on
hold on
%Use the values yvalues and plot the polynomial at the 50
%points in the interval (0,2*pi)
plot(xvalues,yvalues,'r','LineWidth',2)
legend('Data','Interpolation Polynomial')
title('Example 11.3.1')

```

The output for the argument **c_m** produced by the above script is

```

c_m =
Columns 1 through 5
    0    99.9073    0.4016   -17.3844    0.7076
Columns 6 through 10
    0.4041    0.1675   -0.0619    0.0065   -0.0002

```

This output, arranged as a column matrix, is

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \\ a_9 \end{bmatrix} = \begin{bmatrix} 0 \\ 99.9073 \\ 0.4016 \\ -17.3844 \\ 0.7076 \\ 0.4041 \\ 0.1675 \\ -0.0619 \\ 0.0065 \\ -0.0002 \end{bmatrix} \quad (11.3.2)$$

The values **yvalues** are given by the MATLAB output

```

yvalues =
Columns 1 through 8
    0    12.7811   25.3586   37.5220   49.0694   59.8105   69.5687   78.1840
Columns 9 through 16
    85.5150   91.4418   95.8670   98.7182   99.9485   99.5377   97.4926   93.8468

```

```

Columns 17 through 24

 88.6599   82.0173   74.0278   64.8229   54.5535   43.3884   31.5108   19.1159

Columns 25 through 32

 6.4070   -6.4070   -19.1159   -31.5108   -43.3884   -54.5535   -64.8229   -74.0278

Columns 33 through 40

-82.0173   -88.6599   -93.8468   -97.4926   -99.5377   -99.9485   -98.7182   -95.8670

Columns 41 through 48

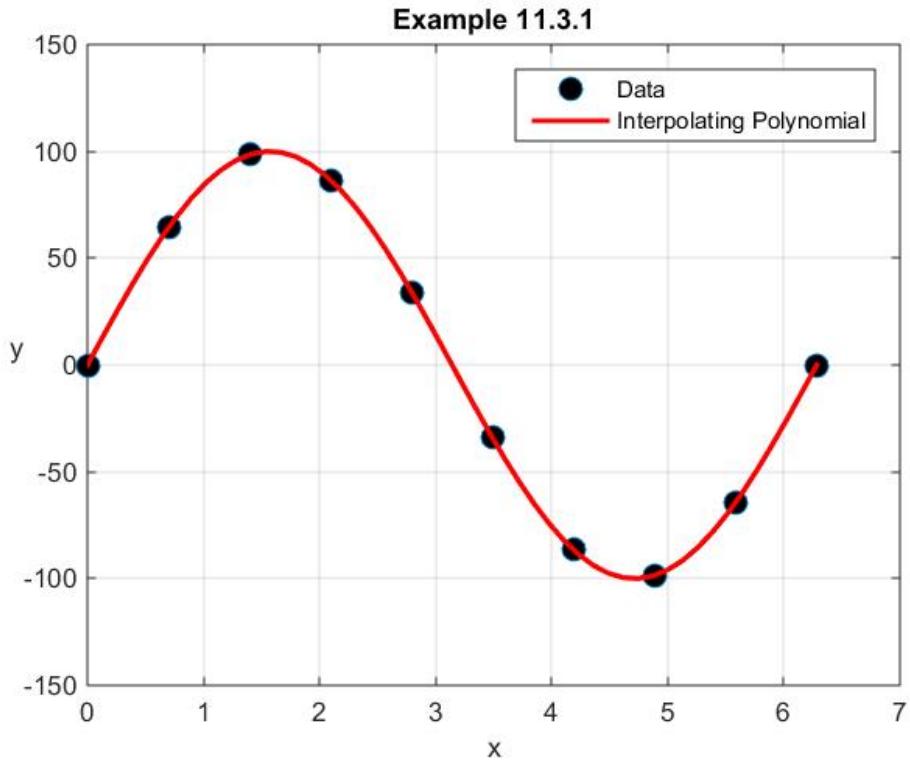
-91.4418   -85.5150   -78.1840   -69.5687   -59.8105   -49.0694   -37.5220   -25.3586

Columns 49 through 50

-12.7811   0.0000

```

The plot produced by the above script is



The number of data points for this example did not create any MATLAB warnings about the ill-conditioned nature of the matrix A_M . It is interesting to note that the various condition numbers introduced in Section 7.4 assume the values

$$\text{cond}(A, \text{'fro'}) = 9.8992(10)^9 \quad (11.3.3)$$

$$\text{cond}(A, 1) = 1.6519(10)^{10} \quad (11.3.4)$$

$$\text{cond}(A, \text{inf}) = 1.7388(10)^{10} \quad (11.3.5)$$

$$\text{cond}(A, 2) = 9.8903(10)^9 \quad (11.3.6)$$

These large condition numbers display the ill-conditioned nature of A_M in this case.

The errors in the results produced by **monomial.m** for this example, while too small to be reflected in the above figure, can be illustrated by executing a calculation that compares the given values y in the above table with the values **yvalues** predicted by **monomial.m** at the given points x in the table. The MATLAB script

```
[c_m,yvalues]=monomial(x,y,x);
abs(y'-yvalues')
```

displays, as a column matrix, the absolute value of the difference $y' - yvalues'$. The numerical results produced by MATLAB from this script are⁷

```
>> abs(y'-yvalues')

ans =
1.0e-11 *
0
0.0313
0.2430
0.0725
0.0668
0.5052
0.0455
0.1904
0.0540
0.2078
```

(11.3.7)

These results arise from the ill conditioned nature of the matrix of coefficients A_M and any round off errors associated with the calculation of **yvalues** by **monomial.m**. Later, in Section 11.9, we shall see an example where these numerical problems are greater.

⁷ The results (11.3.7) are the same as those calculated from the script **abs(y'-polyval(p,x)')**.

Section 11.4. Newton Interpolation with MATLAB

Next, we shall develop a function m-file that will implement the Newton polynomial interpolation. The explanation of this file will require that we discuss how to cause MATLAB to construct the divided difference table. As illustrated in Example 11.2.4, this table will give us the polynomial coefficients. If one wants to convert the Newton polynomial into a monomial polynomial we also need to generalize the transition matrix (11.2.50) to the case of a transition from a monomial polynomial of degree N to a Newton polynomial basis of the same degree. In this section, we will also discuss how to utilize MATLAB to construct this transition matrix.

First, we shall discuss how to cause MATLAB to construct the $(N+1) \times (N+1)$ matrix representing the divided difference table introduced in Section 11.2. We are still interested in interpolations of the $N+1$ data points $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$, and the N points that appear in the definition of the $N+1$ Newton polynomials are the values x_1, x_2, \dots, x_N given as a part of the for the data. As always, we continue to assume the data points are ordered as $x_1 < x_2 < \dots < x_{N+1}$. The $(N+1) \times (N+1)$ matrix that represents the divided difference table is

$$M = \begin{bmatrix} y_1 & f[x_2, x_1] & f[x_3, x_2, x_1] & \cdot & \cdot & \cdot & f[x_N, x_{N-1}, \dots, x_1] & f[x_{N+1}, x_N, \dots, x_1] \\ y_2 & f[x_3, x_2] & f[x_4, x_3, x_2] & \cdot & \cdot & \cdot & f[x_{N+1}, x_N, \dots, x_2] & 0 \\ y_3 & f[x_4, x_3] & f[x_5, x_4, x_3] & \cdot & \cdot & \cdot & 0 & 0 \\ \cdot & \cdot & & & & & \cdot & \cdot \\ \cdot & \cdot & & & & & \cdot & \cdot \\ \cdot & \cdot & & & & & \cdot & \cdot \\ y_N & f[x_N, x_{N-1}] & f[x_{N+1}, x_N, x_{N-1}] & 0 & \cdot & \cdot & 0 & 0 \\ y_{N+1} & f[x_{N+1}, x_N] & 0 & 0 & \cdot & \cdot & 0 & 0 \end{bmatrix} \quad (11.4.1)$$

where zeros have been entered for all of the elements that are not contributing to the table. The key to the construction of this matrix in MATLAB is to recognize that all of the nonzero elements in the columns after the first obey the recursive relationship

$$M_{jk} = \frac{M_{j+1,k-1} - M_{j,k-1}}{x_{j+k-1} - x_j} \quad \text{for } k = 2, \dots, N+1 \text{ and } j = 1, \dots, N-k \quad (11.4.2)$$

Because of (11.4.2), the MATLAB script that will create the divided difference matrix is

```
M=zeros(N+1,N+1); %Preallocate
M(:,1)=y';
for k=2:N+1
    for j=1:N+1-(k-1)
```

```

M(j,k)=(M(j+1,k-1)-M(j,k-1))/(x(j+k-1)-x(j))
end
end

```

In order to construct the transition matrix, we first need a few preliminaries. The Newton basis for \mathcal{P}_N is the set of polynomials $\{n_1, n_2, n_3, \dots, n_N, n_{N+1}\}$, where

$$\begin{aligned} n_1(x) &= 1 \\ n_j(x) &= \prod_{k=1}^{j-1} (x - x_k) \quad \text{for } j = 2, \dots, N+1 \end{aligned} \tag{11.4.3}$$

In order to manipulate the various products that appear in these basis elements, we need convenient formulas for products of the form $\prod_{k=1}^M (x - x_k)$. If these products are formed, they can always be expanded and rearranged into the M^{th} order polynomial of the form

$$\prod_{k=1}^M (x - x_k) = \sum_{k=0}^M (-1)^k \sigma_k(x_1, x_2, x_3, \dots, x_M) x^{M-k} \tag{11.4.4}$$

where

$$\sigma_0(x_1, x_2, x_3, \dots, x_M) = 1 \tag{11.4.5}$$

and⁸

$$\sigma_j(x_1, x_2, x_3, \dots, x_M) = \sum_{\substack{k_1, k_2, \dots, k_j=1 \\ 1 \leq k_1 < k_2 < \dots < k_j \leq M}} x_{k_1} x_{k_2} \cdots x_{k_j} \quad \text{for } j = 1, 2, \dots, M \tag{11.4.6}$$

It follows from (11.4.6) that

⁸ As in Section 6.1, equation (11.4.6) defines an *elementary symmetric polynomial*. A brief but good discussion of these polynomials can be found at http://en.wikipedia.org/wiki/Elementary_symmetric_polynomial.

$$\begin{aligned}
\sigma_1(x_1, x_2, x_3, \dots, x_M) &= x_1 + x_2 + \dots + x_M \\
\sigma_2(x_1, x_2, x_3, \dots, x_M) &= x_1 x_2 + x_1 x_3 + \dots + x_1 x_M + x_2 x_3 + x_2 x_4 + \dots + x_2 x_M + \dots + x_{M-1} x_M \\
\sigma_3(x_1, x_2, x_3, \dots, x_M) &= x_1 x_2 x_3 + x_1 x_2 x_4 + \dots + x_1 x_2 x_M + x_1 x_3 x_4 + x_1 x_3 x_5 + \dots + x_1 x_3 x_M + \dots + x_{M-2} x_{M-1} x_M \\
&\vdots \\
&\vdots \\
\sigma_M(x_1, x_2, x_3, \dots, x_M) &= x_1 x_2 \cdots x_M
\end{aligned} \tag{11.4.7}$$

In this specialized notation, the result (11.2.50) for $N = 3$ takes the form

$$T = \begin{bmatrix} T_k^1 & T_k^2 & T_k^3 & T_k^4 \end{bmatrix} = \begin{bmatrix} T_1^1 & T_2^1 & T_3^1 & T_4^1 \\ T_1^2 & T_2^2 & T_3^2 & T_4^2 \\ T_1^3 & T_2^3 & T_3^3 & T_4^3 \\ T_1^4 & T_2^4 & T_3^4 & T_4^4 \end{bmatrix} = \begin{bmatrix} 1 & -\sigma_1(x_1) & \sigma_2(x_1, x_2) & -\sigma_3(x_1, x_2, x_3) \\ 0 & 1 & -\sigma_1(x_1, x_2) & \sigma_2(x_1, x_2, x_3) \\ 0 & 0 & 1 & -\sigma_1(x_1, x_2, x_3) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11.4.8}$$

and the basis change (11.2.12) takes the form

$$\begin{bmatrix} 1 \\ x - x_1 \\ (x - x_1)(x - x_2) \\ (x - x_1)(x - x_2)(x - x_3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\sigma_1(x_1) & 1 & 0 & 0 \\ \sigma_2(x_1, x_2) & -\sigma_1(x_1, x_2) & 1 & 0 \\ -\sigma_3(x_1, x_2, x_3) & \sigma_2(x_1, x_2, x_3) & -\sigma_1(x_1, x_2, x_3) & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} \tag{11.4.9}$$

The details leading to the transition matrix (11.4.8) suggest that for a basis transformation $\{1, x, x^2, x^3, \dots, x^{N-1}, x^N\}$

$\rightarrow \{1, x - x_1, (x - x_1)(x - x_2), (x - x_1)(x - x_2)(x - x_3), \dots, (x - x_1)(x - x_2)(x - x_3) \cdots (x - x_N)\}$

the transition matrix is

$$T = \begin{bmatrix} T_k^j \end{bmatrix} = \begin{bmatrix} 1 & -\sigma_1(x_1) & \sigma_2(x_1, x_2) & -\sigma_3(x_1, x_2, x_3) & \cdot & \cdot & \cdot & (-1)^N \sigma_N(x_1, x_2, x_3, \dots, x_N) \\ 0 & 1 & -\sigma_1(x_1, x_2) & \sigma_2(x_1, x_2, x_3) & \cdot & \cdot & \cdot & (-1)^{N-2} \sigma_{N-1}(x_1, x_2, x_3, \dots, x_N) \\ 0 & 0 & 1 & -\sigma_1(x_1, x_2, x_3) & \cdot & \cdot & \cdot & (-1)^{N-2} \sigma_{N-2}(x_1, x_2, x_3, \dots, x_N) \\ 0 & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & -\sigma_1(x_1, x_2, x_3, \dots, x_N) \\ 0 & 0 & 0 & \cdot & \cdot & \cdot & 0 & 1 \end{bmatrix} \quad (11.4.10)$$

The generalization to a polynomial of degree N that replaces (11.2.51)₁ can be read off from (11.4.10) and is

$$\begin{bmatrix} a_o \\ a_1 \\ a_2 \\ \cdot \\ \cdot \\ a_{N-1} \\ a_N \end{bmatrix} = \begin{bmatrix} 1 & -\sigma_1(x_1) & \sigma_2(x_1, x_2) & -\sigma_3(x_1, x_2, x_3) & \cdot & \cdot & \cdot & (-1)^N \sigma_N(x_1, x_2, x_3, \dots, x_N) \\ 0 & 1 & -\sigma_1(x_1, x_2) & \sigma_2(x_1, x_2, x_3) & \cdot & \cdot & \cdot & (-1)^{N-2} \sigma_{N-1}(x_1, x_2, x_3, \dots, x_N) \\ 0 & 0 & 1 & -\sigma_1(x_1, x_2, x_3) & \cdot & \cdot & \cdot & (-1)^{N-2} \sigma_{N-2}(x_1, x_2, x_3, \dots, x_N) \\ \cdot & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot \\ \cdot & -\sigma_1(x_1, x_2, x_3, \dots, x_N) \\ 0 & 0 & 0 & \cdot & \cdot & \cdot & 0 & 1 \end{bmatrix} \begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ \cdot \\ \cdot \\ c^N \\ c^{N+1} \end{bmatrix} \quad (11.4.11)$$

The entries in (11.4.10) can be calculated in a couple of different ways. We shall briefly discuss these calculations. It will turn out that the two approaches are rather slow for large data sets. The most efficient calculation seems to be to implement the basis transformation by utilizing script that creates directly the matrix product (11.4.11) rather than to form the square matrix (11.4.10) and to then calculate the product in (11.4.11). In any case, there are reasons to know how to build the transition matrix (11.4.10).

Our first approach will be to use the following script. The script that will create the matrix (11.4.10) for the case $N = 3$ is

```

N=3
syms x1 x2 x3
x=[x1,x2,x3]
T=sym(eye(N+1,N+1))
for k=N+1:-1:2

```

```

for j=(k-1):-1:1
    T(1:(N+1),k)=T(1:(N+1),k)-x(j).*circshift(T(:,k),-1)
end
end

```

The 4×4 matrix created by MATLAB is

```

T =

```

$$\begin{bmatrix} 1, -x_1, x_1 \cdot x_2, & -x_1 \cdot x_2 \cdot x_3 \\ 0, 1, -x_1 - x_2, x_1 \cdot (x_2 + x_3) + x_2 \cdot x_3 \\ 0, 0, 1, & -x_1 - x_2 - x_3 \\ 0, 0, 0, & 1 \end{bmatrix}$$

In the above script the quantities x_1, x_2 and x_3 have been defined as symbols. This choice allows the resulting matrix to be displayed in symbolic form and displays that it does create (11.2.50). The **eye** function was introduced in Chapter 7 and Appendix A. In this case, it creates a symbolic 4×4 identity matrix. The two **for-end** loops replace the elements of the identity matrix with the appropriate entries. The command **circshift** has not been discussed previously. It takes the elements in the k^{th} column of T and shifts it up one row with each choice of the index j . While not necessary illuminating, if the above script is modified as follows to

```

N=3
syms x1 x2 x3
x=[x1,x2,x3]
T=sym(eye(N+1,N+1))
for k=N+1:-1:2
    k
    for j=(k-1):-1:1
        j
        circshift(T(:,k),-1)
        T(1:(N+1),k)=T(1:(N+1),k)-x(j).*circshift(T(:,k),-1)
    end
end

```

the output will display the role of **circshift** at each step in the iteration. The output of this modified script for **k=4**, the fourth column, is

(Builds the identity matrix)

```

T =

```

$$\begin{bmatrix} 1, 0, 0, 0 \\ 0, 1, 0, 0 \\ 0, 0, 1, 0 \\ 0, 0, 0, 1 \end{bmatrix}$$

```
(Assigns k=4)
k =
4
(Assigns j=3)
j =
3
(Shifts elements of column 4 of T up one row)
ans =
0
0
1
0
(Result is multiplied by x3 and added to T)
T =
[ 1, 0, 0,    0]
[ 0, 1, 0,    0]
[ 0, 0, 1, -x3]
[ 0, 0, 0,    1]

(Assign j=2)
j =
2
(Shifts elements of column 4 of T up one row)

ans =
0
-x3
1
0
(Result is multiplied by x2 and added to T)

T =
[ 1, 0, 0,      0]
[ 0, 1, 0, x2*x3]
[ 0, 0, 1, - x2 - x3]
[ 0, 0, 0,      1]

(Assign j=1)
```

```
j =
1
(Shifts elements of column 4 of T up one row)

ans =
x2*x3
- x2 - x3
1
0
(Result is multiplied by x1 and added to T)

T =
[ 1, 0, 0,           -x1*x2*x3]
[ 0, 1, 0, x1*(x2 + x3) + x2*x3]
[ 0, 0, 1,           - x1 - x2 - x3]
[ 0, 0, 0,           1]
```

The remaining three columns are built by the same kinds of steps. It should be evident that the above construction will generalize for arbitrary integer N .

The second approach we wish to discuss involves the use of MATLAB's **charpoly** command.⁹ The syntax for this command is that if A is a square matrix, **charpoly(A)** returns a row vector of the coefficients of the characteristic polynomial of A . If A is a symbolic matrix, **charpoly** returns a symbolic vector. Otherwise, it returns a vector of double precision values. For example, the following script produces

```
%charpoly example
syms x1 x2 x3
charpoly(diag([x1]))
charpoly(diag([x1,x2]))
charpoly(diag([x1,x2,x3]))
```



```
ans =
[ 1, -x1]
```



```
ans =
```

⁹ **charpoly** is a command introduced in MATLAB version R2012a. For earlier versions of MATLAB use the **poly** command.

```
[ 1, - x1 - x2, x1*x2]

ans =

[ 1, - x1 - x2 - x3, x3*(x1 + x2) + x1*x2, -x1*x2*x3]
```

These results are the upper diagonal elements of (11.4.8). These results are generalized by the formula

$$\text{charpoly}\left(\text{diag}([x_1, x_2, x_3, \dots, x_N])\right)^T = \begin{bmatrix} 1 \\ -\sigma_1(x_1, x_2, x_3, \dots, x_N) \\ . \\ . \\ (-1)^{N-2} \sigma_{N-2}(x_1, x_2, x_3, \dots, x_N) \\ (-1)^{N-1} \sigma_{N-1}(x_1, x_2, x_3, \dots, x_N) \\ (-1)^N \sigma_N(x_1, x_2, x_3, \dots, x_N) \end{bmatrix} \quad (11.4.12)$$

where the transpose of $\text{charpoly}(\text{diag}(x_1, x_2, x_3, \dots, x_N))$ has been displayed to aid with the display of the equation on the page.

For the case $N = 3$, the script

```
syms x1 x2 x3
x=[x1,x2,x3]
T=sym(eye(4))
for j=2:4
    T(j:-1:1,j)=charpoly(diag(x(1:j-1)))
end
```

produces the matrix

```
T =

[ 1, -x1,      x1*x2,           -x1*x2*x3]
[ 0,   1, - x1 - x2, x3*(x1 + x2) + x1*x2]
[ 0,   0,       1,           - x1 - x2 - x3]
[ 0,   0,       0,                   1]
```

This result is the MATLAB *symbolic version* of the transition matrix (11.2.50). The above script, as with the first case we discussed, easily generalizes for polynomials of order N .

Unfortunately, the two approaches described seem to be unacceptably slow for large matrices. We shall attempt to illustrate this fact with examples later in this section. We shall avoid this problem, as indicated above, by the introduction of script that will create the matrix product (11.4.11) directly. The key to this script is equation (11.2.56), repeated,

$$\begin{aligned}
 & \begin{bmatrix} c^1 - x_1 c^2 + x_1 x_2 c^3 - x_1 x_2 x_3 c^4 \\ c^2 - (x_1 + x_2) c^3 + (x_1 x_2 + x_1 x_3 + x_2 x_3) c^4 \\ c^3 - (x_1 + x_2 + x_3) c^4 \\ c^4 \end{bmatrix} = \begin{bmatrix} 1 & -x_1 & x_1 x_2 & -x_1 x_2 x_3 \\ 0 & 1 & -(x_1 + x_2) & x_1 x_2 + x_1 x_3 + x_2 x_3 \\ 0 & 0 & 1 & -(x_1 + x_2 + x_3) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c^1 \\ c^2 \\ c^3 \\ c^4 \end{bmatrix} \\
 &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} c^1 + \begin{pmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - x_1 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{pmatrix} c^2 \\
 &\quad + \begin{pmatrix} \begin{bmatrix} 0 \\ -x_1 \\ 1 \\ 0 \end{bmatrix} - x_2 \begin{bmatrix} -x_1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \end{pmatrix} c^3 \\
 &\quad + \begin{pmatrix} \begin{bmatrix} 0 \\ x_1 x_2 \\ -(x_1 + x_2) \\ 1 \end{bmatrix} - x_3 \begin{bmatrix} x_1 x_2 \\ -(x_1 + x_2) \\ 1 \\ 0 \end{bmatrix} \end{pmatrix} c^4 \tag{11.4.13}
 \end{aligned}$$

where the left side multiplication has been carried out. The right side of (11.4.13) represents the product on the left side in a form that can be produced by the MATLAB script

```

N=3
syms c1 c2 c3 c4
syms x1 x2 x3
x=[x1,x2,x3];
c=[c1;c2;c3;c4];
a=[1;zeros(N,1)];
p=a*c(1);
for k=2:N+1
    a=circshift(a,1)-x(k-1)*a;
    p=p+a*c(k);
end
p

```

The MATLAB output from this script is

p =

```

c1 - c2*x1 + c3*x1*x2 - c4*x1*x2*x3
c2 - c3*(x1 + x2) + c4*(x3*(x1 + x2) + x1*x2)
c3 - c4*(x1 + x2 + x3)
c4

```

which is the left side of (11.4.13). We shall see below how this third approach is utilized as a part of the function m-file that implements the Newton interpolation.

Our objective is to produce a function m-file that will have as its output, the Newton interpolation coefficients, the corresponding monomial interpolation coefficients and values of the interpolated polynomials at specified points. We shall model the function m-file after the file **monomial.m** discussed in Section 11.3. If we utilize the above preliminaries for Newton polynomial interpolation, the following script defines the function m-file **newton.m**.

```

function [c_m,c_n,yvalues]=newton(x,y,xvalues)
%newton: Newton polynomial interpolation
%N=degree of polynomial
%N+1=number of data pairs
%input:
% x = row matrix of independent variable
% values = [x1, x2, ..., xN+1]
% y = row matrix of dependent variable
% values = [y1, y2, ..., yN+1]
% xvalues=row matrix of points where the
% interpolated values are to be calculated.
% The last argument can be omitted.
%output:
% c_m = monomial polynomial coefficients, a
% row matrix of dimension N+1 ordered with increasing
% powers of the variable
% c_n = Newton polynomial coefficients
% yvalues=values of the interpolated polynomial
% at points xvalues
N=length(x)-1;
if length(y)~=N+1
    error('x and y must be of the same length')
end
%Build the matrix of divided differences
M=zeros(N+1,N+1); %Preallocate
M(:,1)=y';
for k=2:N+1
    for j=1:N+1-(k-1)
        M(j,k)=(M(j+1,k-1)-M(j,k-1))/(x(j+k-1)-x(j));
    end
end
c_n=M(1,:);

```

```
%Calculate the monomial polynomial coefficients
a=[1;zeros(N,1)]
g=a*c_n(1)
for k=2:N+1
    a=circshift(a,1)-x(k-1)*a;
    g=g+a*c_n(k);
end
c_m=g'
if nargin==3
    %Use nested Horner form to evaluate polynomial
    %at xvalues
    h=c_n(N+1)
    for q=N:-1:1
        h=c_n(q)+h.* (xvalues-x(q))
    end
    yvalues=h
end
```

Give the two inputs **x** and **y**, the function m-file yields two $N + 1$ dimensional row matrices **c_n**, the polynomial coefficients for the Newton interpolation and **c_m**, the corresponding polynomial coefficients for the monomial interpolation. If, in addition, the row matrix **xvalues** is prescribed, the function file produces the corresponding polynomial values **yvalues** in the form of a row matrix.

In order to illustrate the use of this function file, consider the following example, which is a repeat of Example 11.3.1 except that the interpolation is Newton.

Example 11.4.1: You are again given the ten pair data table

x	0	0.6981	1.3963	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
y	0	64.28	98.48	86.60	34.20	-34.20	86.60	98.48	64.28	0

and the objective is to utilize **newton.m** to perform the interpolation. The following MATLAB script utilizes the function m-file **newton.m** to calculate the various unknowns and create a graph of the input data and the output.

```
clc
clear
%Construct the data table
N=9;
x=linspace(0,2*pi,N+1);
y=100*sin(x);
%Assign the 50 equally spaced points x where the
%resulting polynomial will be evaluated
xvalues=linspace(0,2*pi,50);
%Calculate the newton and monomial row matrices of
```

```
%polynomial coefficients and the yvalues
[c_m,c_n,yvalues]=newton(x,y,xvalues)

%Plot the data points
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([0,7,-150,150])
grid on
hold on
%Use the values yvalues and plot the polynomial at the 50
%points in the interval (0,2*pi)
plot(xvalues,yvalues,'r','LineWidth',2)
legend('Data','Newton Interpolating Polynomial')
title('Example 11.4.1')
```

The figure created by this script is indistinguishable from the one created in Example 11.3.1. The Newton polynomial coefficients, c_n , are given by

```
c_n =
Columns 1 through 7
0 92.0725 -30.8550 -7.8388 3.7820 -0.1307 -0.1064
Columns 8 through 10
0.0132 0.0007 -0.0002
```

The associated monomial polynomial coefficients, c_m are given by

```
c_m =
Columns 1 through 7
0 99.9073 0.4016 -17.3844 0.7076 0.4041 0.1675
Columns 8 through 10
-0.0619 0.0065 -0.0002
```

As one would hope, these numbers agree with those shown in equation (11.3.2).

As utilized in Example 11.3.1, as a measure of the numerical error associated with this Newton interpolation example, we can calculate the values of the interpolation polynomial at the given points x in the data table and compare the calculated results with the given values y in the data table. These errors can be calculated by executing the MATLAB script **abs(y' -**

yvalues'), where these **yvalues** are calculated from **[c_m,c_n,yvalues]=newton(x,y,x)**. The result is

```
>> abs(y'-yvalues')

ans =

1.0e-13 *

0
0
0
0.1421
0.0711
0.5684
0.1421
0.2842
0.7105
0.6480
```

(11.4.14)

These errors are substantially less than those shown in (11.3.7) for monomial interpolation. When we worked Example 11.3.1, we pointed out that the matrix to invert is the transposed Vandermonde matrix which is ill conditioned. The condition numbers for this matrix in Example 11.3.1 are given by equations (11.3.3) through (11.3.6). The improved accuracy of Newton interpolation as illustrated by (11.4.14) suggests that the matrix to invert for the Newton interpolation of Example 11.4.1, is better conditioned than the monomial one. The next example shows how to construct the matrix which we called A_N in Section 11.2 (see equation (11.2.27)) and to calculate its condition numbers for the data table utilized in Example 11.4.1.

Example 11.4.2: As explained, it is instructive to construct the matrix A_N and to calculate its condition numbers based upon the data table of Example 11.4.2. The following MATLAB script will generate these results:

```
clc
clear
N=9;
x=linspace(0,2*pi,N+1);
A_N=zeros(length(x)); %Preallocate
A_N(:,1)=1;
for j=2:length(x);
    for k=1:length(x);
        if k<j;
            A_N(k,j)=0;
        else
            A_N(k,j)=(x(k)-x(j-1))*A_N(k,j-1);
        end
    end
end
```

```

    end
end
A_N
cond(A_N, 'fro')
cond(A_N,1)
cond(A_N,inf)
cond(A_N,2)

```

The numerical output for the four condition numbers turn out to be

$$\text{cond}(A_N, \text{'fro'}) = 1.3263(10)^5 \quad (11.4.15)$$

$$\text{cond}(A_N, 1) = 1.3651(10)^5 \quad (11.4.16)$$

$$\text{cond}(A_N, \text{inf}) = 2.4572(10)^5 \quad (11.4.17)$$

$$\text{cond}(A_N, 2) = 1.1803(10)^5 \quad (11.4.18)$$

If these numbers are compared to their counterparts for monomial interpolation, equations (11.3.3) through (11.3.6), it becomes evident that A_N is better conditioned than A_M .

Example 11.4.3: The purpose of this example is to illustrate by examples the benefits of utilizing the approach based upon (11.4.13) to calculate the monomial polynomial coefficients from the Newton polynomial coefficients. Our approach will be to calculate these coefficients for data tables built from the equation $y = 100 \sin x$ as with Examples 11.3.1 and 11.4.1. Each choice of N results in a particular data table and, given that table, we shall calculate the coefficients c_m three ways. The first way will be utilizing the function m-file **newton.m** as given. The second way will be to use a function m-file obtained by replacing the script

```

a=[1;zeros(N,1)]
g=a*c_n(1)
for k=2:N+1
    a=circshift(a,1)-x(k-1)*a;
    g=g+a*c_n(k);
end
c_m=g'

```

in **newton.m** with

```

T=eye(N+1,N+1)
for k=N+1:-1:2
    for j=(k-1):-1:1
        T(1:(N+1),k)=T(1:(N+1),k)-x(j).*circshift(T(:,k),-1)
    end
end

```

```

end
c_m=(T*c_n)'

```

the resulting function m-file will be called **newton1.m**. The third way will be to replace the script in **newton.m** by

```

T=eye(N+1,N+1)
for k=N+1:-1:2
    for j=2:N+1
        T(j:-1:1,j)=charpoly(diag(x(1:j-1)))
    end
end
c_m=(T*c_n)'

```

the resulting function m-file will be called **newton2.m**. The script that will implement this calculation is

```

clc
clear
%Given N and construct the data table
N=9;
x=linspace(0,2*pi,N+1);
y=100*sin(x);
tic
[c_m1,c_n]=newton(x,y)
time1=toc
tic
[c_m1,c_n]=newton2(x,y)
time2=toc
tic
[c_m2,c_n]=newton3(x,y)
time3=toc
[c_m',c_m1',c_m2']
times=[time1,time2,time3]

```

This script is for the case $N = 9$. Note the insertion of the commands **tic** and **toc**. These commands calculate the time in seconds to execute the commands between **tic** and **toc**. In the above script, the line **time1=toc** displays the number of seconds required to execute the command **[c_m1,c_n]=newton(x,y)**. Likewise, **time2=toc** displays the number of seconds required to execute the command **[c_m2,c_n]=newton2(x,y)** and **time3=toc** the number of seconds required to execute **[c_m3,c_n]=newton3(x,y)**. ¹⁰

¹⁰ The elapsed time calculations utilizing **tic** and **toc** are unique to the author's computer and the version of MATLAB being used. If the above script is executed on a different computer, different elapsed times will be generated. The important point is the relative values of the times to execute the calculations in the three cases.

If the above script is executed for $N = 9$, the same polynomial degree used in Example 11.4.1, the script `times=[time1,time2,time3]` yields the results

```
times =  
0.0011    0.0100    1.5449
```

Thus, a calculation based upon **charpoly** is the slowest of the three and the calculation based upon (11.4.13) is the fastest. The calculation based upon the second case, the one where the transition matrix is created, is fast but approximately ten times slower than the first case. The problem with the third case is that **charpoly** is solving an eigenvalue problem at each step. It is an eigenvalue problem where the answers are already known in advance. The problem with the second case is that double loops will always be slower than single loops. If the degree of the polynomial is increased to the case $N = 20$, the results are

```
times =  
0.0046    0.2019    27.6983
```

This result further displays the advantages of a calculation of **c_m** based upon the result (11.4.13).

Section 11.5. Lagrange Interpolation with MATLAB¹¹

In Section 11.1, we gave an example of a Lagrange polynomial of degree one. The example is equation (11.1.9), repeated here,

$$f_1(x) = \frac{x - x_2}{x_1 - x_2} y_1 + \frac{x - x_1}{x_2 - x_1} y_2 \quad (11.5.1)$$

In Section 11.2, we discussed an example based upon the data set

$\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$. In this case, the polynomial is equation (11.2.61), repeated,

$$\begin{aligned} f_3(x) &= c^1 l_1(x) + c^2 l_2(x) + c^3 l_3(x) + c^4 l_4(x) \\ &= \underbrace{\frac{(x - x_2)(x - x_3)(x - x_4)}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} y_1}_{\text{3rd degree polynomial in } x} + \underbrace{\frac{(x - x_1)(x - x_3)(x - x_4)}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} y_2}_{\text{3rd degree polynomial in } x} \\ &\quad + \underbrace{\frac{(x - x_1)(x - x_2)(x - x_4)}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_4)} y_3}_{\text{3rd degree polynomial in } x} + \underbrace{\frac{(x - x_1)(x - x_2)(x - x_3)}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} y_4}_{\text{3rd degree polynomial in } x} \end{aligned} \quad (11.5.2)$$

If we are given the data set $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$, the generalization of the Lagrange polynomial is to replace (11.2.1) with another, but algebraically identical, N^{th} order polynomial

$$\begin{aligned} f_N(x) &= c^1 l_1(x) + c^2 l_2(x) + c^3 l_3(x) + \dots + c^N l_N(x) + c^{N+1} l_{N+1}(x) \\ &= \underbrace{\frac{(x - x_2)(x - x_3) \cdots (x - x_{N+1})}{(x_1 - x_2)(x_1 - x_3) \cdots (x_1 - x_{N+1})} y_1}_{\text{N degree polynomial in } x} + \underbrace{\frac{(x - x_1)(x - x_3) \cdots (x - x_{N+1})}{(x_2 - x_1)(x_2 - x_3) \cdots (x_2 - x_{N+1})} y_2}_{\text{N degree polynomial in } x} \\ &\quad + \dots + \underbrace{\frac{(x - x_1)(x - x_2) \cdots (x - x_N)}{(x_{N+1} - x_1)(x_{N+1} - x_2) \cdots (x_{N+1} - x_N)} y_{N+1}}_{\text{N degree polynomial in } x} \end{aligned} \quad (11.5.3)$$

The isolation of the values $\{y_1, y_2, y_3, \dots, y_N, y_{N+1}\}$ in (11.5.3) makes explicit the fact that the polynomial $f_N(x)$ matches the data points, i.e.,

$$f_N(x_j) = y_j \quad \text{for } j = 1, 2, \dots, N+1 \quad (11.5.4)$$

If we write the formula (11.5.3)₁ as

¹¹These polynomials are named after the Italian mathematician Joseph-Louis Lagrange. Additional information can be found at http://en.wikipedia.org/wiki/Joseph_Louis_Lagrange

$$f_N(x) = \sum_{k=1}^{N+1} l_k y_k \quad (11.5.5)$$

then

$$l_k = \prod_{\substack{j=1 \\ k \neq j}}^{N+1} \frac{x - x_j}{x_k - x_j} = \frac{x - x_1}{x_k - x_1} \cdots \frac{x - x_{j-1}}{x_k - x_{j-1}} \frac{x - x_{j+1}}{x_k - x_{j+1}} \cdots \frac{x - x_{N+1}}{x_k - x_{N+1}} \quad (11.5.6)$$

It is evident from (11.5.6) that l_k is a polynomial of order N with the property that

$$l_k(x_j) = \begin{cases} 1 & \text{if } k = j \\ 0 & \text{if } k \neq j \end{cases} \quad (11.5.7)$$

This special feature of the basis $\{l_1, l_2, l_3, \dots, l_N, l_{N+1}\}$ was utilized in Section 11.2 for the case $N = 3$.

While we will not need it here, there is a useful formula for the Lagrange polynomials. Actually, it was given in Section 6.4 in the context of a discussion of the exponential linear transformation. The particular formula, adjusted to fit the notation of this chapter, is equation (6.4.5), repeated

$$\left[\begin{array}{cccccc} 1 & 1 & 1 & \cdot & \cdot & \cdot & 1 & 1 \\ x_1 & x_2 & x_3 & \cdot & \cdot & \cdot & x_N & x_{N+1} \\ x_1^2 & x_2^2 & x_3^2 & \cdot & \cdot & \cdot & x_N^2 & x_{N+1}^2 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ x_1^{N-1} & x_2^{N-1} & x_3^{N-1} & \cdot & \cdot & \cdot & x_N^{N-1} & x_{N+1}^{N-1} \\ x_1^N & x_2^N & x_3^N & \cdot & \cdot & \cdot & x_N^N & x_{N+1}^N \end{array} \right] \begin{bmatrix} l_1(x) \\ l_2(x) \\ l_3(x) \\ \vdots \\ \vdots \\ l_N(x) \\ l_{N+1}(x) \end{bmatrix} = \begin{bmatrix} 1 \\ x \\ x^2 \\ \cdot \\ \cdot \\ \cdot \\ x^{N-1} \\ x^N \end{bmatrix} \quad (11.5.8)$$

As indicated in Section 6.4, the form of (11.5.8) makes its solution by Cremer's rule convenient. For example, Cremer's rule gives the formula

$$l_1(x) = \frac{\begin{vmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ x & x_2 & x_3 & \cdots & x_N & x_{N+1} \\ x^2 & x_2^2 & x_3^2 & \cdots & x_N^2 & x_{N+1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x^{N-1} & x_2^{N-1} & x_3^{N-1} & \cdots & x_N^{N-1} & x_{N+1}^{N-1} \\ x^N & x_2^N & x_3^N & \cdots & x_N^N & x_{N+1}^N \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ x_1 & x_2 & x_3 & \cdots & x_N & x_{N+1} \\ x_1^2 & x_2^2 & x_3^2 & \cdots & x_N^2 & x_{N+1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_1^{N-1} & x_2^{N-1} & x_3^{N-1} & \cdots & x_N^{N-1} & x_{N+1}^{N-1} \\ x_1^N & x_2^N & x_3^N & \cdots & x_N^N & x_{N+1}^N \end{vmatrix}} \quad (11.5.9)$$

Equation (11.5.9) is just another way to write equation (11.5.6) for $k = 1$

As illustrated in Example 11.2.6, it is trivial to utilize the data table and evaluate each factor in (11.5.3). The conversion of that result to a polynomial with respect to the monomial basis requires the transition matrix. For the case $N = 3$ the transition matrix follows from equation (2.6.24) and takes the explicit form

$$T = \begin{bmatrix} -\frac{x_2 x_3 x_4}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} & -\frac{x_1 x_3 x_4}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} & -\frac{x_1 x_2 x_4}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_1)} & -\frac{x_1 x_2 x_3}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \\ \frac{x_3 x_4 + x_2 x_4 + x_2 x_3}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} & \frac{x_1 x_4 + x_3 x_4 + x_1 x_3}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} & \frac{x_1 x_2 + x_1 x_4 + x_2 x_4}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_4)} & \frac{x_1 x_2 + x_1 x_3 + x_2 x_3}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \\ -\frac{x_2 + x_3 + x_4}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} & -\frac{x_1 + x_3 + x_4}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} & -\frac{x_1 + x_2 + x_4}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_4)} & -\frac{x_1 + x_2 + x_3}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \\ \frac{1}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} & \frac{1}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} & \frac{1}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_4)} & \frac{1}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} \end{bmatrix} \quad (11.5.10)$$

As indicated in equation (2.6.25), the inverse of the transition matrix (11.5.10) is the transposed Vandermonde matrix

$$T^{-1} = \hat{T} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \end{bmatrix} \quad (11.5.11)$$

This fact is the reason the matrix in (11.5.10) and the one in (11.2.19) are the same. The transformation formula for components of vectors, equation (2.6.31), gives the formula that connects the components with respect to the Lagrange basis, y_1, y_2, y_3 and y_4 , to the corresponding components with respect to the monomial basis. This result is

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} -\frac{x_2x_3x_4}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & -\frac{x_1x_3x_4}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & -\frac{x_1x_2x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & -\frac{x_1x_2x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ \frac{x_3x_4+x_2x_4+x_2x_3}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & \frac{x_1x_4+x_3x_4+x_1x_3}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & \frac{x_1x_2+x_1x_4+x_2x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & \frac{x_1x_2+x_1x_3+x_2x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ -\frac{x_2+x_3+x_4}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & -\frac{x_1+x_3+x_4}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & -\frac{x_1+x_2+x_4}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & -\frac{x_1+x_2+x_3}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \\ \frac{1}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & \frac{1}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & \frac{1}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & \frac{1}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \quad (11.5.12)$$

It is because of the relationship (11.5.11) that (11.5.12) and equation (11.2.20) are identical.

Our next objective is to formulate a function file that will play the same role for Lagrange polynomial interpolation as did the function file **newton.m** for Newton polynomial interpolation and the function file **monomial.m** for monomial polynomial interpolation. We shall model the function m-file after the file **monomial.m** discussed in Section 11.3 and the file **newton.m** discussed in Section 11.4. However, because of the relationship (11.2.58), the function file is less complicated than **monomial.m** and **newton.m**. If we utilize the above preliminaries for Lagrange polynomial interpolation, the following script defines the function m-file **lagrange.m**.¹²

```
function [c_m,yvalues]=lagrange(x,y,xvalues)
%lagrange: lagrange polynomial interpolation
%N=degree of polynomial
%N+1=number of data pairs
%input:
% x = row matrix of independent variable
% values = [x1, x2, ..., xN+1]
% y = row matrix of dependent variable
```

¹² The **yvalues** calculation in the script for **lagrange.m** is taken from the MATLAB File Exchange, <http://www.mathworks.com/matlabcentral/fileexchange/4822-using-numerical-computing-with-matlab-in-the-classroom/content/polyinterp.m>. This file is also part of the collection of m-files that are provided with the online textbook, Numerical Computing with MATLAB, by Cleve Moler. As indicated in previous references, this textbook can be found at <http://www.mathworks.com/moler/chapters.html>.

```
% values = [y1, y2, ..., yN+1]
% xvalues=row matrix of points where the
% interpolated values are to be calculated.
% The last argument can be omitted.
%output:
% c_m = monomial polynomial coefficients, a
% row matrix of dimension N+1 ordered with increasing
% powers of the variable
% % yvalues=values of the interpolated polynomial
% at points xvalues
N=length(x)-1;
if length(y)~=N+1
    error('x and y must be of the same length')
end
%Build the transition matrix lagrange to monomial
for m=1:N+1
    Q=1;
    for k=1:N+1
        if k~=m
            Q=conv(Q,[1,-x(k)])/(x(m)-x(k));
        end
    end
    T(N+1:-1:1,m)=Q;
end
%Calculate the monomial polynomial coefficients
c_m=(T*y)';
%Calculate yvalues
if nargin==3
L=ones(N+1,length(xvalues)); %Preallocate
for i=1:N+1
    for j=1:N+1
        if (i~=j)
            L(i,:)=L(i,:).*(xvalues-x(j))/(x(i)-x(j));
        end
    end
end
yvalues=y*L;
end
```

Note that the above script utilizes the MATLAB polynomial command **conv**. This command, as explained in Section 9.6, computes the row matrix that defines the polynomial that is the product of the two polynomials in the argument of **conv**.

As we did in Sections 11.3 and 11.4, we shall illustrate the use of the function file **lagrange.m** by performing a Newton interpolation for the data table used in Examples 11.3.1 and 11.4.1.

Example 11.5.1: You are again given the ten pair data table

x	0	0.6981	1.3963	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
y	0	64.28	98.48	86.60	34.20	-34.20	86.60	98.48	64.28	0

and the objective is to utilize **lagrange.m** to perform the interpolation. The following MATLAB script utilizes the function m-file **lagrange.m** to calculate the various unknowns and create a graph of the input data and the output.

```

clc
clear
%Construct the data table
N=9;
x=linspace(0,2*pi,N+1);
y=100*sin(x);
%Assign the 50 equally spaced points x where the
%resulting polynomial will be evaluated
xvalues=linspace(0,2*pi,50);
%Calculate the monomial row matrix of
%polynomial coefficients and the yvalues
[c_m,yvalues]=lagrange(x,y,xvalues)

%Plot the data points
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y' , 'Rotation',0)
axis([0,7,-150,150])
grid on
hold on
%Use the values yvalues and plot the polynomial at the 50
%points in the interval (0,2*pi)
plot(xvalues,yvalues,'r','LineWidth',2)
legend('Data','Lagrange Interpolating Polynomial')
title('Example 11.5.1')

```

The figure created by this script is indistinguishable from the one created with Example 11.3.1. The monomial polynomial coefficients, **c_m** produced by the above script

```

c_m =
Columns 1 through 7
0 99.9073 0.4016 -17.3844 0.7076 0.4041 0.1675
Columns 8 through 10

```

-0.0619 0.0065 -0.0002

As one would hope, these numbers agree with those shown in equation (11.3.2). After Example 11.3.1, we compared the y values in the above table to the values computed from the polynomial that interpolates the data. We performed the same calculation after Example 11.4.1. Our conclusion was that, for this example at least, the Newton interpolation scheme was more accurate. If we make the same analysis for the polynomial produced in Example 11.5.1, the result is that the given y values and the values calculated from the Lagrange interpolation polynomial are identical. Thus, at least for this example, the Lagrange interpolation scheme is the most accurate. There are disadvantages to the Lagrange interpolation scheme. An obvious one is the complexity of the polynomial. This fact makes operations like differentiation and integration more difficult.

Section 11.6. Interpolation by MATLAB's **polyfit** Command

The interpolation problems we are studying can also be worked with **polyfit**. This MATLAB command was briefly introduced in Section 10.4 in the context of a linear regression and, again, in the context of polynomial regression in Section 10.5. When we discussed regression in Chapter 10, the degree of the polynomial, denoted by S in equation (10.1.2), was required to obey the relationship $K > S + 1$, where K equaled the number of distinct data points. In this chapter, where interpolation is the topic, we have the case where $K = S + 1$. The notation we have adopted in this chapter tends to suppresses this requirement. In this chapter we always have $K = N + 1$ and $S = N$, where N is both the degree of the polynomial and one less than the number of data points.

The syntax for **polyfit** is that given in Section 10.5, namely,

$$\text{polyfit}(\mathbf{x}, \mathbf{y}, \mathbf{N}) \quad (11.6.1)$$

where \mathbf{x} and \mathbf{y} are the data points expressed as a row or column vector, and \mathbf{N} , in this case, is the order of the polynomial. As explained above, in this case $\mathbf{N+1}$ equals the number of data points. The output of this command is a row vector

$$(p_1, p_2, p_3, \dots, p_N, p_{N+1}) \quad (11.6.2)$$

whose elements are the coefficients of a polynomial written in the form (11.2.2), repeated,

$$y(x) = p_1x^N + p_2x^{N-1} + \dots + p_Nx + p_{N+1} \quad (11.6.3)$$

Admittedly, the convention we have used where the polynomials are written in the form (11.2.1) rather than in the MATLAB form (11.6.3) is sometimes inconvenient. One must simply adopt the relationship, reflected in equation (10.5.20), that

$$(a_0, a_1, a_2, \dots, a_{N-1}, a_N) = (p_{N+1}, p_N, p_{N-1}, \dots, p_2, p_1) \quad (11.6.4)$$

or, equivalently,

$$(a_N, a_{N-1}, \dots, a_2, a_1, a_0) = (p_1, p_2, \dots, p_{N-1}, p_N, p_{N+1}) \quad (11.6.5)$$

Example 11.6.1: You are again given the ten pair data table

x	0	0.6981	1.3963	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
y	0	64.28	98.48	86.60	34.20	-34.20	86.60	98.48	64.28	0

This is the same data table used in Examples 11.3.1, 11.4.1 and 11.5.1. The following MATLAB script utilizes **polyfit** to calculate the various unknowns and create a graph of the input data and the output.

```
%Example 11.6.1
clc
clear
%Construct the data table
N=9;
x=linspace(0,2*pi,N+1);
y=100*sin(x);
%Assign the 50 equally spaced points x where the
%resulting polynomial will be evaluated
xvalues=linspace(0,2*pi,50);
%Calculate the monomial row matrix of
%polynomial coefficients by use of polyfit, polyval and the
%yvalues
p=polyfit(x,y,N)
%Calculate monomial coefficients by use of convention used
%in the text.
c_m=zeros(1,N+1);
c_m(:)=p(N+1:-1:1)
yvalues=polyval(p,xvalues)
%Plot the data points
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([0,7,-150,150])
grid on
hold on
%Use the calculated values yvalues and plot the polynomial
%at the 50 %points in the interval (0,2*pi)
plot(xvalues,yvalues,'r','LineWidth',2)
legend('Data','Polyfit Interpolating Polynomial')
title('Example 11.6.1')
```

As before, the figure created by this script is indistinguishable from the one created in Example 11.3.1. The monomial polynomial coefficients, **c_m**, are given by

```
c_m =
Columns 1 through 7
    0.0000    99.9073     0.4016   -17.3844      0.7076
    0.4041     0.1675
Columns 8 through 10
```

```
-0.0619    0.0065   -0.0002
```

Of course, these results appear to be the same as those obtained in Examples 11.3.1, 11.4.1 and 11.5.1. As with our other calculations based upon the above data set, we can measure the accuracy of the calculation for this particular example by forming **abs(y' - polyval(p, x)')**. The results of this calculation are

```
errorP =
1.0e-11 *
0.3404
0.0398
0.0213
0.0142
0.0092
0.0121
0.0057
0.0867
0.2146
0.2893
```

(11.6.6)

These results are close to those shown in equation (11.3.7) that we found for Example 11.3.1. The source of the difference lies in the details of how **polyfit** inverts the matrix (11.2.10). The MATLAB script **edit polyfit** will display the contents of the MATLAB function file **polyfit.m** and reveal that **polyfit** utilizes the *QR decomposition* mentioned in Section 4.14, 4.15 and 5.5. Another important feature is that it warns you about the ill conditioned nature of the coefficient matrix. In this example, the output resulting from the use of **polyfit** produces the warning

Warning: Polynomial is badly conditioned. Add points with distinct X values, reduce the degree of the polynomial, or try centering and scaling as described in HELP POLYFIT.

We shall discuss the accuracy of our various interpolation methods in more detail in Section 11.9.

Section 11.7. Extrapolations of Interpolations

Extrapolation is the use of interpolation to extend the data outside of the range of the data set. The result is an extension of the polynomial outside of the known range. Such practice can result in a poor approximation. The following example illustrates this assertion.

Example 11.7.1: In this example, we shall utilize the results of the data table we have used in several of our previous examples, namely,

x	0	0.6981	1.3963	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
y	0	64.28	98.48	86.60	34.20	-34.20	86.60	98.48	64.28	0

The polynomial that interpolates this data table is given in Examples 11.3.1, 11.4.1, 11.5.1 and 11.6.1. While there are some questions about which polynomial is the most accurate reflection of the formula used to generate the table, namely, $y = 100\sin(x)$, the various graphs suggests that the interpolating polynomials are all rather accurate. The question of this example is whether or not this accuracy is retained if we extrapolate the polynomial to a value of $x = 9$. In other words, how close does the interpolating polynomial evaluated at $x = 9$ replicate the value of $100\sin(9) = 41.2118$. The following MATLAB script derives the interpolating polynomial from the above data table utilizing the function file **newton.m**, plots the resulting polynomial for values of x in the range $[0, 9]$, and plots on the same axes the function $y = 100\sin(x)$.

```
% Example 11.7.1
clc
clear
%Construct the data table
N=9;
x=linspace(0,2*pi,N+1);
y=100*sin(x);

%Plot the data points
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)

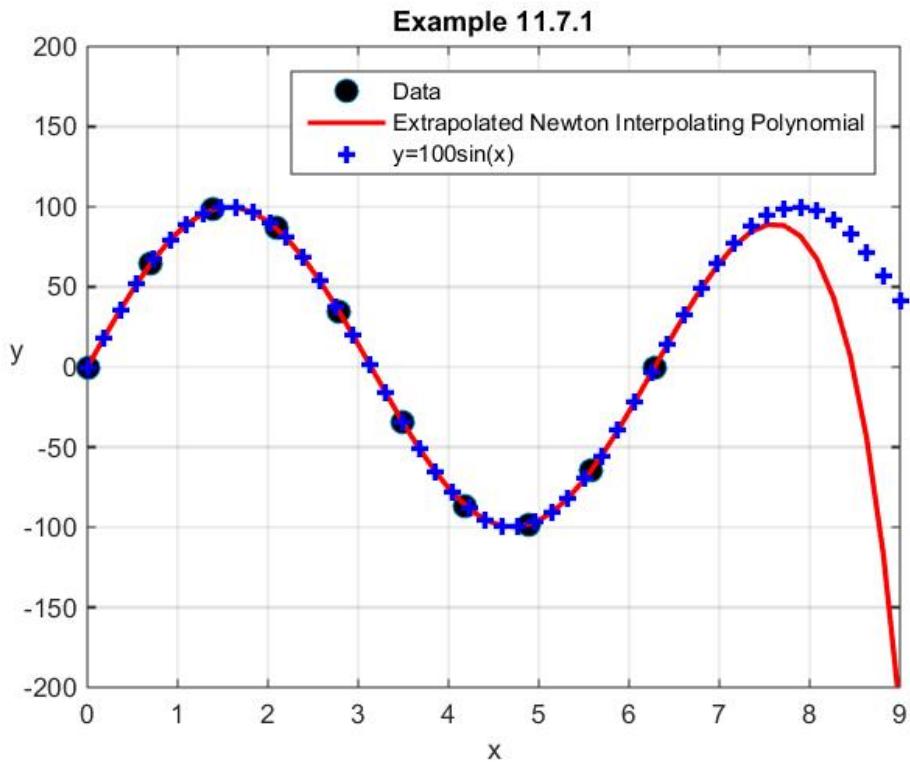
axis([0,9,-200,200])
grid on
hold on
%Calculate the newton and monomial row matrices of
%polynomial coefficients
[c_m,c_n]=newton(x,y)

%Plot the values at 50 xvalues in the range [0,9]
xvalues=linspace(0,9,50)
```

```
%Use nested Horner method to evaluate Newton
%interpolating polynomial at xvalues
h=c_n(N+1)
for q=N:-1:1
    h=c_n(q)+h.*(xvalues-x(q))
end
yvalues=h
plot(xvalues,yvalues,'r','LineWidth',2)

%Plot exact function, y=100sin(x) for values xvalues
plot(xvalues,100*sin(xvalues),'+b','LineWidth',2)
legend('Data','Extrapolated Newton Interpolating
Polynomial','y=100sin(x)')
title('Example 11.7.1')
```

The result is the plot



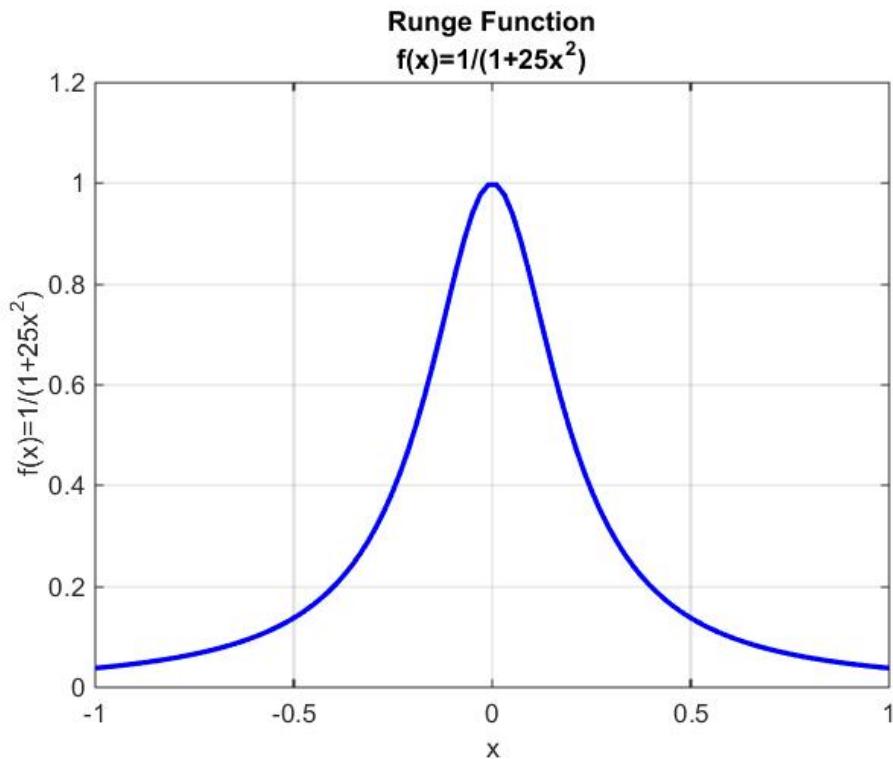
This figure illustrates that the data can fit the polynomial in the range, while at the same time be greatly in error when one tries to extrapolate outside of the range.

Section 11.8. Approximation of a Known Function: Oscillations

Given the capacity to create polynomials that interpolates data, it is natural to ask the question whether or not it is sensible to approximate a known function that is *not a polynomial* by a *polynomial*. Give a function that is *not a polynomial*, one could always evaluate it at a collection of points and create from those values a polynomial of appropriate degree. Since we have begun with a known function, we can plot the function, plot the polynomial that we hope is a good approximation and make a judgment. It turns out that a counter intuitive result can be obtained. The *more* points we use as data points for the known function, the *less* accurate is the interpolating polynomial. This assertion is usually illustrated by starting with the Runge function¹³

$$f(x) = \frac{1}{1+25x^2} \quad (11.8.1)$$

If we plot this function in the interval $[-1,1]$, it takes the shape



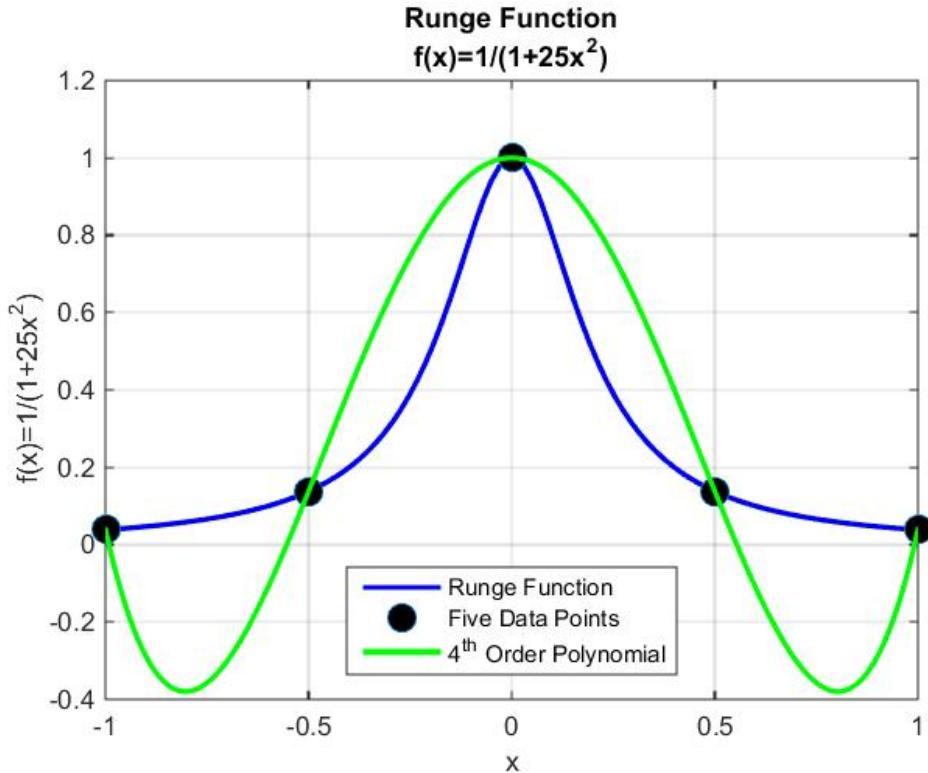
¹³ Runge, Carl Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten, Zeitschrift für Mathematik und Physik, vol. 46, 1901. Information about the German mathematician Carl David Tolmé Runge can be found at http://en.wikipedia.org/wiki/Carl_David_Tolm%C3%A9_Runge.

Next, we ask whether or not we can fit a polynomial of degree 4 to the values of

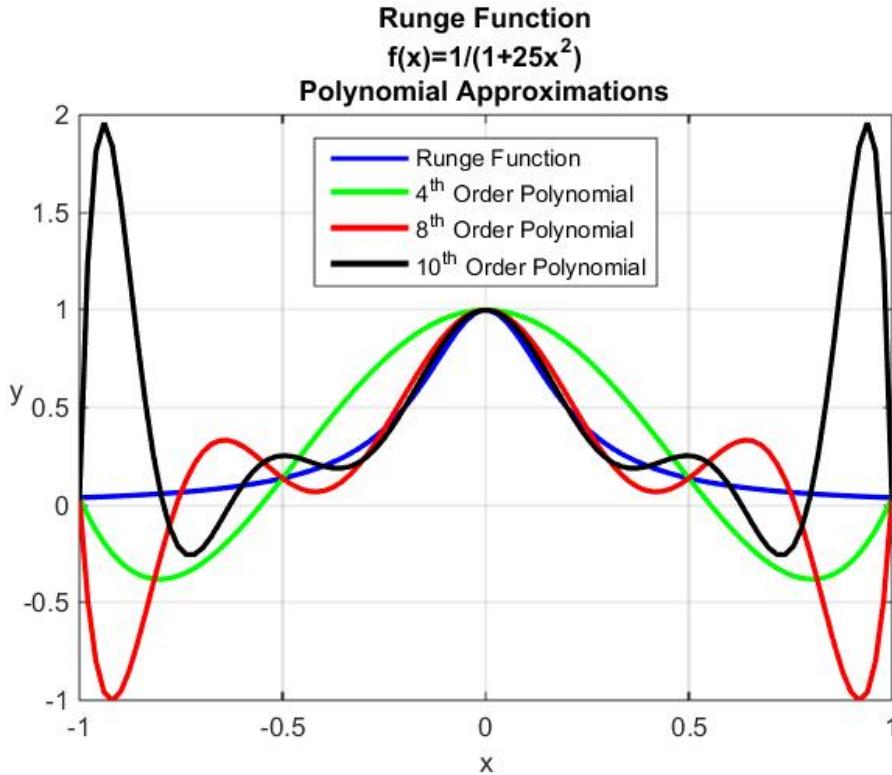
$f(x) = \frac{1}{1+25x^2}$ at five equal points. The data set that corresponds to this choice is

x	-1.0000	-0.5000	0	0.5000	1.0000
y	0.0385	0.1379	1.0000	0.1379	0.0385

The next step is to use **monomial.m**, **newton.m**, **lagrange.m** or **polyfit** to generate the fourth order polynomial that passes through these points. In this case, we shall use **polyfit**. If a plot of this polynomial is superimposed on the above figure, we obtain



In an effort to obtain a more accurate approximation, it is reasonable to repeat the above calculation but with more data points. If we use a data set based upon *nine* equally spaced data points (an eighth order polynomial) and another one based upon *eleven* equally spaced data points (a tenth order polynomial) and then superimpose these curves on the last figure, the result is



This figure confirms the point made above. Additional data points do not increase the accuracy. Near the boundaries, the higher order polynomials oscillate greatly and become less accurate than lower order ones.

It is perhaps useful to record the MATLAB script that will produce the above figure.

```

clc
clear
%create the first set of data.  5 points
x1=linspace(-1,1,5)
y1=1./(1+25*x1.^2)
%Use polyfit to create the 4th order polynomial that
%interpolates these five points
p4=polyfit(x1,y1,4)

%create the second set of data.  9 points
x2=linspace(-1,1,9)
y2=1./(1+25*x2.^2)
%Use polyfit to create the 8th order polynomial that
%interpolates these nine points
p8=polyfit(x2,y2,8)

%create the third set of data.  11 points
x3=linspace(-1,1,11)

```

```
y3=1./(1+25*x3.^2)
%Use polyfit to create the polynomial that interpolates
these 10 %points
p10=polyfit(x3,y3,10)

%Plot these three polynomials and the actual
%function using 100 points
x=linspace(-1,1)
% evaluate the function and the three polynomials
% at these 100 %points
y=1./(1+25*x.^2)
Y4=polyval(p4,x)
Y8=polyval(p8,x)
Y10=polyval(p10,x)

% plot four curves on a common axis
plot(x,y,'b',x,Y4,'g',x,Y8,'r',x,Y10,'k','LineWidth',2)
grid on
xlabel('x')
ylabel('y','Rotation',0)
legend('Runge Function','4^{th} Order Polynomial',...
    '8^{th} Order Polynomial',...
    '10^{th} Order Polynomial','Location','North')
title({'Runge Function','f(x)=1/(1+25x^2)','Polynomial
Approximations'})
```

Section 11.9. Issues of Numerical Accuracy

At a certain fundamental level, the polynomial interpolation schemes discussed in this chapter are all equivalent. They differ from each other by a change of basis. In other words, analytically the interpolating polynomials determined by the choice of monomial, Newton or Lagrange polynomials are identical. However, as we have briefly mentioned in Sections 11.2 and 11.3, there are numerical advantages associated with the Newton and Lagrange polynomial choices. In this Section, we shall briefly explore these advantages. One source of numerical errors arises from the ill conditioned matrix (11.2.17), the transposed Vandermonde matrix. In this Section, we shall see an example that illustrates the serious nature of this problem. Our example in Section 11.3 did involve an ill conditioned matrix, but the errors were not evident in the answer. One scheme that is frequently used for monomial interpolation schemes is to precondition the data by shifting and scaling the numbers. We shall discuss this method in this section. It will be shown that this method is no more than yet another change of basis. It is a basis change from one monomial basis to another that is selected in a special fashion. During the discussion of this method, it will be explained how **polyfit** will implement a version of this preconditioning.

In Examples 11.3.1, 11.4.1, 11.5.1 and 11.6.1 we started with the same data table, namely,

x	0	0.6981	1.3963	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
y	0	64.28	98.48	86.60	34.20	-34.20	86.60	98.48	64.28	0

and derived interpolating polynomials based upon the Monomial, the Newton, the Lagrange and the **polyfit** schemes. We measured the accuracy of each calculation by comparing the given y values to those calculated from the interpolating polynomial. We can summarize the results (11.3.7), (11.4.14) and (11.6.6) in the following table. Also included in the table are the results of the observation in Section 11.5 that the Lagrange interpolation scheme produces zeros for the calculated errors.

	Monomial	Newton	Lagrange	Polyfit
x	(abs(y' - yvalues'))x(10)^{13}			
0	0.0000	0.0000	0	0.3404
0.6981	0.0313	0.0000	0	0.0398
1.3963	0.2430	0.0000	0	0.0213
2.0944	0.0725	0.001421	0	0.0142
2.7925	0.0668	0.000711	0	0.0092
3.4907	0.5052	0.005684	0	0.0121
4.1888	0.0455	0.001421	0	0.0057
4.8869	0.1904	0.002842	0	0.0867
5.5851	0.0540	0.007105	0	0.2146
6.2832	0.2078	0.006480	0	0.2893

This table only holds for the particular examples in Sections 11.3, 11.4, 11.5 and 11.6. Thus, one cannot make broad conclusions about the relative accuracy of the four interpolation schemes. The fact that we are comparing the output of the interpolating polynomials to the y values in the table also makes the Lagrange column of the table a misleading measure of its accuracy. Never the less, the above table suggests, but does not prove, that the Newton interpolation scheme improves on the Monomial scheme. It also suggests that **polyfit** and the Monomial scheme yield results that have comparable accuracy. Again, it is risky to make generalizations based upon one example.

It is instructive to look at an example that displays in a more dramatic fashion the problems of Monomial interpolations. The following example is one where the ill conditioned nature of the coefficient matrix causes a significant error.

Example 11.9.1¹⁴: You are given the following data table and the problem is to use the Monomial, Newton, and Lagrange interpolating schemes to derive interpolating polynomials. The data table is

x	1050	1050.5	1051	1052	1053	1054
y	3	2	-1	1	0	-2

If one simply performs the Monomial interpolation calculation with the script

```
clc
clear
N=5
x=[1050,1050.5,1051,1052,1053,1054]
```

¹⁴ This example is motivated by one that can be found at
http://www.nada.kth.se/kurser/kth/2d1213/05_06/utdelat/kap3.pdf

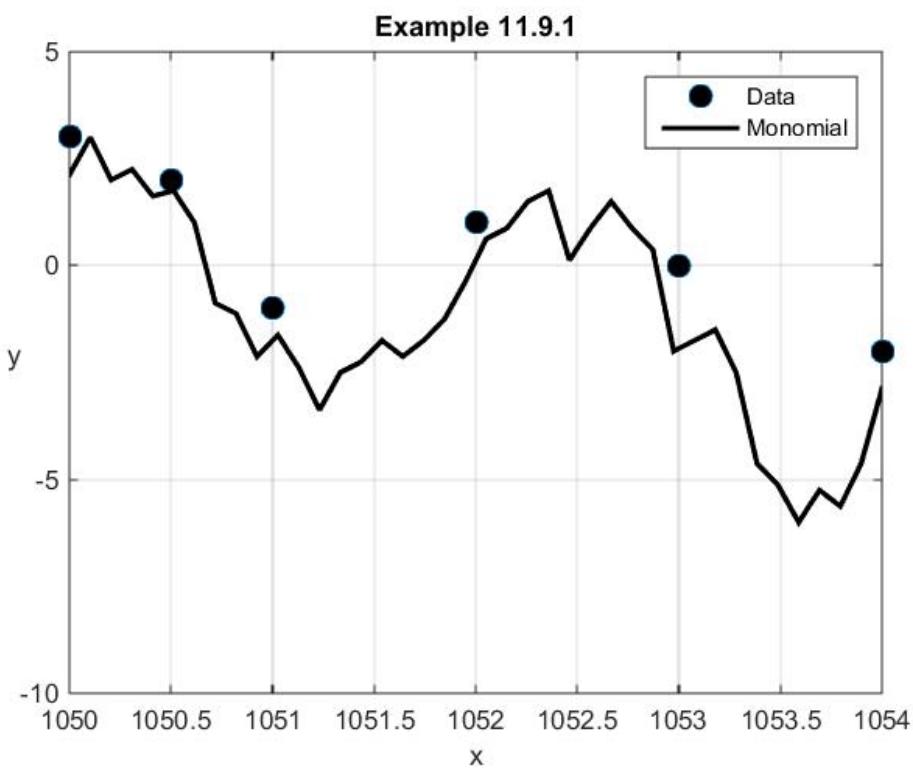
```

y=[3,2,-1,1,0,-2]
%Plot of data
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([1050,1054,-10,5])
title('Example 11.9.1')
grid on
hold on

%Monomial Interpolation
xvalues=linspace(1050,1054,40)
[c_m,yvalues]=monomial(x,y,xvalues)
%Plot of Monomial Interpolation
plot(xvalues,yvalues,'k','LineWidth',2)
legend('Data','Monomial')

```

The following unacceptable result is obtained



The numerical errors in this case are so significant that the interpolating polynomial does not even pass through the given x values. If one examines the MATLAB script in Section 11.3 for the function m-file, **monomial.m**, there are two sources of the numerical errors. They are

- 1) The script, $\mathbf{c_m}=(\mathbf{A}\backslash \mathbf{y}')'$, that involves the inverse of the ill conditioned matrix A_M .

- 2) The script, `yvalues=polyval(p,xvalues)`, that takes the polynomial coefficients calculated in 1) and calculates the corresponding values of the interpolating polynomial. Among other things, `polyval` utilizes the Horner method which can, for the data table above, involve round off from the calculation of differences of large numbers.

While `polyval` can introduce errors, the one we wish to discuss is the one arising from the ill conditioned matrix A_M . It is elementary to cause MATLAB to show that in this case

$$A_M = (10)^{15} \begin{bmatrix} 0.000000000000001 & 0.000000000001050 & 0.000000001102500 & 0.000001157625000 & 0.001215506250000 & 1.276281562500000 \\ 0.000000000000001 & 0.000000000001051 & 0.000000001103550 & 0.000001159279538 & 0.001217823154275 & 1.279323223565953 \\ 0.000000000000001 & 0.000000000001051 & 0.000000001104601 & 0.000001160935651 & 0.001220143369201 & 1.282370681030251 \\ 0.000000000000001 & 0.000000000001052 & 0.000000001106704 & 0.000001164252608 & 0.001224793743616 & 1.288483018284032 \\ 0.000000000000001 & 0.000000000001053 & 0.000000001108809 & 0.000001167575877 & 0.001229457398481 & 1.294618640600493 \\ 0.000000000000001 & 0.000000000001054 & 0.000000001110916 & 0.000001170905464 & 0.001234134359056 & 1.300777614445024 \end{bmatrix} \quad (11.9.1)$$

The large variation of the values of the elements of (11.9.1) is an indication of the ill conditioned nature of A_M . Given the matrix (11.9.1), the various condition numbers introduced in Section 7.4 and calculated in Sections 11.3 and 11.4 assume the values

$$\text{cond}(A, 'fro') = 1.9471(10)^{30} \quad (11.9.2)$$

$$\text{cond}(A, 1) = 3.2381(10)^{30} \quad (11.9.3)$$

$$\text{cond}(A, \text{inf}) = 1.5757(10)^{30} \quad (11.9.4)$$

$$\text{cond}(A, 2) = 5.5241(10)^{26} \quad (11.9.5)$$

These large condition numbers confirm that A_M is ill-conditioned. Each MATLAB calculation leading to the results (11.9.2) through (11.9.5) also gives the warning

**Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 3.088208e-31.**

s

As explained in Section 7.4, in equation (7.4.33), `rcond` is the *reciprocal condition number*.

The ill conditioned nature of (11.9.1) tells us that the monomial polynomial coefficients `c_m=(A\y')'` are incorrect. When these incorrect values are used to calculate `yvalues` even more errors are introduced.

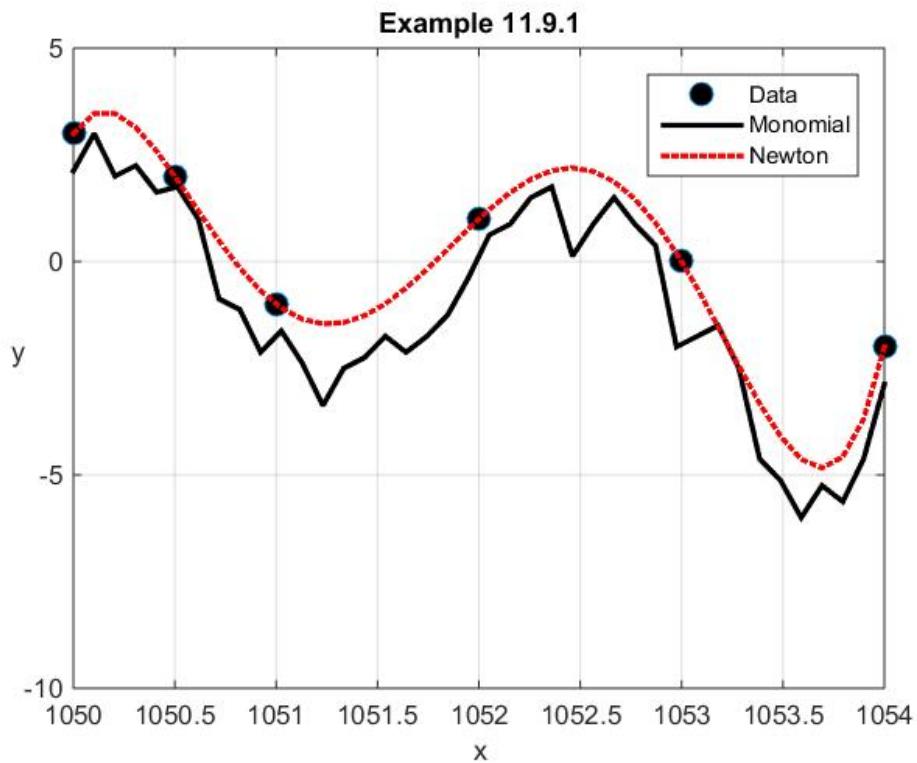
Given the poor results reflected in the above figure, it is instructive to work the same problem utilizing a Newton interpolation. The graph in this case is obtained by augmenting the above script as follows:

```
clc
clear
N=5
x=[1050,1050.5,1051,1052,1053,1054]
y=[3,2,-1,1,0,-2]
%Plot of data
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([1050,1054,-10,5])
title('Example 11.9.1')
grid on
hold on

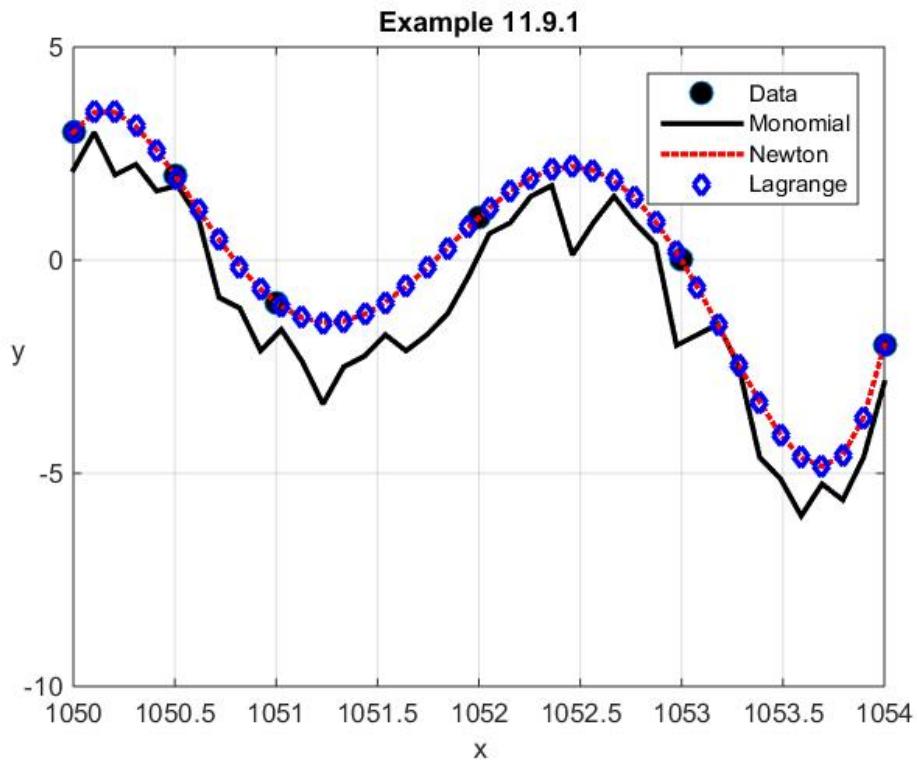
%Monomial Interpolation
xvalues=linspace(1050,1054,40)
[c_m,yvalues]=monomial(x,y,xvalues)
%Plot of Monomial Interpolation
plot(xvalues,yvalues,'k','LineWidth',2)
legend('Data','Monomial')

%Newton Interpolation
[c_mn,c_n,ynvalues]=newton(x,y,xvalues)
%Plot of Newton Interpolation
plot(xvalues,ynvalues,'r:','LineWidth',2)
legend('Data','Monomial','Newton')
```

The figure this script produces is



This figure shows, in a dramatic fashion, the increased accuracy of the Newton interpolation method. If the Lagrange method is also used, the above figure is modified to be



This figure illustrates that the Lagrange interpolation method appears to be as accurate as the Newton method, at least for this example.¹⁵

In the introduction to this section it was mentioned that *Monomial interpolation* can be effectively implemented when one preconditions the data in a special way. The preconditioning shifts and scales the numbers by a change of variables of the form

$$z = \frac{x - \mu_1}{\mu_2} \quad (11.9.6)$$

where μ_1 and μ_2 are real numbers to be prescribed. For example, we can use

$$\mu_1 = \frac{1}{2}(x_{N+1} + x_1) \quad (11.9.7)$$

and

$$\mu_2 = \frac{1}{2}(x_{N+1} - x_1) \quad (11.9.8)$$

The new data table will be such that the y values will be given at points relative to the original interval's midpoint and at points normalized by the one half range of the original interval. The normalization in this case has the feature that $-1 \leq z \leq 1$. Another important pair of choices for μ_1 and μ_2 are

$$\mu_1 = \frac{1}{N+1} \sum_{j=1}^{N+1} x_j \quad (11.9.9)$$

and

$$\mu_2 = \sqrt{\frac{1}{N} \sum_{j=1}^{N+1} (x_j - \mu_1)^2} \quad (11.9.10)$$

It follows from (10.2.10) that μ_1 is the *arithmetic mean* of the x values and, from (10.2.13), that μ_2 is the *standard deviation* of the x values. As explained in Section 10.2, the mean and the

¹⁵ There are numerical advantages of the Newton method over the Lagrange method. These fall into the category of numerical stability. A discussion of these advantages, as well as the MATLAB script for a graphic user interface (GUI) that shows the advantages, can be found at

http://www.maths.cam.ac.uk/undergrad/course/na/ib/stability_of_interpolation/stability_of_interpolation.php#3. A discussion of the error estimates for Lagrange interpolations can be found in Prenter, P. M., *Spines and Variational Methods*, Dover Publications, 2008.

standard deviation are given by the MATLAB commands **mean(x)** and **std(x)**, respectively.

If we view the transformation (11.9.6) in the context of a basis change for the vector space of polynomials \mathcal{P}_N , then the basis of polynomials $\{q_1, q_2, \dots, q_{N+1}\}$, where

$$q_j(x) = x^{j-1} \quad \text{for } j = 1, 2, \dots, N+1 \quad (11.9.11)$$

are connected to the basis of polynomials $\{\hat{q}_1, \hat{q}_2, \dots, \hat{q}_{N+1}\}$, where

$$\hat{q}_j(z) = z^{j-1} \quad \text{for } j = 1, 2, \dots, N+1 \quad (11.9.12)$$

by the change of basis (11.2.12), repeated,

$$\hat{q}_j(z) = \sum_{k=1}^{N+1} T_j^k q_k(x) \quad (11.9.13)$$

If we make use of (11.9.12), (11.9.11) and (11.9.6), it follows that

$$\begin{aligned} \hat{q}_j(z) &= z^{j-1} = \left(\frac{x - \mu_1}{\mu_2} \right)^{j-1} = \frac{1}{\mu_2^{j-1}} \sum_{k=1}^j \binom{j-1}{k-1} x^{k-1} (-\mu_1)^{j-k} \\ &= \frac{1}{\mu_2^{j-1}} \sum_{k=1}^j \binom{j-1}{k-1} q_k(x) (-\mu_1)^{j-k} \end{aligned} \quad (11.9.14)$$

where $\binom{j-1}{k-1}$ is the binomial coefficient

$$\binom{j-1}{k-1} = \frac{(j-1)!}{(k-1)!(j-k)!} \quad (11.9.15)$$

Therefore, the transition matrix $[T_j^k]$ has the components

$$T_j^k = \begin{cases} \frac{1}{\mu_2^{j-1}} \binom{j-1}{k-1} (-\mu_1)^{j-k} & \text{for } k = 1, 2, \dots, j \\ 0 & \text{for } k = j+1, \dots, N+1 \end{cases} \quad (11.9.16)$$

For example, when $N = 5$, from (11.9.16), the transition matrix that defines the basis $\{\hat{q}_1, \hat{q}_2, \dots, \hat{q}_{N+1}\}$ is

$$\left[T_j^k \right] = \begin{bmatrix} 1 & -\frac{\mu_1}{\mu_2} & \frac{\mu_1^2}{\mu_2^2} & -\frac{\mu_1^3}{\mu_2^3} & \frac{\mu_1^4}{\mu_2^4} & -\frac{\mu_1^5}{\mu_2^5} \\ 0 & \frac{1}{\mu_2} & -\frac{2\mu_1}{\mu_2^2} & \frac{3\mu_1^2}{\mu_2^3} & -\frac{4\mu_1^3}{\mu_2^4} & \frac{5\mu_1^4}{\mu_2^5} \\ 0 & 0 & \frac{1}{\mu_2^2} & -\frac{3\mu_1}{\mu_2^3} & \frac{6\mu_1^2}{\mu_2^4} & -\frac{10\mu_1^3}{\mu_2^5} \\ 0 & 0 & 0 & \frac{1}{\mu_2^3} & -\frac{4\mu_1}{\mu_2^4} & \frac{10\mu_1^2}{\mu_2^5} \\ 0 & 0 & 0 & 0 & \frac{1}{\mu_2^4} & -\frac{5\mu_1}{\mu_2^5} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\mu_2^5} \end{bmatrix} \quad (11.9.17)$$

Example 11.9.2: You are given the data table of Example 11.9.2 and the problem is to adopt the preconditioning defined by (11.9.6), (11.9.9) and (11.9.10) above and apply the Monomial interpolating scheme to the new set of (z, y) data. After performing the Monomial scheme, the answers will be converted back to the original (x, y) data. Given the data table of Example 11.9.1, repeated,

x	1050	1050.5	1051	1052	1053	1054
y	3	2	-1	1	0	-2

it follows from (11.9.9) that

$$\mu_1 = \frac{1}{N+1} \sum_{j=1}^{N+1} x_j = 1051.75 \quad (11.9.18)$$

and from (11.9.10) that

$$\mu_2 = \sqrt{\frac{1}{N} \sum_{j=1}^{N+1} (x_j - \mu_1)^2} = 1.541 \quad (11.9.19)$$

In terms of the independent variable z , defined by (11.9.6), the above table becomes

z	-1.1355	-0.8111	-0.4867	0.1622	0.8111	1.4600
y	3	2	-1	1	0	-2

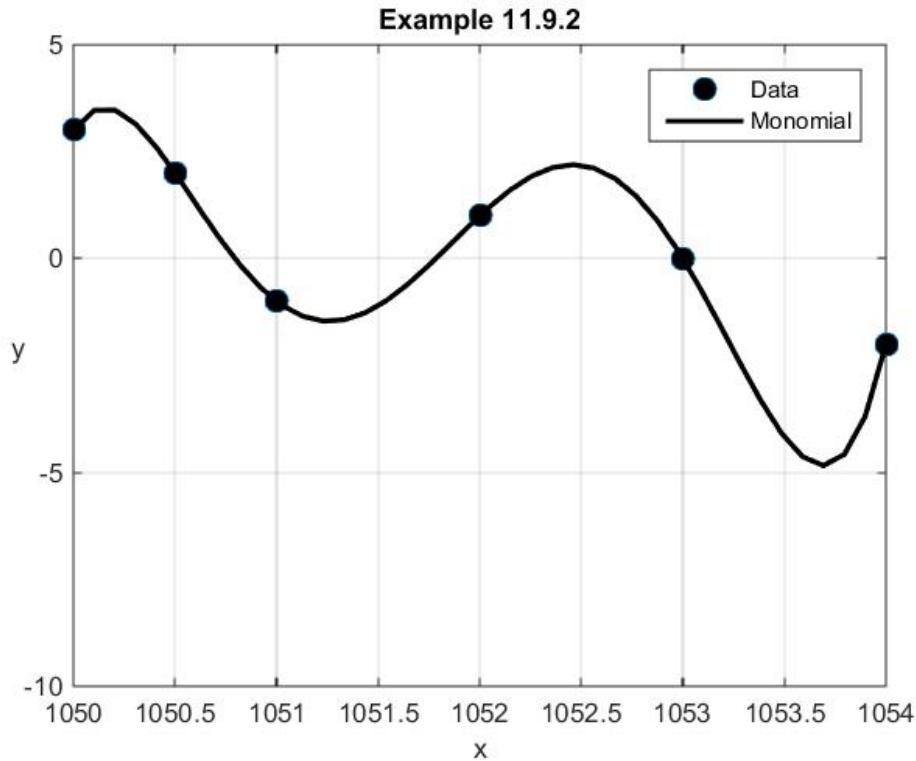
The MATLAB script that performs the preconditioning, the Monomial interpolation and shifts the results back to (x, y) variables is

```
clc
clear
N=5
x=[1050,1050.5,1051,1052,1053,1054]
mu1=mean(x)
mu2=std(x)
z=(x-mu1)/mu2
y=[3,2,-1,1,0,-2]

%Plot of data: x vs y
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([1050,1054,-10,5])
title('Example 11.9.2')
grid on
hold on

%Monomial Interpolation: z vs y
zvalues=linspace(z(1),z(6),40)
[c_m,yvalues]=monomial(z,y,zvalues)
%Plot of Monomial Interpolation:xvalues vs y
xvalues=mu2*zvalues+mu1
plot(xvalues,yvalues,'k','LineWidth',2)
legend('Data','Monomial')
```

The result is the plot



This figure displays none of the problems displayed for Monomial interpolation in Example 11.9.1. A similar improvement is obtained if the preconditioning is based upon equations (11.9.7) and (11.9.8).

The improvement of Monomial interpolation achieved by the preconditioning described above can be efficiently implemented by use of a special feature of **polyfit**. The syntax **[p,s,mu]=polyfit(x,y,N)** will produce the polynomial coefficients **p** of a polynomial in the variable $z = \frac{x - \mu_1}{\mu_2}$ and a structure **s** that can be used with **polyval** to obtain error

estimates and a matrix **mu=[mean(x), std(x)]** that defines the variable z . If this feature of **polyfit** is utilized the above figure can be created by the revised script

```

clc
clear
N=5
x=[1050,1050.5,1051,1052,1053,1054]
y=[3,2,-1,1,0,-2]
%Plot of data: x vs y
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([1050,1054,-10,5])
title('Example 11.9.2 with polyfit')
grid on

```

```
hold on

%Monomial Interpolation Utilizing Polyfit
[p,s,mu]=polyfit(x,y,N)

%Plot of Interpolating Polynomial:xvalues vs y
xvalues=linspace(1050,1054,40)
zvalues=(xvalues-mu(1))/mu(2)
yvalues=polyval(p,zvalues)
plot(xvalues,yvalues,'k','LineWidth',2)
legend('Data','polyfit')
```

The resulting figure is indistinguishable from the one above.

Section 11.10. Piecewise Lagrange Interpolation

Interpolating schemes have more variations that we are able to discuss in this work. One that we have not discussed and that is especially deserving of a small amount of attention is the use of piecewise polynomials to patch together an approximation of a data set. In this section, we shall discuss interpolation based upon piecewise Lagrange polynomials. This particular choice has application in the study of the one dimensional finite element method.

As in Section 11.2, we are given the $N + 1$ data set $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$, where $x_1 < x_2 < \dots < x_{N+1}$. However, rather than find a N^{th} order polynomial that interpolates the data, we shall partition the interval $[x_1, x_{N+1}]$ into a finite set of subintervals and calculate interpolating polynomials for each subinterval. We shall force the individual polynomials to agree at their common points at the juncture of the subintervals. For example, if $N = 9$, as was the case in Example 11.3.1, Example 11.4.1, Example 11.5.1 and Example 11.6.1, we can partition the interval $[x_1, x_9]$ into the nine segments

$$\begin{aligned}[x_1, x_9] = & [x_1, x_2] \cup [x_2, x_3] \cup [x_3, x_4] \cup [x_4, x_5] \\ & \cup [x_5, x_6] \cup [x_6, x_7] \cup [x_7, x_8] \cup [x_8, x_9] \cup [x_9, x_{10}]\end{aligned}\tag{11.10.1}$$

and attempt to implement a piecewise interpolation involving nine linear polynomials as follows:

$$f(x) = \begin{cases} a_0 + a_1 x & \text{for } x \in [x_1, x_2] \\ a_2 + a_3 x & \text{for } x \in [x_2, x_3] \\ a_4 + a_5 x & \text{for } x \in [x_3, x_4] \\ a_6 + a_7 x & \text{for } x \in [x_4, x_5] \\ a_8 + a_9 x & \text{for } x \in [x_5, x_6] \\ a_{10} + a_{11} x & \text{for } x \in [x_6, x_7] \\ a_{12} + a_{13} x & \text{for } x \in [x_7, x_8] \\ a_{14} + a_{15} x & \text{for } x \in [x_8, x_9] \\ a_{16} + a_{17} x & \text{for } x \in [x_9, x_{10}]\end{cases}\tag{11.10.2}$$

The eighteen polynomial coefficients are determined by evaluating (11.10.2) at the ten points of the data set and, in addition, force the matching of the linear polynomials at the eight common points $x_2, x_3, x_4, x_5, x_6, x_7, x_8$ and x_9 . Also, we can partition the interval $[x_1, x_9]$ into the three segments

$$[x_1, x_9] = [x_1, x_4] \cup [x_4, x_7] \cup [x_7, x_{10}]\tag{11.10.3}$$

and attempt to implement a piecewise interpolation involving three cubic polynomials as follows:

$$f(x) = \begin{cases} a_0 + a_1x + a_2x^2 + a_3x^3 & \text{for } x \in [x_1, x_4] \\ a_4 + a_5x + a_6x^2 + a_7x^3 & \text{for } x \in [x_4, x_7] \\ a_8 + a_9x + a_{10}x^2 + a_{11}x^3 & \text{for } x \in [x_7, x_{10}] \end{cases} \quad (11.10.4)$$

The twelve polynomial coefficients are determined by evaluating (11.10.4) at the ten points of the data set and, in addition, force the matching of the first two polynomials at the common point x_4 and the matching of the second pair of polynomials at the common point x_7 .

Given a data set with $N+1$ points, the degree of each interpolating polynomial is not arbitrary. If M is that degree, the ratio $\frac{N}{M}$ must be a positive integer. In Sections 11.1 through 11.9, we always had $\frac{N}{M}=1$. In the example (11.10.2), we have $\frac{N}{M}=\frac{9}{1}=9$. In a similar fashion, in the example (11.10.4), we have $\frac{N}{M}=\frac{9}{3}=3$. It should be evident that the positive integer $\frac{N}{M}$ represents the number of interpolating polynomials which in turn represents the number of segments of the piecewise polynomial. For the example where $N=9$, the three cases where $\frac{N}{M}$ is a positive integer are for $M=1$, nine segments and first order polynomials, $M=3$, three segments and cubic polynomials and $M=9$, one segment and a ninth order polynomial.

Within each segment of the piecewise polynomial to be calculated, the interpolation methods earlier in this chapter can be used to determine the polynomial coefficients. A practical approach at this point is to utilize the Lagrange interpolating scheme and replace (11.10.2) by

$$f(x) = \begin{cases} \frac{x - x_2}{x_1 - x_2} y_1 + \frac{x - x_1}{x_2 - x_1} y_2 & \text{for } x \in [x_1, x_2] \\ \frac{x - x_3}{x_2 - x_3} y_2 + \frac{x - x_2}{x_3 - x_2} y_3 & \text{for } x \in [x_2, x_3] \\ \frac{x - x_4}{x_3 - x_4} y_3 + \frac{x - x_3}{x_4 - x_3} y_4 & \text{for } x \in [x_3, x_4] \\ \frac{x - x_5}{x_4 - x_5} y_4 + \frac{x - x_4}{x_5 - x_4} y_5 & \text{for } x \in [x_4, x_5] \\ \frac{x - x_6}{x_5 - x_6} y_5 + \frac{x - x_5}{x_6 - x_5} y_6 & \text{for } x \in [x_5, x_6] \\ \frac{x - x_7}{x_6 - x_7} y_6 + \frac{x - x_6}{x_7 - x_6} y_7 & \text{for } x \in [x_6, x_7] \\ \frac{x - x_8}{x_7 - x_8} y_7 + \frac{x - x_7}{x_8 - x_7} y_8 & \text{for } x \in [x_7, x_8] \\ \frac{x - x_9}{x_8 - x_9} y_8 + \frac{x - x_8}{x_9 - x_8} y_9 & \text{for } x \in [x_8, x_9] \\ \frac{x - x_{10}}{x_9 - x_{10}} y_9 + \frac{x - x_9}{x_{10} - x_9} y_{10} & \text{for } x \in [x_9, x_{10}] \end{cases} \quad (11.10.5)$$

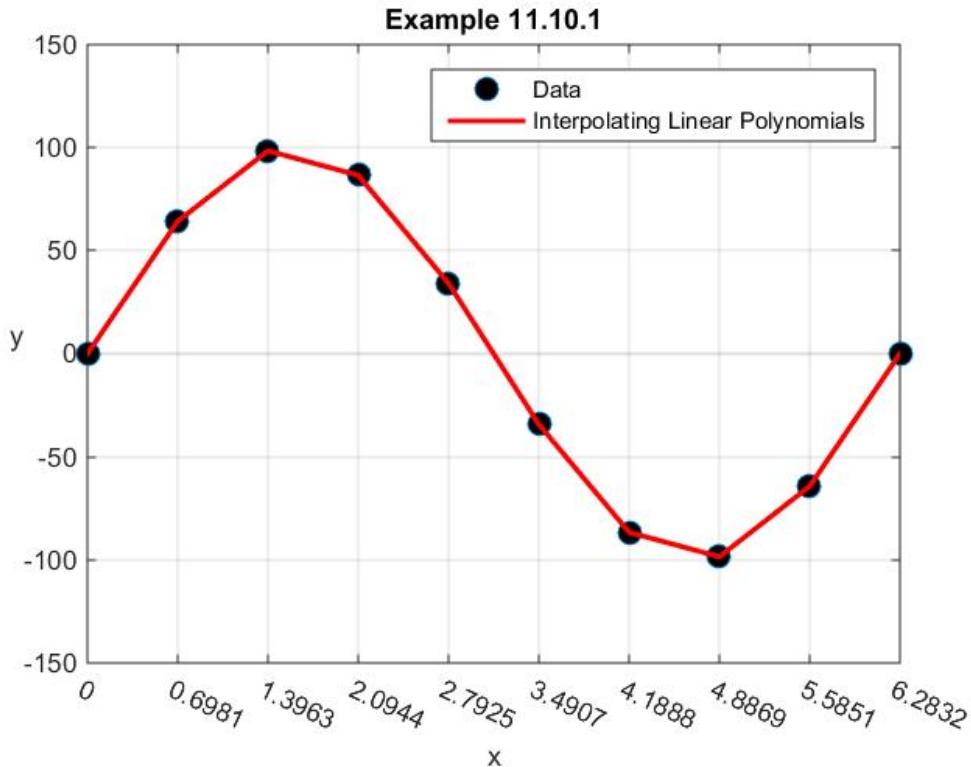
and (11.10.4) by

$$f(x) = \begin{cases} \frac{(x - x_2)(x - x_3)(x - x_4)}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} y_1 + \frac{(x - x_1)(x - x_3)(x - x_4)}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} y_2 \\ \quad + \frac{(x - x_1)(x - x_2)(x - x_4)}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_4)} y_3 + \frac{(x - x_1)(x - x_2)(x - x_3)}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} y_4 & \text{for } x \in [x_1, x_4] \\ \frac{(x - x_5)(x - x_6)(x - x_7)}{(x_4 - x_5)(x_4 - x_6)(x_4 - x_7)} y_4 + \frac{(x - x_4)(x - x_6)(x - x_7)}{(x_5 - x_4)(x_5 - x_6)(x_5 - x_7)} y_5 \\ \quad + \frac{(x - x_4)(x - x_5)(x - x_7)}{(x_6 - x_4)(x_6 - x_5)(x_6 - x_7)} y_6 + \frac{(x - x_4)(x - x_5)(x - x_6)}{(x_7 - x_4)(x_7 - x_5)(x_7 - x_6)} y_7 & \text{for } x \in [x_4, x_7] \\ \frac{(x - x_8)(x - x_9)(x - x_{10})}{(x_7 - x_8)(x_7 - x_9)(x_7 - x_{10})} y_7 + \frac{(x - x_7)(x - x_9)(x - x_{10})}{(x_8 - x_7)(x_8 - x_9)(x_8 - x_{10})} y_8 \\ \quad + \frac{(x - x_7)(x - x_8)(x - x_{10})}{(x_9 - x_7)(x_9 - x_8)(x_9 - x_{10})} y_9 + \frac{(x - x_7)(x - x_8)(x - x_9)}{(x_{10} - x_7)(x_{10} - x_8)(x_{10} - x_9)} y_{10} & \text{for } x \in [x_7, x_{10}] \end{cases} \quad (11.10.6)$$

Example 11.10.1: As we have done several times, we shall begin with the data table

x	0	0.6981	1.3963	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
y	0	64.28	98.48	86.60	34.20	-34.20	86.60	98.48	64.28	0

and apply the solution (11.10.5). The resulting plot of this piecewise interpolation turns out to be



The MATLAB script that will produce the above figure is

```

clc
clear
%Construct the data table
N=9;
x=linspace(0,2*pi,N+1);
y=100*sin(x);

%Plot the data points
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([0,2*pi,-150,150])
grid on
hold on

set(gca,'XTick',x,'XTickLabelRotation',-25)
xplot=[x(1:N);x(2:N+1)]'

```

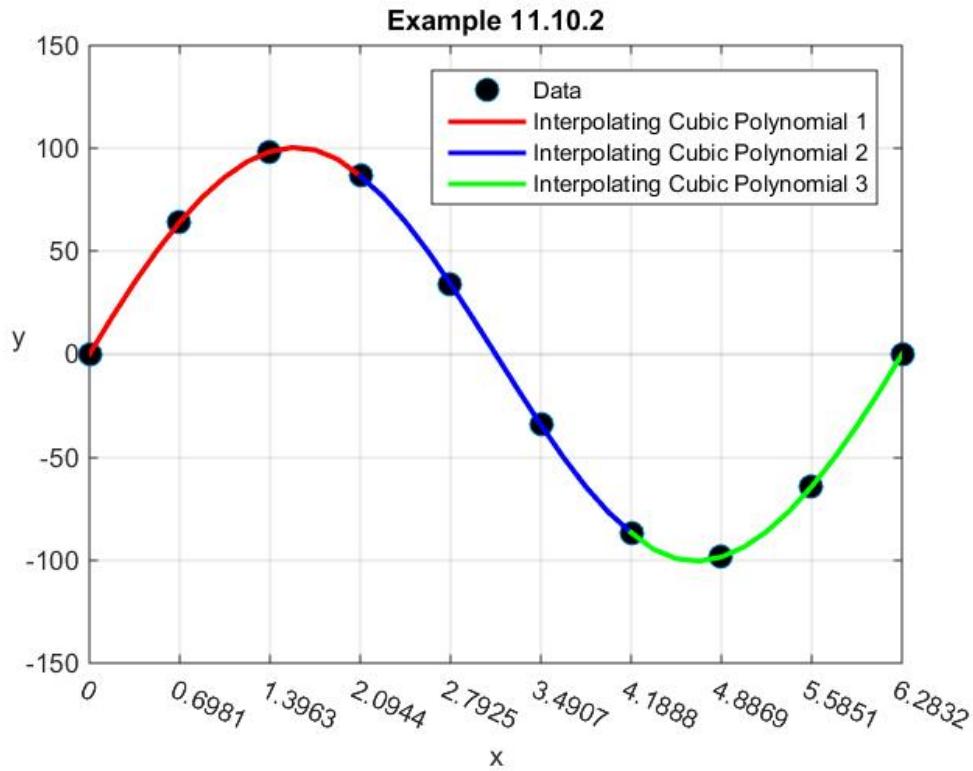
```

yplot=[y(1:N);y(2:N+1)]'
plot(xplot,yplot,'r','LineWidth',2)
legend('Data','Interpolating Linear Polynomials')
title('Example 11.10.1')

```

The above figure illustrates the feature that the various linear segments agree at the common points but their slopes do not.

Example 11.10.2: If we repeat Example 11.10.1 except partition the interval $[0, 6.2832]$ into the three segments (11.10.3), equations (11.10.6) yield the figure



The MATLAB script that will generate this figure is

```

clc
clear
%Construct the data table
N=9;
x=linspace(0,2*pi,N+1);
y=100*sin(x);
%Assign the 4N+1=37 equally spaced points x where the
%resulting polynomial will be evaluated
xvalues=linspace(0,2*pi,4*N+1);

%Partition the xvalues into the three segments

```

```

%Select the values in [x(1),x(4)]
xvalues1=xvalues.*(x(1)<=xvalues&xvalues<=x(4))
% xvalues1=[x(1),xvalues1(xvalues1~=0),x(4)]
xvalues1=[x(1),xvalues1(xvalues1~=0)]
%Select the values in [x(4),x(7)]
xvalues2=xvalues.*((x(4)<=xvalues&xvalues<=x(7)))
% xvalues2=[x(4),xvalues2(xvalues2~=0),x(7)]
xvalues2=[xvalues2(xvalues2~=0)]
%Select the values in [x(7),x(10)]
xvalues3=xvalues.*((x(7)<=xvalues&xvalues<=x(10)))
% xvalues3=[x(7),xvalues3(xvalues3~=0),x(10)]
xvalues3=[xvalues3(xvalues3~=0)]

%Use function file lagrange.m to calculate yvalues
%for each segment
[c_m,yvalues1]=lagrange(x(1:4),y(1:4),xvalues1)
[c_m,yvalues2]=lagrange(x(4:7),y(4:7),xvalues2)
[c_m,yvalues3]=lagrange(x(7:10),y(7:10),xvalues3)
%Plot the data points
plot(x,y,'o','MarkerFaceColor','k','MarkerSize',9)
xlabel('x')
ylabel('y','Rotation',0)
axis([0,2*pi,-150,150])
grid on
hold on
%Use the values yvalues1, yvalues2 and yvalues3 and
%plot the polynomial at the 37 points
%in the interval (0,2*pi)
plot(xvalues1,yvalues1,'r','LineWidth',2)
plot(xvalues2,yvalues2,'b','LineWidth',2)
plot(xvalues3,yvalues3,'g','LineWidth',2)

set(gca,'XTick',x,'XTickLabelRotation',-25)
legend('Data','Interpolating Cubic Polynomial 1',...
    'Interpolating Cubic Polynomial 2',...
    'Interpolating Cubic Polynomial 3')
title('Example 11.10.2')

```

Examples 11.10.1 and 11.10.2 illustrate the calculation of piecewise Lagrange polynomial interpolation. While somewhat tedious to setup, with MATLAB it is a straight forward calculation process. One simply has to create a table with $N + 1$ data pairs and identify the polynomials of degree M that have the property that $\frac{N}{M}$ is a positive integer.

It is customary when performing piecewise Lagrange polynomial interpolations to express the result in the form

$$f(x) = \sum_{j=1}^{N+1} y_j \varphi_j(x) \quad (11.10.7)$$

where the $N + 1$ functions $\varphi_1(x), \varphi_2(x), \dots, \varphi_{N+1}(x)$ are defined on the interval $[x_1, x_{N+1}]$ and have the properties that

$$\varphi_j(x_k) = \delta_{jk} \quad \text{for } j, k = 1, 2, \dots, N + 1 \quad (11.10.8)$$

Their values at other points on the interval are determined by the type of piecewise polynomial interpolation being formed. This assertion is best explained by specific examples. If the piecewise polynomial interpolation is the one represented by (11.10.5), the task is to write (11.10.5) in the form (11.10.7) and identify the functions $\varphi_1(x), \varphi_2(x), \dots, \varphi_{N+1}(x)$ that obey (11.10.8). It follows from (11.10.5) that it can be written

$$\begin{aligned}
f(x) = & y_1 \begin{cases} \frac{x - x_2}{x_1 - x_2} \text{ for } x \in [x_1, x_2] \\ 0 \text{ for } x \in [x_2, x_{10}] \end{cases} + y_2 \begin{cases} \frac{x - x_1}{x_2 - x_1} \text{ for } x \in [x_1, x_2] \\ \frac{x - x_3}{x_2 - x_3} \text{ for } x \in [x_2, x_3] \\ 0 \text{ for } x \in [x_3, x_{10}] \end{cases} \\
& + y_3 \begin{cases} 0 \text{ for } x \in [x_1, x_2] \\ \frac{x - x_2}{x_3 - x_2} \text{ for } x \in [x_2, x_3] \\ \frac{x - x_4}{x_3 - x_4} \text{ for } x \in [x_3, x_4] \\ 0 \text{ for } x \in [x_4, x_{10}] \end{cases} + y_4 \begin{cases} 0 \text{ for } x \in [x_1, x_3] \\ \frac{x - x_3}{x_4 - x_3} \text{ for } x \in [x_3, x_4] \\ \frac{x - x_5}{x_4 - x_5} \text{ for } x \in [x_4, x_5] \\ 0 \text{ for } x \in [x_5, x_{10}] \end{cases} \\
& + y_5 \begin{cases} 0 \text{ for } x \in [x_1, x_4] \\ \frac{x - x_4}{x_5 - x_4} \text{ for } x \in [x_4, x_5] \\ \frac{x - x_6}{x_5 - x_6} \text{ for } x \in [x_5, x_6] \\ 0 \text{ for } x \in [x_6, x_{10}] \end{cases} + y_6 \begin{cases} 0 \text{ for } x \in [x_1, x_5] \\ \frac{x - x_5}{x_6 - x_5} \text{ for } x \in [x_5, x_6] \\ \frac{x - x_7}{x_6 - x_7} \text{ for } x \in [x_6, x_7] \\ 0 \text{ for } x \in [x_7, x_{10}] \end{cases} \\
y_7 \begin{cases} 0 \text{ for } x \in [x_1, x_6] \\ \frac{x - x_6}{x_7 - x_6} \text{ for } x \in [x_6, x_7] \\ \frac{x - x_8}{x_7 - x_8} \text{ for } x \in [x_7, x_8] \\ 0 \text{ for } x \in [x_8, x_{10}] \end{cases} & + y_8 \begin{cases} 0 \text{ for } x \in [x_1, x_7] \\ \frac{x - x_7}{x_8 - x_7} \text{ for } x \in [x_7, x_8] \\ \frac{x - x_9}{x_8 - x_9} y_8 \text{ for } x \in [x_8, x_9] \\ 0 \text{ for } x \in [x_9, x_{10}] \end{cases} \\
& + y_9 \begin{cases} 0 \text{ for } x \in [x_1, x_8] \\ \frac{x - x_8}{x_9 - x_8} \text{ for } x \in [x_8, x_9] \\ \frac{x - x_{10}}{x_9 - x_{10}} \text{ for } x \in [x_9, x_{10}] \end{cases} + y_{10} \begin{cases} 0 \text{ for } x \in [x_1, x_9] \\ \frac{x - x_9}{x_{10} - x_9} \text{ for } x \in [x_9, x_{10}] \end{cases} \tag{11.10.9}
\end{aligned}$$

The form of (11.10.9) along with (11.10.7) allows us to conclude that

$$\varphi_1(x) = \begin{cases} \frac{x - x_2}{x_1 - x_2} & \text{for } x \in [x_1, x_2] \\ 0 & \text{for } x \in [x_2, x_{N+1}] \end{cases}$$

$$\varphi_j(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{j-1}] \\ \frac{x - x_{j-1}}{x_j - x_{j-1}} & \text{for } x \in [x_{j-1}, x_j] \\ \frac{x - x_{j+1}}{x_j - x_{j+1}} & \text{for } x \in [x_j, x_{j+1}] \\ 0 & \text{for } x \in [x_{j+1}, x_{N+1}] \end{cases} \quad \text{for } j = 2, \dots, N \quad (11.10.10)$$

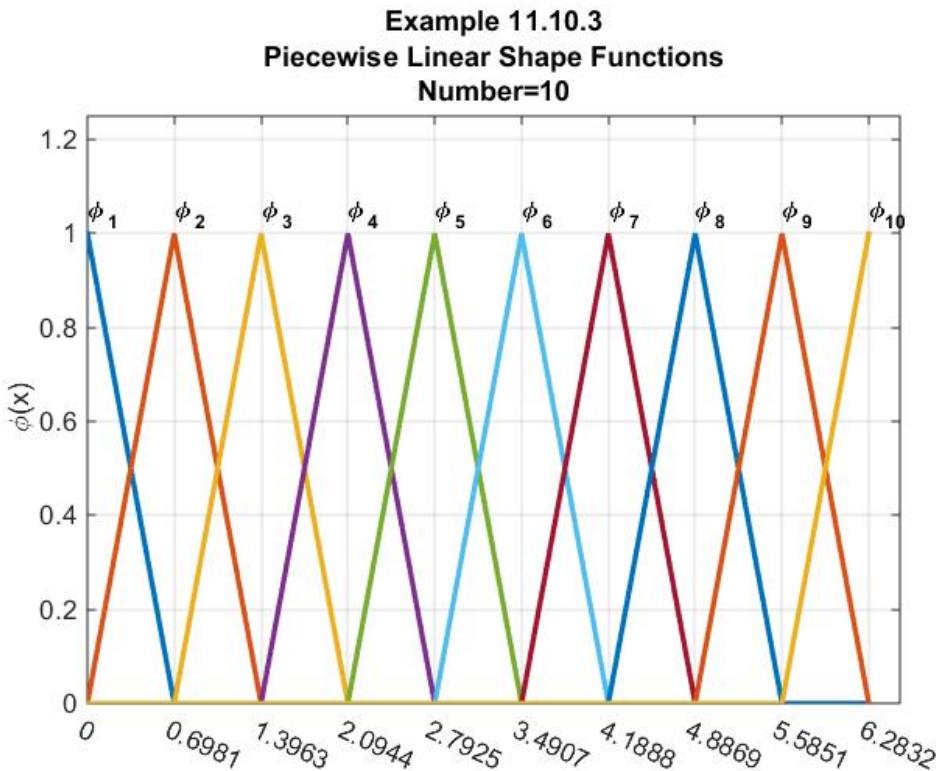
$$\varphi_{N+1}(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_N] \\ \frac{x - x_N}{x_{N+1} - x_N} & \text{for } x \in [x_N, x_{N+1}] \end{cases}$$

The piecewise linear functions defined by (11.10.10) are usually called *shape functions*.

Example 11.10.3: If we adopt the set of x values used in Example 11.10.1 and in several other examples, we can plot the functions in (11.10.10). As given in Example 11.10.1, $N = 9$ and the x values are

x	0	0.6981	1.3963	2.0944	2.7925	3.4907	4.1888	4.8869	5.5851	6.2832
-----	---	--------	--------	--------	--------	--------	--------	--------	--------	--------

The plot of the resulting ten shape functions turns out to be



This figure shows that there is one family of shape functions made up of the triangle shaped objects shown. They are simply translated down the axis as the figure shows. The shape functions at each end are simply half of the triangular shapes as shown. The MATLAB script that will produce the above figure is

```

clc
clear all
N=9
x=linspace(0,2*pi,N+1);

%Shape Function Calculations
%For j=1
v(1,:)=(x(2)-x)/(x(2)-x(1)).*(x(1)<=x&x<x(2))
for j=2:N
    v(j,:)=(x-x(j-1))/(x(j)-x(j-1)).*(x(j-1)<x&x<=x(j))+...
        (x(j+1)-x)/(x(j+1)-x(j)).*(x(j)<x&x<x(j+1));
end
%For j=N+1
v(N+1,:)=(x-x(N))/(x(N+1)-x(N)).*(x(N)<x&x<=x(N+1))

%Plot Shape Functions
plot(x,v,'LineWidth',2)
grid on
axis([0,x(N+1)+.25,0,1.25])

```

```

title({'Example 11.10.3';...
    'Piecewise Linear Shape Functions';...
    ['Number=' num2str(N+1)]})
ylabel('\phi(x)')
set(gca,'XTick',x,'XTickLabelRotation',-25)

%Label shape functions
xposition=x
yposition=1.04*[ones(1,N+1)]
labels=cellstr([num2str([1:N+1])])
labels=strcat('{',labels,'}')
my_labels=strcat('\phi_',labels)
text(xposition,yposition, ...
    my_labels,'HorizontalAlignment','left',...
    'FontWeight','bold','FontSize',9)

```

Given the idea of a shape function, it is useful to rework Example 10.11.1 and display how the piecewise interpolation represented in the form (11.10.7) builds the solution. The figure that results from superimposing the shape functions multiplied by the appropriate y value turns out to be,¹⁶

¹⁶ Because of the length of the text in the legend, it is convenient to make the entry, Interpolating Segments, print in two lines. An online search will display a few ways to cause MATLAB to print in this way. The script that is sufficient is

```

legend([h0,h1(1),h2(2:N)],'Data',sprintf('Interpolating \nSegments'),...
    '\phi_2','\phi_3','\phi_4','\phi_5','\phi_6','\phi_7','\phi_8','\phi_9',...
    'Location','SouthEastOutside')

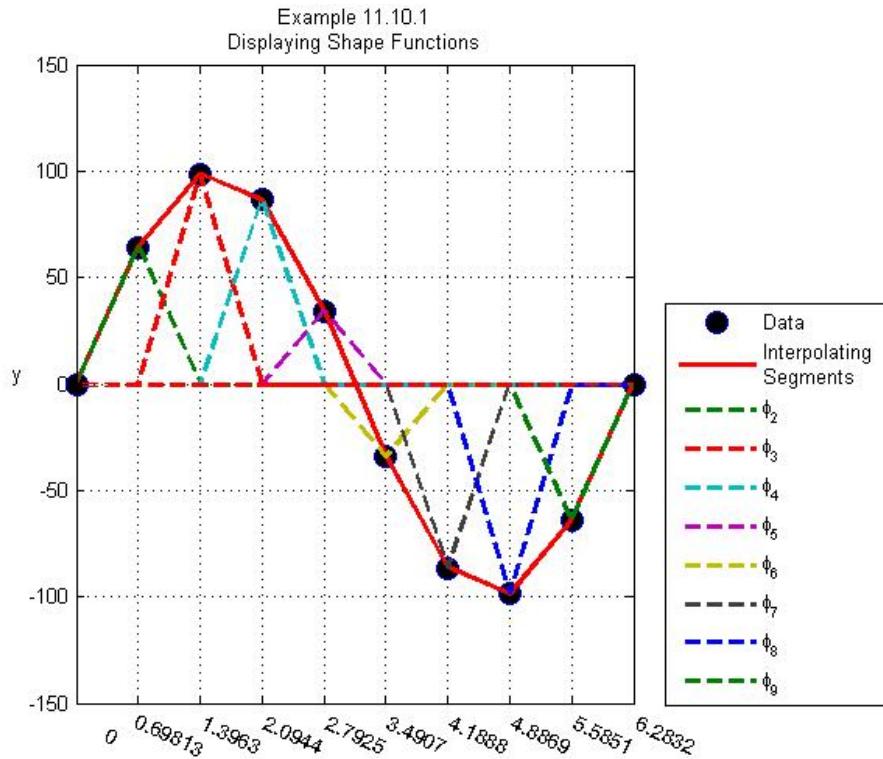
```

Another way to achieve the same result is the script

```

legend([h0,h1(1),h2(2:N)],'Data', ['Interpolatinga' char(10) 'Segments'],...
    '\phi_2','\phi_3','\phi_4','\phi_5','\phi_6','\phi_7','\phi_8','\phi_9',...
    'Location','SouthEastOutside')

```



This figure shows eight of the ten shape functions. The two remaining shape functions, φ_1 and φ_{10} , do not show because the corresponding y values are zero. The continuity of the piecewise linear segments and the discontinuity in their slopes at the connecting points x_2, \dots, x_N are evident from the figure.

For the case of piecewise interpolation for cubic interpolation, we can also write (11.10.6) in the form (11.10.7). The result is

$$\begin{aligned}
f(x) = & y_1 \begin{cases} \frac{(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & \text{for } x \in [x_1, x_4] \\ 0 & \text{for } x \in [x_4, x_{10}] \end{cases} \\
& + y_2 \begin{cases} \frac{(x-x_1)(x-x_3)(x-x_4)}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} & \text{for } x \in [x_1, x_4] \\ 0 & \text{for } x \in [x_4, x_{10}] \end{cases} \\
& + y_3 \begin{cases} \frac{(x-x_1)(x-x_2)(x-x_4)}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} & \text{for } x \in [x_1, x_4] \\ 0 & \text{for } x \in [x_4, x_{10}] \end{cases} \\
& + y_4 \begin{cases} \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} & \text{for } x \in [x_1, x_4] \\ \frac{(x-x_5)(x-x_6)(x-x_7)}{(x_4-x_5)(x_4-x_6)(x_4-x_7)} & \text{for } x \in [x_4, x_7] \\ 0 & \text{for } x \in [x_7, x_{10}] \end{cases} \\
& + y_5 \begin{cases} 0 & \text{for } x \in [x_1, x_4] \\ \frac{(x-x_4)(x-x_6)(x-x_7)}{(x_5-x_4)(x_5-x_6)(x_5-x_7)} & \text{for } x \in [x_4, x_7] \\ 0 & \text{for } x \in [x_7, x_{10}] \end{cases} \\
& + y_6 \begin{cases} 0 & \text{for } x \in [x_1, x_4] \\ \frac{(x-x_4)(x-x_5)(x-x_7)}{(x_6-x_4)(x_6-x_5)(x_6-x_7)} & \text{for } x \in [x_4, x_7] \\ 0 & \text{for } x \in [x_7, x_{10}] \end{cases} \\
& + y_7 \begin{cases} 0 & \text{for } x \in [x_1, x_4] \\ \frac{(x-x_4)(x-x_5)(x-x_6)}{(x_7-x_4)(x_7-x_5)(x_7-x_6)} & \text{for } x \in [x_4, x_7] \\ \frac{(x-x_8)(x-x_9)(x-x_{10})}{(x_7-x_8)(x_7-x_9)(x_7-x_{10})} & \text{for } x \in [x_7, x_{10}] \end{cases} \\
& + y_8 \begin{cases} 0 & \text{for } x \in [x_1, x_7] \\ \frac{(x-x_7)(x-x_9)(x-x_{10})}{(x_8-x_7)(x_8-x_9)(x_8-x_{10})} & \text{for } x \in [x_7, x_{10}] \end{cases} \\
& + y_9 \begin{cases} 0 & \text{for } x \in [x_1, x_7] \\ \frac{(x-x_7)(x-x_8)(x-x_{10})}{(x_9-x_7)(x_9-x_8)(x_9-x_{10})} & \text{for } x \in [x_7, x_{10}] \end{cases} \\
& + y_{10} \begin{cases} 0 & \text{for } x \in [x_1, x_7] \\ \frac{(x-x_7)(x-x_8)(x-x_9)}{(x_{10}-x_7)(x_{10}-x_8)(x_{10}-x_9)} & \text{for } x \in [x_7, x_{10}] \end{cases}
\end{aligned} \tag{11.10.11}$$

Therefore, for a piecewise cubic interpolation,

$$\varphi_1(x) = \begin{cases} \frac{(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} & \text{for } x \in [x_1, x_4] \\ 0 & \text{for } x \in [x_4, x_{N+1}] \end{cases} \quad (11.10.12)$$

$$\varphi_j(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{j-1}] \\ \frac{(x-x_{j-1})(x-x_{j+1})(x-x_{j+2})}{(x_j-x_{j-1})(x_j-x_{j+1})(x_j-x_{j+2})} & \text{for } x \in [x_{j-1}, x_{j+2}] \\ 0 & \text{for } x \in [x_{j+2}, x_{N+1}] \end{cases} \quad \text{for } j=2,5,8,\dots,N-1 \quad (11.10.13)$$

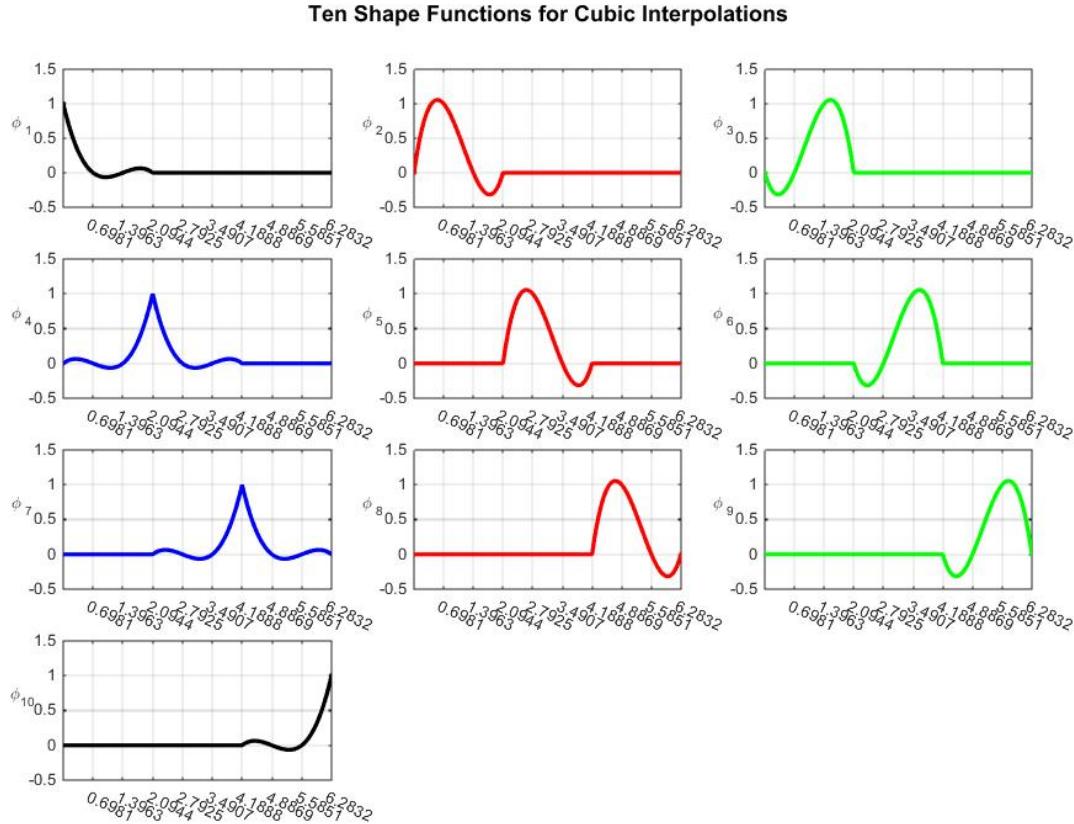
$$\varphi_j(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{j-2}] \\ \frac{(x-x_{j-2})(x-x_{j-1})(x-x_{j+1})}{(x_j-x_{j-2})(x_j-x_{j-1})(x_j-x_{j+1})} & \text{for } x \in [x_{j-2}, x_{j+1}] \\ 0 & \text{for } x \in [x_{j+1}, x_{N+1}] \end{cases} \quad \text{for } j=3,6,9,\dots,N \quad (11.10.14)$$

$$\varphi_j(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{j-3}] \\ \frac{(x-x_{j-3})(x-x_{j-2})(x-x_{j-1})}{(x_j-x_{j-3})(x_j-x_{j-2})(x_j-x_{j-1})} & \text{for } x \in [x_{j-3}, x_j] \\ \frac{(x-x_{j+1})(x-x_{j+2})(x-x_{j+3})}{(x_j-x_{j+1})(x_j-x_{j+2})(x_j-x_{j+3})} & \text{for } x \in [x_j, x_{j+3}] \\ 0 & \text{for } x \in [x_{j+3}, x_{N+1}] \end{cases} \quad \text{for } j=4,7,10,\dots,N-2 \quad (11.10.15)$$

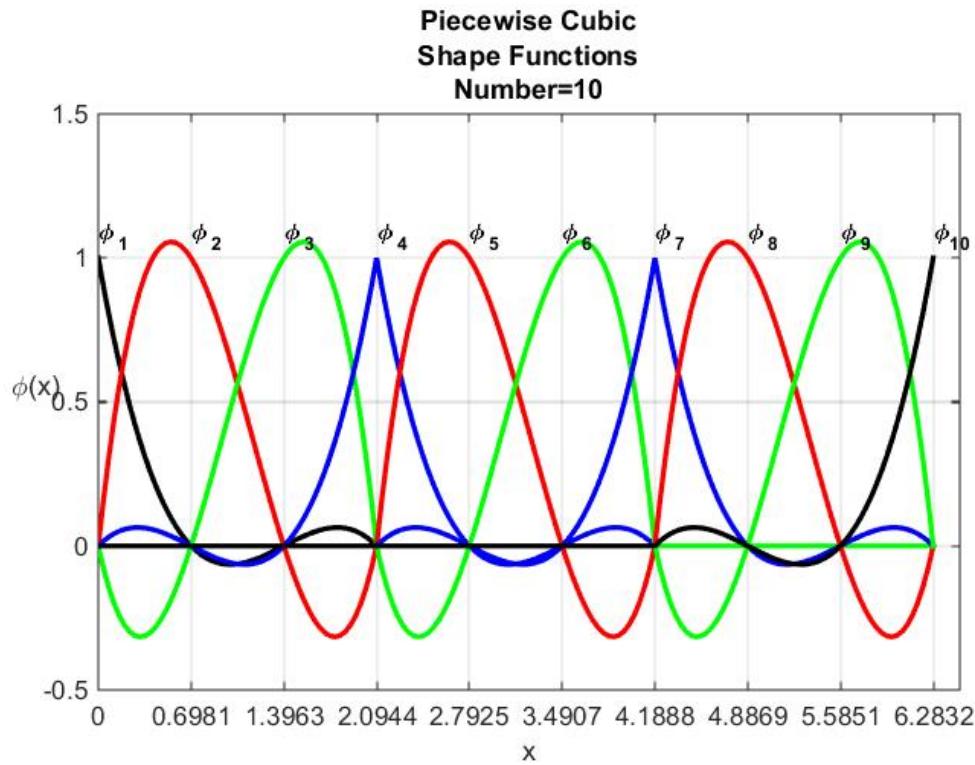
and

$$\varphi_{N+1}(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{N-2}] \\ \frac{(x-x_{N-2})(x-x_{N-1})(x-x_N)}{(x_{N+1}-x_{N-2})(x_{N+1}-x_{N-1})(x_{N+1}-x_N)} & \text{for } x \in [x_{N-2}, x_{N+1}] \end{cases} \quad (11.10.16)$$

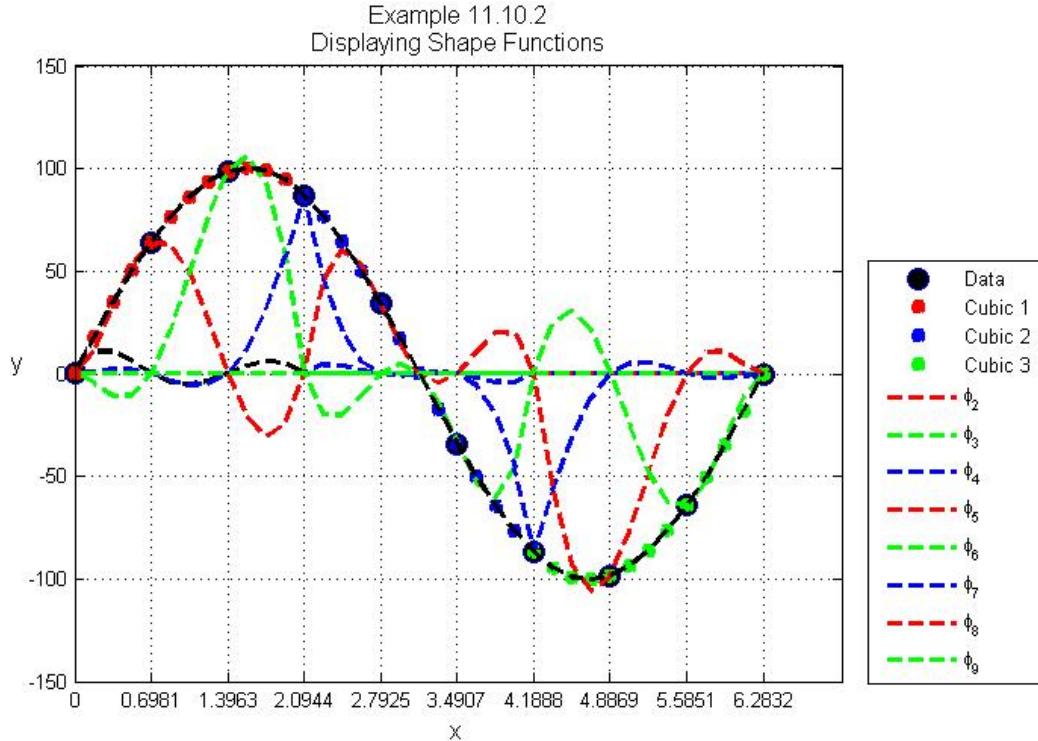
Example 11.10.4: If we continue with the case $N = 9$ and x having the values shown in Example 11.10.3, the plots of the nine shape functions take the forms



These ten figures display three families of functions, shown in blue, red and green, respectively supplemented by functions φ_1 and φ_{10} . The graphs in black correspond to the ends of the interval and are plots of (11.10.12) and (11.10.16), respectively. Note that they are simply translations of each other along the x axis. The graphs in red are plots of (11.10.13). The graphs in blue are plots of (11.10.14) and are also translations of each other along the x axis. Finally, the graphs in green are plots of (11.10.15). Of course they, like the other families of shape functions, are translations of each other along the x axis. Note that the graph of φ_1 is essentially the leading one half of the blue figure. Likewise the graph of φ_{10} is the trailing one half of the blue figure. If the ten graphs are superimposed on a common axis, the result is the complicated looking figure



As we did with Example 11.10.1, it is useful to rework Example 10.11.2 and display how the piecewise interpolation represented in the form (11.10.7) builds the solution. The figure that results from superimposing the shape functions multiplied by the appropriate y value turns out to be



As with Example 11.10.1, this figure shows eight of the ten shape functions. The two remaining shape functions, ϕ_1 and ϕ_{10} , do not show because the corresponding y values are zero. While complicated, the above figure does show how the formula (11.10.7) and the shape functions build the piecewise cubic interpolating polynomial. The scale of the figure does not reveal the behavior of the three cubics at the connecting points x_4 and x_7 . However, because of the relationship (11.10.7), the figure above, showing the ten shape functions, shows that the first, second and third derivatives have discontinuities at x_4 and x_7 .

Exercises

11.10.1: If the data set is again $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$, where $x_1 < x_2 < \dots < x_{N+1}$, a piecewise quadratic interpolation can be performed in those cases where $\frac{N}{2}$ is a positive integer. The value of this positive integer represents the number of segments of the piecewise polynomial. Therefore, we have partitioned the interval $[x_1, x_{N+1}]$ into the $\frac{N}{2}$ segments

$$[x_1, x_{N+1}] = [x_1, x_3] \cup [x_3, x_5] \cup [x_5, x_7] \dots \cup [x_{N-1}, x_{N+1}] \quad (11.10.17)$$

Show that the shape functions in this case are given by

$$\begin{aligned}
\varphi_1(x) &= \begin{cases} \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} & \text{for } x \in [x_1, x_3] \\ 0 & \text{for } x \in [x_3, x_{N+1}] \end{cases} \\
\varphi_j(x) &= \begin{cases} 0 & \text{for } x \in [x_1, x_{j-1}] \\ \frac{(x-x_{j-1})(x-x_{j+1})}{(x_j-x_{j-1})(x_j-x_{j+1})} & \text{for } x \in [x_{j-1}, x_{j+1}] \\ 0 & \text{for } x \in [x_{j+1}, x_{N+1}] \end{cases} \quad \text{for } j = 2, 4, 6, \dots, N \\
\varphi_j(x) &= \begin{cases} 0 & \text{for } x \in [x_1, x_{j-2}] \\ \frac{(x-x_{j-2})(x-x_{j-1})}{(x_j-x_{j-2})(x_j-x_{j-1})} & \text{for } x \in [x_{j-2}, x_j] \\ \frac{(x-x_{j+1})(x-x_{j+2})}{(x_j-x_{j+1})(x_j-x_{j+2})} & \text{for } x \in [x_j, x_{j+2}] \\ 0 & \text{for } x \in [x_{j+2}, x_{N+1}] \end{cases} \quad \text{for } j = 3, 5, 7, \dots, N-1 \\
\varphi_{N+1}(x) &= \begin{cases} 0 & \text{for } x \in [x_1, x_{N-1}] \\ \frac{(x-x_{N-1})(x-x_N)}{(x_{N+1}-x_{N-1})(x_{N+1}-x_N)} & \text{for } x \in [x_{N-1}, x_{N+1}] \end{cases} \quad (11.10.18)
\end{aligned}$$

In the case where $N = 8$ and where we again make the choice

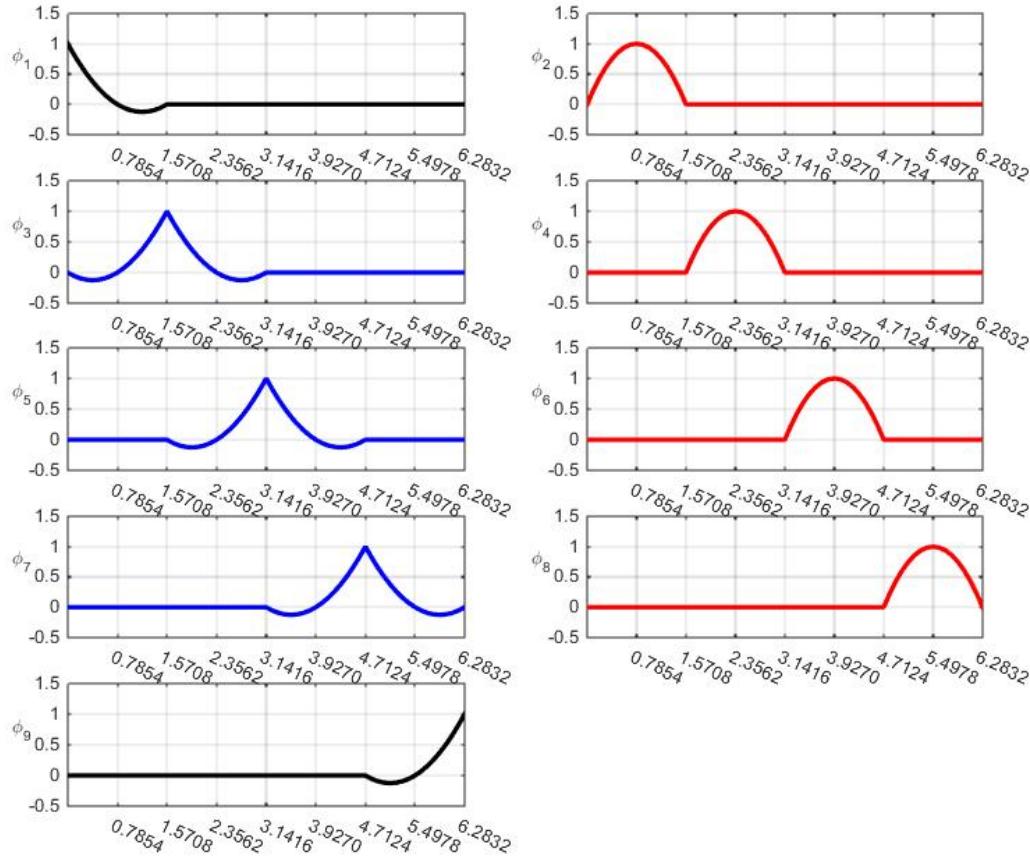
x	0	0.7854	1.5708	2.3562	3.1416	3.9270	4.7124	5.4978	6.2832
-----	---	--------	--------	--------	--------	--------	--------	--------	--------

the segments of the partition are

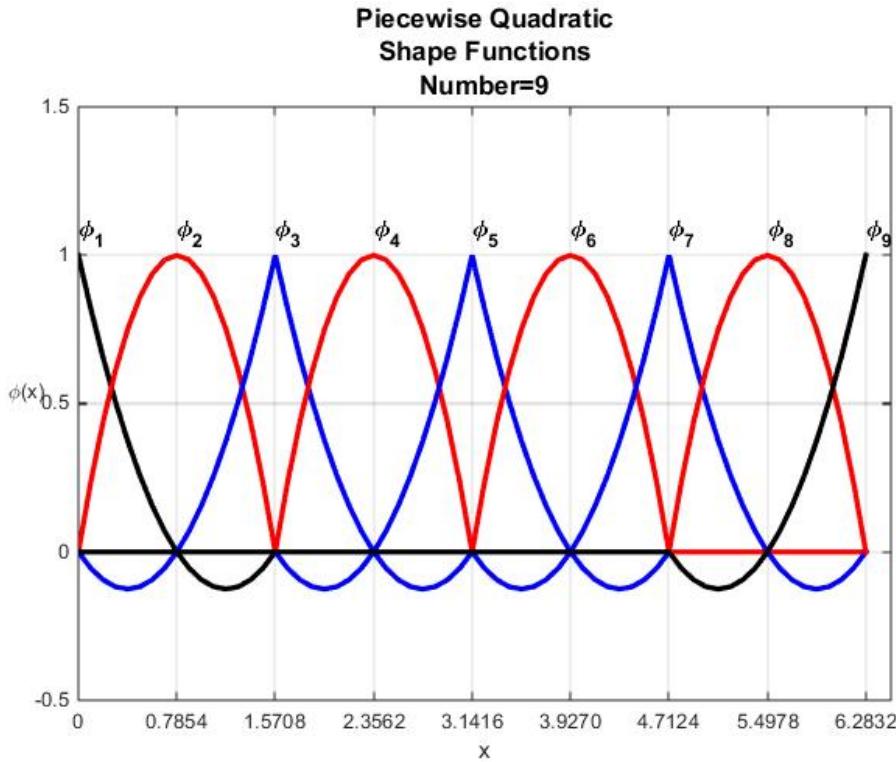
$$[0, 6.2832] = [0, 1.5708] \cup [1.5708, 3.1416] \cup [3.1416, 4.7124] \cup [4.7124, 6.2832] \quad (11.10.19)$$

and the nine shape functions in the quadratic case are

Nine Shape Functions for Quadratic Interpolations



This figure displays the two families of shape functions for this case. The first is shown in blue and the other in red. As with the cubic case, the end shape functions are, in the case of ϕ_1 , the leading one half of the blue shape function. Likewise, the shape function ϕ_9 is the trailing one half of the blue shape function. If the nine shape functions are superimposed on the same set of axes, the result is the figure



The figures for the shape functions reveal that piecewise interpolations based upon these functions will produce curves that are continuous at the connecting points x_3, x_5 and x_7 . At the same points the slopes and curvatures are discontinuous.

11.10.2: If the data set is again $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$, where

$x_1 < x_2 < \dots < x_{N+1}$, a piecewise quartic interpolation can be performed in those cases where $\frac{N}{4}$ is a positive integer. The value of this positive integer represents the number of segments of the piecewise polynomial. Therefore, we have partitioned the interval $[x_1, x_{N+1}]$ into the

$\frac{N}{3}$ segments

$$[x_1, x_{N+1}] = [x_1, x_5] \cup [x_5, x_9] \cup [x_9, x_{13}] \dots \cup [x_{N-3}, x_{N+1}] \quad (11.10.20)$$

Show that the shape functions in this case are given by

$$\varphi_1(x) = \frac{(x - x_2)(x - x_3)(x - x_4)(x - x_5)}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)(x_1 - x_5)} \quad \text{for } x \in [x_1, x_5] \quad (11.10.21)$$

$$\varphi_j(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{j-1}] \\ \frac{(x - x_{j-1})(x - x_{j+1})(x - x_{j+2})(x - x_{j+3})}{(x_j - x_{j-1})(x_j - x_{j+1})(x_j - x_{j+2})(x_j - x_{j+3})} & \text{for } x \in [x_{j-1}, x_{j+3}] \\ 0 & \text{for } x \in [x_{j+3}, x_{N+1}] \end{cases} \quad \text{for } j = 2, 6, 10, \dots, N-2 \quad (11.10.22)$$

$$\varphi_j(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{j-2}] \\ \frac{(x - x_{j-2})(x - x_{j-1})(x - x_{j+1})(x - x_{j+2})}{(x_j - x_{j-2})(x_j - x_{j-1})(x_j - x_{j+1})(x_j - x_{j+2})} & \text{for } x \in [x_{j-2}, x_{j+2}] \\ 0 & \text{for } x \in [x_{j+2}, x_{N+1}] \end{cases} \quad \text{for } j = 3, 7, 11, \dots, N-1 \quad (11.10.23)$$

$$\varphi_j(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{j-3}] \\ \frac{(x - x_{j-3})(x - x_{j-2})(x - x_{j-1})(x - x_{j+1})}{(x_j - x_{j-3})(x_j - x_{j-2})(x_j - x_{j-1})(x_j - x_{j+1})} & \text{for } x \in [x_{j-3}, x_{j+1}] \\ 0 & \text{for } x \in [x_{j+1}, x_{N+1}] \end{cases} \quad \text{for } j = 4, 8, 12, \dots, N \quad (11.10.24)$$

$$\varphi_j(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{j-4}] \\ \frac{(x - x_{j-4})(x - x_{j-3})(x - x_{j-2})(x - x_{j-1})}{(x_j - x_{j-4})(x_j - x_{j-3})(x_j - x_{j-2})(x_j - x_{j-1})} & \text{for } x \in [x_{j-4}, x_j] \\ \frac{(x - x_{j+1})(x - x_{j+2})(x - x_{j+3})(x - x_{j+4})}{(x_j - x_{j+1})(x_j - x_{j+2})(x_j - x_{j+3})(x_j - x_{j+4})} & \text{for } x \in [x_j, x_{j+4}] \\ 0 & \text{for } x \in [x_{j+4}, x_{N+1}] \end{cases} \quad \text{for } j = 5, 9, \dots, N-3 \quad (11.10.25)$$

$$\varphi_{N+1}(x) = \begin{cases} 0 & \text{for } x \in [x_1, x_{N-3}] \\ \frac{(x - x_{N-3})(x - x_{N-2})(x - x_{N-1})(x - x_N)}{(x_{N+1} - x_{N-3})(x_{N+1} - x_{N-2})(x_{N+1} - x_{N-1})(x_{N+1} - x_N)} & \text{for } x \in [x_{N-3}, x_{N+1}] \end{cases} \quad (11.10.26)$$

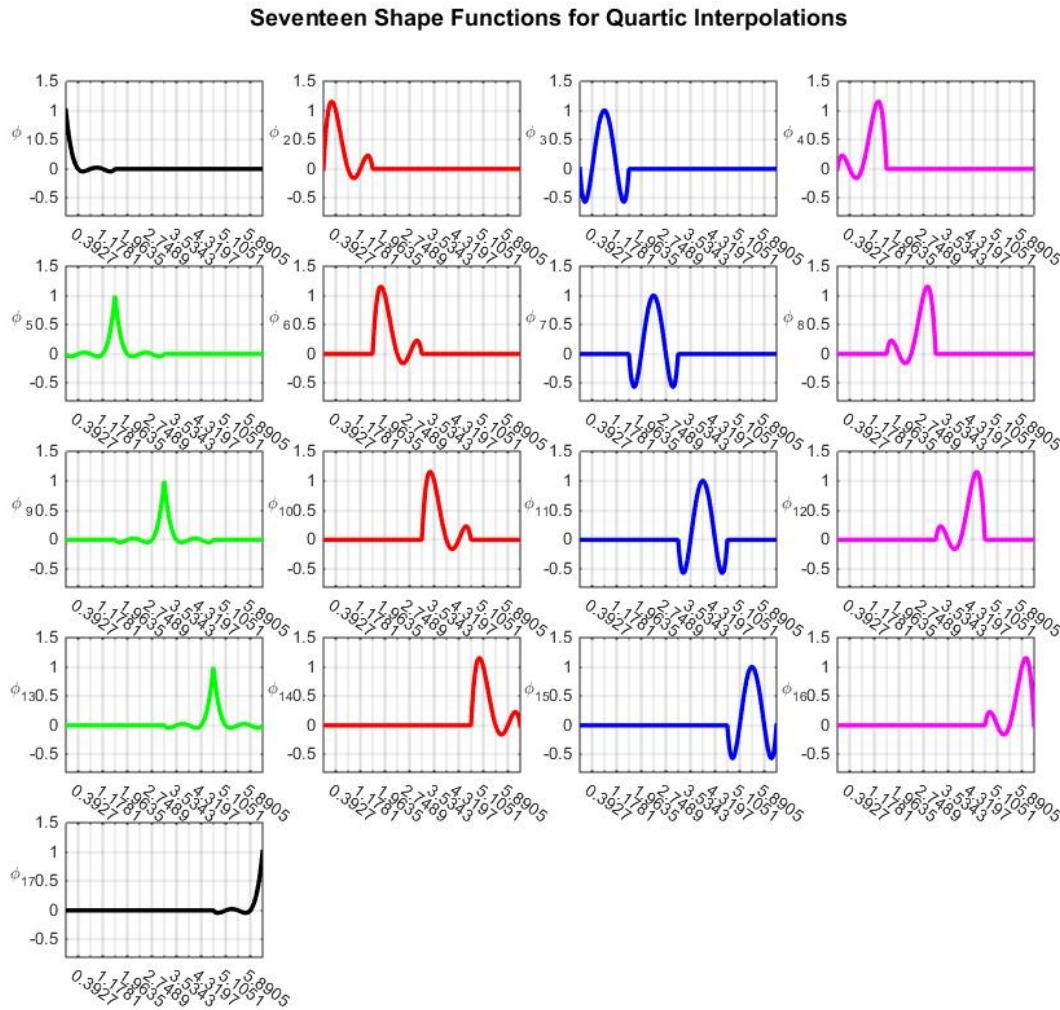
In the case where $N = 16$ and where we make the choice

x	0	0.3927	0.7854	1.1781	1.5708	1.9635	2.3562	2.7489	
	3.1416	3.5343	3.9270	4.3197	4.7124	5.1051	5.4978	5.8905	6.2832

the segments of the partition are

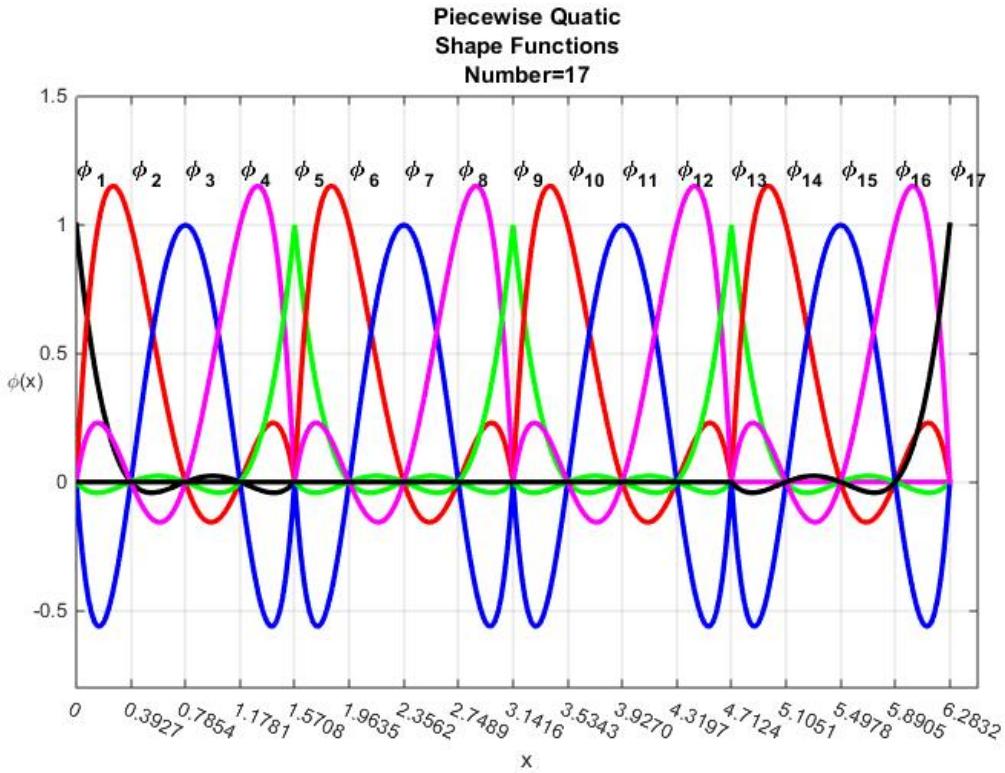
$$[0, 6.2832] = [0, 1.5708] \cup [1.5708, 3.1416] \cup [3.1416, 4.7124] \cup [4.7124, 6.2832] \quad (11.10.27)$$

and the 17 shape functions in the quartic case are



This figure displays the four families of shape functions for this case. The first is shown in green and the others in red, blue and magenta, respectively. As with our other examples, the end shape functions are, in the case of φ_1 , the leading one half of the blue shape function. Likewise, the

shape function ϕ_{17} is the trailing one half of the blue shape function. If the seventeen shape functions are superimposed on the same set of axes, the result is the figure



The figures for the shape functions reveal that piecewise interpolations based upon these functions will produce curves that are continuous at the connecting points x_5, x_9 and x_{13} . At the same points the first four derivatives are discontinuous.

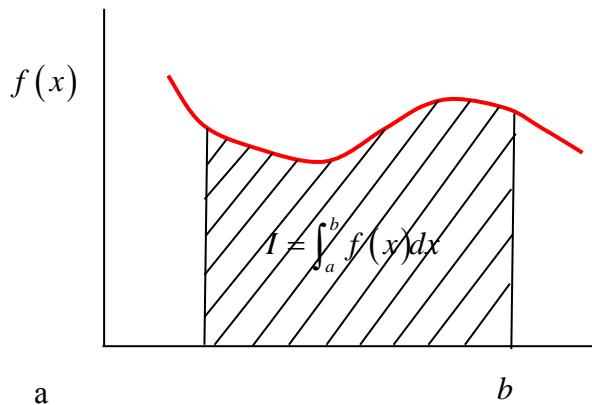
Section 11.11. Numerical Integration and Piecewise Interpolation

An important application of piecewise polynomial interpolation is *numerical integration*. In this section we shall utilize the results in Section 11.10 to discuss this topic. The fundamental ideas are few and are elementary. Given a real valued function $f : (a,b) \rightarrow \mathbb{R}$, the question is how to evaluate the integral

$$I = \int_a^b f(x) dx \quad (11.11.1)$$

Depending upon the function $f : (a,b) \rightarrow \mathbb{R}$, one simply integrates utilizing first principals as in introductory calculus courses. For some more complicated situations, the integral can be evaluated utilizing tables or, sometimes, the symbolic capabilities of MATLAB or something equivalent. However, it is frequently the case where the function to be integrated does not have an integral that can be expressed analytically. In such cases, the techniques of numerical integration become important.

It is helpful to recall that the integral (11.11.1) is simply the area of the curve resulting from the plot of $f : (a,b) \rightarrow \mathbb{R}$. The following figure suggests this relationship



The first formal step is to create a *partition* of the interval $[a,b]$ into N equal intervals. We achieve this partition by defining the *step size* h by the formula

$$h = \frac{b-a}{N} \quad (11.11.2)$$

and by dividing the interval into the equal segments by the formulas

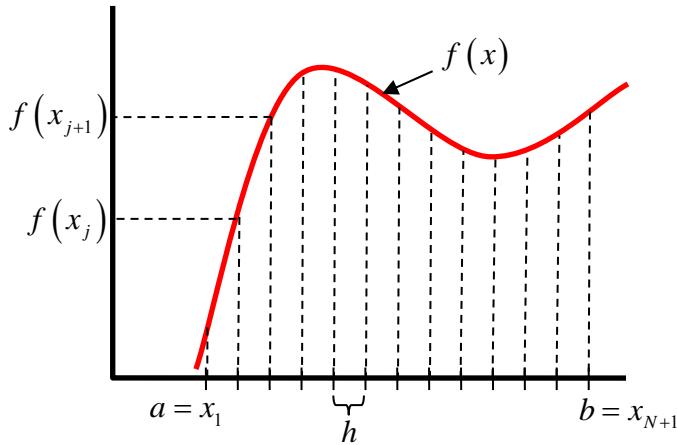
$$\begin{aligned}x_1 &= a \\x_2 &= h + x_1 \\x_3 &= h + x_2 \\\vdots \\x_{N+1} &= h + x_N = b\end{aligned}\tag{11.11.3}$$

The next step is to evaluate f at each point of the interval. The result is the data set

$\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$, where $a = x_1 < x_2 < \dots < x_{N+1} = b$ and

$$y_j = f(x_j) \quad \text{for } j = 1, 2, \dots, N+1\tag{11.11.4}$$

The following figure suggests how the partition is constructed.



As explained in Section 11.10, the data set

$\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N), (x_{N+1}, y_{N+1})\}$ allows one to create a piecewise polynomial interpolation that approximates the given function $f : (a, b) \rightarrow \mathbb{R}$. If we denote this approximating polynomial by f_N , it follows from (11.11.1) and (11.10.5) that

$$I = \int_a^b f(x) dx \cong \int_a^b f_N(x) dx = \sum_{j=1}^{N+1} y_j \int_a^b \varphi_j(x) dx\tag{11.11.5}$$

Equation (11.11.5) reduces our numerical integration to an integration of the shape functions associated with the particular piecewise polynomial used to approximate f . The piecewise polynomial nature of the shape functions means that the integrations are elementary.

The result of the integrations in (11.11.5) is a result of the general form

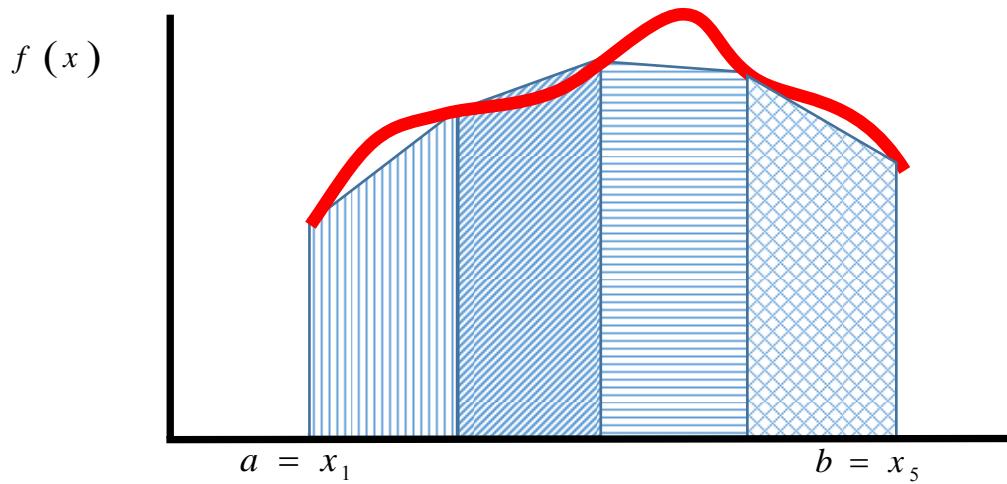
$$I = \int_a^b f(x) dx \equiv \sum_{j=1}^{N+1} \underbrace{w_j}_{\substack{\text{weighting} \\ \text{factor}}} \underbrace{f(x_j)}_{\substack{\text{weighted sum of } f(x_j)}} \quad (11.11.6)$$

The weighting factor w_j is the integral

$$w_j = \int_a^b \varphi_j(x) dx \quad (11.11.7)$$

and depends upon the particular piecewise polynomial being used to approximate the curve $f : (a, b) \rightarrow \mathcal{R}$. Our purpose is to identify various choices for the weighting factors and to understand the approximations associated with these choices.

As our first illustration consider the case where a piecewise linear interpolation is adopted. Thus, we are approximating the area under a given curve by the area of the several trapezoids that are associated with the linear segments. The following figure suggests the geometric arrangement.



In order to obtain an expression that approximates the integral, we need to evaluate the integrals (11.11.7) for the shape functions defined by equation (11.10.8). The results are simply the areas under the figures for the triangular shape functions. The figure for Example 11.10.3 illustrates these shapes. It follows from (11.10.8) that

$$w_1(x) = \frac{h}{2} \quad (11.11.8)$$

$$w_j(x) = 2h \quad \text{for } j = 2, \dots, N \quad (11.11.9)$$

and

$$w_{N+1}(x) = \frac{h}{2} \quad (11.11.10)$$

the areas under the figures for the triangular shape functions. Given (11.11.8), (11.11.9) and (11.11.10), (11.11.6) becomes

$$I \approx \sum_{j=1}^{N+1} w_j f(x_j) = \frac{h}{2} \left(f(x_1) + 2 \sum_{j=2}^N f(x_j) + f(x_{N+1}) \right) \quad (11.11.11)$$

A version of (11.11.11) that displays the contribution of the N linear segments is

$$I \approx h \left(\frac{1}{2} \sum_{j=1}^N (f(x_j) + f(x_{j+1})) \right) \quad (11.11.12)$$

Equation (11.11.11) or, equivalently, (11.11.12) is the *Composite Trapezoidal Rule*.

If, instead of piecewise linear interpolation, we adopt piecewise *quadratic* interpolation as discussed in Exercise 11.10.1, we must evaluate the integrals (11.11.7) for the shape functions given by Equation (11.10.15). In this case, in order to apply piecewise quadratic interpolation, we must partition the interval such that N is an even number. Evaluating these integrals is just the process of finding the areas under the curves shown in Exercise 11.10.1. The results are easily shown to be

$$w_1 = \frac{h}{3} \quad (11.11.13)$$

$$w_j = \frac{4h}{3} \quad \text{for } j = 2, 4, 6, \dots, N \quad (11.11.14)$$

$$w_j = \frac{2h}{3} \quad \text{for } j = 3, 5, 7, \dots, N-1 \quad (11.11.15)$$

and

$$w_{N+1} = \frac{h}{3} \quad (11.11.16)$$

Note that w_1 is one half of w_j for $j = 2, 4, \dots, N$ as the figures of Exercise 11.10.1 suggest. A similar observation follows for w_{N+1} .

Given (11.11.13) through (11.11.16), equation (11.11.6) reduces to

$$I \cong \frac{h}{3} \left(f(x_1) + 4 \sum_{j=2,4,\dots}^N f(x_j) + 2 \sum_{j=3,5,\dots}^{N-1} f(x_j) + f(x_{N+1}) \right) \quad (11.11.17)$$

A version of (11.11.17) that displays the contribution of the $\frac{N}{2}$ quadratic segments is

$$I \cong \frac{h}{3} \left(\sum_{j=1}^{N/2} \left(f(x_{2j-1}) + 4f(x_{2j}) + f(x_{2j+1}) \right) \right) \quad (11.11.18)$$

Equation (11.11.17) or, equivalently, (11.11.18) is known as the *Composite Simpson 1/3 Rule*.¹⁷

Next, if we utilize piecewise *cubic* interpolation as discussed in Example 11.10.4, we must evaluate the integrals (11.11.7) for the shape functions given by Equation (11.10.10) through (11.10.14). In this case, in order to apply piecewise cubic interpolation, we must partition the interval such that N is a positive integer multiple of 3. As with quadratic interpolation, evaluating these integrals is just the process of finding the areas under the curves shown in Example 11.10.4. The results are easily shown to be

$$w_1 = \frac{3h}{8} \quad (11.11.19)$$

$$w_j = \frac{9h}{8} \quad \text{for } j = 2, 5, 8, \dots, N-1 \quad (11.11.20)$$

$$w_j = \frac{9h}{8} \quad \text{for } j = 3, 6, 9, \dots, N \quad (11.11.21)$$

$$w_j = \frac{3h}{4} \quad \text{for } j = 4, 7, 10, \dots, N-2 \quad (11.11.22)$$

and

$$w_{N+1} = \frac{3h}{8} \quad (11.11.23)$$

¹⁷ Information about the British mathematician, Thomas Simpson, can be found at http://en.wikipedia.org/wiki/Thomas_Simpson.

Note that w_1 is one half of w_j for $j = 4, 7, 10, \dots, N-2$ as the figures of Exercise 11.10.4 suggest. A similar observation follows for w_{N+1} .

Given (11.11.19) through (11.11.23), equation (11.11.6) reduces to

$$I \equiv \frac{3h}{8} \left(f(x_1) + 3 \sum_{j=2,5,8,\dots}^{N-1} f(x_j) + 3 \sum_{j=3,6,9,\dots}^N f(x_j) + 2 \sum_{j=4,7,10,\dots}^{N-2} f(x_j) + f(x_{N+1}) \right) \quad (11.11.24)$$

A version of (11.11.24) that displays the contribution of the $\frac{N}{3}$ cubic segments is

$$I \equiv \frac{3h}{8} \left(\sum_{j=1}^{N/3} (f(x_{3j-2}) + 3f(x_{3j-1}) + 3f(x_{3j}) + f(x_{3j+1})) \right) \quad (11.11.25)$$

Equation (11.11.24) or its equivalent (11.11.25) is known as the *Composite Simpson 3/8 Rule*.

The three formulas (11.11.11), (11.11.17) and (11.11.24) are three examples of what are called *Newton-Cotes formulas*.¹⁸ They arise out of the same fundamental construction, i.e., approximating the area under a curve by a) straight line, b) a quadratic and c) a cubic. Higher order Newton-Cotes formulas can be obtained by the obvious choice of simply utilizing polynomials of order higher than three for the piecewise polynomial interpolations. For example, for the case of piecewise quartic polynomial interpolations the resulting composite integration formula is known as the *Composite Boole's Rule*.¹⁹ In this case the partition must be such that N is a positive integer multiple of 4. It can be shown that the resulting integration rule is

$$I \equiv \frac{2h}{45} \left(7f(x_1) + 32 \sum_{j=2,6,10,\dots}^{N-2} f(x_j) + 12 \sum_{j=3,7,11,\dots}^{N-1} f(x_j) + 32 \sum_{j=4,8,12,\dots}^N f(x_j) + 14 \sum_{j=5,9,13,\dots}^{N-3} f(x_j) + 7f(x_{N+1}) \right) \quad (11.11.26)$$

¹⁸ The English mathematician Roger Cotes was a contemporary of Sir Isaac Newton. Information about Cotes can be found at http://en.wikipedia.org/wiki/Roger_Cotes.

¹⁹ Information about the English mathematician George Boole can be found at http://en.wikipedia.org/wiki/George_Boole.

A version of (11.11.26) that displays the contribution of the $\frac{N}{4}$ quartic segments is ²⁰

$$I \equiv \frac{2h}{45} \left(\sum_{j=1}^{N/4} (7f(x_{4j-3}) + 32f(x_{4j-2}) + 12f(x_{4j-1}) + 32f(x_{4j}) + 7f(x_{4j+1})) \right) \quad (11.11.27)$$

The four results (11.11.11), (11.11.17), (11.11.24) and (11.11.26) are approximations to the integral (11.11.1). The truncation error associated with each of the four approximations can be shown to be as follows: ²¹

Interpolation	Formula	Truncation Error
Composite Trapezoidal	Equation (11.11.11)	$-\frac{(b-a)^3}{12N^3} \sum_{j=1}^N f^{(2)}(\xi_j)$
Composite Simpson 1/3	Equation (11.11.17)	$-\frac{(b-a)^5}{90N^5} \sum_{j=1}^{N/2} f^{(4)}(\xi_i)$
Composite Simpson 3/8	Equation (11.11.24)	$-\frac{3(b-a)^5}{80N^5} \sum_{j=1}^{N/3} f^{(4)}(\xi_i)$
Composite Boole	Equation (11.11.26)	$-\frac{8(b-a)^7}{945N^5} \sum_{j=1}^{N/4} f^{(6)}(\xi_i)$

The truncation errors shown in this table are derived from the remainder term in Taylor's Theorem, equation (8.1.5). The values ξ_j for the j^{th} trapezoid is a value in $[x_j, x_{j+1}]$. The values ξ_j for the j^{th} quadratic is a value in $[x_{2j-1}, x_{2j+1}]$. The values ξ_j for the j^{th} cubic is a

²⁰ The coefficients in the formulas (11.11.12), (11.11.18), (11.11.25) and (11.11.27) are called Newton Coates coefficients. For the trapezoidal rule they are $\left(\frac{h}{2}, \frac{h}{2}\right)$, for Simpson 1/3 rule they are $\left(\frac{h}{3}, \frac{4h}{3}, \frac{h}{3}\right)$, for Simpson 3/8

they are $\left(\frac{3h}{8}, \frac{9h}{8}, \frac{9h}{8}, \frac{3h}{8}\right)$ and for Boole's rule $\left(\frac{28h}{90}, \frac{128h}{90}, \frac{48h}{90}, \frac{128h}{90}, \frac{28h}{90}\right)$. The symmetry of these

coefficients these examples show is a general property of Newton Coates coefficients. The all positive number property is not a general property of Newton Coefficients. It turns out, for example, that the eighty order coefficients are not all positive. The result is that round off errors in addition to truncation errors become an issue in these high order cases. Details about this observation can be found, for example, at

http://people.oregonstate.edu/~peterseb/mth351/docs/newton_cotes.pdf.

²¹ These formulas are derived by applying Taylor's Theorem in the form (8.1.4) to each segment of the partition of $[a, b]$. The truncation error for each segment is derived, for example, at the site

<http://mathworld.wolfram.com/Newton-CotesFormulas.html>.

value in $[x_{3j-2}, x_{3j+1}]$. Finally, the values ξ_j for the j^{th} quartic is a value in $[x_{4j-3}, x_{4j+1}]$.

We shall illustrate the application of the above numerical integration approaches by a series of examples. It is possible to summarize certain features of these methods as follows:

1. Simpson's 1/3 Rule is usually the method of preference because it attains third-order accuracy (error term is proportional to forth derivative) with three points rather than the four points required for the 3/8 version.
2. The 3/8 rule is useful when the number of segments is odd.
 - a. For the 3/8 rule N must be divisible by 3.
 - b. For the 1/3 rule N must be divisible by 2.
3. Sometimes a mix of the 1/3 rule and the 3/8 rule is used in order to accommodate an odd number of segments without having to use the 3/8 rule entirely.

We shall illustrate the numerical integration schemes we have discussed by performing integration of a known function that has a known integral. In this way, we can examine the truncation errors in various cases. The function we shall select is the Bessel function of the first kind of order n , where n is an integer.²² This function is defined by the series

$$J_n(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!(m+n)!} \left(\frac{x}{2}\right)^{2m+n} \quad (11.11.28)$$

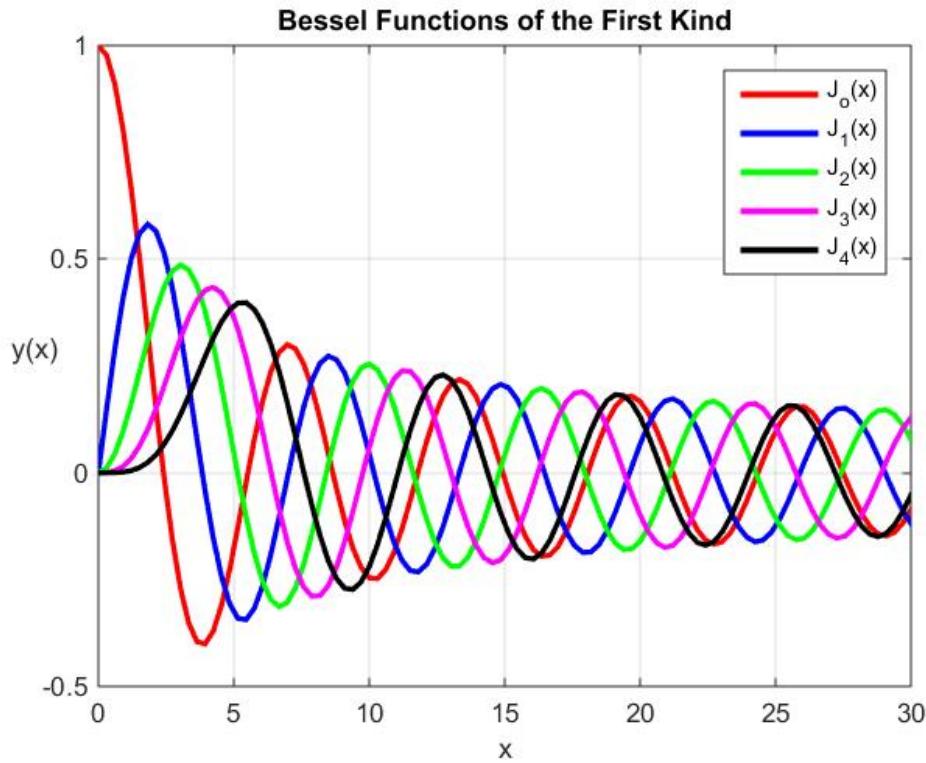
It is an identity among Bessel functions that²³

$$\frac{d(x^n J_n(x))}{dx} = x^n J_{n-1}(x) \quad (11.11.29)$$

Therefore, we can construct examples that involve numerical integrations of $x^n J_{n-1}(x)$ and compare the results to exact answers provided by (11.11.29). Of course, we shall make considerable use of MATLAB as we work these examples. The MATLAB syntax for $J_n(x)$ is **besselj(n,x)**. In order to understand some of the properties of the Bessel functions we shall utilize, it is useful to use MATLAB to generate the following plot for the Bessel functions $J_0(x), J_1(x), J_2(x), J_3(x)$ and $J_4(x)$.

²² Information about the German mathematician and astronomer Friedrich Bessel can be found at http://en.wikipedia.org/wiki/Friedrich_Bessel

²³ Our use of Bessel functions in this work is limited. Everything about Bessel functions this work will require is available from an internet search. For example, the identity (11.11.29) can be found at <http://mathworld.wolfram.com/BesselFunctionoftheFirstKind.html>. A classical reference that contains a great amount of information about Bessel functions Abramowitz, M. and I A. Stegun, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Table, National Bureau of Standards, 1964. This work is available in a later edition published by Dover.



The MATLAB script that will create this figure is

```

clc
clear all
x=linspace(0,30,100);
color='rbgmk'
for n=[1:5]
    plot(x,besselj(n-1,x),color(n),'LineWidth',2)
    grid on
    hold on
    axis([0,30,-.5,1])
end
legend('J_0(x)', 'J_1(x)', 'J_2(x)', 'J_3(x)', 'J_4(x)')
title('Bessel Functions of the First Kind')
xlabel('x')
ylabel('y(x)', 'Rotation', 0)

```

The above figure shows that $J_0(0)=1$ and the other Bessel functions on the graph are zero at $x=0$. In very rough terms, they behave like damped trigometric functions.

Example 11.11.1: In this example, we shall utilize the composite trapezoidal rule to evaluate the

integral

$$I = \int_{x=a}^b x J_0(x) dx \quad (11.11.30)$$

Because of the relationship (11.11.29), (11.11.30) has the exact integral

$$I = \int_{x=a}^b x J_0(x) dx = \int_{x=a}^b \frac{d(x J_1(x))}{dx} dx = b J_1(b) - a J_1(a) \quad (11.11.31)$$

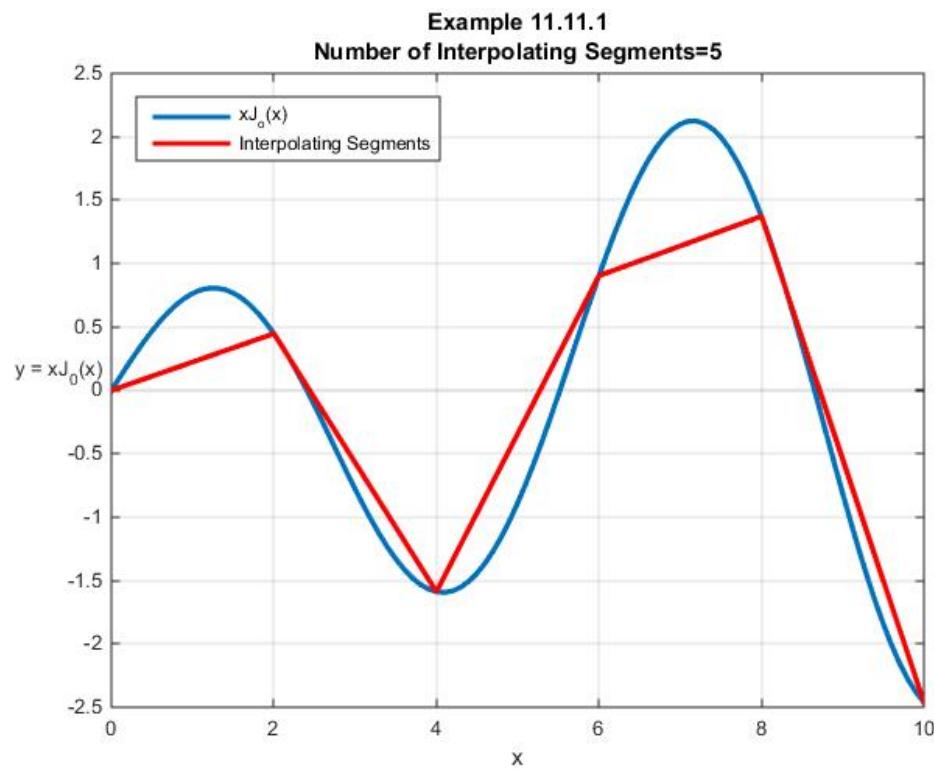
We shall use the function m-file **trapazoid.m** to calculate the approximation to (11.11.30). The MATLAB script for **trapezoid.m** is²⁴

```
function I=trapazoid(func,a,b,N)
%trapazoid: composite trapezoidal rule integration
% I=trapazoid(func,a,b,N)
%input:
% func=name of function to be integrated
%         func defined in vectorized form,
%         for example, by
%         func=inline('x.*exp(x)'), or
%         equivalently by func=@(x)(x.*exp(x))
% a, b=integration limits
% N=number of intervals (default = 100)
%%output:
% I=integral estimate
if nargin<3,error('at least 3 input arguments required'),end
if ~(b>a),error('upper bound must be greater than lower'),end
if nargin<4|isempty(N),N=100;end
h=(b-a)/N;
x=[a:h:b];
y=func(x);
I=(h/2)*(y(1)+2*sum(y(2:end-1))+y(end));
```

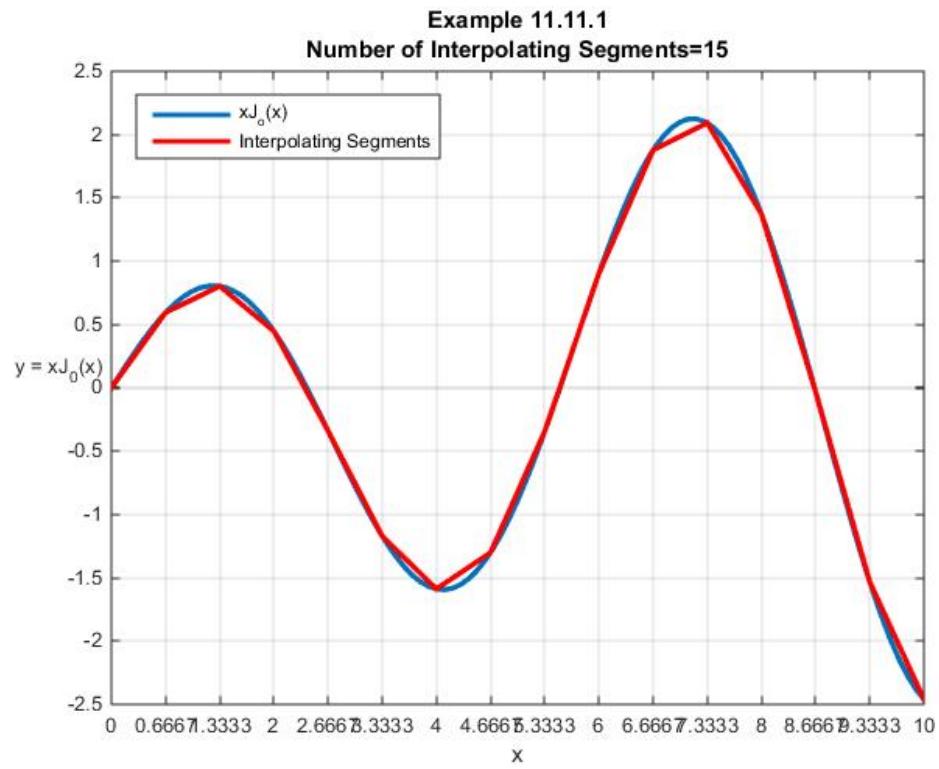
The structure of this m-file yields an error message if the relationship $b > a$ is not satisfied. Also, if the integer N is not prescribed, a default value of $N = 100$ will be inserted.

It is interesting to look at two plots of the function $x J_0(x)$ with different choices of the number of intervals N . These plots will be constructed with $a = 0$ and $b = 10$ for the cases $N = 5$ and $N = 15$. The results are

²⁴ Many versions of the equivalent of **trapezoid.m** can be found online.



and



It should be evident from the two figures that $N = 5$ will produce a poor approximation to the integral, and that $N = 15$ will produce a substantially better approximation. The exact result (11.11.31) can be used to show that

$$I = \int_{x=0}^{10} x J_0(x) dx = 10 J_1(10) = 0.4347 \quad (11.11.32)$$

Given the result (11.11.32), the truncation error, expressed as a percentage, is the difference in the exact value and the approximate calculated value normalized by the exact value. This measure of error was discussed in Section 8.2.

In order to display the dependence of the answer for the approximate integral on N , the following MATLAB script is used

```

clc
clear
a=0
b=10
%Calculate exact value
I=b*besselj(1,b)-a*besselj(1,a);
%Put a title line on the table
fprintf('\n\t\t\tComposite Trapezoidal Rule\n\n')
%Display the exact value
fprintf(['\t\tExact Integral=' num2str(I)])
%Label columns of table
fprintf('\n\n\t\tN\tStep Size=h\t\tApproximate
Value\tTruncation Error %% \n')

%Define function to integrate
f=@(x)x.*besselj(0,x);
%Number of segments.
for N=5:5:40;
%Calculate the step size
    h=(b-a)/N;
%Calculate the approximate integral
    Iapprox=trapazoid(f,a,b,N);
%Calculate the truncation error as a percentage
    epsilon(N)=(I-Iapprox)*100/I;
%Print the table
    fprintf('\t%5.0f \t%5.4f\t\t%5.4f
\t\t\t%5.2f\n',N,h,Iapprox,epsilon(N))
end

```

The above script makes considerable use of the MATLAB **fprintf** command. This rather complicated command is discussed in MATLAB **help**. If the above script is executed, the result is the table

Composite Trapezoidal Rule

Exact Integral=0.43473

N	Step Size=h	Approximate Value	Truncation Error
5	2.0000	-0.1868	142.98
10	1.0000	0.2912	33.02
15	0.6667	0.3718	14.47
20	0.5000	0.3995	8.10
25	0.4000	0.4122	5.17
30	0.3333	0.4191	3.59
35	0.2857	0.4233	2.64
40	0.2500	0.4260	2.02

It is evident from this example that a large number of terms are required to obtain an acceptable value for the integral.

Example 11.11.2: In this example, we shall utilize the composite Simpson 1/3 rule to evaluate the integral (11.11.30). The function m-file defined by the script

```

function I=simpson13(func,a,b,N)
%simpson13: composite Simpson 1/3 rule
% I=simp13_357(func,a,b,N)
%input:
% func=name of function to be integrated
%         func defined in vectorized form,
%         for example, by
%         func=inline('x.*exp(x)'), or
%         equivalently by func=@(x)(x.*exp(x))
% a, b=integration limits
% N=number of intervals (default=100).
%     N must be an even integer
%output:
% I=integral estimate
if nargin<3,error('at least 3 input arguments required'),end
if nargin<4|isempty(N),N=100;end
if ~(b>a),error('upper bound must be greater than lower'),end
if mod(N,2)~=0
    disp('N must be an even integer, it will be replaced by
N+1')
    N=N+1
end
x=linspace(a,b,N+1);

```

```

h=(b-a)/N;
y=func(x);
I=(h/3)*(y(1)+4*sum(y(2:2:end-1))+2*sum(y(3:2:end-
2))+y(end));

```

will be used to calculate the approximate value of the integral for various values of N . The structure of this m-file is essentially the same as the one for **trapezoid.m** utilized earlier. One important difference is the presence of the script

```

if mod(N,2)~=0
    disp('N must be an even integer, it will be replaced by
N+1')
    N=N+1
end

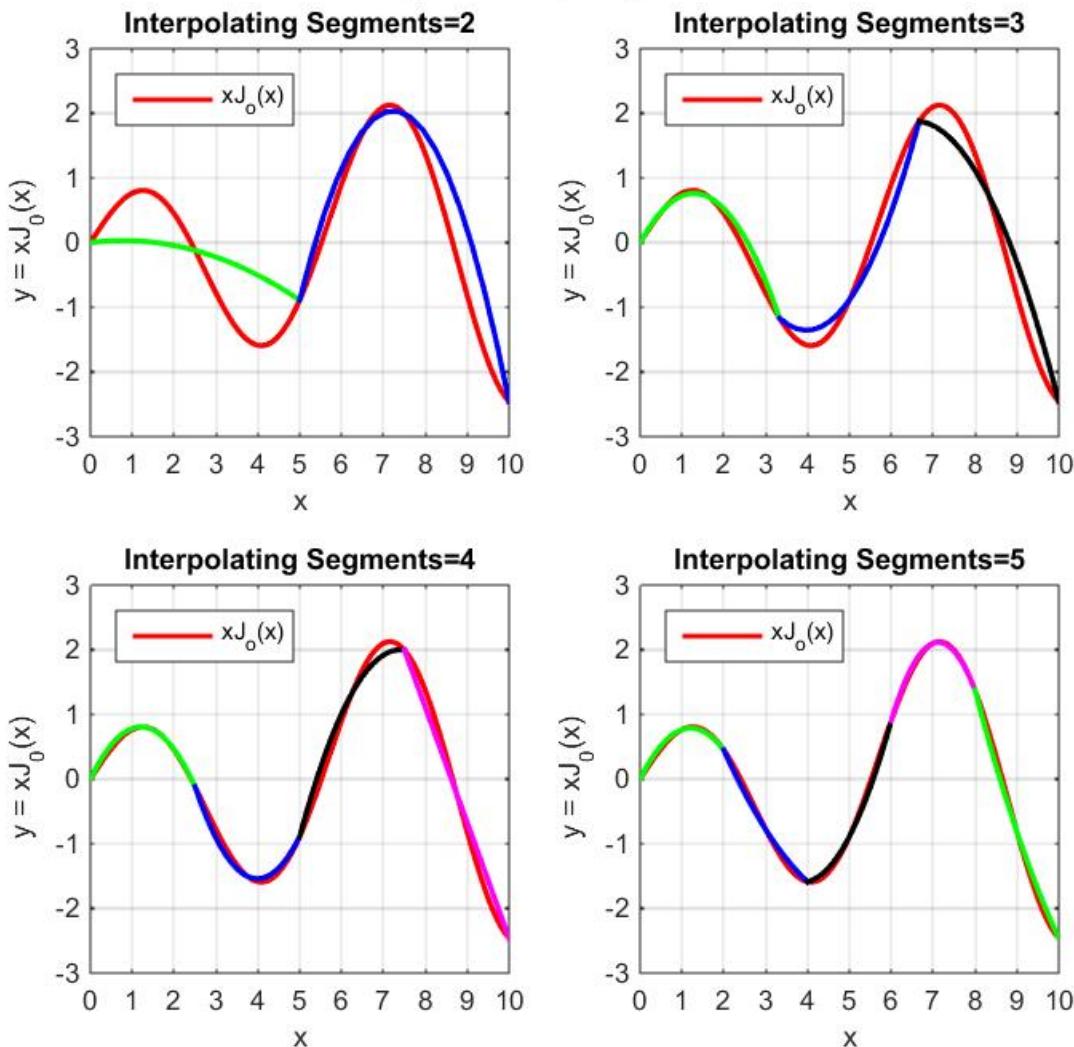
```

This script is included to reflect the requirement that N must be an even number for the Simpson 1/3 rule. The script **mod(N,2)** yields 0 if N is an even number and 1 if N is an odd number. In the odd case, N is replaced by the even number $N + 1$.²⁵

A feel for the accuracy of the Simpson 1/3 rule can be obtained by creating graphs of the function to be integrated, $xJ_0(x)$, and quadratic polynomials that form the underlying piecewise interpolation. In the cases $\frac{N}{2} = 2, \frac{N}{2} = 3, \frac{N}{2} = 4$ and $\frac{N}{2} = 5$ the following graphs suggest how close the area under the curve of $xJ_0(x)$ is approximated by the piecewise quadratic polynomial.
²⁶

²⁵ The MATLAB **mod** command is explained at <http://www.mathworks.com/help/matlab/ref/mod.html>. Essentially, as it is utilized here, it simply yields the remainder of the division **N/2**. If **N** is even the remainder is 0, if it is odd the remainder is 1.

²⁶The script sufficient to produce the figure for Example 11.11.2 is left as an exercise. MATLAB does not have a built in command to create a common title for subplots. If the online resources for MATLAB are examined, one finds there are several virtually equivalent ways to create a common title. For example, at the site <http://www.mathworks.com/matlabcentral/answers/100459-how-can-i-insert-a-title-over-a-group-of-subplots> one can download a function m-file **subtitle.m** that will create the common title.

Example 11.11.2, Simpson 1/3 Rule

These figures suggest that $\frac{N}{2} = 2$ in this case will provide a poor approximation. The case $\frac{N}{2} = 3$ will be substantially better. The case $\frac{N}{2} = 5$ should give good results. These informal observations are reinforced if we utilize MATLAB to generate a table like the one after Example 11.11.1. The resulting table is

Composite Simpson 1/3 Rule

Exact Integral=0.43473

%	Segments	Step Size=h	Approximate Value	Truncation Error
	2	2.5000	2.7259	-527.03
	3	1.6667	0.6041	-38.96
	4	1.2500	0.4772	-9.77
	5	1.0000	0.4505	-3.64
	6	0.8333	0.4420	-1.67
	7	0.7143	0.4385	-0.87
	8	0.6250	0.4369	-0.50
	9	0.5556	0.4361	-0.31
	10	0.5000	0.4356	-0.20
	11	0.4545	0.4353	-0.14
	12	0.4167	0.4351	-0.10

As one would expect, this table shows the improved accuracy of the Simpson 1/3 rule over the Trapazoid Rule.

If the problem should require the use of the slightly more accurate Simpson 3/8 rule, the function m-file **simpson38.m** can be used. The MATLAB script for this file is

```

function I=simpson38(func,a,b,N)
%simpson38: composite Simpson 3/8 rule
% I=simpson38(func,a,b,N)
%input:
% func=name of function to be integrated
%         func defined in vectorized form,
%         for example, by
%         func=inline('x.*exp(x)'), or
%         equivalently by func=@(x)(x.*exp(x))
% a, b=integration limits
% N=number of intervals (default=99).
%     N must be an divisible by three
%output:
% I=integral estimate
if nargin<3,error('at least 3 input arguments required'),end
if nargin<4|isempty(N),N=99;end
if ~(b>a),error('upper bound must be greater than lower'),end
if mod(N,3)~=0
    disp('N must be an integer multiple of 3, it will be
replaced by N+(3-mod(N,3)) ')
    N=N+(3-mod(N,3));
end
x=linspace(a,b,N+1);
h=(b-a)/N;
y=func(x);
I=(3*h/8)*( y(1) + 3*sum(y(2:3:end-2))...
+ 3*sum(y(3:3:end-1)))...

```

```
+2*sum(y(4:3:end-3))+ y(end));
```

The most important change was to replace the formula for I by (11.11.25). In addition to fixing the formula for I , the if-end statement has been modified to cause N to be corrected if a value is selected that is not divisible by 3. Essentially, the script checks the remainder of $\frac{N}{3}$ and adds 1 or 2 to its value in order to calculate with a value that is divisible by 3. Also, the default value of N has been set to 99 in order that it is divisible by 3.

Example 11.11.3: The *Gamma Function*, which is given the symbol Γ , is defined by the indefinite integral

$$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} dx \quad \text{for } z > 0 \quad (11.11.33)$$

The Gamma function arises in various applications, especially in the study of Bessel functions where n is not an integer. When z is a positive integer n , the definition (11.11.33) can be used to show that

$$\Gamma(n) = (n-1)! \quad (11.11.34)$$

For our purposes, it is just an integral we wish to evaluate as an illustration of the multiple application Simpson 3/8 rule. However, we do encounter a small problem. We have established techniques to work problems where the integral is of the form (11.11.1), repeated,

$$I = \int_a^b f(x) dx \quad (11.11.35)$$

The integral in (11.11.33) has a zero lower limit and an *infinite* upper limit. The technical meaning of the integral in (11.11.33) is

$$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} dx \equiv \lim_{b \rightarrow \infty} \int_0^b x^{z-1} e^{-x} dx \quad (11.11.36)$$

As we have observed, computers do discrete operations. They do not take limits. What one does when evaluating (11.11.36) is to numerically evaluate the integral for a sequence of large upper limits and select the one, if it exists, that this sequence approaches.

Because we have MATLAB available, in our simply example, we always know the answer in advance. We shall evaluate the integral (11.11.33) in the case $z=10$. The MATLAB script **gamma(10)** yields the answer 362,880. Note in passing that the MATLAB script **factorial(9)** yields the same result, as (11.11.34) requires.

We will use the function file **I=simpson38** and the script

```

clc
clear
a=0
b=200
N=37*3
func=@(x)(x.^10-1).*exp(-x))
I=simpson38(func,a,b,N)

```

The answer that results is

$$I=363,740 \quad (11.11.37)$$

If we change N to be 300, the correct answer 362,880 is obtained.

Example 11.11.4: If we repeat Example 11.11.1 and 11.11.2 but utilize the Composite Boole Rule, the resulting table turns out to be

Composite Boole Rule

Exact Integral=0.43473

Segments	Step Size=h	Approximate Value	Truncation Error
2	1.2500	0.3273	24.71
3	0.8333	0.4312	0.82
4	0.6250	0.4342	0.11
5	0.5000	0.4346	0.03
6	0.4167	0.4347	0.01
7	0.3571	0.4347	0.00
8	0.3125	0.4347	0.00

The function m-file that will calculate the approximate integral is the file **boole.m** with the script

```

function I=boole(func,a,b,N)
%boole: composite Boole rule
% I=boole(func,a,b,N)
%input:
% func=name of function to be integrated
%       func defined, for example, by
%       func=inline('x*exp(x)'), or
%       equivalently by func=@(x)(x*exp(x))
% a, b=integration limits
% N=number of segments (default=100).
%   N must be an divisible by three
%output:
% I=integral estimate
if nargin<3,error('at least 3 input arguments required'),end
if nargin<4|isempty(N),N=100;end

```

```

if ~ (b>a), error('upper bound must be greater than lower'), end
if mod(N,4) ~= 0
    disp('N must be an integer multiple of 4, it will be
replaced by N+(4-mod(N,4)) ')
    N=N+(4-mod(N,4));
end
x=linspace(a,b,N+1);
h=(b-a)/N;
y=func(x);
I=(2*h/45)*( 7*y(1) + 32*sum(y(2:4:end-2))...
+ 12*sum(y(3:4:end-1))...
+32*sum(y(4:4:end))...
+14*sum(y(5:4:end-3))+ 7*y(end));

```

As one would expect, MATLAB has commands that implement numerical integration schemes. The first of these is the command **trapz**. This command computes an approximation of an integral utilizing the trapezoidal method. The syntax is

$$I = \text{trapz}(x, y) \quad (11.11.38)$$

where **x** is a row or column vector of values of the independent variable and **y** a vector of corresponding dependent variables arrayed in the same dimension as **x**. An *important feature* of **trapz** is that it will integrate data sets that are not necessarily given by a function. Certain applications require the area under a curve where the curve is given as experimental data and not the result of evaluating a function.

The second MATLAB integration command is **quad**. The syntax is

$$I = \text{quad}(\text{func}, a, b) \quad (11.11.39)$$

where **func** is the function to be integrated expressed as an inline function or, equivalently, as an anonymous function. As usual the quantities **a** and **b** represent the limits of the integration. **quad** uses an adaptive Simpson method of integration. Finally, MATLAB has the command **quadl** that has the same syntax as **quad**

$$I = \text{quadl}(\text{func}, a, b) \quad (11.11.40)$$

This command uses what is called an *adaptive Lobatto method*.²⁷ MATLAB provides a family of integration tools in addition to those mentioned here. Examples are **integral**, **quadgk**, and **polyint**.

Often integrations appear as double integrals of the form²⁸

²⁷ Information about the Dutch mathematician Rehuel Lobatto can be found at http://en.wikipedia.org/wiki/Rehuel_Lobatto.

$$I = \int_c^d \int_a^b f(x, y) dx dy \quad (11.11.41)$$

For a function $f(x, y)$ defined on a rectangle $a \leq x \leq b$ and $c \leq y \leq d$. Such integrals arise in the applications when, for example, one wants to find the moment of a load distributed over a surface. Triple integrals also arise. They are of the form

$$I = \int_e^g \int_c^d \int_a^b f(x, y) dx dy dz \quad (11.11.42)$$

The approach to evaluating such integrals numerically is essentially the same as the iterative integral approach usually discussed in elementary Calculus. The procedure is to express the integral as a series of single integrals. For example,

$$I = \int_c^d \int_a^b f(x, y) dx dy = \int_c^d \left(\underbrace{\int_a^b f(x, y) dx}_{y \text{ held constant}} \right) dy \quad (11.11.43)$$

Except for certain poorly behaved functions, we know from Calculus that the order of the integration is not important. Therefore, we could have written

$$I = \int_c^d \int_a^b f(x, y) dx dy = \int_a^b \left(\underbrace{\int_c^d f(x, y) dy}_{x \text{ held constant}} \right) dx \quad (11.11.44)$$

Example 11.11.5: Consider the function

$$f(x, y) = e^{-x^2 y^2} \quad (11.11.45)$$

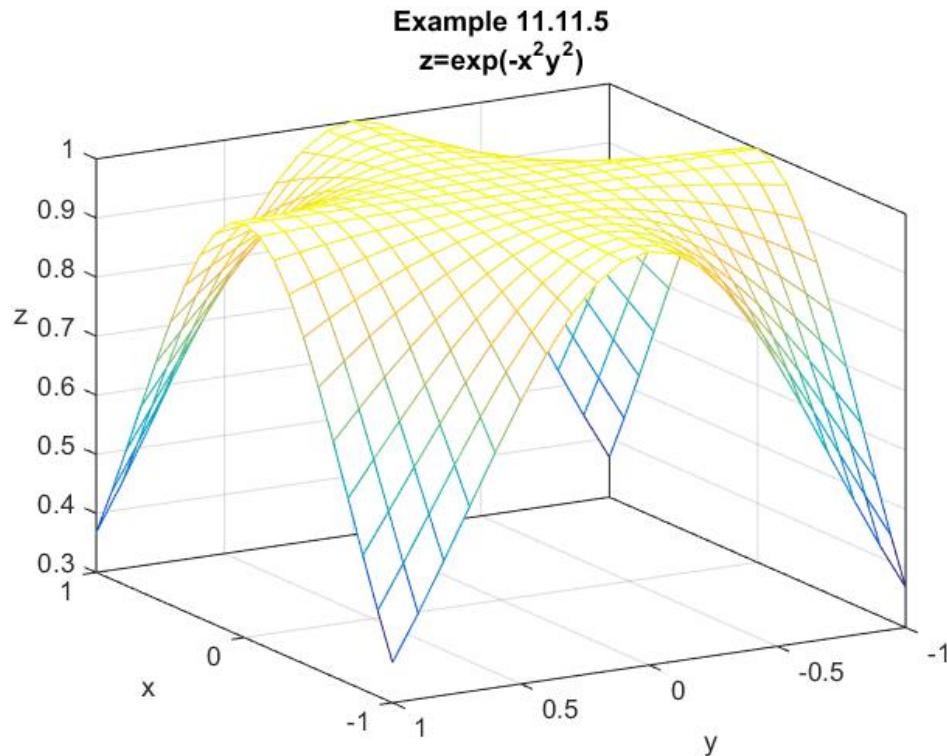
and the problem of evaluating the double integral

$$I = \int_{-1}^1 \int_{-1}^1 f(x, y) dx dy \quad (11.11.46)$$

The plot of this function is

²⁸ Multiple integrals also arise where the limits of the first integral depend upon the next variable of integration. In such a case, (11.11.41) is replaced by

$$I = \int_c^d \int_{a(y)}^{b(y)} f(x, y) dx dy$$



In order to work problems (using the Simpson 3/8 rule) that require double integration, we can modify the function m-file **simpson38.m** by the following:

```

function I=simpsonmulti38(func,a,b,c,d,N1,N2)
%simpsonmulti38: Double integral using Simpson 3/8 method
% I=simpsonmulti38(func,a,b,c,d,N1,N2)
%input:
% func=name of function to be integrated
% a,b,c,d limits of integration
% N1=number of segments in first independent variable
% N2=number of segments in second independent variable
%     default on both=99
%     both must be divisible by 3
%output:
% I=integral estimate
if nargin<5,error('at least 5 input arguments required'),end
if nargin<6|isempty([N1,N2]),N1=99,N2=99;end
if ~(b>a)|~(d>c),error('upper bounds must be greater than
lower'),end
if mod(N1,3)~=0
    disp('N1 must be an integer multiple of 3, it will be replaced
by N1+(3-mod(N1,3)) ')
    N1=N1+(3-mod(N1,3));
end
if mod(N2,3)~=0

```

```

    disp('N2 must be an integer multiple of 3, it will be replaced
by N2+(3-mod(N2,3)) ')
    N2=N2+(3-mod(N2,3));
end
x=linspace(a,b,N1+1);
y=linspace(c,d,N2+1);
[X,Y]=meshgrid(x,y)
h1=(b-a)/N1;
h2=(d-c)/N2;
Z=func(X,Y)
for i=1:N2+1
    Ix(i)=(3*h1/8)*( Z(i,1) + 3*sum(Z(i,2:3:end-2)) +
3*sum(Z(i,3:3:end-1))...
    +2*sum(Z(i,4:3:end-3))+ Z(i,end))
end
for i=1:N1+1
    I=(3*h2/8)*( Ix(1) + 3*sum(Ix(2:3:end-2)) + 3*sum(Ix(3:3:end-
1))...
    +2*sum(Ix(4:3:end-3))+ Ix(end))
end

```

The output from MATLAB that is produced by use of the above function file is

	$N_I=3$	$N_I=6$	$N_I=9$	$N_I=12$
$N_2=3$	3.6566	3.6395	3.6394	3.6393
$N_2=6$	3.6395	3.6246	3.6243	3.6242
$N_2=9$	3.6394	3.6243	3.6239	3.6239
$N_2=12$	3.6393	3.6242	3.6239	3.6238

Because the function being integrated, $f(x, y) = e^{-x^2 y^2}$, is symmetric in x and y , it should be a surprise that the numbers in the above table form a symmetric matrix.

MATLAB also has built in functions that will perform numerical multiple integrals. Examples are **integral2**, **integral3**, and **quad2d**

Exercises

11.11.1: Confirm the results (11.11.8) through (11.11.10) for the trapezoidal rule.

11.11.2: Confirm the results (11.11.13) through (11.11.16) for the Simpson 1/3 rule.

11.11.3: Confirm the results (11.11.19) through (11.11.23) for the Simpson 3/8 rule.

11.11.4: The *error function* is a function defined by an integral. Its formal definition is

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-\eta^2} d\eta \quad (11.11.47)$$

Utilize the Simpson 1/3 Rule to obtain an estimate of $\operatorname{erf}(0.5)$. This function is one of the built in functions in MATLAB. MATLAB gives the value of the error function at the point $z = .5$ to be $\operatorname{erf}(0.5) = 0.5205$.

11.11.5: Utilize the Simpson 1/3 Rule to obtain an estimate of the integral

$$I = \int_0^{10} \left(x^2 e^{-x^2/10} \right) dx \quad (11.11.48)$$

11.11.6: Utilize the Simpson 3/8 rule to obtain an estimate of the integral

$$I = \int_0^{\pi} \frac{1}{e^x + e^{-x}} dx \quad (11.11.49)$$

11.11.7: Utilize the Simpson 3/8 rule to obtain an estimate of the integral

$$I = \int_0^{\frac{\pi}{3}} \frac{x e^x}{\cos(x^2)} dx \quad (11.11.50)$$

11.11.8: Use the Composite Boole Rule to obtain an estimate of the integral

$$I = \int_{x=0}^{30} f(x) dx = \int_{x=0}^{30} \left(200 \left(\frac{x}{x+7} \right) e^{-\frac{2.5x}{30}} \right) dx \quad (11.11.51)$$

11.11.9: An alternate way to define the Bessel function of order zero is the formula ²⁹

$$J_0(x) = \frac{1}{\pi} \int_0^{\pi} \cos(x \sin \theta) d\theta \quad (11.11.52)$$

Use the Composite Boole Rule to obtain an estimate of $J_0(8)$.

11.11.10: Evaluate the integral

$$I = \int_{1/10}^{10} Y_0(x) dx \quad (11.11.53)$$

²⁹ See, for example, http://en.wikipedia.org/wiki/Bessel_function.

where $Y_0(x)$ is the *Bessel function of the second kind* and of zero order. As with Bessel functions of the first kind, the numerical values of $Y_0(x)$ are in MATLAB and called by the syntax **bessely(0,x)**. The function $Y_0(x)$ goes to $-\infty$ as $x \rightarrow 0$. The integration schemes discussed in this section all have step sizes of fixed length. The built in MATLAB functions have adaptable step sizes that are more accurate near singularities.

Calculate estimates of the integral (11.11.53) for the following cases

- a. Simpson 1/3 rule for N=20.
- b. Simpson 3/8 rule for N=21.
- c. **quad**
- d. **quadl**

You will obtain your best answers for parts c. and d.

11.11.11: Evaluate the integral

$$I = \int_{-1}^1 \int_{-1}^1 e^{-x^2} J_0(xy) dx dy \quad (11.11.54)$$

Utilize both the Simpson 3/8 Rule with $N_1 = N_2 = 6$, and MATLAB's built in function **integral2**.

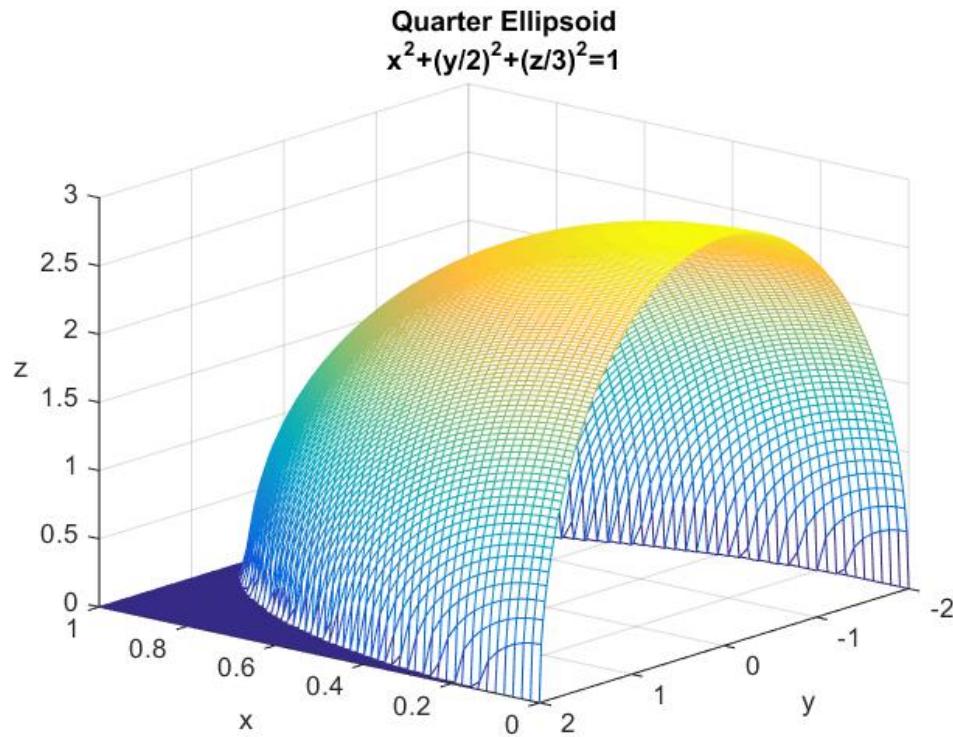
11.11.12: Utilize **integral2** to evaluate

$$I = 3 \int_{y=-2}^2 \int_{x=0}^{\sqrt{1-\left(\frac{y}{2}\right)^2}} \sqrt{1-x^2 - \left(\frac{y}{2}\right)^2} dx dy \quad (11.11.55)$$

Equation (11.11.55) is the equation for one fourth of the volume of the ellipsoid

$$x^2 + \left(\frac{y}{2}\right)^2 + \left(\frac{z}{3}\right)^2 = 1 \quad (11.11.56)$$

The volume to be determined is shown in the following figure



11.11.13: Evaluate the integral

$$I = \int_0^6 \int_0^6 e^{-x^2} \operatorname{erf}(xy) dx dy \quad (11.11.57)$$

Utilize both the Simpson 3/8 Rule with $N_1 = N_2 = 60$, and MATLAB's built in function `integral2`.

Chapter 12

ORDINARY DIFFERENTIAL EQUATIONS ¹

In Chapter 5, we discussed eigenvalue problems and their application in finding the solution of systems of ordinary differential equations. In the introduction to Chapter 9, we classified, in a simplified way, mathematical problems by the techniques one uses when finding its solution. Roughly speaking, the classifications were as follows:

- Systems of nonlinear algebraic equations, expressed in vector notation,

$$\mathbf{f}(\mathbf{y}) = \mathbf{0} \quad (12.1.1)$$

- Systems of nonlinear ordinary differential equations ²

$$\left\{ \begin{array}{l} \frac{dy_1}{dx} = f_1(x, y_1, y_2, \dots, y_m) \\ \frac{dy_2}{dx} = f_2(x, y_1, y_2, \dots, y_m) \\ \cdot \\ \cdot \\ \cdot \\ \frac{dy_m}{dx} = f_N(x, y_1, y_2, \dots, y_m) \end{array} \right. \quad (12.1.2)$$

m equations m unknowns

which one would normally write in a vector notation as

¹ There are a large number of reference textbooks that provide great coverage of the numerical solution of ordinary differential equations. Certain of these books will be referenced as we proceed through this chapter. Three references that are especially useful to students are

1. Polking, John C., and David Arnold, *Ordinary Differential Equations using MATLAB*, Third Edition, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2004.
2. Moler, Cleve, *Numerical Computing with MATLAB*, SIAM, Philadelphia, 2004. The electronic edition is at <http://www.mathworks.com/moler>.
3. Shampine, L. F., I. Gladwell and S. Thompson, *Solving ODEs with MATLAB*, Cambridge University Press, 2003. Electronic versions of this reference can be found on the web.

² Initial conditions are also required.

$$\frac{dy}{dx} = \mathbf{f}(x, \mathbf{y}) \quad (12.1.3)$$

c) Systems of nonlinear partial differential equations.

In Chapter 9, we discussed certain aspects of finding the solution of (12.1.1). In this chapter we are interested in numerical schemes that can be used to find the solution of various systems that fit the form (12.1.3). To be more specific, our first interest is to obtain numerical solutions to the following *initial value problem*:

Find the solution for the $m \times 1$ column vector $\mathbf{y} = \mathbf{y}(x)$ of the ordinary differential equation ³

$$\frac{d\mathbf{y}}{dx} = \mathbf{f}(x, \mathbf{y}) \quad (12.1.4)$$

on an interval (a, b) which satisfies, at some $x_0 \in (a, b)$, an *initial condition*

$$\mathbf{y}(x_0) = \mathbf{y}_0 \quad (12.1.5)$$

where \mathbf{y}_0 is given.

Section 12.1. Normal Form of a System of Ordinary Differential Equations

With rather great generality, almost all systems of ordinary differential equations can be put into the form (12.1.4). As an illustration, consider a manipulation that is essentially like the one used in Section 5.5 where the normal form of system of *linear* ordinary differential equations was discussed. You are given an m^{th} order nonlinear ordinary differential equation in the form

$$\frac{d^m u}{dx^m} = g(x, u, \frac{du}{dx}, \dots, \frac{d^{m-1}u}{dx^{m-1}}) \quad (12.1.6)$$

³ Depending upon the particular example or application, we shall write the initial value problem as illustrated in (12.1.4) and (12.1.5) or we shall use the equivalent notation

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x}) \quad \text{subject to } \mathbf{x}(t_0) = \mathbf{x}_0$$

We used this alternate notation for the special cases discussed in Sections 5.5 through 5.7 and in Section 6.5.

The *initial value problem* for this nonlinear ordinary differential equation is the problem to find a solution of the ordinary differential equation on an interval (a, b) which satisfies at

$x_0 \in (a, b)$ the following initial conditions:

$$\begin{aligned} u(x_0) &= u_0, \\ \frac{du(x_0)}{dx} &= u_1, \\ &\vdots \\ \frac{d^{m-1}u(x_0)}{dx^{m-1}} &= u_{m-1}, \end{aligned} \tag{12.1.7}$$

where u_0, u_1, \dots, u_{m-1} are given constants. Our objective is to write the initial value problem (12.1.6) and (12.1.7) in the form of the initial value problem (12.1.4) and (12.1.5), i.e., in the form

$$\frac{dy}{dx} = \mathbf{f}(x, y) \tag{12.1.8}$$

on an interval (a, b) which satisfies, at some $x_0 \in (a, b)$, an *initial condition*

$$y(x_0) = y_0 \tag{12.1.9}$$

where y_0 is given. The importance of the form (12.1.8) in the study of ordinary differential equations causes it to be given a special name. It is called the *normal form* of a system of ordinary differential equations.

As a first step, we *define* the column vector $\mathbf{y}(x)$ by

$$\mathbf{y}(x) = \begin{bmatrix} y_1(x) \\ y_2(x) \\ y_3(x) \\ \vdots \\ y_m(x) \end{bmatrix} \equiv \begin{bmatrix} u(x) \\ \frac{du(x)}{dx} \\ \frac{d^2u(x)}{dx^2} \\ \vdots \\ \frac{d^{m-1}u(x)}{dx^{m-1}} \end{bmatrix} \tag{12.1.10}$$

The derivative of the definition (12.1.10) yields

$$\frac{d\mathbf{y}(x)}{dx} = \begin{bmatrix} \frac{dy_1(x)}{dx} \\ \frac{dy_2(x)}{dx} \\ \frac{dy_3(x)}{dx} \\ \vdots \\ \frac{dy_m(x)}{dx} \end{bmatrix} = \begin{bmatrix} \frac{du(x)}{dx} \\ \frac{d^2u(x)}{dx^2} \\ \frac{d^3u(x)}{dx^3} \\ \vdots \\ \frac{d^mu(x)}{dx^m} \end{bmatrix} = \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_m \\ g(x, y_1, y_2, \dots, y_m) \end{bmatrix} \quad (12.1.11)$$

Next, we simply *define* the vector valued function $\mathbf{f}(x, \mathbf{y})$ by

$$\mathbf{f}(x, \mathbf{y}) = \begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_m \\ g(x, y_1, y_2, \dots, y_m) \end{bmatrix} \quad (12.1.12)$$

and the single m^{th} order nonlinear ordinary differential equation (12.1.6) has been replaced by the system of m first order nonlinear ordinary differential equations (12.1.8). The *initial condition* on the first order vector equation (12.1.8) is inherited from the initial conditions (12.1.7) and the definition (12.1.10). It follows from these two equations and (12.1.9) that

$$\mathbf{y}_0 = \mathbf{y}(x_0) = \begin{bmatrix} y_1(x_0) \\ y_2(x_0) \\ y_3(x_0) \\ \vdots \\ y_m(x_0) \end{bmatrix} \equiv \begin{bmatrix} u(x_0) \\ \frac{du(x_0)}{dx} \\ \frac{d^2u(x_0)}{dx^2} \\ \vdots \\ \frac{d^{m-1}u(x_0)}{dx^{m-1}} \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{m-1} \end{bmatrix} \quad (12.1.13)$$

The importance of the above manipulation to us is that all of our numerical methods begin with writing the ordinary differential equations to be solved in the *normal form* (12.1.8)

The manipulation leading from (12.1.6) to (12.1.13) is even more general than it might appear. It is not difficult to see that we could have started with a *system* of n , m^{th} order, nonlinear ordinary differential equations of the form

$$\frac{d^m \mathbf{u}}{dx^m} = \mathbf{g}(x, \mathbf{u}, \frac{d\mathbf{u}}{dx}, \dots, \frac{d^{m-1}\mathbf{u}}{dx^{m-1}}) \quad (12.1.14)$$

and by steps entirely similar to (12.1.10) through (12.1.12) again reached (12.1.8), except at this time, the vector \mathbf{y} is of dimension $nm \times 1$.

Example 12.1.1: You are given an initial value problem for the third order nonlinear ordinary differential equation

$$\begin{aligned} \frac{d^3u}{dx^3} + 2\frac{d^2u}{dx^2} + \left(\frac{du}{dx}\right)^2 + 3u &= 0 \\ u(5) = 1, \frac{du(5)}{dx} = 0 \quad \text{and} \quad \frac{d^2u(5)}{dx^2} = 0 \end{aligned} \quad (12.1.15)$$

and the task is to express this system of equations in the normal form (12.1.8). The first step is always to define the column vector \mathbf{y} that appears in (12.1.8). In this case, as follows from (12.1.10), is

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} u \\ \frac{du}{dx} \\ \frac{d^2u}{dx^2} \end{bmatrix} \quad (12.1.16)$$

The next step is always to form the left side of (12.1.8) by differentiation of (12.1.16). The result is

$$\frac{d\mathbf{y}}{dx} = \frac{d}{dx} \begin{bmatrix} u \\ \frac{du}{dx} \\ \frac{d^2u}{dx^2} \end{bmatrix} = \begin{bmatrix} \frac{du}{dx} \\ \frac{d^2u}{dx^2} \\ \frac{d^3u}{dx^3} \end{bmatrix} \quad (12.1.17)$$

The following step is to use the given ordinary differential equations (12.1.15) to express the right hand side of (12.1.17) in the form of the right hand side of (12.1.8). This calculation is

$$\frac{dy}{dx} = \begin{bmatrix} \frac{du}{dx} \\ \frac{d^2u}{dx^2} \\ \frac{d^3u}{dx^3} \end{bmatrix} = \begin{bmatrix} \frac{du}{dx} \\ \frac{d^2u}{dx^2} \\ -2\frac{d^2u}{dx^2} - \left(\frac{du}{dx}\right)^2 - 3u \end{bmatrix} = \begin{bmatrix} y_2 \\ x_3 \\ -2y_3 - y_2^2 - 3y_1 \end{bmatrix} \equiv \mathbf{f}(x, \mathbf{y}) \quad (12.1.18)$$

Example 12.1.2: You are given a system of two coupled ordinary differential equations of the form

$$\begin{aligned} \frac{d^2y}{dt^2} + y + 4\frac{dx}{dt} - 4x &= 4e^t \\ \frac{dy}{dt} - y + \frac{dx}{dt} + 9x &= 0 \\ y(0) = 5 \quad \frac{dy(0)}{dt} = 0 \quad x(0) = \frac{1}{2} \end{aligned} \quad (12.1.19)$$

and the task is to express this system of equations in the normal form (12.1.8). The first step is always to define the column vector \mathbf{y} that appears in (12.1.8). In this case, equation (12.1.10) tells us that

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{dy}{dt} \end{bmatrix} \quad (12.1.20)$$

The next step is always to form the left side of (12.1.8) by differentiation of (12.1.20). The result is

$$\frac{dy}{dt} = \frac{d}{dt} \begin{bmatrix} x \\ y \\ \frac{dy}{dt} \end{bmatrix} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{d^2y}{dt^2} \end{bmatrix} \quad (12.1.21)$$

The following step is to use the given ordinary differential equations (12.1.19) to express the right hand side of (12.1.21) in the form of the right hand side of (12.1.8). This calculation is

$$\begin{aligned}
 \frac{d\mathbf{y}}{dt} = \frac{d}{dt} \begin{bmatrix} x \\ y \\ \frac{dy}{dt} \\ \frac{d^2y}{dt^2} \end{bmatrix} &= \begin{bmatrix} \frac{dx}{dt} \\ -9x + y - \frac{dy}{dt} \\ \frac{dy}{dt} \\ -y - 4\frac{dx}{dt} + 4x + 4e^t \end{bmatrix} = \begin{bmatrix} -9x + y - \frac{dy}{dt} \\ \frac{dy}{dt} \\ -y - 4\left(-9x + y - \frac{dy}{dt}\right) + 4x + 4e^t \end{bmatrix} \\
 &= \begin{bmatrix} -9x + y - \frac{dy}{dt} \\ \frac{dy}{dt} \\ 40x - 5y + 4\frac{dy}{dt} + 4e^t \end{bmatrix} = \begin{bmatrix} -9x + y - \frac{dy}{dt} \\ \frac{dy}{dt} \\ 40x - 5y + 4\frac{dy}{dt} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 4e^t \end{bmatrix} \\
 &= \begin{bmatrix} -9 & 1 & -1 \\ 0 & 0 & 1 \\ 40 & -5 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ \frac{dy}{dt} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 4e^t \end{bmatrix} \equiv \mathbf{f}(t, \mathbf{y})
 \end{aligned} \tag{12.1.22}$$

Equation (12.1.22) is the normal form (12.1.8) in this case. The initial condition for this form of the ordinary differential equation arises from the definition (12.1.20) and the given conditions in (12.1.19). The explicit form of the initial condition is

$$\mathbf{y}_0 = \mathbf{y}(0) = \begin{bmatrix} x(0) \\ y(0) \\ \frac{dy(0)}{dt} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ 5 \\ 0 \end{bmatrix} \tag{12.1.23}$$

There are *exceptional cases* where an ordinary differential equation cannot be put in the form (12.1.14). In such cases, it cannot be put into the normal form (12.1.8). The following example is one where the normal form can only be obtained by imposing a restriction on the independent variable x :

Example 12.1.3: The ordinary differential equation that defines the Bessel functions discussed in Section 11.11 is

$$x^2 \frac{d^2y}{dx^2} + x \frac{dy}{dx} + (x^2 - p^2)y = 0 \tag{12.1.24}$$

where $p > 0$ is a parameter.⁴ If we adopt the starting place where the ordinary differential equation is written in the form (12.1.6), we would need to rewrite (12.1.24) as

$$\frac{d^2y}{dx^2} + \frac{1}{x} \frac{dy}{dx} + \left(1 - \frac{p^2}{x^2}\right)y = 0 \quad (12.1.25)$$

and proceed with the definitions (12.1.10)

$$\mathbf{y}(t) = \begin{bmatrix} y_1(x) \\ y_2(x) \end{bmatrix} \equiv \begin{bmatrix} y(x) \\ \frac{dy(x)}{dx} \end{bmatrix} \quad (12.1.26)$$

and (12.1.12)

$$\mathbf{f}(x, \mathbf{y}) = \begin{bmatrix} \frac{dy}{dx} \\ -\frac{1}{x} \frac{dy}{dx} - \left(1 - \frac{p^2}{x^2}\right)y \end{bmatrix} = \begin{bmatrix} y_2 \\ -\frac{1}{x} y_2 - \left(1 - \frac{p^2}{x^2}\right) y_1 \end{bmatrix} \quad (12.1.27)$$

The *problem* is the division by x which allowed the transformation of (12.1.24) into (12.1.25). Implicitly we made the assumption that the problem does not require knowledge of the solution at or near $x = 0$. It turns out that this particular ordinary differential equation has applications which need the solution at and near $x = 0$. This kind of exceptional case is addressed by *not* forcing the answer to be in the normal form (12.1.4). The approach in this kind of case is to adopt a more general normal form of the form

$$\mathbf{M}(x, \mathbf{y}) \frac{d\mathbf{y}}{dx} = \mathbf{f}(x, \mathbf{y}) \quad (12.1.28)$$

where \mathbf{M} is a $m \times m$ square matrix, possibly depending upon x and \mathbf{y} . If we do not allow the restriction $x \neq 0$, we can easily put (12.1.24) in the form of (12.1.28). The result is

$$\begin{bmatrix} 1 & 0 \\ 0 & x^2 \end{bmatrix} \frac{d\mathbf{y}}{dx} = \begin{bmatrix} y_2 \\ -xy_2 - (x^2 - p^2)y_1 \end{bmatrix} \quad (12.1.29)$$

If the matrix $\mathbf{M}(x, \mathbf{y})$ in (12.1.28) is invertible for all values of its argument, then (12.1.28) and the normal form (12.1.4) are equivalent. In the exceptional cases where $\mathbf{M}(x, \mathbf{y})$ is not

⁴ In Section 11.11, we discussed the case where the parameter p was an integer.

invertible and we cannot replace (12.1.28) by (12.1.4), the solution procedures we are going to discuss become much more complicated.⁵

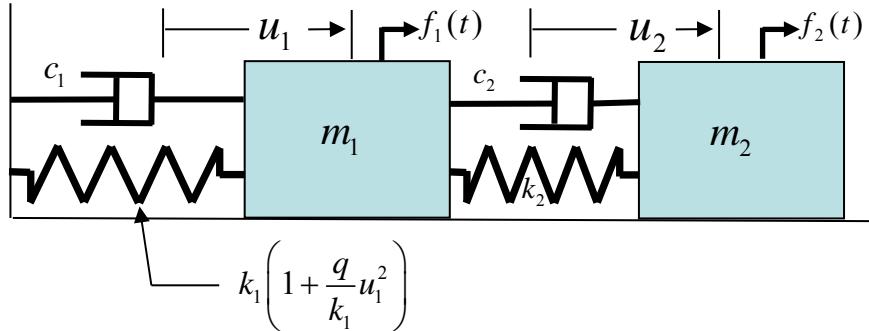
For our discussion of numerical schemes, we shall assume the equations to be solved have been put into the normal form (12.1.4). We shall briefly discuss the case where (12.1.28) is the normal form in Section 12.13.

Exercises

12.1.1: Convert the following initial value problem to an initial value problem for a system in normal form

$$\begin{aligned} 3 \frac{d^2x}{dt^2} + 5x - 2y &= 0 \\ 4 \frac{d^2y}{dt^2} + 2y - 6x &= 0 \\ x(0) = -1 \quad \frac{dx(0)}{dt} &= 0 \\ y(0) = 1 \quad \frac{dy(0)}{dt} &= 2 \end{aligned} \tag{12.1.30}$$

12.1.2: Given a coupled two degree of freedom vibrating system shown in the following figure:



You are given that the first spring is nonlinear with the force-displacement relationship shown. When one takes into account that the first spring is nonlinear, the equations of motion are

⁵ In discussions of the numerical solution of (12.1.28), the matrix $\mathbf{M}(x, y)$ is known as the *mass matrix*. The origin of this name lies in applications, like those discussed in Chapter 5, where $\mathbf{M}(x, y)$ is, in fact, related to the mass of different components of a vibrating systems.

$$\begin{aligned} m_1 \ddot{u}_1 &= -c_1 \dot{u}_1 - k_1 \left(1 + \frac{q}{k_1} u_1^2 \right) u_1 + c_2 (\dot{u}_2 - \dot{u}_1) + k_2 (u_2 - u_1) + f_1(t) \\ m_2 \ddot{u}_2 &= -c_2 (\dot{u}_2 - \dot{u}_1) - k_2 (u_2 - u_1) + f_2(t) \end{aligned} \quad (12.1.31)$$

where m_1 and m_2 are the two masses, k_1 , k_2 and q are spring constants, c_1 and c_2 are damping constants and $f_1(t)$ and $f_2(t)$ are forcing functions. Express the system (12.1.31) in normal form.

Section 12.2. Picard's Theorem

Without additional assumptions there is no guarantee that the initial value problem based upon the first order ordinary differential equation (12.1.8) subject to the initial condition (12.1.9) has a unique solution. Sufficient conditions to guarantee the existence of a unique solution are embodied in a theorem known as *Picard's Theorem*.⁶ It is usually stated in the one dimensional case where the initial value problem (12.1.8) and (12.1.9) reduce to

$$\frac{dy}{dx} = f(x, y) \quad (12.2.1)$$

and

$$y(t_0) = y_0 \quad (12.2.2)$$

In this case, the usual statement of Picard's Theorem is as follows:

Theorem 12.2.1: If the following three conditions are valid:⁷

1. f is a continuous function in a region \mathcal{D} of the (x, y) plane that contains the rectangle

$$\mathcal{N} = \{(x, y) \mid x_0 \leq x \leq x_1, |y - y_0| \leq k\} \quad (12.2.3)$$

where $x_1 > x_0$ and $k > 0$ are constants.

2. There exists a positive constant M such that

$$|f(x, y) - f(x, z)| \leq M |y - z| \quad (12.2.4)$$

is valid when (x, y) and (x, z) are in the rectangle \mathcal{N} .

3. If the number L is defined by

$$L = \max \{ |f(x, y)| \mid (x, y) \in \mathcal{N} \} \quad (12.2.5)$$

and if $L(x_1 - x_0) \leq k$

⁶ Information about the French mathematician, Émile Picard, can be found at http://en.wikipedia.org/wiki/%C3%89mile_Picard.

⁷ The example that is often given of an initial value problem that does not obey the three conditions in the Theorem is $\frac{dy}{dx} = y^{\frac{2}{3}}, y(0) = 0$. The two solutions of this initial value problem are $y(x) = 0$ and $y(x) = \frac{x^3}{27}$. This example fails the condition (12.2.4).

Then, there exists a unique continuously differentiable function $y(x)$ defined on the closed interval $[x_0, x_1]$ that obeys (12.2.1) and (12.2.2).

The condition (12.2.4) is known as a *Lipschitz condition* and M is the *Lipschitz constant*.⁸

The proof of Picard's Theorem will not be given here. It can be found in almost any textbook on ordinary differential equations. Of course, a simple internet search will also find the proof in many places. In addition, the generalization to a system of first order ordinary differential equations like (12.1.8) can also be found.⁹

⁸ Information about the German mathematician, Rudolf Lipschitz, can be found at http://en.wikipedia.org/wiki/Rudolf_Lipschitz.

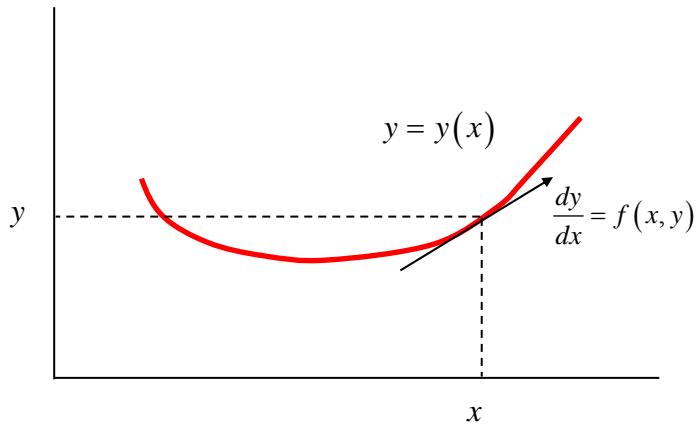
⁹ A suggested textbook that contains the single equation and system versions of Picard's Theorem is the book, Kreider, Donald L., Robert G. Kuller, and Donald R. Ostberg, *Elementary Differential Equations*, Addison-Wesley, 1968.

Section 12.3. Direction Field

If we simplify our discussion and look at the special case of (12.1.8) when $m = 1$, the result is the differential equation

$$\frac{dy}{dx} = f(x, y) \quad (12.3.1)$$

The solution of this equation is a function $y = y(x)$ that obeys some prescribed initial condition. The differential equation (12.3.1) simply states that at any point (x, y) the *slope* of the solution curve is given by $f(x, y)$. As the next figure suggests, one can plot the slopes at every point (x, y) . The result is a *field of vectors* that define the direction of the solution curve through a particular point.



More simply, the *direction field* for the ordinary differential equation (12.3.1) is the field of tangents that defines the evolution of the solution. One element of this direction field is shown in the figure,

Example 12.3.1: Given the simple nonlinear ordinary differential equation

$$\frac{dy}{dx} = x\sqrt{y} \quad (12.3.2)$$

we can generate the direction field in the region $0 < t < 2$ and $0 < x < 10$ by executing the following MATLAB script¹⁰

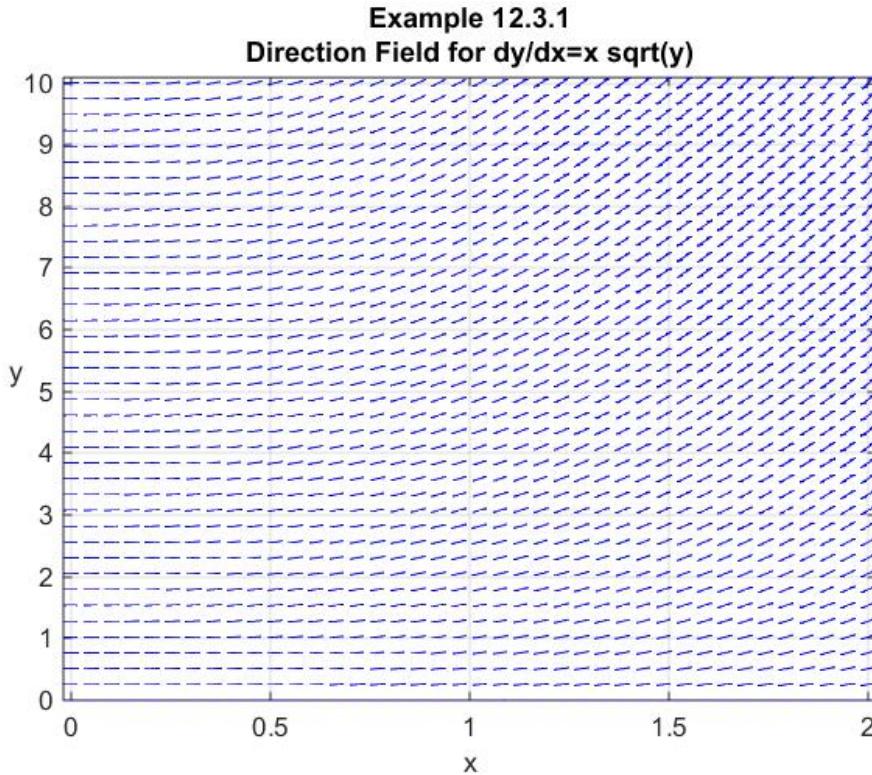
```
clc
clear
yprime = @(x,y)(x.*sqrt(y));
a=0
b=2
xmin=a
xmax=b
ymin=0
ymax=10
N=40
draw_dir_field(yprime,a,b,ymin,ymax,N);
grid on
title({'Example 12.3.1',...
    'Direction Field for dy/dx=x*sqrt(y)'})
xlabel('x')
ylabel('y','Rotation',0)
```

The result is the figure

¹⁰ The script for Example 12.3.1 uses the function m-file **draw_dir_field.m** that has the script

```
function draw_dir_field(f,xmin, xmax, ymin, ymax, N)
%creates direction field
xval = linspace(xmin,xmax,N);
yval = linspace(ymin,ymax,N);
[xm,ym]=meshgrid(xval,yval);
dx = xval(2) - xval(1);
dy = yval(2) - yval(1);
yp=arrayfun(f,xm,ym);
s = 1./max(1/dx,abs(yp)./dy)*0.35;
quiver(xval,yval,s,s.*yp,0,'Color',[0 0 1]); hold on;
quiver(xval,yval,-s,-s.*yp,0,'Color',[0 0 1]);
axis tight;
```

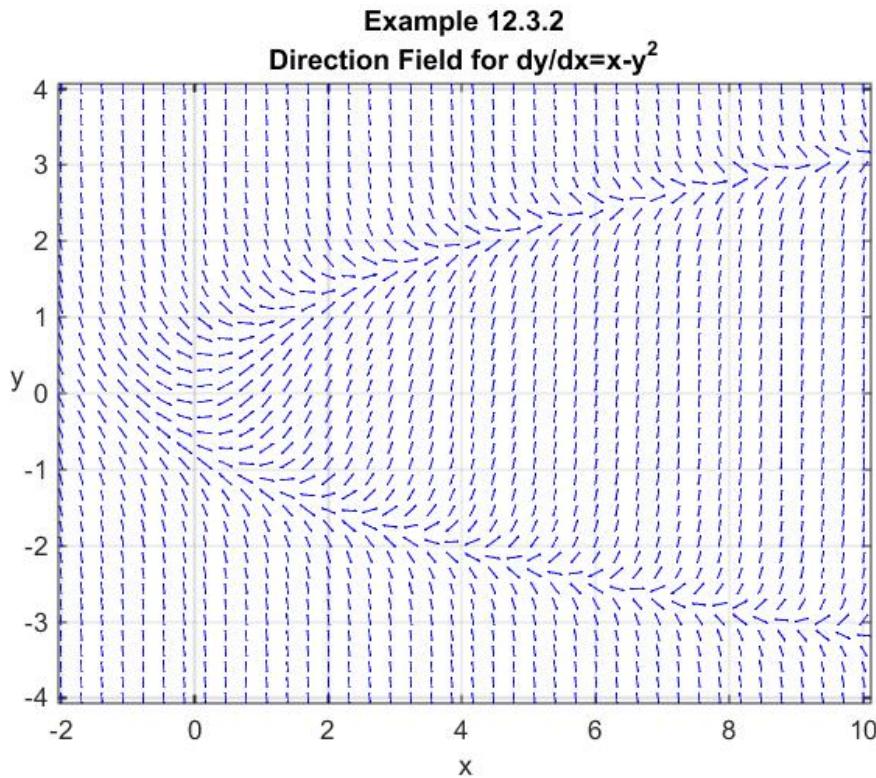
This file uses MATLAB's **quiver** command to generate the direction field. This particular function m-file was written by Dr. Waqar Malik.



Example 12.3.2: Given the simple nonlinear ordinary differential equation

$$\frac{dy}{dx} = x - y^2 \quad (12.3.3)$$

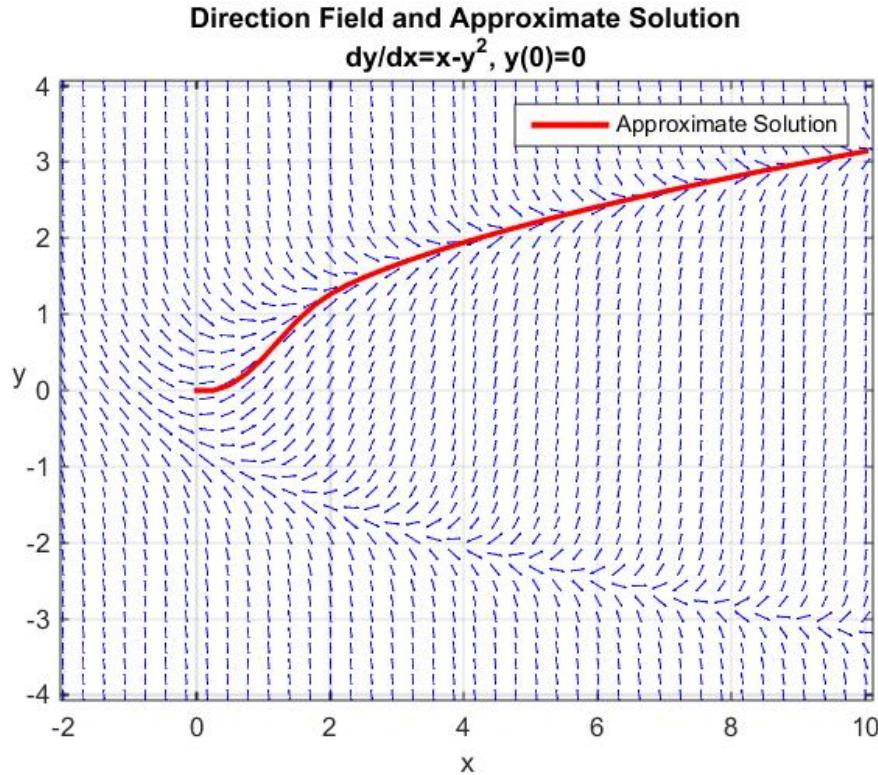
we modify the MATLAB script used in Example 12.3.2 and obtain the direction field in the region $-2 < x < 10$ and $-4 < y < 4$. The resulting figure is



The last two figures suggest an approach to finding a solution to the initial value problem based upon (12.3.1). One simply selects an initial condition and constructs a solution incrementally by drawing tangents to the direction field starting at the initial point. For example, the solution of (12.3.3), repeated,

$$\frac{dy}{dx} = x - y^2 \quad (12.3.4)$$

subject to the initial condition $y(0) = 0$ yields the figure



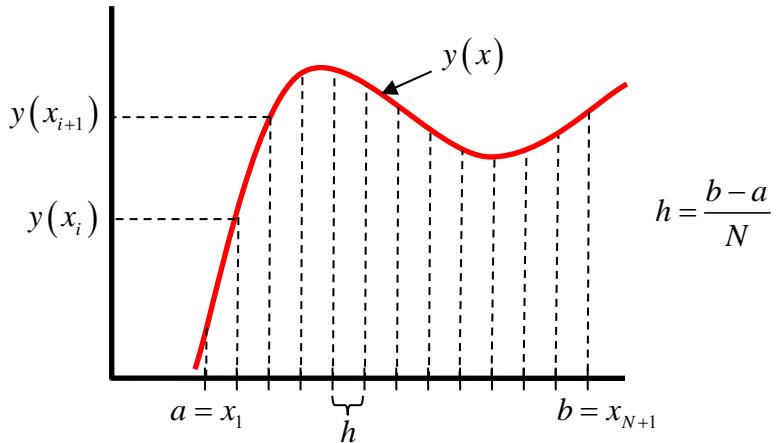
The kinds of numerical schemes we shall discuss rely on the fundamental idea derived from (12.3.1) that, locally, the solution is determined by the direction field. As we shall see, the complexity and the accuracy of our approximation schemes depend upon how much of the direction field in the neighborhood of a point (x, y) is utilized to determine the values of y in the neighborhood of (x, y) .

Section 12.4. Euler's Method: A One Step Iteration Method¹¹

Euler's method is an elementary numerical procedure that is often discussed in introductory ordinary differential equations courses. It attempts to find an approximate numerical solution of the initial value problem

$$\frac{dy}{dx} = f(x, y) \quad y(x_0) = y_0 \quad (12.4.1)$$

and builds an *iterative scheme* by approximating the slope $f(x, y)$ by its value at the beginning of an interval. If you are given a partition of the interval (a, b) into N intervals as shown in the following figure:



the Euler method involves adopting a step size h and a *forward difference approximation* to the derivative at x_i of the form

$$\frac{dy(x_i)}{dx} \cong \frac{y(x_{i+1}) - y(x_i)}{h} \quad (12.4.2)$$

This approximation and the ordinary differential equation (12.4.1) yields the result

$$\frac{y(x_{i+1}) - y(x_i)}{h} \cong f(x_i, y) \quad (12.4.3)$$

¹¹ Leonhard Paul Euler lived from April 15, 1707 until September 18, 1783. He was Swiss mathematician who spent most of his life in Russia and Germany. You can read about him in many places. One place to start is http://en.wikipedia.org/wiki/Leonhard_Euler.

The Euler method is when one adopts (12.4.3) as an *iteration condition*

$$y_{i+1} = y_i + f(x_i, y_i)h \quad (12.4.4)$$

The Euler method is a *one step* method because the value at the point x_{i+1} , y_{i+1} , is determined by values one step from x_{i+1} , at the single point x_i .

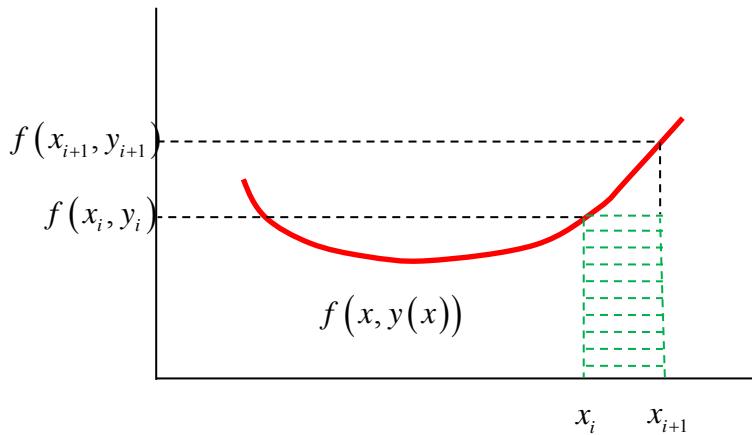
Another way to look at the derivation of (12.4.4) is to integrate (12.4.1) between limits x_i and x_{i+1} to obtain

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} f(x, y(x))dx \quad (12.4.5)$$

The iteration formula is obtained by approximating the area under the integral in (12.4.5) by

$$\int_{x_i}^{x_{i+1}} f(x, y(x))dx \approx f(x_i, y_i)(x_{i+1} - x_i) = f(x_i, y_i)h \quad (12.4.6)$$

In other words, the area under the curve $f(x, y(x))$



is approximated by the rectangle of height $f(x_i, y_i)$ and width $x_{i+1} - x_i = h$.¹²

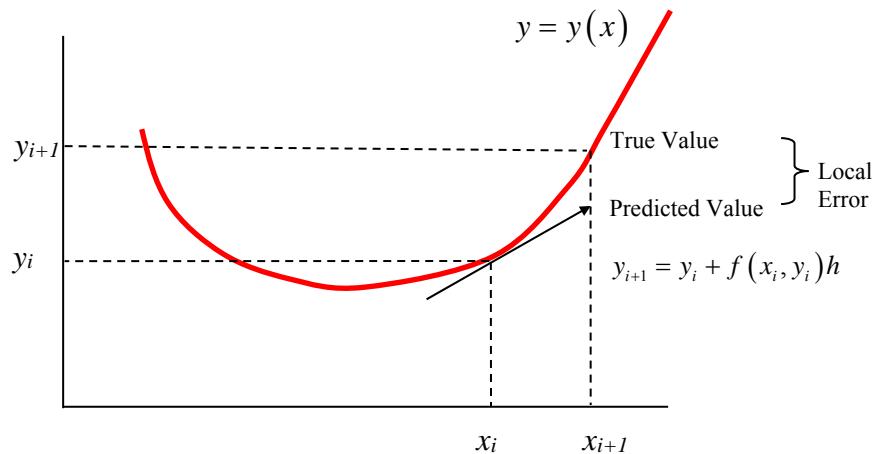
¹² The Euler method based upon (12.4.4) is sometimes referred to as the *explicit Euler method*. The *implicit Euler method* is one based upon the iteration formula $y_{i+1} = y_i + f(x_{i+1}, y_{i+1})h$. The solution method in this case involves solving an implicit equation for each y_{i+1} . This iteration formula can be viewed as the result of approximating the integral in (12.4.5) by $f(x_{i+1}, y_{i+1})h$.

As suggested by the last figure, when (12.4.4) is adopted, it is inevitable that errors are present. This error, a truncation error, is displayed by the Taylor's Theorem representation, equation (8.1.11),

$$\begin{aligned} y_{i+1} &= y_i + \frac{dy(x_i)}{dx} h + \frac{1}{2!} \frac{d^2 y(x_i)}{dx^2} h^2 + \cdots + \frac{1}{n!} \frac{d^n y(x_i)}{dx^n} h^n + O(h^{n+1}) \\ &= y_i + f(x_i, y_i)h + \frac{1}{2!} f'(x_i, y_i)h^2 + \cdots + \frac{1}{n!} h^n f^{(n-1)}(x_i, y_i) + O(h^{n+1}) \end{aligned} \quad (12.4.7)$$

The *truncation error* is everything past the term $O(h)$ in the formula

$$y_{i+1} = y_i + f(x_i, y_i)h + \underbrace{\frac{1}{2!} f'(x_i, y_i)h^2 + \cdots + \frac{1}{n!} h^n f^{(n-1)}(x_i, y_i)}_{\text{Truncation Error} = O(h^2)} + O(h^{n+1}) \quad (12.4.8)$$



Truncation errors consist of the following two elements::

- a. Local Truncation Error: Error from a single step in the iteration.
- b. Propagated Truncation Error: Accumulated error from the previous steps. The error in the $n+1$ iteration consists of the local truncation error for the $n+1$ iteration plus the accumulated errors from the previous n iterations.
 - a. Sometimes the sum of local and propagated truncation errors is called the *global truncation error*.

While the local truncation error is $O(h^2)$, it turns out that the global truncation error is $O(h)$. For this reason, the Euler method is sometimes referred to as a *first order* method. The proof that the global truncation error is $O(h)$ will not be given here.¹³

Example 12.4.1: As an example that utilizes the Euler method, consider the ordinary differential equation (12.3.2), repeated,

$$\frac{dy}{dx} = x\sqrt{y} \quad (12.4.9)$$

For our initial condition, we shall require that

$$y(1) = 4 \quad (12.4.10)$$

By an elementary separation of variable argument, the exact solution of (12.4.9) subject to the initial condition (12.4.10) is

$$y = \frac{1}{16}(x^2 + 7)^2 \quad (12.4.11)$$

If a step size of $h = 0.125$ is adopted, the iteration scheme (12.4.4) can be implemented in the following table

Computation Table				
$h=$	0.125			
i	x_i	y_i	Exact Solution	Error Relative To Exact
1	1	4.00000	4.00000	0.00%
2	1.125	4.25000	4.27003	0.47%
3	1.25	4.53991	4.58228	0.92%
4	1.375	4.87283	4.94020	1.36%
5	1.5	5.25223	5.34766	1.78%
6	1.625	5.68194	5.80885	2.18%
7	1.75	6.16613	6.32837	2.56%
8	1.875	6.70932	6.91115	2.92%
9	2	7.31641	7.56250	3.25%

¹³ The proof can be found in the textbook, Butcher, J. C., Numerical Methods for Ordinary Differential Equations, Second Edition, John Wiley, 2008. The proof can also be found online by executing the appropriate online search.

As is probably evident, the row corresponding to $i=1$ reflects the initial condition on y_i given by (12.4.10). The subsequent entries in the y_i column are calculated from (12.4.4). The Exact Solution column is calculated from (12.4.11) and formula

$$Error = 100 \left| \frac{y(x_i) - y_i}{y(x_i)} \right| \quad (12.4.12)$$

Thus, the Error column is simply a percentage measuring the absolute value of the difference between the exact solution value and the corresponding y_i normalized by the exact solution value. This table can easily be modified by refining the grid. However, for our purposes, it is important to implement the iteration scheme with MATLAB.

MATLAB script that will generate the third column of the above table is

```

a=1;
b=2;
N=8
h=(b-a)/N
x=linspace(a,b,N+1);

yprime=@(x,y)(x.*sqrt(y));
y(1)=4;
for m=1:1:N;
    y(m+1)=y(m)+h*yprime(x(m),y(m));
end

```

This elementary script defines the function $f(x, y)$ in (12.4.1) as the anonymous function **fprime**. The **for-end** loop creates the iteration (12.4.4). The MATLAB **fprintf** command can be combined with the above script to produce the above table. As an illustration, the script

```

clc
clear
a=1;
b=2;
N=8
h=(b-a)/N
x=linspace(a,b,N+1);
yprime=@(x,y)(x.*sqrt(y));
y(1)=4;
fprintf(' \n\n\tti\ttx_i\tty_i\texact y\tError')
fprintf(' \n\tt_____')
fprintf(' \n\t%5.0f \t%5.4f\t%5.4f\t%5.4f%',...
    1,x(1),y(1),y(1),0)

```

```

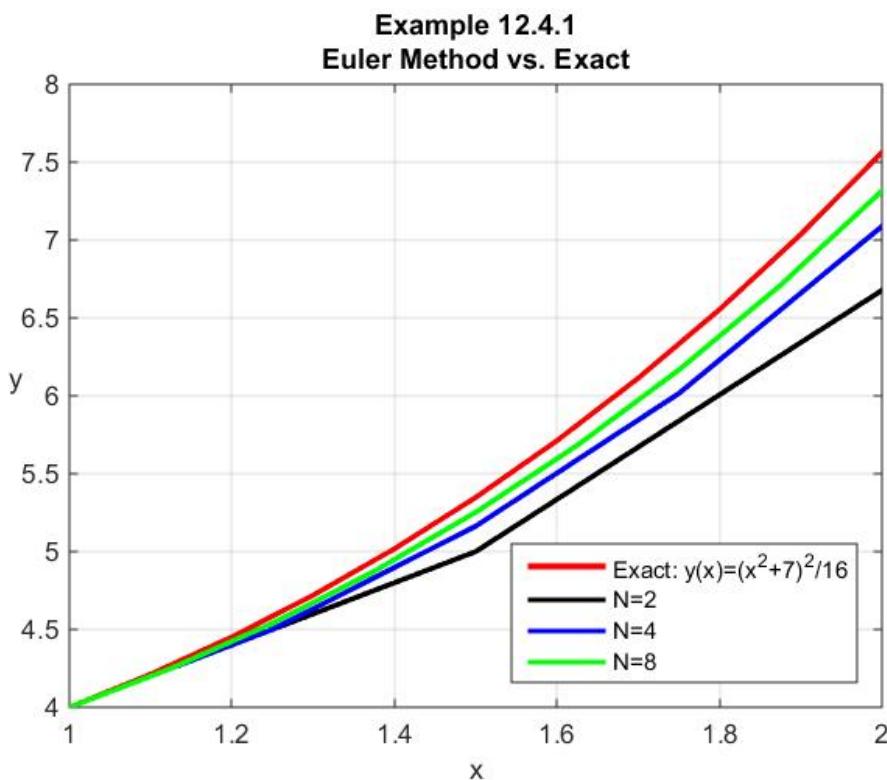
for m=1:1:N;
y(m+1)=y(m)+h*yprime(x(m),y(m));
yexact(m+1)=(x(m+1).^2+7).^2/16;
d(m+1)=abs(100*(yexact(m+1)-y(m+1))/yexact(m+1));
fprintf('\n\t%5.0f \t%5.4f\t%5.4f \t%5.4f\t%5.4f%%',...
m+1,x(m+1),y(m+1),yexact(m+1),d(m+1))
end

```

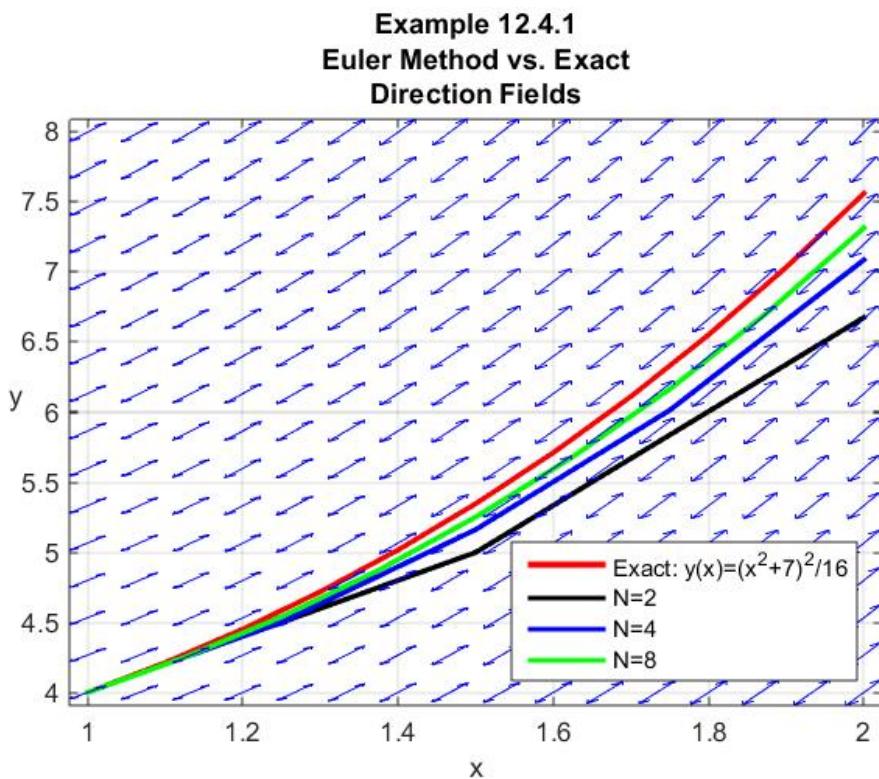
will, for the case $N = 8$, yield the table

i	x_i	y_i	exact y	Error
1	1.0000	4.0000	4.0000	0.0000%
2	1.1250	4.2500	4.2700	0.4692%
3	1.2500	4.5399	4.5823	0.9246%
4	1.3750	4.8728	4.9402	1.3638%
5	1.5000	5.2522	5.3477	1.7844%
6	1.6250	5.6819	5.8089	2.1848%
7	1.7500	6.1661	6.3284	2.5637%
8	1.8750	6.7093	6.9111	2.9203%
9	2.0000	7.3164	7.5625	3.2541%

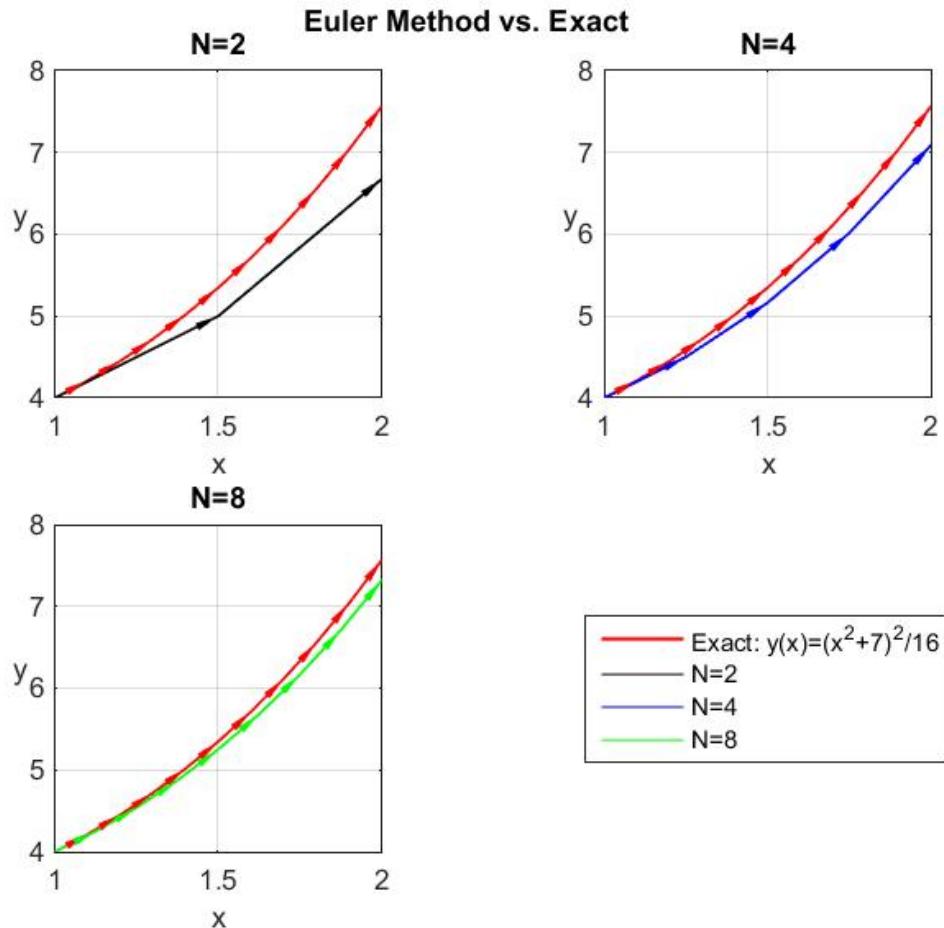
The sensitivity of error to the choice of N can easily be checked by running the above script for other choices. A more instructive evaluation of this dependence is obtained if one simply plots the exact solution (12.4.11) vs various approximate solutions obtained by different choices of N . The following figure illustrates this kind of plot for the choices $N = 8, 16$ and 32 :



It is perhaps instructive to overlay on the above figure the direction field for the ordinary differential equation (12.4.9). The result is



A figure which, perhaps, improves the display of the exact solution, the approximate solution and the underlying direction field is



The elementary Example 12.4.1 does show that the accuracy of the Euler method increases with reduced step size. This accuracy is obtained at the expense of a larger computational effort. While the Euler method is somewhat inefficient, it is simple to implement.

Example 12.4.2: As an additional example that utilizes the Euler method, consider the ordinary differential equation (12.3.3), repeated,

$$\frac{dy}{dx} = x - y^2 \quad (12.4.13)$$

Our purpose with this example is to illustrate an interesting point about nonlinear ordinary differential equations. It is a feature of linear ordinary differential equations that a small change in the initial conditions will produce a small change in the solution. As this example will illustrate, a small change in the initial conditions will, for some initial conditions for some differential equations, produce a large change in the solution. We shall illustrate this feature by

solving, by the Euler method, subject to a range of initial conditions. The initial conditions we shall adopt are

$$y(x_0) = 4 \quad (12.4.14)$$

for the eight choices

$$x_0 = [-2, -1.9, -1.804093, -1.804092, -1.6, -1.4, -1.2, -1.0] \quad (12.4.15)$$

The particular choices have been selected to illustrate the feature of nonlinear ordinary differential equations just mentioned. Like Example 12.4.1, this example has an exact solution. However, it is a little more complicated to derive, and it is not essential to our discussion.¹⁴ The MATLAB script that utilizes the Euler method for the eight initial conditions and that presents the solutions as a graph is¹⁵

```
clc
clear
a=-2;
b=10;
%Draw Direction Field
yprime=@(x,y)(x-y.^2)
N=40;
xmin=-2;
xmax=10;
ymin=-4;
ymax=4;
draw_dir_field(yprime,xmin,xmax,ymin,ymax,N);
```

¹⁴Equation (12.4.13) is an ordinary differential equation of a type known as Riccati's equation. It is named after the Italian mathematician Jacopo Francesco Riccati. Information about Riccati can be found at http://en.wikipedia.org/wiki/Jacopo_Francesco_Riccati. As explained, for example, in the textbook Ince, E. L., Ordinary Differential Equations, Dover Publications, Inc., 1956, the change of variables

$$y(x) = \frac{1}{u(x)} \frac{du(x)}{dx}$$

converts the nonlinear first order ordinary differential equation (12.4.13) into the second order linear ordinary differential equation

$$\frac{d^2u}{dx^2} - xu = 0$$

This ordinary differential equation is known as the *Airy equation* or the *Stokes equation*. Its solution can be expressed in terms of Airy functions or, equivalently, Bessel functions. Equation (12.4.13) can also be solved by utilizing the MATLAB **dsolve** command.

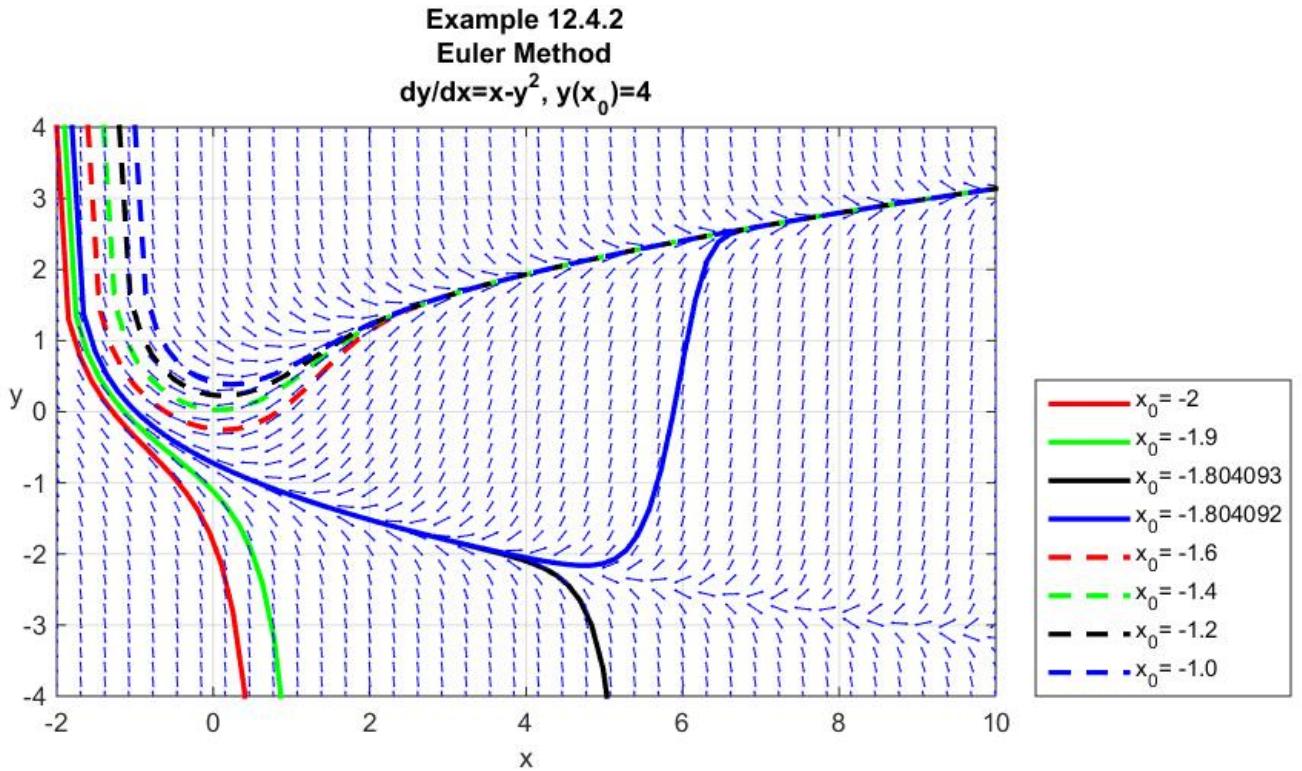
¹⁵The **get** command in the script is utilized to identify the graphic handles for the eight curves that represent the solution. Given these handles, the legend can be created for each curve in the manner shown. A discussion of the **legend** command can be found at <http://www.mathworks.com/help/matlab/ref/legend.html>.

```
%Solve ode for various initial conditions
N=80;
%Possible starting points
x0=[-2,-1.9,-1.804093,-1.804092,-1.6,-1.4,-1.2,-1.0];
%Value of y the same at all starting points
y(1)=4;
%Set eight line colors and line styles
color='rgkbrgkb';
lines=char('---','--','-.','-.','--','--','-.','--');
hold on
grid on
for k=1:length(x0);
    h=(b-x0(k))/N;
    x=linspace(x0(k),b,N+1);
    for n=1:N;
        y(n+1)=y(n)+h*yprime(x(n),y(n));
    end
    plot(x,y,'color',color(k),...
        'LineWidth',2,'linestyle',lines(k,:))
    axis([-2,10,-4,4])
end

xlabel('x')
ylabel('y','Rotation',0)
title({['Example 12.4.2'],'Euler Method','dy/dx=x-y^2',...
y(x_0)=4'});

%The following get command determines the graphic handles
%for the eight curves that represent the solutions. There
%are ten graphic handles for the axis object. Two of these
%are created by the quiver command that is a part of
%draw_dir.
h=get(gca,'children')
legend([h(8:-1:1)],'x_0= -2','x_0= -1.9',...
    'x_0= -1.804093','x_0= -1.804092',...
    'x_0= -1.6','x_0= -1.4',...
    'x_0= -1.2','x_0= -1.0','Location','SouthEastOutside')
```

The figure created by the above script is



The first three curves, the red one, the green one and the black one, start at the initial position shown in the legend at the value (12.4.14) and follow the paths shown to large negative values. The blue curve starts at an initial position that is infinitesimally close to the proceeding black curve and evolves to a fundamentally different solution. The following four curves, those shown in dashed lines, follow the upper branch suggested by the underlying direction field. The numerical values of the initial condition that result in the bifurcation of the two solutions was found by the simple brute force method of trying various values until desired plot was obtained. The initial condition resulting in the bifurcation depend upon the number of intervals N selected.

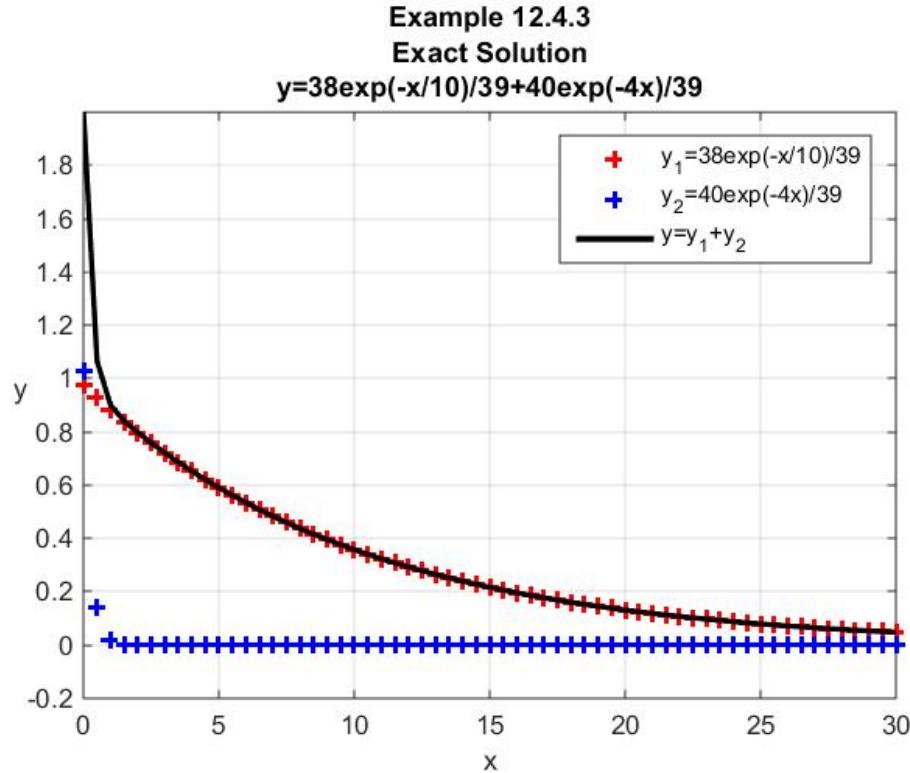
Example 12.4.3: You are given the linear ordinary differential equation

$$\frac{dy}{dx} + \frac{1}{10}y = -4e^{-4x} \quad (12.4.16)$$

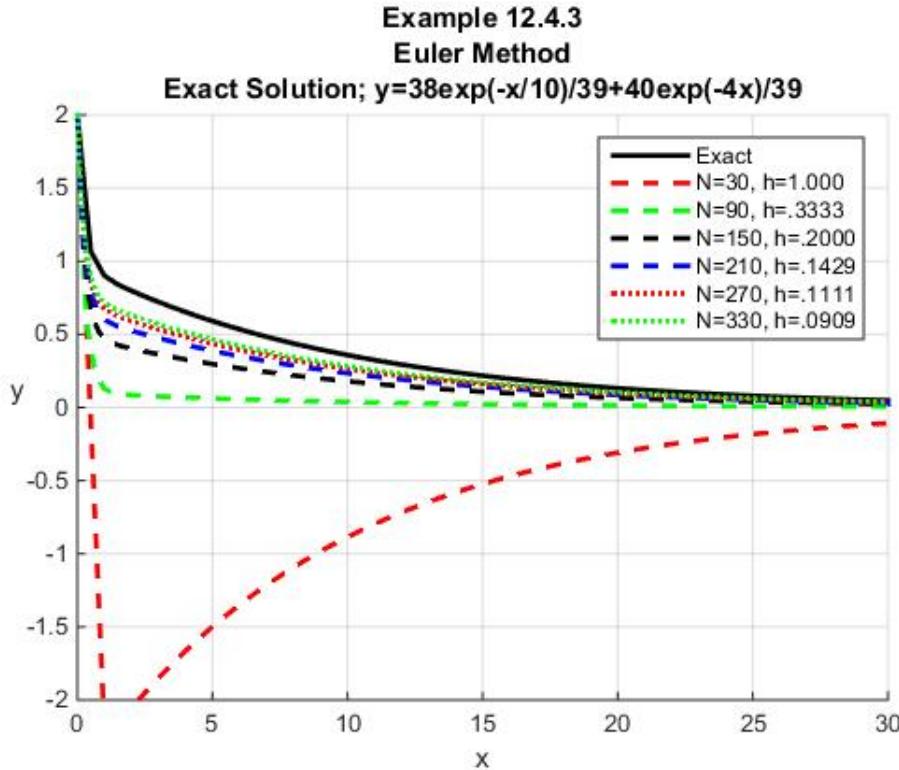
and we wish to utilize the Euler method to find the solution subject to the initial condition $y(0) = 2$. This elementary differential equation has the exact solution

$$y(x) = \frac{40}{39}e^{-4x} + \frac{38}{39}e^{-\frac{1}{10}x} \quad (12.4.17)$$

The simplicity of the exact solution reveals features that might complicate the numerical solution. If we think of the independent variable x as a time variable, the first term decays rapidly relative to the second term. A plot of each term superimposed on the full solution is



The blue line reflects an almost instantaneous decay to zero, while the red line takes substantially longer. When contemplating the problem of designing a numerical scheme that will create an approximation to the black line, it is evident that if the step size is too large, the blue line part of the solution would simply be missed by an iteration scheme based upon (12.4.4). It is suggested by the above figure that a step size of $h = 1$ would miss the rapid decay of the blue part of the solution. The following figure displays the exact solution along with six choices of step sizes:



This figure shows that the approximate solution is terrible for $N = 30$, $N = 90$ and not much better for the larger values of N . It is also interesting to record the time required to generate the approximate solutions as a function of the step size. If we utilize the `tic` and `toc` commands mentioned in Section 11.4. The various times turn out to be¹⁶

N	30	90	150	210	270	330
Time	0.0095 sec	0.0284 sec	0.0807 sec	0.1783 sec	0.2963 sec	0.5829 sec

While small numbers, the time required to obtain the best solution, for $N = 330$, is excessive for an elementary ordinary differential equation. An even more extreme example is in the case $N = 1100$ which consumed 7.3575 seconds.

The peculiar results illustrated by Example 12.4.3, reflects a significant deficiency of the Euler method and of fixed step size methods for a category of ordinary differential equations known as *stiff*. Whether or not a differential equation is stiff or not is not a precise concept. Roughly speaking, as Example 12.4.3 illustrates, a stiff ordinary differential equation is one that has multiple time scales (thinking of the independent variable x as time) of different orders of magnitude. In a numerical scheme, the step size must be small enough to “see” the fastest transient. In other words, the step size is controlled by the fastest transient. As such, the computational effort is necessarily greater which results in errors and costs. For nonlinear

¹⁶ As mentioned in Section 11.4, the time calculations utilizing `tic` and `toc` are dependent on the author’s computer and the version of MATLAB being utilized. The trends as reflected in the numbers calculated are not dependent on these choices.

ordinary differential equations the concept is even less precise. We shall have additional discussion of the concept of a stiff ordinary differential equation in Section 12.9.

Exercises

12.4.1: Utilize the approach utilized in Example 12.4.2 and determine the solution of

$$\frac{dy}{dx} = y^2 \cos(2x) \quad (12.4.18)$$

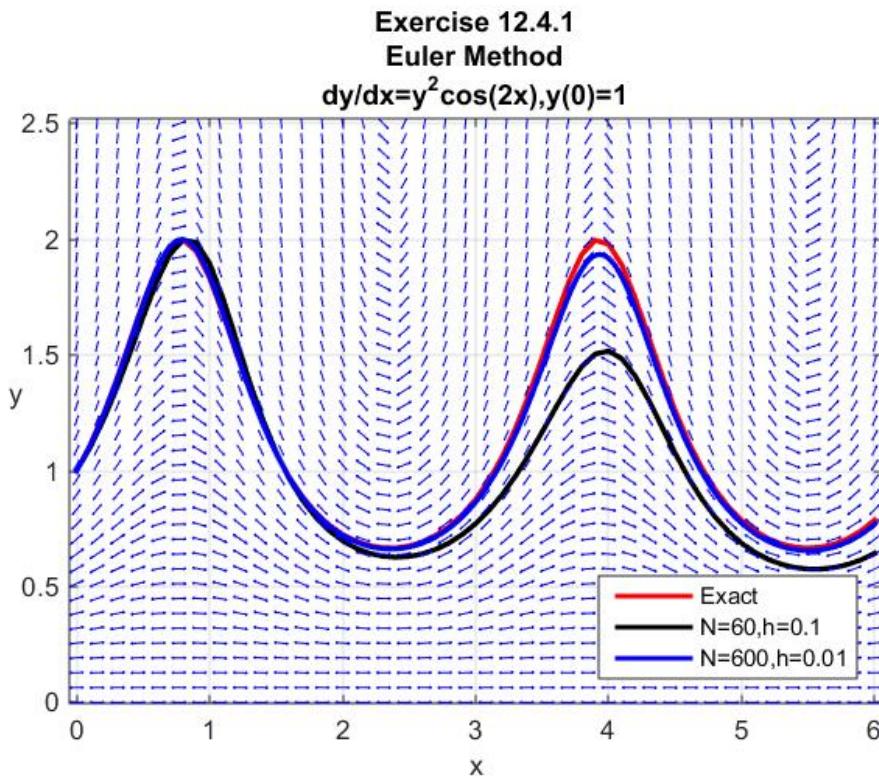
subject to the initial condition

$$y(0) = 1 \quad (12.4.19)$$

You are given that the exact solution for this initial value problem is

$$y(x) = \frac{2}{2 - \sin(2x)} \quad (12.4.20)$$

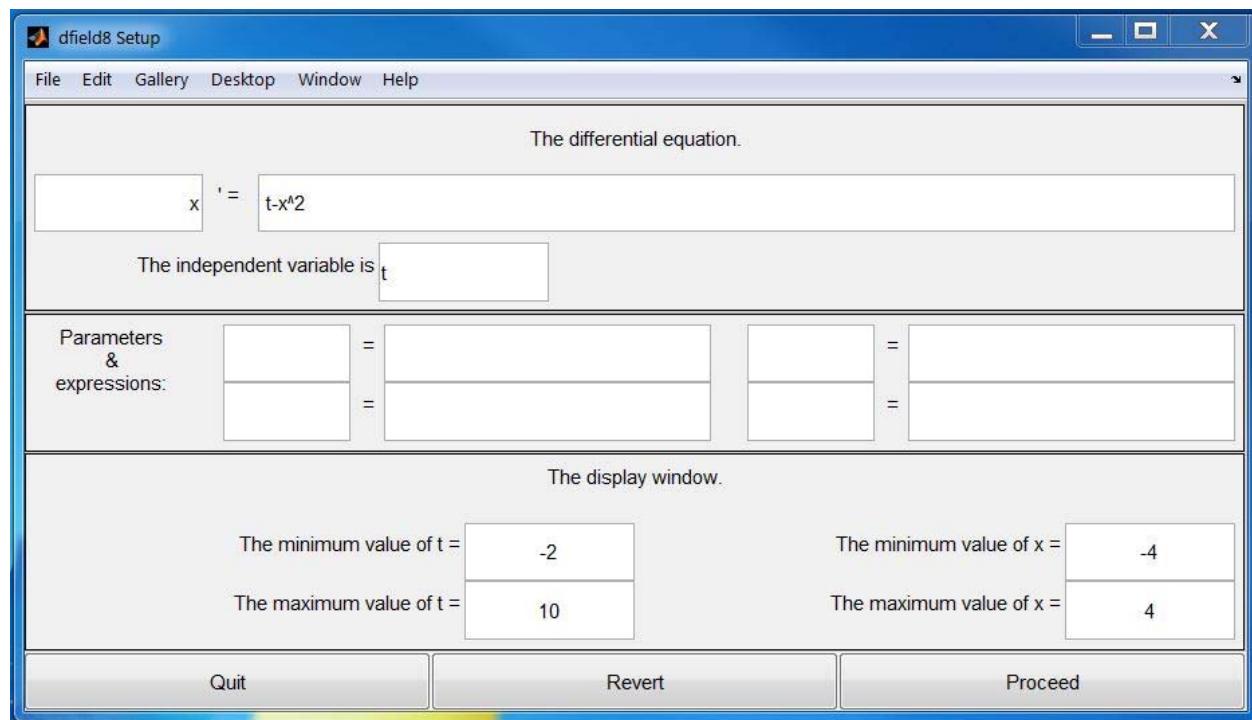
Plot the exact solution and the approximate solution based upon the Euler method for $N = 60$ and $N = 600$. The result should look something like



Section 12.5. MATLAB Implementations of the Euler Method

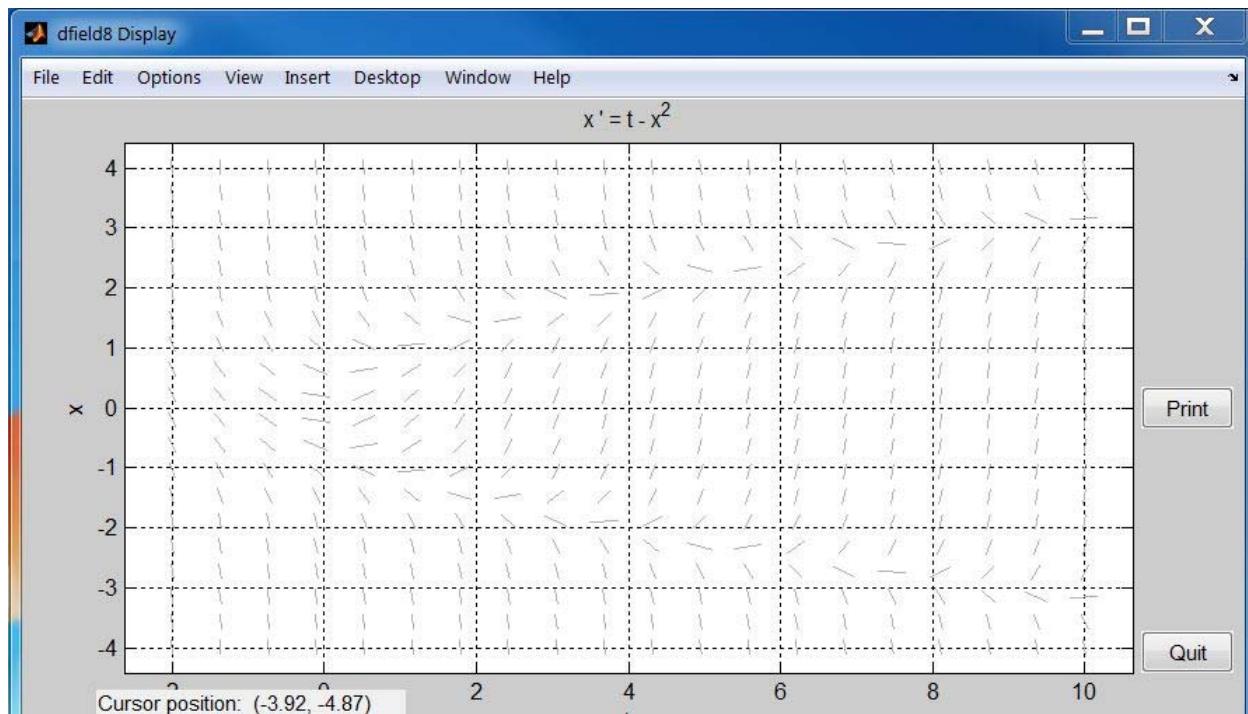
In this section, we shall look at function files that are useful when utilizing the Euler method. We are building up to a discussion of the built in solvers in MATLAB that utilize methods far superior than the Euler method. The utility of the discussion in this section is that the structure of the solution method is similar to that used for the MATLAB solvers.

Before we proceed with a discussion of function files that implement the Euler method, it is useful to mention a set of MATLAB tools that are available online. The website <http://math.rice.edu/~dfield/> contains a program **dfield8** which is a graphical user interface that generates solutions of ordinary differential equations.¹⁷ If the command **dfield8** is executed within MATLAB, the following graphical user interface is obtained.

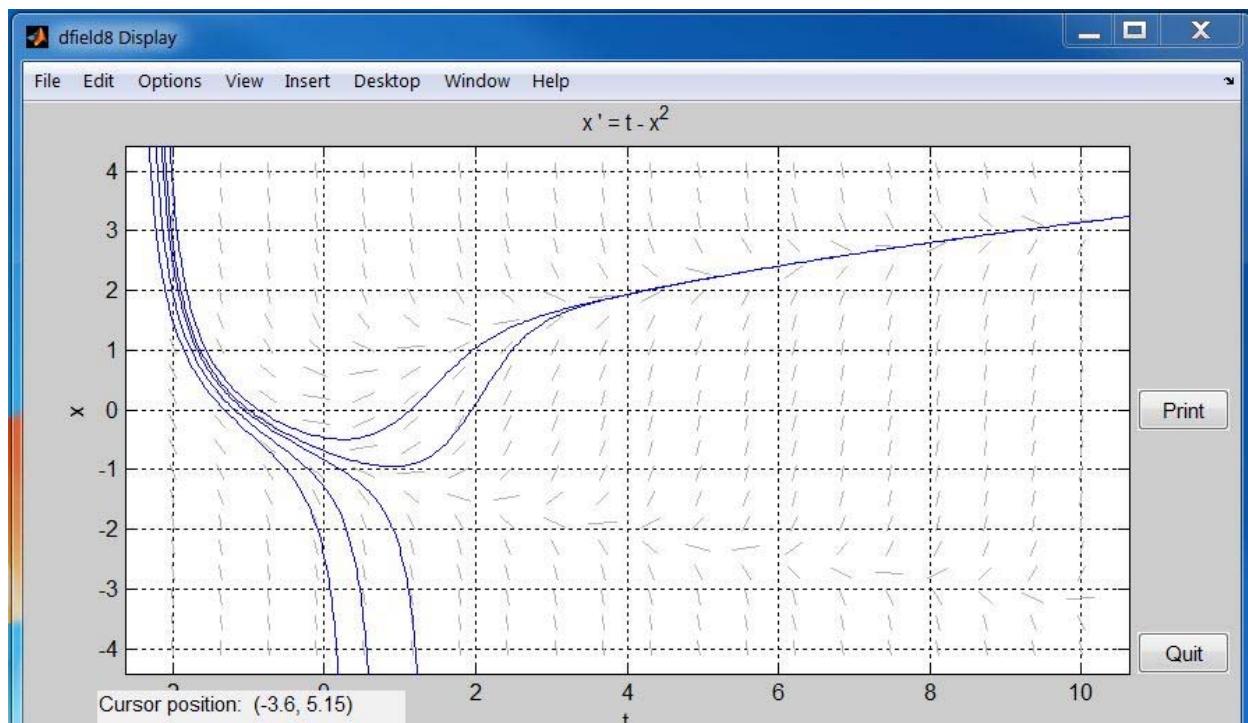


The information about the ordinary differential equation to be solved is entered in the proper slots above and one executes “Proceed” and the direction field associated with the ordinary differential equation is displayed. The differential equation shown in this case is the one discussed in Example 12.4.2 above. The result of executing the **Proceed** command is the graph:

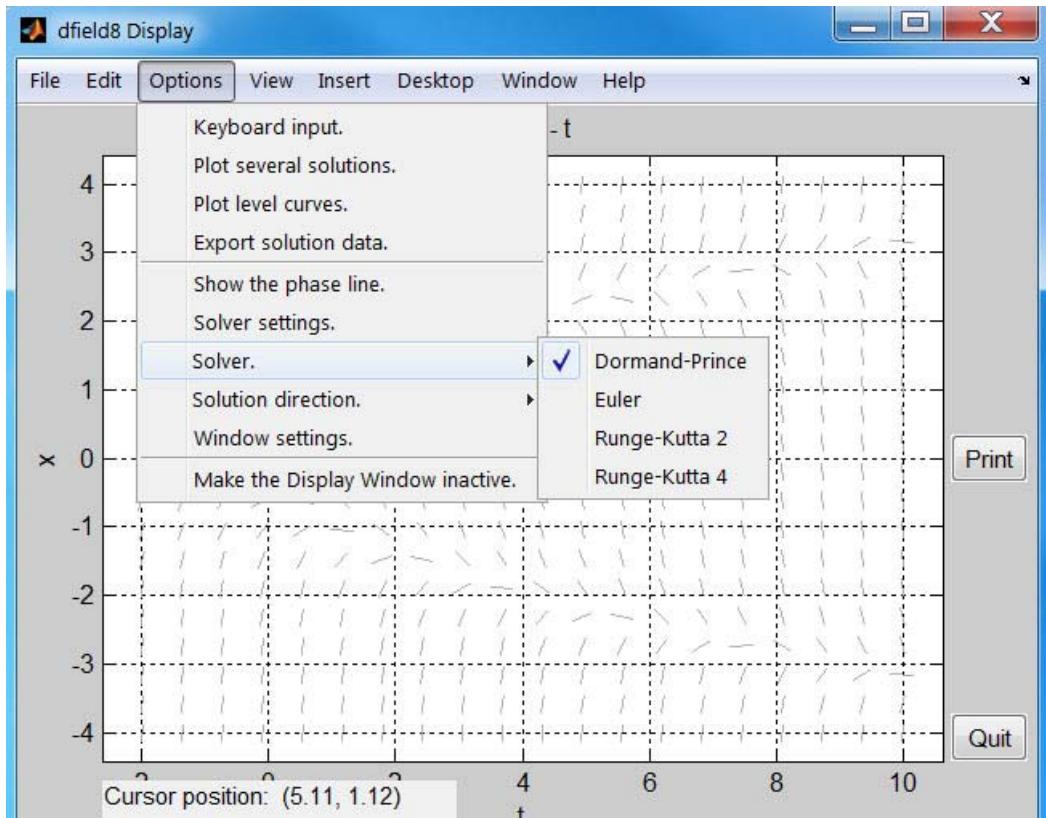
¹⁷ The **dfield8** program is a creation of Dr. John Polking of Rice University. This program and a related one, **pplane8** are described in detail in the textbook, Polking, John C., and David Arnold, *Ordinary Differential Equations using MATLAB, Third Edition*, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2004.



The **dfield8** interface allows one to generate solutions to initial value problems by simply clicking on the initial point (t_0, x_0) . The next figure illustrates several choices that in an approximate sense replicates those discussed in Example 12.4.2.



The computational scheme used to generate the above solutions is built into the **dfield8** tool. The following figure



reveals, within the menu structure of **dfield8**, the four solvers it will utilize. In addition to the Euler method, **dfield8** will use solvers that go by the names Dormand-Prince and Runge-Kutta. We shall discuss Runge-Kutta solvers in Sections 12.6 and 12.7.¹⁸

Our next objective is to introduce a MATLAB function file we shall call **euler357.m** that can be called in such a way that it produces an approximate solution based upon the Euler method. As indicated in the introduction to this Section, the idea is to create a function file that uses syntax similar to that of the several ODE solvers built into MATLAB. These solvers will be discussed in Section 12.8. Interestingly, an internet search will reveal an almost uncountable number of function files of the type we shall discuss. Most of these differ in trivial ways from each other. The one to be introduced here is certainly not original. In simplest of terms, it restructures the script used in Section 12.4 into a function file.

The script that defines the function file **euler357.m** is

```
function [x,y]=euler357(yprime,xspan,y0,ssize)
```

¹⁸ The website <http://math.rice.edu/~dfield/#8.0> contains links to download three function files eul.m, rk2.m and rk4.m that implement the Euler method and two versions of the Runge-Kutta method that we will discuss in Section 12.6.

```
% [x,y]=euler357(yprime,xspan,y0,ssize)
%           uses Euler's method to integrate an ODE
%input:
% yprime=name of the anonymous function, inline function
% or m-file that evaluates the ODE system of M equations
% xspan=[a,b] or [a,b]' where a and b = initial and
%         final values of the independent variable
% y0=M x 1 column vector of initial values of the
% dependent variable
% ssize=step size
%output:
% x=column vector of independent variable values
% y=M column matrix of solutions for dependent variables

if nargin<4,error('at least 4 input arguments required'),end
a=xspan(1);b=xspan(2);
if ~(b>a),error('upper limit must be greater than lower'),end
%Given ssize and xspan, calculate number of steps
N=floor((b-a)/ssize);
%Partition x to fit number of steps. Note, ssize is
%adjusted and named h
x=linspace(a,b,N+1)';
h=(b-a)/N
y=[y0,zeros(length(y0),N)]; %Preallocate
for n=1:N %implement Euler Method
    y(:,n+1)=y(:,n)+yprime(x(n),y(:,n))*h;
end
%Transpose y matrix
y=y'
```

As explained inside the script of the function file **euler357.m**, the syntax to call the approximate solution is

$$[x,y] = \text{euler357}(yprime, xspan, y0, ssize) \quad (12.5.1)$$

where **yprime** is the right hand of the ordinary differential equation, **xspan** is the *interval* of the independent variable

$$xspan = [a, b] \quad (12.5.2)$$

and **y0** is the initial condition. The script for **euler375.m** allows for the possibility that the ordinary differential equation to be solved is a first order *system of equations*. Thus, it will generate approximate solutions for problems more complicated than those discussed in Section 12.4. To accommodate this generalization and as explained in the above script, the initial condition **y0** is a column vector. The independent variable is defined by its end points as shown

by (12.5.2). The structure of **euler375.m** is such that **xspan** can be entered as a column vector or as the row vector (12.5.2). The *output* consists of a column vector of independent variables **x** and a matrix **y** of dependent variables. The number of rows of **y** equals the number of rows of **x**, and the number of columns of **y** corresponds to the number of independent variables, i.e., the number of first order ordinary differential equations being solved. Unlike the examples in Section 12.4, one of the inputs to **euler375.m** is the step size **ssize**. In Section 12.4 we would specify the number of intervals, **N**, and calculate the step size by the formula

$$\text{ssize} = (\mathbf{b}-\mathbf{a})/\mathbf{N} \quad (12.5.3)$$

Given the step size, **ssize**, equation (12.5.3) does not necessarily produce an integer for **N**. This problem has been avoided in the script for **euler375.m** by use of the **floor** function.¹⁹ The line of script **N=floor((b-a)/ssize)** calculates the value **(b-a)/ssize** and rounds the result downward to the nearest integer. Given this integer, the next line of script, **x=linspace(a,b,N+1)'**, creates a column vector whose values represent a partition of **[a,b]** into **N** intervals. The step size for this new partition will be close to but not always identical to **ssize**. It is possible to avoid this step by retaining the original **ssize** and adding, if necessary, an additional interval at the end to produce **xspan = [a,b]**.

It is instructive to work an example that utilizes **euler375.m**. In doing so, we shall illustrate three different approaches that MATLAB allows to create the function **yprime**, which defines the system of first ordinary differential equations. Consider the following example:

Example 12.5.1: This example seeks to find an approximate solution of the second order nonlinear ordinary differential equation

$$\frac{d^2y}{dt^2} + \frac{1}{4} \frac{dy}{dt} + \left(1 + \frac{y}{10}\right)y = 0 \quad (12.5.4)$$

in the interval $(0, 40)$ subject to the initial conditions

$$y(0) = 1 \quad \text{and} \quad \frac{dy(0)}{dt} = 0 \quad (12.5.5)$$

As illustrated in Section 12.1, the second order equation (12.5.4) can be replaced by the system of two first order nonlinear equations

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ -\frac{1}{4}x_2(t) - \left(1 + \frac{x_1}{10}\right)x_1(t) \end{bmatrix} \quad (12.5.6)$$

¹⁹ See http://www.mathworks.com/help/symbolic/mupad_ref/floor.html.

where

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} y(t) \\ \frac{dy(t)}{dt} \end{bmatrix} \quad (12.5.7)$$

Given the definition (12.5.7), the initial condition for the system (12.5.6) becomes

$$\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (12.5.8)$$

Solution 1: This method of solution involves creating a *function m-file* which we shall call **xprime.m**. This file contains the script

```
function dxdt=xprime(t,x)
dxdt=zeros(2,1); %Preallocate
dxdt=[x(2);-x(2)/4-(1+x(1))/10*x(1)];
```

The notation in **xprime.m** is intended to be suggestive. The factor **dxdt** could be given any symbol, as could the name of the function **xprime**. The second line of the script, **dxdt=zeros(2,1)**, preallocates memory for the array representing the column vector **dxdt**. Given the definition of the differential equation as in **xprime.m**, the script

```
clc
clear
a=0
b=60
tspan=[a,b]
x0=[1;0]
ssize=.2
[t,x]=euler357(@xprime,tspan,x0,ssize)
```

will generate the solution utilizing the Euler method. In this case, the script **[t,x]** will produce a three column matrix. The first column consists of the values **t**, the second column the values **x₁(t)** and the third column the values **x₂(t)**. In this case, the matrix **[t,x]** has size **301x3** and, of course, is too large to list here.

Solution 2: The next method of setting up this problem, in a rough sense, places the function file **xprime.m** inside of the same file as the one containing the script that defines **tspan**, **x0** and **ssize**.

The particular script is

```
function example1251
clc
```

```

clear
a=0
b=60
tspan=[a,b]
x0=[1;0]
ssize=.2
[t,x]=euler357(@xprime,tspan,x0,ssize)
%*****
function dxdt=xprime(t,x)
dxdt=zeros(2,1); %Preallocate
dxdt=[x(2);-x(2)/4-(1+x(1)/10)*x(1)]

```

The output from this script is the same as above. This particular structure uses an idea we have not discussed, namely, the idea of a function file *within* a function file. The structure of the “first function” is such that it has no inputs. It simply allows the function file to call the function **xprime**.

Solution 3: This method uses the idea of an anonymous function similar to those used in Section 12.4 to avoid the creation of the second function as in Solutions 1 and 2. The script is

```

clc
clear
a=0
b=60
tspan=[a,b]
x0=[1;0]
ssize=.2
xprime=@(t,x)([x(2);-x(2)/4-(1+x(1)/10)*x(1)])
[t,x]=euler357(xprime,tspan,x0,ssize)

```

In some sense, this method of solution is the most simple. However, there are ordinary differential equations that are difficult if not impossible to represent by the anonymous function as used above. It will be our practice to always use the approach outlined under Solution 1.

After one obtains the numerical solution, it is usually informative to plot the solution in order to better understand the predictions of the ordinary differential equation. The output in the form of the two matrices **t** and **x** can be plotted by methods that should now be familiar.

If we adopt the Solution 1 approach for Example 12.5.1, the script

```

clc
clear
a=0
b=60
tspan=[a,b]
x0=[1;0]
ssize=.1
[t,x]=euler357(@xprime,tspan,x0,ssize)

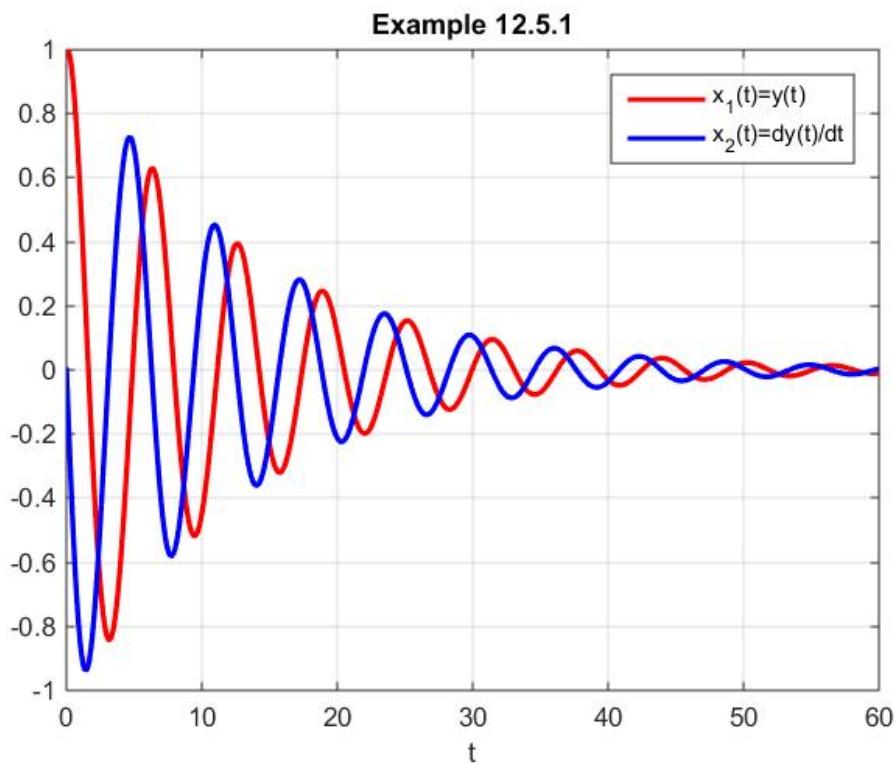
```

```

plot(t,x(:,1),'r','LineWidth',2)
hold on
plot(t,x(:,2),'b','LineWidth',2)
axis([0,60,-1,1])
grid on
xlabel('t')
legend('x_1(t)=y(t)', 'x_2(t)=dy(t)/dt',...
    'Location','NorthEast')
title('Example 12.5.1')

```

produces the figure



It is useful to use Example 12.5.1 to illustrate how MATLAB can be used to create a figure with two y axes, one for each of the two solutions. There are a couple of ways to create a two axis figure. The most convenient is to utilize the built in MATLAB function **plotyy**. If the above script is modified to the result

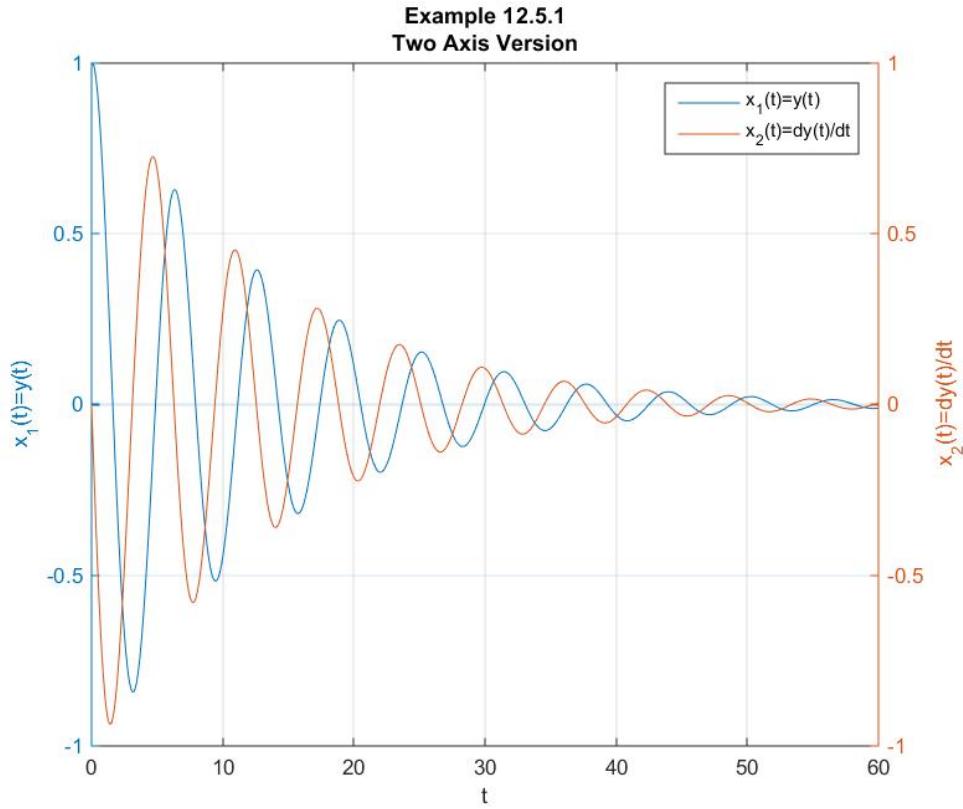
```

clc
clear
a=0
b=60
tspan=[a,b]
x0=[1;0]
ssize=.1

```

```
[t,x]=euler357(@xprime,tspan,x0,ssize)
[hAx,hLine1,hLine2]=plotyy(t,x(:,1),t,x(:,2))
ylabel(hAx(1),'x_1(t)=y(t)')
ylabel(hAx(2),'x_2(t)=dy(t)/dt')
xlabel('t')
grid on
legend([hLine1,hLine2], 'x_1(t)=y(t)', 'x_2(t)=dy(t)/dt', ...
'Location', 'NorthEast')
title({'Example 12.5.1', 'Two Axis Version'})
```

the resulting figure is

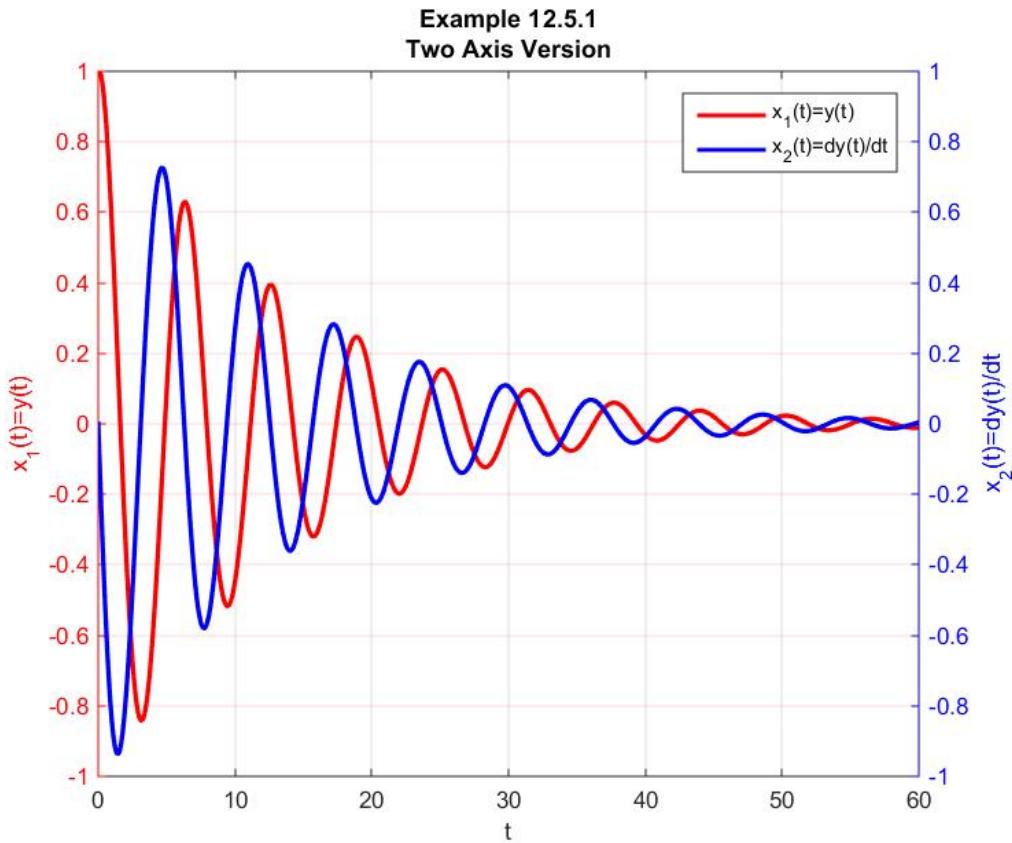


This figure utilizes the defaults of MATLAB R2014b for `plotyy`. For this reason, the colors and line widths are not the same as those for the first figure above. These defaults can be overridden by modifying the above script to be

```
clc
clear
a=0
b=60
tspan=[a,b]
x0=[1;0]
ssize=.1
```

```
[t,x]=euler357(@xprime,tspan,x0,ssize)
[hAx,hLine1,hLine2]=plotyy(t,x(:,1),t,x(:,2))
%Set line width to 2 for the two curves
hLine1.LineWidth=2
hLine2.LineWidth=2
%Set the line color for Line 1 to red
hLine1.Color=[1 0 0]
%Set the line color for Line 2 to blue
hLine2.Color=[0 0 1]
%Set the YTick values for the left axis to
%[-1,-0.8,-0.6,-0.4,-0.2,0,0.2,0.4,0.6,0.8,1]
hAx(1).YTick=[-1,-0.8,-0.6,-0.4,-0.2,0,0.2,0.4,0.6,0.8,1]
%Set the color of the left axis to red
hAx(1).YColor=[1 0 0]
%Set the YTick values for the right axis to
%[-1,-0.8,-0.6,-0.4,-0.2,0,0.2,0.4,0.6,0.8,1]
hAx(2).YTick=[-1,-0.8,-0.6,-0.4,-0.2,0,0.2,0.4,0.6,0.8,1]
%Set the color of the right axis to blue
hAx(2).YColor=[0 0 1]
%Enter in red the y axis label for the left axis
ylabel(hAx(1),'x_1(t)=y(t)','Color','red')
%Enter in blue the y axis label for the right axis
ylabel(hAx(2),'x_2(t)=dy(t)/dt','Color','blue')
xlabel('t')
grid on
legend([hLine1,hLine2],'x_1(t)=y(t)','x_2(t)=dy(t)/dt',...
    'Location','NorthEast')
title({'Example 12.5.1','Two Axis Version'})
```

This script modifies the last figure to the following one



The script makes ample use of what MATLAB calls *handle graphics*. The script

$$[hAx, hLine1, hLine2] = \text{plotyy}(t, x(:, 1), t, x(:, 2)) \quad (12.5.9)$$

creates the plot and assigns four handles to the figure. The two curves have handles **hLine1** and **hLine2**, respectively. The quantity **hAx** is a 1×2 array that represents the axes handles for the two axes of the figure. The role of these handles and how they relate to the line and axes properties is illustrated and briefly explained in the above script. If one executes the script

$$\text{get}(hAx(1)) \quad (12.5.10)$$

the output is a list of all properties of the graphics object represented by **hAx(1)**, the first axis of the figure. The same command applied to the handles **hAx(2)**, **hLine1** and **hLine2** gives lists of those graphics properties. The above script shows the syntax for changing the several properties that convert the next to last figure above to the last one above.²⁰

In the next section, we shall look at generalizations of the Euler method known as Runge-Kutta methods. We shall see from the details of these methods how the Euler's method solver

²⁰ The script utilizes MATLAB's *RGB triple* for specifying the colors. See <http://www.mathworks.com/help/matlab/ref/colors.html>.

euler357.m can be modified for these more general methods. Perhaps more importantly, we shall look at the solvers provided as a part of MATLAB and explain how they are used by the same kinds of script illustrated by **euler357.m**.

Section 12.6. Runge-Kutta Methods: Improved One Step Methods

In this Section, we shall build upon the discussion in Section 12.4. In that section, we discussed the Euler method for finding an approximate solution to the initial value problem for the single ordinary differential equation (12.4.1). A fundamental starting place of that discussion was the topic introduced in Section 12.3, namely, the direction field. The Euler method utilized the direction field in its most elementary fashion. The iteration equation (12.4.4), repeated,

$$y_{i+1} = y_i + f(x_i, y_i)h \quad (12.6.1)$$

utilizes the direction field at the point (x_i, y_i) to calculate the value of the solution $y(x)$ at a step away. The reality is that the value at a step away depends upon more information about the vector field than just the one at (x_i, y_i) . In a simple sense, the improvements of the Euler method, so which will be discussed in this section, involve utilizing more information about the vector field in a neighborhood of (x_i, y_i) . The Runge-Kutta methods we shall discuss are still one step methods, but, as we shall see, utilize in a selective way more properties of the vector field.²¹

The first step in our discussion is to replace the iteration formula (12.6.1) by

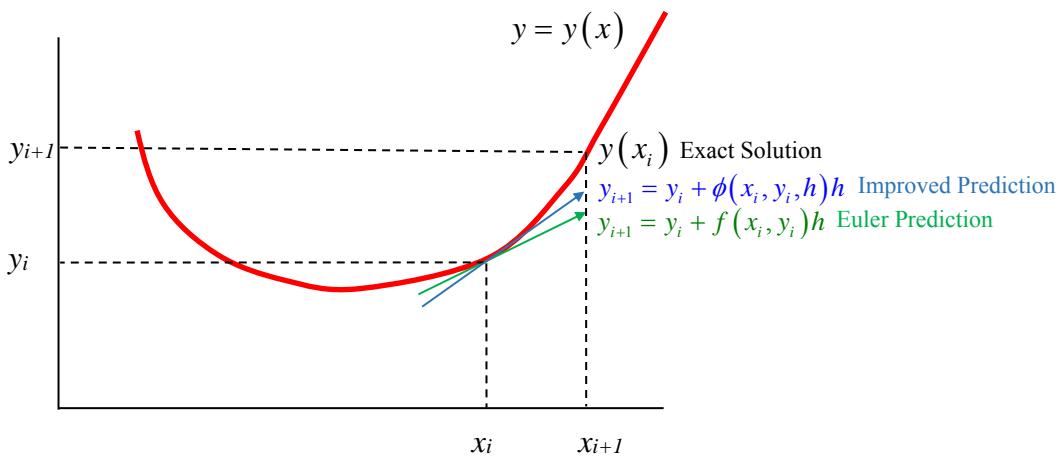
$$y_{i+1} = y_i + \phi h \quad (12.6.2)$$

where,

$$\phi = \phi(x_i, y_i, h) \quad (12.6.3)$$

as a generalization of the simple Euler method. By different choices of the function ϕ we can obtain a hierarchy of iteration schemes ranging from the Euler method just discussed to more general and more accurate schemes. Like the Euler method, the iteration based upon (12.6.3) tries to estimate the location of the solution curve *one step* forward in x . As a hoped for improvement of the Euler method, one can think of (12.6.3) as being characterized by the following figure:

²¹ Information about the German mathematicians Carl David Tolm   Runge and Martin Wilhelm Kutta can be found at http://en.wikipedia.org/wiki/Carl_David_Tolm%C3%A9_Runge and http://en.wikipedia.org/wiki/Martin_Wilhelm_Kutta, respectively.



As the figure suggests, an improved choice of the slope $\phi = \phi(x_i, y_i, h)$ will result in a prediction for the solution at x_{i+1} that is closer to the exact solution than the one predicted by the Euler iteration formula (12.6.1).

Of course, the fundamental purpose of the Runge-Kutta methods we are going to discuss is to improve the accuracy of the underlying iteration scheme that generates the approximate solution. Recall that the Euler method iteration scheme was obtained by converting the Taylor series representation (12.4.8), rewritten here,

$$\begin{aligned} y_{i+1} &= y_i + f(x_i, y_i)h + \underbrace{\frac{1}{2!}f'(x_i, y_i)h^2 + \cdots + \frac{1}{n!}h^n f^{(n-1)}(x_i, y_i)}_{\text{Truncation Error} = O(h^2)} + O(h^{n+1}) \\ &= y_i + f(x_i, y_i)h + O(h^2) \end{aligned} \quad (12.6.4)$$

into the iteration formula (12.6.1). The *improvement* over the Euler method leading to the iteration scheme in the form (12.6.2) is built upon our ability to replace (12.6.4)₂ with a formula like

$$y_{i+1} = y_i + \phi(x_i, y_i, h)h + O(h^n) \quad (12.6.5)$$

where $n > 2$. If the appropriate function $\phi(x_i, y_i, h)$ can be identified that obeys (12.6.5), then the associated iteration scheme (12.6.2) has improved the accuracy of the approximate solution.

We shall not do all of the analysis sufficient to justify the steps below. Essentially one selects the index n , and then proposes expressions for $\phi(x_i, y_i, h)$. Next, the proposed

expressions are forced to satisfy the condition (12.6.5). The details are, at best, tedious. We shall simply state some of the results.²²

The classification begins with the *Runge-Kutta nth order method*. In this case, the starting place is to look at expressions for $\phi(x_i, y_i, h)$ of the form

$$\phi(x_i, y_i, h) = a_1 k_1(x_i, y_i, h) + a_2 k_2(x_i, y_i, h) + \cdots + a_n k_n(x_i, y_i, h) \quad (12.6.6)$$

where a_1, a_2, \dots, a_n are *known constants*, and k_1, k_2, \dots, k_n are *values* of $f(x, y)$ calculated by the following *rules*:

$$\begin{aligned} k_1(x_i, y_i, h) &= f(x_i, y_i) \\ k_2(x_i, y_i, h) &= f(x_i + p_1 h, y_i + q_{11} k_1 h) \\ k_3(x_i, y_i, h) &= f(x_i + p_2 h, y_i + q_{21} k_1 h + q_{22} k_2 h) \\ &\vdots \\ k_n(x_i, y_i, h) &= f(x_i + p_{n-1} h, y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + \cdots + q_{n-1,n-1} k_{n-1} h) \end{aligned} \quad (12.6.7)$$

where the p 's and q 's are *known constants*. Note that k_2 depends upon k_1 , k_3 depends upon k_2 and k_1 and so forth. Equation (12.6.7), while formally complicated, constitutes a framework for classifying our various cases.

Runge-Kutta Hierarchy of Iteration Schemes

Case $n=1$

Euler's Method:

$$n=1 \quad \text{and} \quad a_1=1 \quad (12.6.8)$$

Case $n=2$

a. Heun Method²³

$$n=2 \quad \text{and} \quad y_{i+1} = y_i + \left(\frac{1}{2} k_1 + \frac{1}{2} k_2 \right) h \quad (12.6.9)$$

²² The analytics of the case $n=2$ are worked out in detail on page 703 of the textbook, Numerical Methods for Engineers, Fifth Edition, by Steven C. Chapra and Raymond R. Canale. This book was published by McGraw Hill.

²³ Information about the German mathematician Karl Heun can be found at http://en.wikipedia.org/wiki/Karl_Heun.

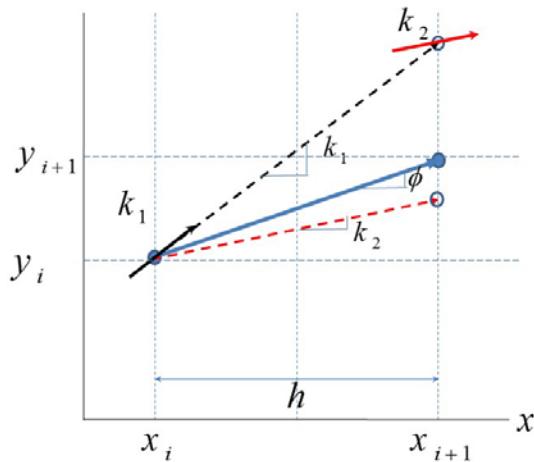
where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + h, y_i + k_1 h) \end{aligned} \quad (12.6.10)$$

If (12.6.10) is combined with (12.6.9)₂ the iteration formula for the *Heun method* is

$$y_{i+1} = y_i + \left(\frac{1}{2} f(x_i, y_i) + \frac{1}{2} f(x_i + h, y_i + f(x_i, y_i)h) \right) h \quad (12.6.11)$$

A graphical depiction of the Heun method is shown in the following figure:



$$k_1 = f(x_i, y_i) \quad k_2 = f(x_i + h, y_i + k_1 h)$$

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

The information on the above figure is a tangent at (x_i, y_i) whose slope is k_1 and a tangent at $(x_i + h, y_i + k_1 h)$ whose slope is k_2 . These two tangents are used to construct a tangent through (x_i, y_i) with slope $\phi = \frac{1}{2}(k_1 + k_2)$ that determines the value y_{i+1} from the formula (12.6.11).

Viewed as a process to utilize the direction field to determine the solution, the Heun method uses two of the elements of the field, the one at (x_i, y_i) and the one at $(x_i + h, y_i + k_1 h)$.

b. Midpoint Method: (Also known as the *second order Runge-Kutta Method*)

$$n=2 \quad \text{and} \quad y_{i+1} = y_i + k_2 h \quad (12.6.12)$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right) \end{aligned} \quad (12.6.13)$$

If (12.6.13) is combined with (12.6.12)₂ the result is

$$y_{i+1} = y_i + f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}f(x_i, y_i)\right)h \quad (12.6.14)$$

c. Ralston's Method:

$$n=2 \quad \text{and} \quad y_{i+1} = y_i + \left(\frac{1}{3}k_1 + \frac{2}{3}k_2\right)h \quad (12.6.15)$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{3}{4}h, y_i + \frac{3}{4}k_1h\right) \end{aligned} \quad (12.6.16)$$

If (12.6.16) is combined with (12.6.15) the result is

$$y_{i+1} = y_i + \left(\frac{1}{3}f(x_i, y_i) + \frac{2}{3}f\left(x_i + \frac{3}{4}h, y_i + \frac{3}{4}f(x_i, y_i)h\right)\right)h \quad (12.6.17)$$

Case $n=3$ (Known as the *third order* Runge-Kutta Method)

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 4k_2 + k_3)h \quad (12.6.18)$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h\right) \\ k_3 &= f(x_i + h, y_i - k_1h + 2k_2h) \end{aligned} \quad (12.6.19)$$

If (12.6.19) is combined with (12.6.18) the result is

$$y_{i+1} = y_i + \frac{1}{6} \left(f(x_i, y_i) + 4f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right) + f\left(x_i + h, y_i - f(x_i, y_i)h + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)\right) \right) h \quad (12.6.20)$$

Case $n = 4$ (Known as the *fourth order* Runge-Kutta Method. This method is one of the more widely used methods of solving nonlinear ordinary differential equations.)

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h \quad (12.6.21)$$

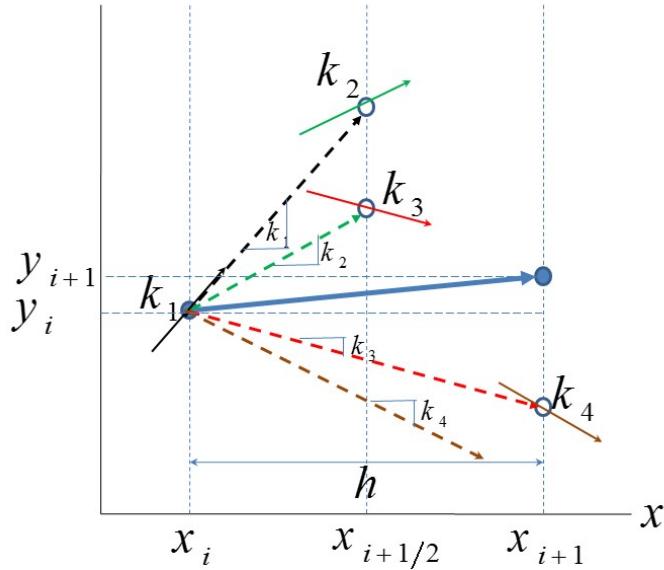
where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h\right) \\ k_3 &= f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2h\right) \\ k_4 &= f(x_i + h, y_i + k_3h) \end{aligned} \quad (12.6.22)$$

If (12.6.22) is combined with (12.6.21) the result is

$$\begin{aligned} y_{i+1} &= y_i + \frac{1}{6} \left(f(x_i, y_i) + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1h\right) + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2h\right) + f(x_i + h, y_i + k_3h) \right) h \\ &= y_i + \frac{1}{6} \left(f(x_i, y_i) + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right) + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)h\right) + f\left(x_i + h, y_i + f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)h\right)h\right) h \end{aligned} \quad (12.6.23)$$

A graphical depiction of the Fourth Order Runge Kutta Method is shown in the following figure:



$$k_1 = f(x_i, y_i)$$

$$k_2 = f\left(x_i + \frac{h}{2}, y_i + k_1 \frac{h}{2}\right)$$

$$k_3 = f\left(x_i + \frac{h}{2}, y_i + k_2 \frac{h}{2}\right)$$

$$k_4 = f(x_i + h, y_i + k_3 h)$$

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

The information on the above figure is a tangent at (x_i, y_i) whose slope is k_1 , a tangent at $\left(x_i + \frac{h}{2}, y_i + k_1 \frac{h}{2}\right)$ whose slope is k_2 , a tangent at $\left(x_i + \frac{h}{2}, y_i + k_2 \frac{h}{2}\right)$ whose slope is k_3 and a tangent at $(x_i + h, y_i + k_3 h)$ whose slope is k_4 . These four tangents are used to calculate the tangent through (x_i, y_i) with slope $\phi = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ that determines the value y_{i+1} from the formula (12.6.11). Viewed as a process to utilize the direction field to determine the solution, the fourth order Runge Kutta method uses four of the elements of the field.

Each of the iteration formulas (12.6.1), (12.6.11), (12.6.14), (12.6.17), (12.6.20) and (12.6.23) carries a truncation error that accumulates during the iteration. The truncation error associated with (12.6.1) was discussed in Section 12.4. Some sense of these errors can be obtained if we view the iteration formulas as arising from the numerical integration results we discussed in Section 11.11. As explained in Section 12.4, the Euler method iteration formula is obtained by approximating the integral (12.4.5), repeated,

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} f(x, y(x)) dx \quad (12.6.24)$$

by (12.4.6), repeated,

$$\int_{x_i}^{x_{i+1}} f(x, y(x)) dx \equiv f(x_i, y_i)(x_{i+1} - x_i) = f(x_i, y_i)h \quad (12.6.25)$$

Likewise, the Heun Method iteration formula (12.6.11) is obtained by approximating the integral in (12.6.24) by

$$\int_{x_i}^{x_{i+1}} f(x, y(x)) dx \equiv \left(\frac{1}{2} f(x_i, y_i) + \frac{1}{2} f(x_i + h, y_i + f(x_i, y_i)h) \right) h \quad (12.6.26)$$

The right side of (12.6.26) is like a *trapezoidal rule* estimate of the area. It is precisely a trapezoidal rule if f does not actually depend upon y .

The *Midpoint Method* (Second order Runge-Kutta Method) is obtained by approximating the integral in (12.6.24) by

$$\int_{x_i}^{x_{i+1}} f(x, y(x)) dx \equiv f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} f(x_i, y_i)\right) h \quad (12.6.27)$$

In this case, the area under the above curve is approximated by the area of a rectangle of height $f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} f(x_i, y_i)\right)$, the value at the *midpoint* of the interval, and of width h . The Ralston Method is obtained by approximating the integral in (12.6.24) by

$$\int_{x_i}^{x_{i+1}} f(x, y(x)) dx \equiv \left(\frac{1}{3} f(x_i, y_i) + \frac{2}{3} f\left(x_i + \frac{3}{4}h, y_i + \frac{3}{4}f(x_i, y_i)h\right) \right) h \quad (12.6.28)$$

This method arises from an optimization procedure which minimizes the bound on the truncation error for iteration schemes that fit the two point (i.e. $n = 2$) assumption.

The *third order* Runge-Kutta Method is obtained by approximating the integral in (12.6.24) by

$$\int_{x_i}^{x_{i+1}} f(x, y(x)) dx \equiv \frac{1}{6} \left(\begin{aligned} & f(x_i, y_i) + 4f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right) \\ & + f\left(x_i + h, y_i - f(x_i, y_i)h + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)\right) \end{aligned} \right) h \quad (12.6.29)$$

This formula is a Simpson 1/3 like rule for finding the area. If the function f did not depend upon x it would be precisely the Simpson 1/3 rule.

The *Fourth order* Runge-Kutta Method is obtained by approximating the integral in (12.6.24) by

$$\int_{x_i}^{x_{i+1}} f(x, y(x)) dx \approx \frac{1}{6} h \left[f(x_i, y_i) + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right) + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)h\right) + f\left(x_i + h, y_i + f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)h\right)h\right] \quad (12.6.30)$$

This formula is a Simpson 3/8 like rule for finding the area. If the function f did not depend upon y it would be precisely the Simpson 3/8 rule.

In Section 12.4, it was explained without proof that the global truncation error of the Euler method is $O(h)$. Also without proof the Huen method can be shown to have a truncation error of $O(h^2)$ as are the Midpoint method and the Ralston method. As the names suggest, the third order Runge-Kutta method has a global truncation error of $O(h^3)$, and the fourth order Runge-Kutta method is $O(h^4)$.

A summary of the various Runge-Kutta iteration formulas is given in the following table:

Runge-Kutta Method	n	$y_{i+1} = y_i + \phi(x_i, y_i, h)h$
Euler	1	$y_{i+1} = y_i + f(x_i, y_i)h$
Heun	2	$y_{i+1} = y_i + \left(\frac{1}{2}f(x_i, y_i) + \frac{1}{2}f(x_i + h, y_i + f(x_i, y_i)h)\right)h$
Midpoint	2	$y_{i+1} = y_i + f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}f(x_i, y_i)\right)h$
Ralston	2	$y_{i+1} = y_i + \left(\frac{1}{3}f(x_i, y_i) + \frac{2}{3}f\left(x_i + \frac{3}{4}h, y_i + \frac{3}{4}f(x_i, y_i)h\right)\right)h$

3 rd Order Runge- Kutta	3	$y_{i+1} = y_i + \frac{1}{6} \left(f(x_i, y_i) + 4f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right) + f\left(x_i + h, y_i - f(x_i, y_i)h + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)\right) \right)$
4 th Order Runge- Kutta	4	$y_{i+1} = y_i + \frac{1}{6} \left(f(x_i, y_i) + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right) + 2f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)h\right) + f\left(x_i + h, y_i + f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f\left(x_i + \frac{1}{2}h, y_i + \frac{1}{2}f(x_i, y_i)h\right)h\right)h\right)$

Section 12.7. MATLAB Implementations of Runge-Kutta Methods

In this section, we shall generalize the function m-file **euler357.m** to accommodate the Runge-Kutta generalizations discussed in Section 12.6. As in Section 12.5, we shall structure the generalization so that it uses syntax similar to that of the several ODE solvers built into MATLAB. These solvers will be discussed in Section 12.8. As with **euler357.m**, we shall take the opportunity to generalize the discussion so that the resulting function m-files hold for systems of ordinary differential equations.

Our approach will be to modify the script for **euler357.m** by replacing the part that implements the iteration defined by (12.6.1) by steps that implement the iterations defined by the various Runge-Kutta methods. The part of the script given in Section 12.5 that must be replaced is

```
for n=1:N %implement Euler Method
    y(:,n+1)=y(:,n)+yprime(x(n),y(:,n))*h;
end
```

Our first illustration will be to replace the above two lines of script for the appropriate script that defines the Heun method. The script that achieves that reflects the iteration (12.6.11) is

```
for n=1:N %implement Heun Method
    k1=yprime(x(n),y(:,n));
    k2=yprime(x(n)+h,y(:,n)+k1*h);
    y(:,n+1)=y(:,n)+(k1+k2)/2*h;
end
```

If this replacement is adopted for the file **euler357.m**, the result is the function m-file **heun357.m** with the script

```
function [x,y]=heun357(yprime,xspan,y0,ssize)
% [x,y]=heun357(yprime,xspan,y0,ssize)
%      uses Heun's method to integrate an ODE
%input:
% yprime=name of the anonymous function, inline function
%or
% m-file that evaluates the ODE system of M equations
% xspan=[a,b] or [a,b]' where a and b = initial and
%       final values of the independent variable
% y0=M x 1 column vector of initial values of the
%dependent variable
% ssize=step size
%output:
% x=column vector of independent variable values
% y=M column matrix of solutions for dependent variables
```

```

if nargin<4,error('at least 4 input arguments
required'),end
a=xspan(1);b=xspan(2);
if ~(b>a),error('upper limit must be greater than
lower'),end
%Given ssize and xspan, calculate number of steps
N=floor((b-a)/ssize);
%Partition x to fit number of steps. Note, ssize is
%adjusted and named h
x=linspace(a,b,N+1)';
h=(b-a)/N
y=[y0,zeros(length(y0),N)]; %Preallocate y to improve
%efficiency
for n=1:N %implement Heun Method
    k1=yprime(x(n),y(:,n));
    k2=yprime(x(n)+h,y(:,n)+k1*h);
    y(:,n+1)=y(:,n)+(k1+k2)/2*h;
end

%Transpose y matrix
y=y';

```

As an example utilizing that utilizes the function m-file **heun357.m**, it is instructive to consider again the system of ordinary differential equations introduced in Example 12.5.1.

Example 12.7.1: This example seeks again to find an approximate solution of the second order nonlinear ordinary differential equation

$$\frac{d^2y}{dt^2} + \frac{1}{4} \frac{dy}{dt} + \left(1 + \frac{y}{10}\right)y = 0 \quad (12.7.1)$$

in the interval $(0, 40)$ subject to the initial conditions

$$y(0) = 1 \quad \text{and} \quad \frac{dy(0)}{dt} = 0 \quad (12.7.2)$$

In this example we shall utilize the Heun method and compare the results to those obtained in Example 12.5.1 which utilized the Euler method. From Example 12.5.1, we know that the normal form of (12.7.1) is

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ -\frac{1}{4}x_2(t) - \left(1 + \frac{x_1}{10}\right)x_1(t) \end{bmatrix} \quad (12.7.3)$$

where

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} y(t) \\ \frac{dy(t)}{dt} \end{bmatrix} \quad (12.7.4)$$

Also, we know that the initial condition on the first order nonlinear system (12.7.3) is

$$\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (12.7.5)$$

The script that defines the differential equation (12.7.3) is again given in the function m-file introduced in Example 12.5.1, **xprime.m**. Recall that this file contains the script

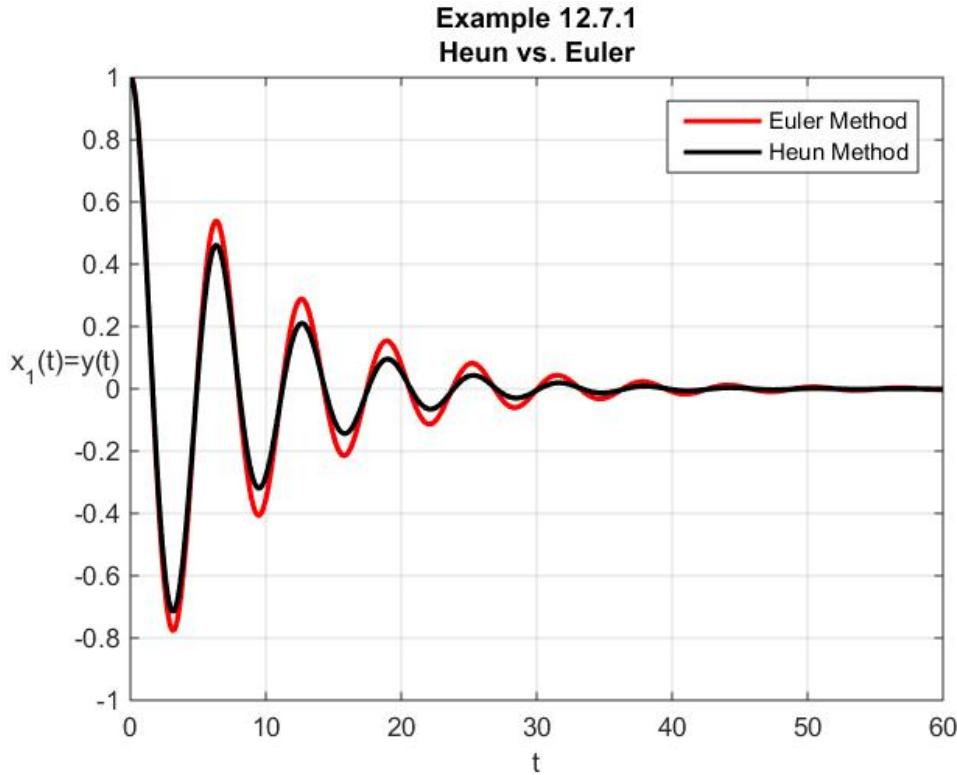
```
function dxdt=xprime(t,x)
dxdt=zeros(2,1); %Preallocate
dxdt=[x(2);-x(2)/4-(1+x(1)/10)*x(1)];
```

The following script generates a solution to the above initial value problem for both the Euler method and the Heun method. It also plots the results for the solution of the first dependent variable, $y(t) = x_1(t)$. A solution that also plots the second dependent variable $\frac{dy(t)}{dt} = x_2(t)$ can be found by generalizing the approach used in Example 12.5.1. The script for Example 12.7 is as follows:

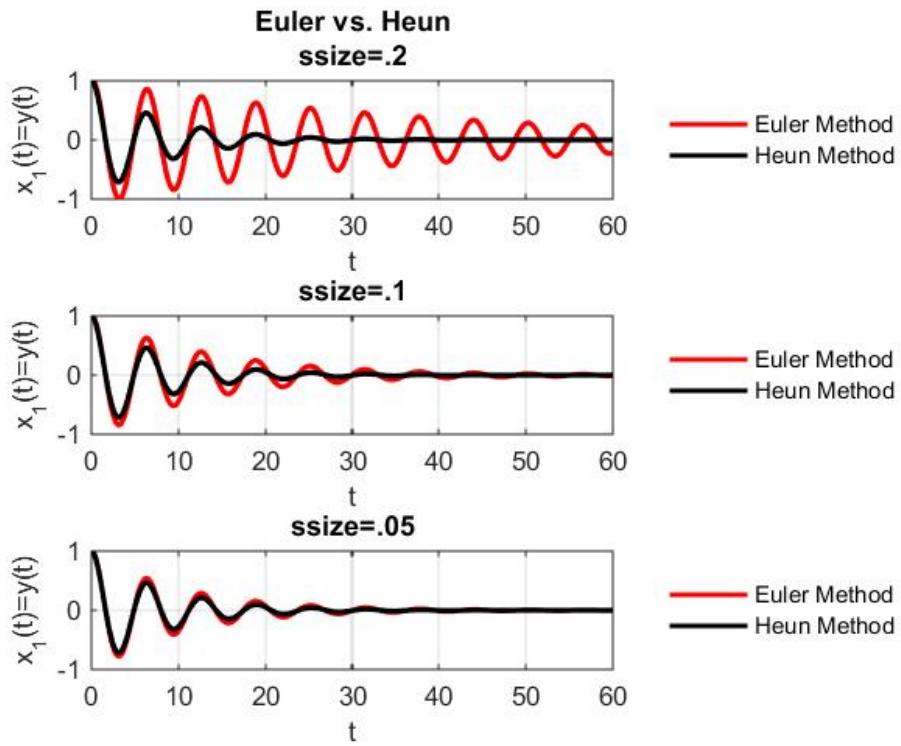
```
clc
clear
a=0
b=60
tspan=[a,b]
x0=[1;0]
ssize=.05
%Euler Method Solution
[t1,x1]=euler357(@xprime,tspan,x0,ssize)
plot(t1,x1(:,1),'r','LineWidth',2)
axis([0,60,-1,1])
xlabel('t')
ylabel('x_1(t)=y(t)', 'Rotation',0)
grid on
hold on
%Heun Method Solution
[t2,x2]=heun357(@xprime,tspan,x0,ssize)
plot(t2,x2(:,1),'k','LineWidth',2)
legend('Euler Method','Heun Method',...)
```

```
'Location','NorthEast')
title({'Example 12.7.1';'Heun vs. Euler'})
```

The resulting plot of the two solutions is



While we do not have an exact solution available for a comparison, at least this figure displays the significant differences in the results of the two methods. It is instructive to illustrate the dependence on step size for this problem and for these two methods. The following plot is the result of repeating the calculation leading to the above plot, except it utilizes three different step sizes.



In rough terms, this figure shows that the Euler method is more sensitive to step size than is the Heun method. In particular, if we can accept the accuracy of the Heun method as suggested by the discussion in Section 12.7, it becomes evident that the Euler method is not accurate until the step size becomes especially small.

Example 12.7.2: An example that illustrates the advantages of Runge-Kutta methods involves finding an approximate solution of the linear ordinary differential equation.

$$\frac{dy}{dx} + \frac{1}{2}y = 10e^{-400(x-2)^2} \quad (12.7.6)$$

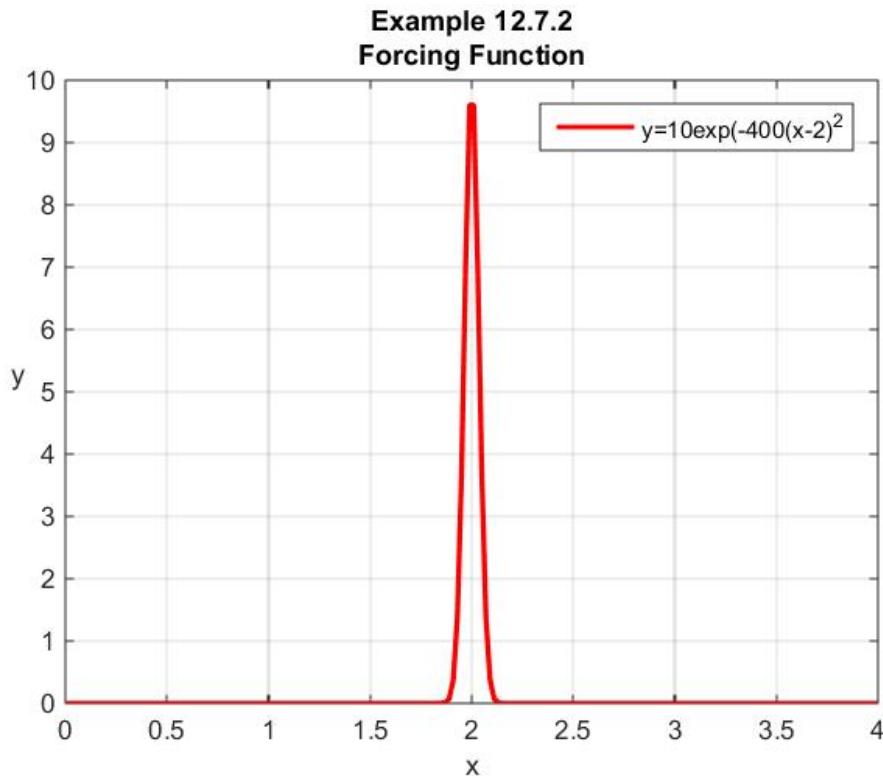
Subject to the initial condition

$$y(0) = 0 \quad (12.7.7)$$

The function

$$f(x) = 10e^{-400(x-2)^2} \quad (12.7.8)$$

has the plot



This example is one where the *rapid change of the forcing function* over a short range of the independent variable suggests that a numerical procedure could be a problem. We shall work this problem exactly and with the solvers **euler357** and **heun357**. We shall also plot the numerical solutions and the exact solution.

The first order linear ordinary differential equation (12.7.6) is one of the elementary forms studied in the first ordinary differential equation course. It is one that has an integrating factor.²⁴ In any case, the exact solution of (12.7.6) that obeys the initial condition (12.7.7) can be shown to be²⁵

²⁴ For ease of reference, it might help to note that the standard form of these kinds of equations is usually written $\frac{dy}{dx} + P(x)y = Q(x)$ and the solution, obtained by use of an integrating factor,

is $y(x) = e^{-\int P(x)dx} \int Q(x)e^{\int P(x)dx} dx + Ce^{-\int P(x)dx}$, where C is a constant of integration. Also, it is useful to note that this solution is a special case of the systems of ordinary differential equations discussed and solved in Section 6.5.

²⁵ If you would like MATLAB's **dsolve** command to do the integration, the script

```
clc
clear
sol=dsolve('Dy=-.5*y+10*exp(-400*(x-2)^2)', 'y(0)=0', 'x')
```

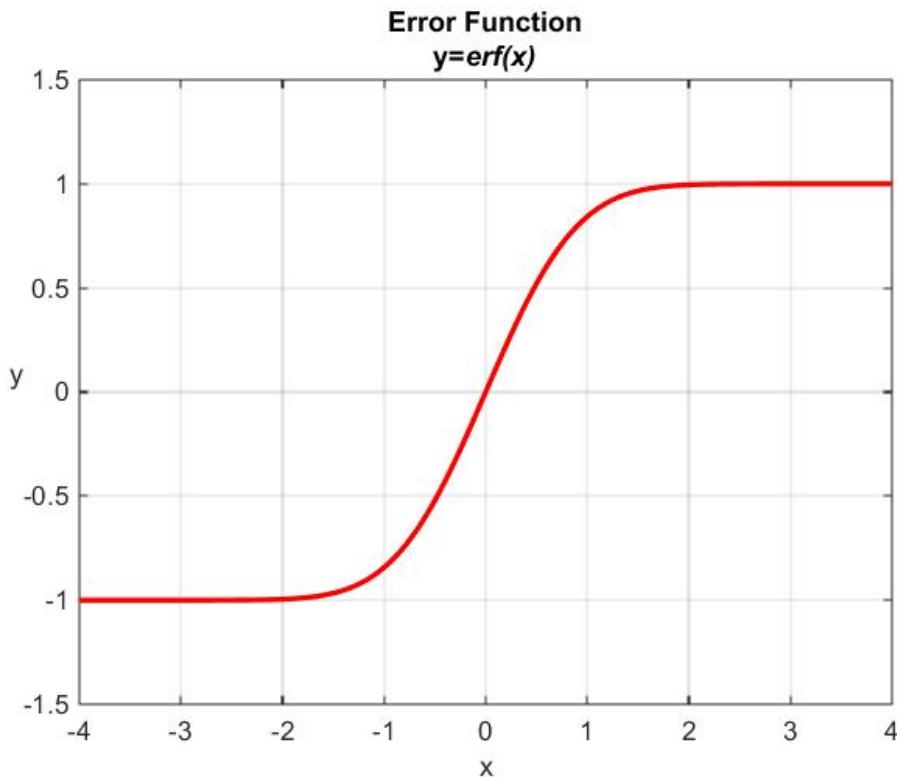
will produce a solution equivalent to (12.7.9)

$$y(x) = \frac{\sqrt{\pi}}{4} e^{\left(\frac{6401-3200x}{6400}\right)} \left(\operatorname{erf}\left(\frac{3201}{80}\right) - \operatorname{erf}\left(\frac{3201-1600x}{80}\right) \right) \quad (12.7.9)$$

where erf is the symbol for the *error function*. This function, which was introduced in Exercise 11.11.4, is defined by the equation

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-\eta^2} d\eta \quad (12.7.10)$$

This function, like most, is tabulated within MATLAB. If this function is utilized, it is easily shown that a plot of the error function is



The first step we must take is to create a function m-file that defines the ordinary differential equation (12.7.6). We shall call this file **f1272.m**. It contains the script

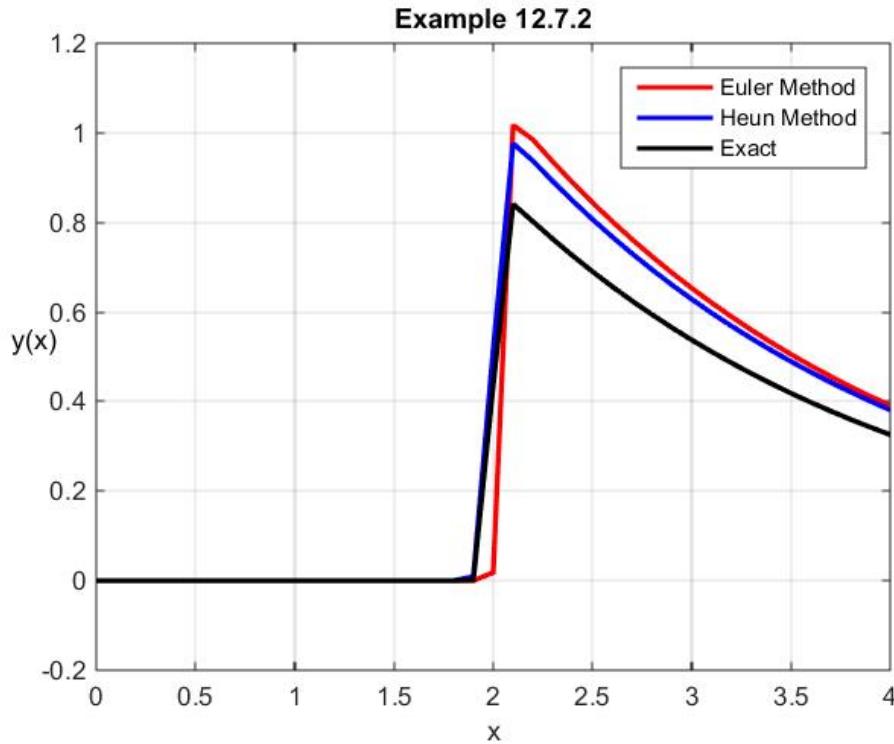
```
function dydx=f1272(x,y)
dydx=-.5*y+10*exp(-400*(x-2)^2);
```

Given this function file, the script

```
clc
clear
```

```
a=0
b=4
xspan=[a,b]
y0=0
ssize=.1
%Euler Method
[x1,y1]=euler357(@f1272,xspan,y0,ssize)
plot(x1,y1,'r','LineWidth',2)
axis([0,4,-.2,1.2])
xlabel('x')
ylabel('y(x)','Rotation',0)
grid on
hold on
%Heun Method
[x2,y2]=heun357(@f1272,xspan,y0,ssize)
plot(x2,y2,'b','LineWidth',2)
%Exact
xrange=a:.1:b
yexact=sqrt(pi)/4*exp((6401-
3200*xrange)/6400).*(erf(3201/80)-erf((3201-
1600*xrange)/80))
plot(xrange,yexact,'k','LineWidth',2)
title('Example 12.7.2')
legend('Euler Method','Heun Method','Exact')
```

produces the figure



As one would expect, for the same step size, the Heun method produces a solution that improves on the Euler method. However, neither is especially good when compared to the exact solution. The source of the problem is the rapid change of the forcing function in the neighborhood of the point $x = 2$. This kind of problem suggests the need for a solver that *adapts* the step function size to the particular forcing function.

A constant step size, as with the solvers discussed thus far, can either misrepresent the forcing function if the step function is too large. If the step size is too small, it can create an unnecessary computing burden. Rather than discuss adaptive solvers in detail, in Section 12.8 we shall take advantage of the built in solvers in MATLAB. As we shall see, these solvers have address a variety of features not present in the simple solvers we have discussed.

Exercises

12.7.1: Utilize MATLAB to find an approximate solution to the initial value problem

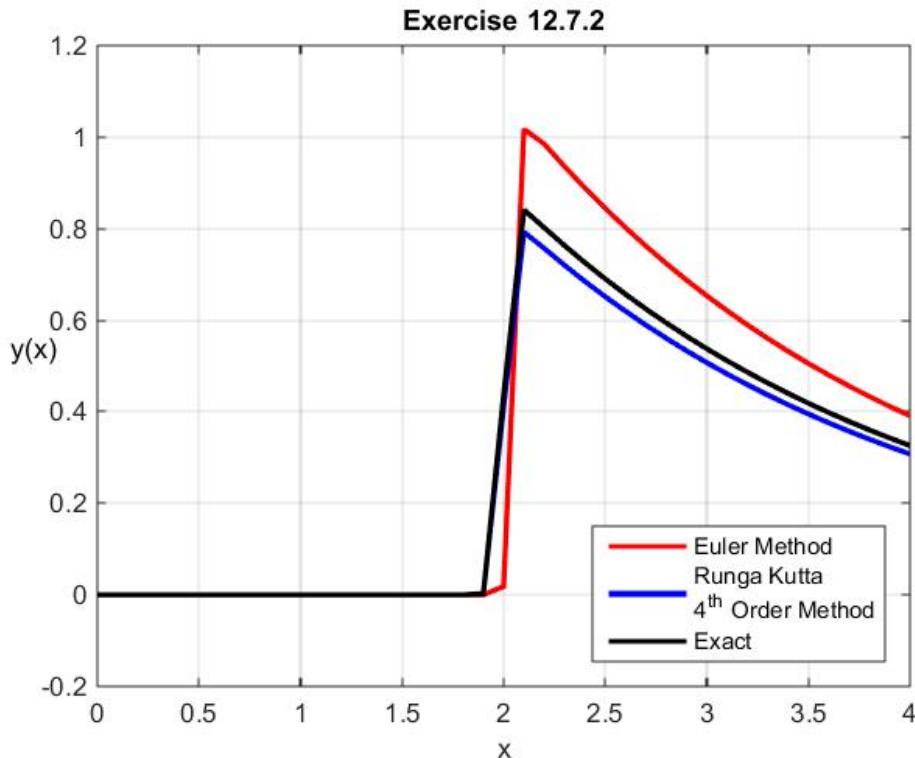
$$\frac{dy}{dx} = yx^2 - 1.1y^2 \quad y(0) = 1 \quad (12.7.11)$$

on the interval $x \in [0, 2]$. Utilize a solver that implements the third order Runge Kutta method. Create this solver by making the appropriate changes to either the solver **euler357** or the solver **heuen357** discussed in Section 12.7.

12.7.2: Rework Example 12.7.2 except utilize a solver that implements the fourth order Runge-Kutta method.²⁶ A solver that simply modifies **euler357.m** by replacing the lines

```
for n=1:N %implement Euler Method
    y(:,n+1)=y(:,n)+yprime(x(n),y(:,n))*h;
end
```

with the appropriate script for the fourth order Runge-Kutta method should produce the figure²⁷



²⁶ The website <http://math.rice.edu/~dfield/> has available a second order Runge-Kutta solver called **rk2.m** and a fourth order Runge-Kutta solver called **rk4.m**.

²⁷ Note that the figure legend has one entry printed on two lines. This format was utilized because of the length of this entry. An online search will display a few ways to cause MATLAB to print in this way. The script that is sufficient is

```
legend({'Euler Method','[Runge Kutta' char(10) '4^{th} Order Method'],'Exact'},...
'Location','SouthEast')
```

Another way to achieve the same result is the script

```
legend({'Euler Method',sprintf('Runge Kutta \n4^{th} Order Method'),'Exact'},...
'Location','SouthEast')
```

Information about **sprintf** can be found at <http://www.mathworks.com/help/matlab/ref/sprintf.html>.

This figure, as compared to the corresponding one for Example 12.7.2, displays the additional accuracy of the fourth order Runge-Kutta method relative to the Euler method and the Heun method. However, it still has accuracy problems resulting from the restriction fixed step size.

Section 12.8. MATLAB ODE Solvers

MATLAB has several built in ODE solvers. Each has specific features that make them useful for specific kinds of systems of ordinary differential equations. They fall into two main categories, *Stiff* and *Nonstiff*. We briefly mentioned stiff ordinary differential equations in our discussion of Example 12.4.3. We shall give additional discussion of this characteristic of a system of ordinary differential equation in Section 12.9.

The following list, which is essentially from the extensive information available online and from the MATLAB **help** command, provides a summary of the MATLAB ODE solvers:²⁸

Nonstiff Solvers:²⁹

- ode45** Based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair.³⁰ It is a *one-step* solver. Therefore, the calculation of $y(x_n)$ only needs the solution at the immediately preceding point, $y(x_{n-1})$. In general, **ode45** is the best function to apply as a "first try" for most problems.
- ode23** Based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine.³¹ It may be more efficient than **ode45** at crude tolerances and in the presence of mild stiffness. Like **ode45**, **ode23** is a one-step solver.
- ode113** Variable order Adams-Basforth-Moulton PECE solver.³² It may be more efficient than **ode45** at stringent tolerances and when the ODE function is particularly expensive to evaluate. **ode113** is a *multistep* solver - it normally needs the solutions at several preceding points to compute the current solution.

Stiff Solvers

²⁸ The list of solvers omits **ode15i**. This solver, which is discussed in the MATLAB documentation, is useful for finding the solution to ordinary differential equations that cannot be solved explicitly for $\frac{dy}{dx}$ and, thus, cannot be written in the form (12.1.8). An example of an implicit ordinary differential equation that is often given is the

$$\text{Weissinger equation, } xy^2 \left(\frac{dy}{dx} \right)^3 - y^3 \left(\frac{dy}{dx} \right)^2 + x(x^2 + 1) \frac{dy}{dx} - x^2 y = 0.$$

²⁹ The source of this summary is Chapter 7 of Moler, Cleve, Numerical Computing with MATLAB, SIAM, Philadelphia, 2004. The electronic edition is at <http://www.mathworks.com/moler>. See also <http://www.mathworks.com/help/matlab/math/ordinary-differential-equations.html>.

³⁰ A discussion of the Dormand-Prince method and its relationship to the Runge-Kutta series of methods can be found at http://en.wikipedia.org/wiki/Dormand%20%93Prince_method.

³¹ A discussion of the Bogacki-Shampine method and its relationship to the Runge-Kutta series of methods can be found at http://en.wikipedia.org/wiki/Bogacki%20%93Shampine_method.

³² Additional information about multistep solvers can be found at http://en.wikipedia.org/wiki/Linear_multistep_method.

- ode15s** Variable-order solver based on the numerical differentiation formulas (NDFs). Optionally it uses the backward differentiation formulas, BDFs, (also known as Gear's method). Like **ode113**, **ode15s** is a multistep solver. If you suspect that a problem is stiff or if **ode45** failed or was very inefficient, try **ode15s**.
- ode23s** Based on a modified Rosenbrock formula of order 2.³³ Because it is a one-step solver, it may be more efficient than **ode15s** at crude tolerances. It can solve some kinds of stiff problems for which **ode15s** is not effective.
- ode23t** An implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.
- ode23tb** An implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order 2. Like **ode23s**, this solver may be more efficient than **ode15s** at crude tolerances.

As we shall see, the syntax for these seven solvers are *all essentially the same*, so we need not worry about learning each case. There are *minor differences* that will be illustrated in our various examples. As mentioned in Sections 12.5 and 12.7, we adopted syntax for the solvers **euler357** and **heun357** that would help later when the MATLAB solvers were discussed.

The MATLAB ode solvers are applied in the context of finding an approximate solution to the initial value problem for the system of M first order ordinary differential equations

$$\frac{dy}{dx} = \mathbf{f}(x, \mathbf{y}) \quad \text{where } \mathbf{y}(x_0) = \mathbf{y}_0 \quad (12.8.1)$$

If we adopt the solver **ode45** to illustrate the syntax, the online MATLAB documentation gives the following:

Syntax:

$$[\mathbf{x}, \mathbf{y}] = \text{ode45}(\mathbf{yprime}, \mathbf{xspan}, \mathbf{y0}) \quad (12.8.2)$$

As with the solvers **euler357** and **heun357**, the arguments are

```
[x,y]=vector consisting of column vector x of evaluation
points of the independent variable and y=solution array in
the form of a M column matrix of solutions for the
dependent variables. Each row of the matrix y corresponds
to the solution for the corresponding row of x.
```

³³ Information about Rosenbrock methods can be found at http://en.wikipedia.org/wiki/Rosenbrock_methods.

yprime=name of the function handle (anonymous function, inline function or m-file) that evaluates the M first order ODE system $\frac{dy}{dx} = f(x, y)$

xspan=[**x0**,**xf**] or [**x0**,**xf**]' where **x0** and **xf** = initial and final values of the independent variable. The solver imposes the initial conditions at **x0** and integrates from **x0** to **xf**. If **xspan** is written [**x0**,**x1**,**x2**,...,**xf**], where **xf**>...>**x2**>**x1**>**x0**, the rows of the solution output **y** give the solution at the points **x0**,**x1**,**x2**,...,**xf**.

Notice that, unlike our solvers **euler357** and **heun357**, the MATLAB solvers *do not* require that the step size **ssize** be prescribed. The MATLAB solvers are *adaptive solvers*. In rough terms, the MATLAB solvers take a step, estimate the error at that step, check to see if the value is greater than or less than a prescribed tolerance and, if necessary, adjust the step size and repeat the calculation.

Each MATLAB solver has certain default integration properties. These properties can be adjusted by specification of **options** by use of the syntax

$$[x, y] = \text{ode45}(yprime, xspan, y0, options) \quad (12.8.3)$$

The **options** are specified by a function named **odeset**.³⁴ Later we shall see examples of how **options** are utilized. If the command **odeset** is executed at the MATLAB command prompt, the following result is obtained:

```
AbsTol: [ positive scalar or vector {1e-6} ]
          RelTol: [ positive scalar {1e-3} ]
          NormControl: [ on | {off} ]
          NonNegative: [ vector of integers ]
          OutputFcn: [ function_handle ]
          OutputSel: [ vector of integers ]
          Refine: [ positive integer ]
          Stats: [ on | {off} ]
InitialStep: [ positive scalar ]
          MaxStep: [ positive scalar ]
          BDF: [ on | {off} ]
          MaxOrder: [ 1 | 2 | 3 | 4 | {5} ]
          Jacobian: [ matrix | function_handle ]
          JPatten: [ sparse matrix ]
```

³⁴ As usual, the MATLAB documentation gives a good discussion of **odeset**. An excellent discussion can also be found in the textbook, Polking, John C., and David Arnold, Ordinary Differential Equations using MATLAB, Third Edition, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2004.

```

Vectorized: [ on | {off} ]
    Mass: [ matrix | function_handle ]
MStateDependence: [ none | {weak} | strong ]
    MvPattern: [ sparse matrix ]
MassSingular: [ yes | no | {maybe} ]
InitialSlope: [ vector ]
Events: [ function_handle ]

```

This output lists the many options that can be controlled by **options**. It also shows the *defaults* that are adopted when **options** are not prescribed.

Among these defaults are two parameters that prescribe the *tolerances* associated with the solvers. They are listed above as **RelTol** and **AbsTol**. As we shall briefly explain, these two parameters combine to tell the solver when a prescribed accuracy has been obtained in a given step, and that it can proceed to the next step. In cases where a greater accuracy than the defaults is required, the **odeset** command can be used to change the tolerances.

The quantity **RelTol** is the *Relative error tolerance*. The default value is

$$\mathbf{RelTol} = 10^{-3} \quad (12.8.4)$$

The quantity **AbsTol** is the *Absolute error tolerance*. This quantity is a **1xM** dimensional vector. Recall that **M** is an integer that specifies the number of equations in the system being solved, i.e., the number of dependent variables. The default **AbsTol** is a **1xM** dimensional vector with the value 10^{-6} for each of its components. Thus, the default **AbsTol** is

$$\mathbf{AbsTol} = 10^{-6} \underbrace{[1, 1, \dots, 1]}_{1 \times M} \quad (12.8.5)$$

The structure of **AbsTol** allows a different value to be assigned for each of the ordinary differential equations being solved.

If $\mathbf{y}(j, k)$ is a calculated value of the solution array at the j^{th} evaluation point of \mathbf{x} for the k^{th} unknown, the question is whether or not this number is sufficiently accurate to accept it and move on to another evaluation point. Or, if not, adjust the step size and repeat the calculation.

The internal structure of the particular MATLAB solver, such as **ode45**, produces a number which measures the error in $\mathbf{y}(j, k)$ at $\mathbf{x}(j)$. We shall denote this error by $\mathbf{e}(j, k)$.³⁵ MATLAB utilizes the numerical values of **RelTol** and **AbsTol** to determine whether or not

³⁵ The discussion of the Dormand-Prince method, which is utilized in **ode45**, at http://en.wikipedia.org/wiki/Dormand%20%93Prince_method explains how it calculates with 4th order and 5th order Runge-Kutta methods and utilizes these results to identify the error.

this error is sufficiently accurate to move on to another evaluation point or if the step size should be adjusted and the calculation repeated. The accuracy of the solution is sufficient to move on if

$$\text{abs}(\mathbf{e}(j,k)) \leq \max(\text{abs}(\mathbf{y}(j,k)) * \text{RelTol}, \text{AbsTol}(k)) \quad (12.8.6)$$

The right side of (12.8.6) is MATLAB's definition of the **tolerance**.

An equivalent version of (12.8.6) is

$$\text{abs}(\mathbf{e}(j,k)) \leq \begin{cases} \text{AbsTol}(k) & \text{if } \text{abs}(\mathbf{y}(j,k)) \leq \frac{\text{AbsTol}(k)}{\text{RelTol}} \\ \text{abs}(\mathbf{y}(j,k)) * \text{RelTol} & \text{otherwise} \end{cases} \quad (12.8.7)$$

Equation (12.8.7) shows that the relative tolerance controls the tolerance unless

$$\text{abs}(\mathbf{y}(j,k)) \leq \frac{\text{AbsTol}(k)}{\text{RelTol}}.$$

We shall see examples in the following where the default values of **RelTol** and **AbsTol** will be altered by use of **odeset**.

A feature of the MATLAB solvers that will prove useful in the following is that the syntax

$$\text{ode45}(\mathbf{yprime}, \mathbf{xspan}, \mathbf{y0}, \text{options}) \quad (12.8.8)$$

will automatically plot the solution **y** vs. the independent variable **x**. This feature is convenient in cases where the numerical values of **y** and **x** are not needed. Example 12.9.1 below is such an example.

Finally, we shall see examples where it is convenient to solve the initial value problem (12.8.1) in several cases corresponding to different parameters specified in the definition of the first order system. If these parameters are specified in the function handle **yprime**, the specific values must be passed to the workspace of **yprime**. This step is achieved by the syntax

$$[\mathbf{x}, \mathbf{y}] = \text{ode45}(\mathbf{yprime}, \mathbf{xspan}, \mathbf{y0}, \text{options}, \mathbf{p1}, \mathbf{p2}, \dots, \mathbf{pn}) \quad (12.8.9)$$

where **p1, p2, ..., pn** are the numerical values of the parameters. If the problem is to be worked without specifying **options**, but it does require that parameters be specified, then the syntax (12.8.9) is replaced by

$$[\mathbf{x}, \mathbf{y}] = \text{ode45}(\mathbf{yprime}, \mathbf{xspan}, \mathbf{y0}, [], \mathbf{p1}, \mathbf{p2}, \dots, \mathbf{pn}) \quad (12.8.10)$$

where the empty matrix `[]` serves as a placeholder for the default **options**. The need for a placeholder first occurred in Section 9.3 in our discussion of the function m-files **bisect.m** and **falspos.m**.

In the examples below and later in this Chapter, we shall see specific examples that utilize the syntax discussed here. In these examples and in the related exercises, we will usually use the **ode45** and **ode15s** solvers. We shall also encounter examples where other features of the solvers are illustrated.

Example 12.8.1: It is instructive to rework Example 12.7.2 and include the results for the solver **ode45**. In this way, we can get determine how **ode45** yields an improved approximation to the exact solution (12.7.9).

As mentioned above, an important feature of the **ode45** syntax is that we *do not* specify a step size **ssize**. The solvers provided by MATLAB utilize a variety of variable step size methods. As explained, the MATLAB adaptive solves compute a solution for a step, estimate the error at that step, check to see if the value is greater than or less than a prescribed tolerance and, if necessary, adjust the step size and repeat the calculation. As Example 12.8.1, we will illustrate how the adaptive feature of **ode45** produces a solution superior to those generated in Example 12.7.2.

If the above script for Example 12.7.2 is modified by including an approximation based upon the solver **ode45** the result is

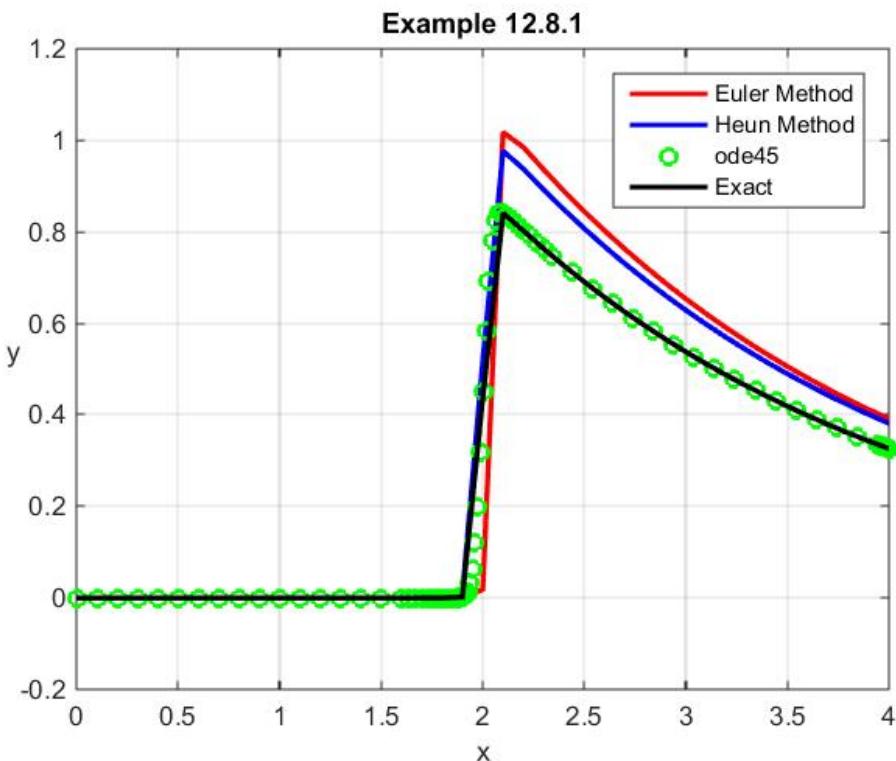
```

clc
clear
a=0
b=4
xspan=[a,b]
y0=0
ssize=.1
%Euler Method
[x1,y1]=euler357(@f1272,xspan,y0,ssize)
plot(x1,y1,'r','LineWidth',2)
axis([0,4,-.2,1.2])
xlabel('x')
ylabel('y','Rotation',0)
grid on
hold on
%Heun Method
[x2,y2]=heun357(@f1272,xspan,y0,ssize)
plot(x2,y2,'b','LineWidth',2)
%ode45
[x3,y3]=ode45(@f1272,xspan,y0)
plot(x3,y3,'g','LineWidth',2)

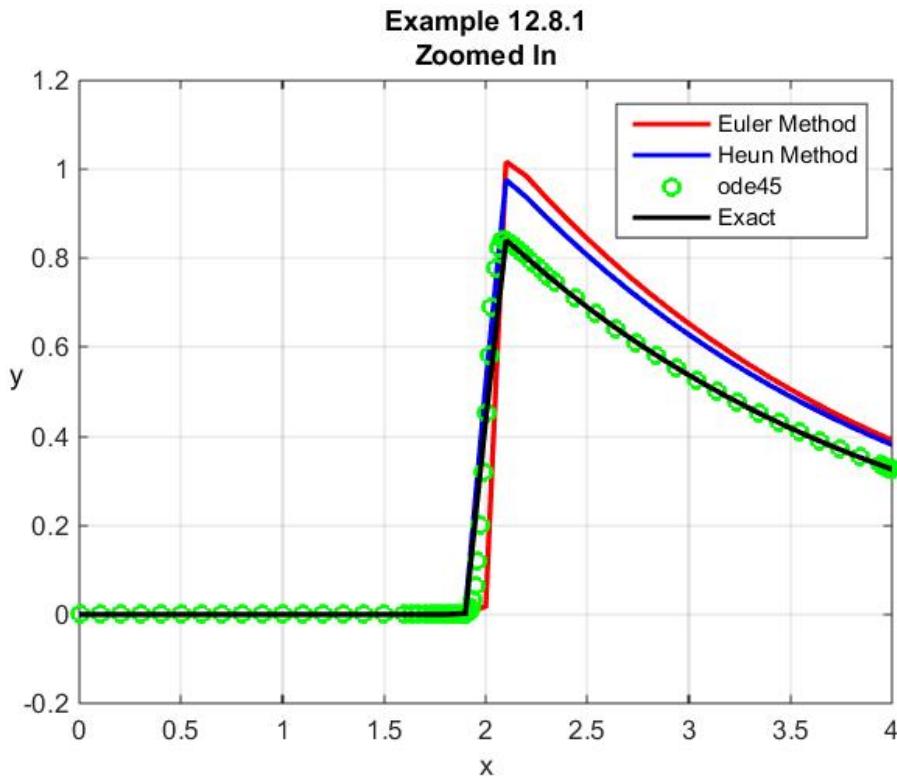
```

```
%Exact
xrange=a:.1:b
yexact=sqrt(pi)/4*exp((6401-...
    3200*xrange)/6400).* (erf(3201/80)-...
    erf((3201-1600*xrange)/80))
plot(xrange,yexact,'k','LineWidth',2)
title('Example 12.8.1')
legend('Euler Method','Heun Method','ode45','Exact')
```

the resulting plot is



The figure shows that the solution based upon the solver **ode45** is very close to the exact solution. As indicated above, the feature of **ode45** that shows up dramatically in this case is the fact that it is *adaptive*. As mentioned in Section 12.7, adaptive means that its step size adjusts to compensate for a rapidly changing functions. This point is illustrated by zooming in near the point $x = 2$. The following shows that the errors produced by the fixed step size are apparent, and, in particular, **ode45** improves the answer near the point $x = 2$.



It is instructive to examine the step sizes that were utilized by **ode45** in Example 12.8.1. The step size is given by the script

$$\text{ssize} = \text{x3}(2:\text{length}(\text{x3})) - \text{x3}(1:\text{length}(\text{x3})-1) \quad (12.8.11)$$

If this script is executed, it will be seen that the step size is reduced in the neighborhood of $x = 2$. A way to display the step sizes is to superimpose what is known as a *stem plot* of the step sizes on the above figure. The script

```

clc
clear
a=0
b=4
xspan=[a,b]
y0=0
ssize=.1
%Euler Method
[x1,y1]=euler357(@f1272,xspan,y0,ssize)
plot(x1,y1,'r','LineWidth',2)
axis([0,4,-.2,1.2])
xlabel('x')
ylabel('y','Rotation',0)
grid on

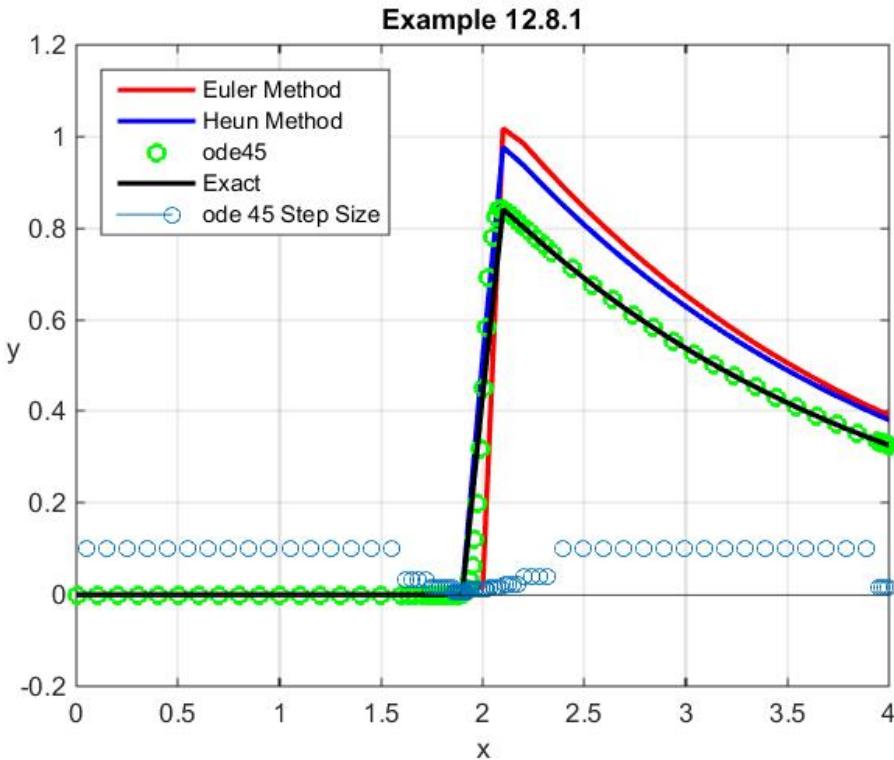
```

```

hold on
%Heun Method
[x2,y2]=heun357(@f1272,xspan,y0,ssize)
plot(x2,y2,'b','LineWidth',2)
%ode45
[x3,y3]=ode45(@f1272,xspan,y0)
plot(x3,y3,'og','LineWidth',2)
%Exact
xrange=a:.1:b
yexact=sqrt(pi)/4*exp((6401-...
    3200*xrange)/6400).*(erf(3201/80)-...
    erf((3201-1600*xrange)/80))
plot(xrange,yexact,'k','LineWidth',2)
%ode45 step size
N=length(x3)
ssize=x3(2:N)-x3(1:N-1)
stem(x3(1:N-1)+ssize/2,ssize)
title('Example 12.8.1')
legend('Euler Method','Heun Method','ode45',...
    'Exact','ode 45 Step Size')

```

produces the revised figure



As explained, the step size reductions for `ode45` are concentrated in the neighborhood of $x = 2$

Example 12.8.2: In our discussion of Example 12.4.2, we encountered the *Airy* ordinary differential equation³⁶

$$\frac{d^2y}{dx^2} - xy = 0 \quad (12.8.12)$$

This ordinary differential equation can be solved in terms of the Airy Functions, or, equivalently, Bessel functions.³⁷ The Airy function of the *first kind* is denoted by $\text{Ai}(x)$. It is defined such that it obeys the initial conditions³⁸

$$\text{Ai}(0) = \frac{1}{3^{\frac{2}{3}} \Gamma\left(\frac{2}{3}\right)}, \quad \frac{d\text{Ai}(0)}{dx} = -\frac{1}{3^{\frac{1}{3}} \Gamma\left(\frac{1}{3}\right)} \quad (12.8.13)$$

where Γ denotes the Gamma Function that was introduced in Section 11.11. Values of Airy functions are given in MATLAB. The script

```
clc
clear
a=-15
b=5
xexact=[a:.1:b]
yexact=airy(xexact)
plot(xexact,yexact,'b','LineWidth',2)
xlabel('x')
ylabel('y(x)','Rotation',0)
grid on
title({'Example 12.8.2','Airy Function'})
legend('Airy Function','Location','SouthEast')
```

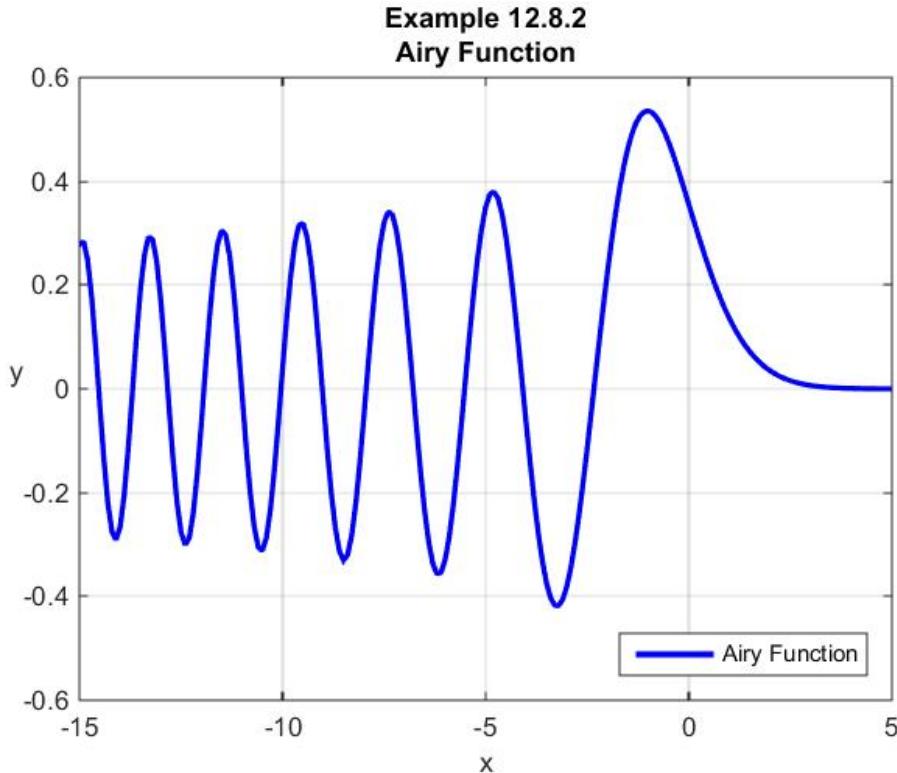
produces the following plot of the Airy function of the first kind in the interval $-15 < x < 5$

³⁶ Information about the English mathematician and astronomer, Sir George Biddell Airy, can be found at http://en.wikipedia.org/wiki/George_Biddell_Airy.

³⁷ It is possible to show that

$$\text{Ai}(x) = \begin{cases} \frac{\sqrt{-x}}{3} \left(J_{\frac{1}{3}}\left(\frac{2}{3}(-x)^{\frac{3}{2}}\right) + J_{-\frac{1}{3}}\left(\frac{2}{3}(-x)^{\frac{3}{2}}\right) \right) & \text{for } x \leq 0 \\ \frac{\sqrt{x}}{3} \left(I_{-\frac{1}{3}}\left(\frac{2}{3}x^{\frac{3}{2}}\right) - I_{\frac{1}{3}}\left(\frac{2}{3}x^{\frac{3}{2}}\right) \right) & \text{for } x \geq 0 \end{cases}$$

³⁸ See http://en.wikipedia.org/wiki/Airy_function.



Given this figure, the solution of (12.8.12) subject to the initial conditions (12.8.13) is obtained by first writing the differential equation in normal form as

$$\frac{d}{dx} \begin{bmatrix} y \\ \frac{dy}{dx} \end{bmatrix} = \begin{bmatrix} \frac{dy}{dx} \\ xy \end{bmatrix} \quad (12.8.14)$$

and define this first order system by the function m-file **f1282.m** with the script

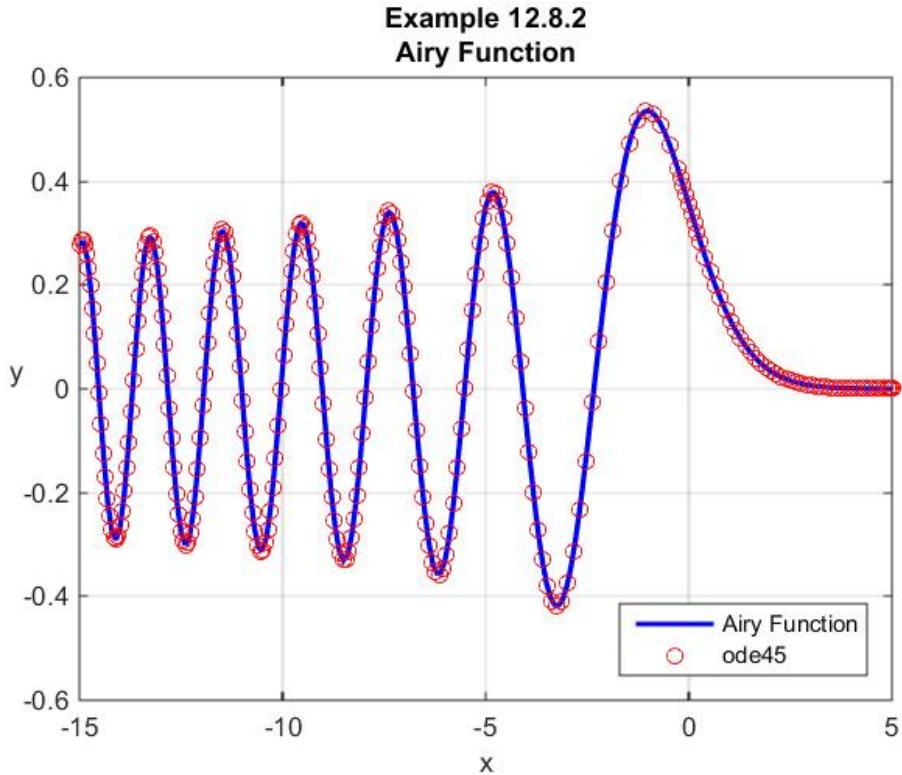
```
function dydx=f1282(x,y)
dydx=zeros(2,1) %Preallocate
dydx=[y(2);x*y(1)];
```

The script required to solve the differential equation (12.8.12) in the interval $-15 < x < 5$ needs to solve two initial value problems, one in the interval $0 < x < 5$ subject to the initial condition (12.8.13) and one in the interval $0 < x < -15$ again subject to the initial condition (12.8.13). After these two solutions are generated, the two solutions are joined and plotted for the combined interval $-15 < x < 5$. The following script reproduces the above figure and superimposes the plot of the two solutions just described:³⁹

³⁹ Note the use of the MATLAB function **flipud**. This function has the property that it will take a row vector or column vector and flip the elements so that the order is reversed. See <http://www.mathworks.com/help/matlab/ref/flipud.html>.

```
clc
clear
a=-15
b=5
xexact=[a:.1:b]
yexact=airy(xexact)
plot(xexact,yexact,'b','LineWidth',2)
xlabel('x')
ylabel('y','Rotation',0)
grid on
hold on
title({'Example 12.8.2','Airy Function'})
%Solve and Plot for positive x
xspan=[0,5]
y0=[1/3^(2/3)/gamma(2/3),-1/3^(1/3)/gamma(1/3)]
%ode45
[x1,y1]=ode45(@f1282,xspan,y0)
%Solve and Plot for negative x.
%Same initial conditions
xspan=[0,-15]
%ode45
[x2,y2]=ode45(@f1282,xspan,y0)
%Construct solution for all x in interval -15<x<5
x3=[flipud(x2);x1]
y3=[flipud(y2);y1]
plot(x3,y3(:,1),'r')
title({'Example 12.8.2','Airy Function'})
legend('Airy Function','ode45','Location','SouthEast')
```

The figure produced by this script is



Exercises

12.8.1: Consider the following nonlinear first order ordinary differential equation

$$\frac{dy}{dx} + y = xy^{\frac{1}{3}} \quad \text{where} \quad y(0) = 1 \quad (12.8.15)$$

This particular ordinary differential equation is one studied in elementary ordinary differential equations courses. It is an example of one known as a *Bernoulli equation*. The exact solution is readily shown to be

$$y(x) = \left(x + \frac{5}{2} e^{-\frac{2}{3}x} - \frac{3}{2} \right)^{\frac{3}{2}} \quad (12.8.16)$$

Use MATLAB's **ode45** solver and obtain a numerical solution to (12.8.15). Display your answer in the form of a plot showing the exact solution and the numerical solution.

12.8.2: Consider the following nonlinear first order ordinary differential equation

$$\frac{dy}{dx} - y^2 = 2x^{-\frac{8}{3}} \quad \text{where} \quad y(1.5) = -0.4 \quad (12.8.17)$$

Use MATLAB's **ode45** and obtain a numerical solution to (12.8.17) in the interval $[1.5, 9]$. Display your answer in the form of a plot showing the numerical solution.

12.8.3: Consider the following nonlinear first order ordinary differential equation of the Bernoulli type

$$\frac{dy}{dx} - 5y = -\frac{5}{2}xy^3 \quad \text{where } y(0) = \frac{1}{\sqrt{20}} \quad (12.8.18)$$

The exact solution turns out to be

$$y(x) = \frac{10}{\sqrt{50x - 5 + 2005e^{-10x}}} \quad (12.8.19)$$

Use the MATLAB solver **ode45** to obtain a numerical solution to (12.8.18). Display your answer in the form of a plot showing the exact solution and the numerical solution.

12.8.4: Use the MATLAB solver **ode45** and solve the initial value problem

$$\frac{dy}{dx} = -|y|^{\frac{5}{2}} + \cos(y) \quad \text{where } y(0) = 1 \quad (12.8.20)$$

in the interval $[0, 4]$. Display your answer in the form of a plot showing the numerical solution.

12.8.5: Use the MATLAB solver **ode45** and solve the initial value problem for the second order nonlinear ordinary differential equation

$$\frac{d^2y}{dt^2} + e^y \frac{dy}{dt} + y^3 = \sin t \quad (12.8.21)$$

in the interval $(0, 60)$ subject to the initial conditions

$$y(0) = 3 \quad \text{and} \quad \frac{dy(0)}{dt} = -1 \quad (12.8.22)$$

Display your answer in the form of a plot showing the numerical solution $y(t)$ vs. t .

12.8.6: Use the MATLAB solver **ode45** and solve the initial value problem for the second order ordinary differential equation

$$\frac{d^3y}{dx^3} + 2\frac{d^2y}{dx^2} + \left(\frac{dy}{dx}\right)^2 + 3y = 0 \quad (12.8.23)$$

in the interval $5 \leq x \leq 12$ subject to the initial conditions

$$y(5) = 1, \frac{dy(5)}{dx} = 0 \quad \text{and} \quad \frac{d^2y(5)}{dx^2} = 0 \quad (12.8.24)$$

Display your answer in the form of a plot showing the numerical solutions for $y(x), \frac{dy(x)}{dx}$ and $\frac{d^2y(x)}{dx^2}$ in the interval $5 \leq x \leq 12$.

12.8.7: Rework Exercise 12.8.6 with equation (12.8.23) replaced by

$$\frac{d^3y}{dx^3} + 2\frac{d^2y}{dx^2} + \left(\frac{dy}{dx}\right)^2 + 3y = 10\sin(6x) \quad (12.8.25)$$

12.8.8: Use the MATLAB solver **ode45** and solve the initial value problem for the second order ordinary differential equation

$$\frac{d^3y}{dx^3} + 2y\frac{d^2y}{dx^2} - \left(\frac{dy}{dx}\right)^2 + 1 = 0 \quad (12.8.26)$$

in the interval $0 \leq x \leq 5$ subject to the initial conditions

$$y(0) = 0, \frac{dy(0)}{dx} = 0 \quad \text{and} \quad \frac{d^2y(5)}{dx^2} = 20 \quad (12.8.27)$$

Display your answer in the form of a plot showing the numerical solutions for $y(x), \frac{dy(x)}{dx}$ and $\frac{d^2y(x)}{dx^2}$ in the interval $0 \leq x \leq 5$.

Section 12.9. More on Stiff Ordinary Differential Equations

In this section, we shall return to a discussion of stiff ordinary differential equations. Recall that this issue arose when we worked Example 12.4.3. As a part of this discussion, we shall try to explain the difference between a stiff and nonstiff ordinary differential equation. As a practical matter, one can always begin a solution approach by assuming the differential equation to be solved is not stiff. If problems arise, such as an unacceptably long computing time or results that are clearly wrong, the next logical step is to change solvers to one designed for stiff equations. This imprecise approach needs to be supported by examples and other background information. Hopefully, the examples of this section and later sections will provide insights into stiff ordinary differential equations.

In Example 12.4.3, we discussed a stiff *linear* ordinary differential equation. In the context of this linear ordinary differential equation, we characterized a stiff ordinary differential equation as one that has multiple time scales (thinking of the independent variable x as time) of different orders of magnitude. For nonlinear differential equations the concept of time scales is even less precise.

There are several definitions of stiffness that one can find:⁴⁰

1. An ordinary differential equation is stiff if the step size required for stability is smaller than the step size required for accuracy.
2. An ordinary differential equation is stiff if it contains some components of the solution that decay rapidly compared to other components of the solution.
3. A system of ordinary differential equations is stiff if at least one eigenvalue of the system is negative and large compared to the other eigenvalues of the system.
4. An ordinary differential equation is stiff if the step size based on computational time is too large to obtain an accurate and stable solution.

Essentially, these definitions try to capture the idea that ordinary differential equations, both linear and nonlinear, have components that grow or decay at different rates. In cases, where these rates differ greatly, the ordinary differential equation is stiff.

Example 12.9.1: In Example 12.4.3, we first encountered the idea of a stiff ordinary differential equation. We adopted the Euler method and looked at various approximate solutions of the initial value problem

$$\frac{dy}{dx} + \frac{1}{10}y = -4e^{-4x} \quad \text{where } y(0) = 2 \quad (12.9.1)$$

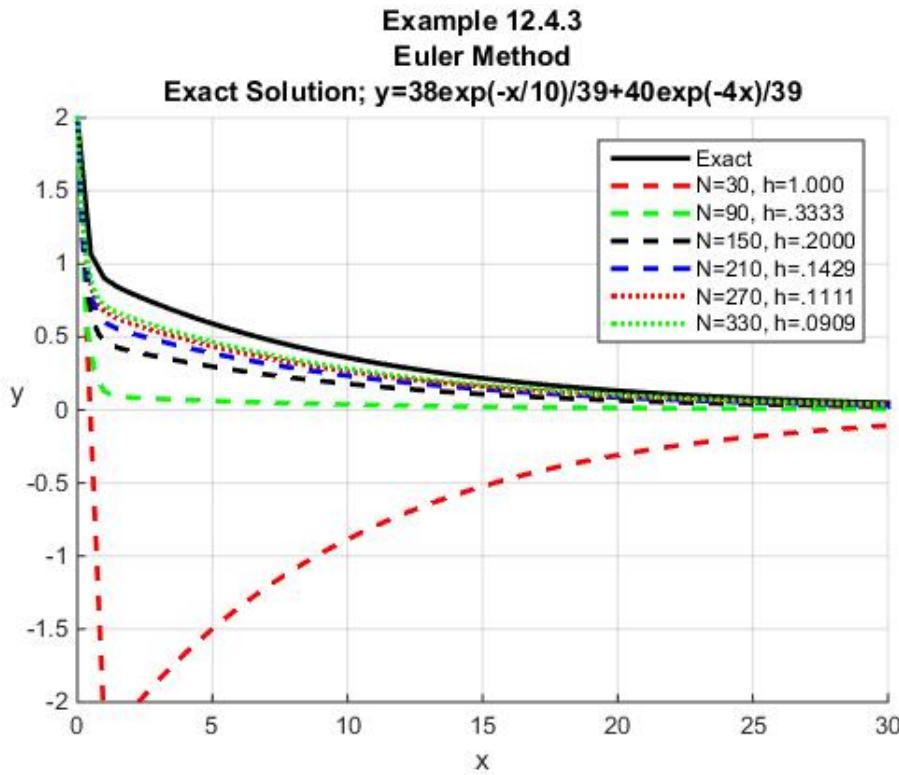
We discovered that the resulting answers were especially sensitive to step sizes and, even for small step sizes, were slow to generate. In particular, the most accurate solution we generated

⁴⁰ This list is essentially a list presented in the textbook, Numerical Methods for Engineers and Scientists, Second Edition, by Joe D. Hoffman, CRC Press, 2001. Example 12.9.1 is similar to an example in this textbook.

required 330 steps in the interval $[0, 30]$ and that solution took almost 0.6 seconds. As a part of the Example 12.4.3, the following table was given:

N	30	90	150	210	270	330
Time	0.0095 sec	0.0284 sec	0.0807 sec	0.1783 sec	0.2963 sec	0.5829 sec

As pointed out, an even more extreme example is in the case $N = 1100$ which consumed 7.3575 seconds. The figure that illustrated this calculation is shown in Section 12.4.3 and is repeated here



This figure shows, among other things, that the solution for $N = 330$ is not accurate when compared to the exact solution. In this example, we shall compare the exact solution to the solution generated by the solver **ode45** and the stiff solver **ode15s**. Our method of displaying these solutions is to utilize the following script to display the results on three subplots.⁴¹

```

clc
clear all
a=0;
b=30;

subplot(3,1,1)

```

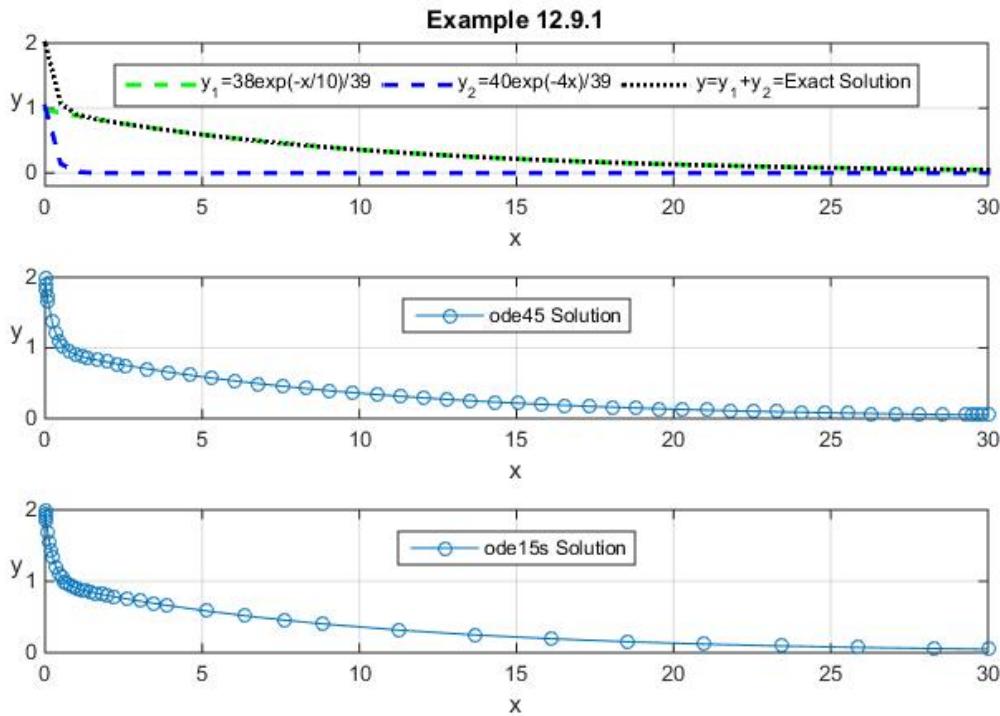
⁴¹ Note that this script adopts the feature mentioned associated with equation (12.8.8) where the solution plots are generated automatically.

```
%Plot of the exact solution
x=[a:1/2:b]
yp=40*exp(-4*x)/39
yh=38*exp(-x/10)/39
plot(x,yh,'g','LineWidth',2,'LineStyle','--')
axis([0,30,-.2,2])
grid on
hold all
plot(x,yp,'b','LineWidth',2,'LineStyle','--')
grid on
yexact=yh+yp
plot(x,yexact,'k','LineWidth',2,'LineStyle',':')
xlabel('x')
ylabel('y','Rotation',0)
legend('y_1=38exp(-x/10)/39','y_2=40exp(-4x)/39',...
'y=y_1+y_2=Exact Solution',...
'Orientation','Horizontal','Location','North')
title('Example 12.9.1')

subplot(3,1,2)
tic
ode45(@f1291,[0,30],2)
toc
grid on
xlabel('x')
ylabel('y(x)','Rotation',0)
legend('ode45 Solution','Location','North')

subplot(3,1,3)
tic
ode15s(@f1291,[0,30],2)
toc
grid on
xlabel('x')
ylabel('y(x)','Rotation',0)
legend('ode15s Solution','Location','North')
```

The figure produced by this script is



The first of the three subplots shows the exact solution and the two components of the exact solution. The first, denoted by y_1 , is the slow part of the solution. The second, denoted by y_2 , is the fast part. As explained in the discussion of Example 12.4.3, a numerical scheme based upon a fixed step size has a problem of being small enough to capture the fast part of the solution without taking too long to calculate the slow part.

The second subplot shows the solution generated by the solver **ode45**. The fact that **ode45** is an adaptive solver is revealed by the concentration of data points in the region of rapid change of the fast solution. The first two subplots support the idea that **ode45** has produced a more accurate solution than those based upon the first step size Euler method. The **tic-toc** script revealed an elapsed time of 0.489886 seconds which is slightly faster than the most accurate solution based upon the Euler method.

The third subplot shows the solution generated by the stiff solver **ode15s**. It is evident from the figure that more data points are concentrated in the region of the fast solution than for the **ode45** solver. In the region of the slow solution, the data points are less concentrated than for **ode45**. The **tic-toc** script revealed an elapsed time of 0.353941 seconds.⁴²

⁴² As explained in our earlier discussions, the elapsed time calculations in Example 12.9.1 are unique to the author's computer and the version of MATLAB being used. If the above script is executed on a different computer or with a different version of MATLAB, different elapsed times will be generated. The key result is that the stiff solver is faster for this simple stiff ordinary differential equation.

Example 12.9.1 involves the solution of an elementary linear ordinary differential equation. As such, it does not in a significant way reveal the advantages of the stiff solver. Our later examples will illustrate these advantages for more complex problems.

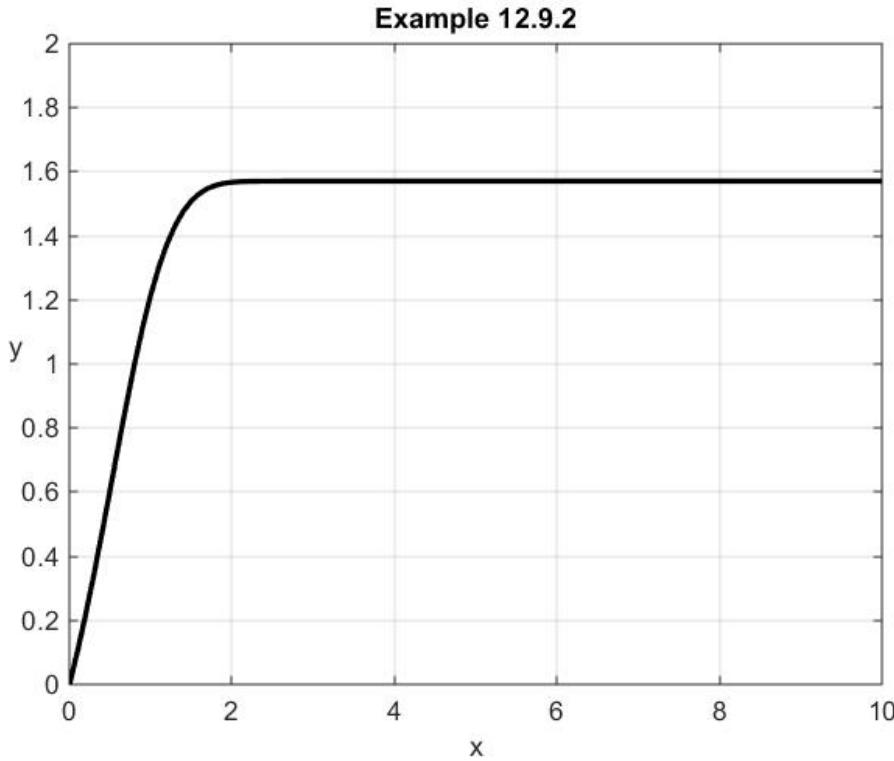
Example 12.9.2: Our objective is to find an approximate numerical solution of the initial value problem

$$\frac{dy}{dx} = e^x \cos y \quad \text{where } y(0) = 0 \quad (12.9.2)$$

on the interval $[0, 10]$.⁴³ Equation (12.9.2) is a first order nonlinear ordinary differential equation whose variables separate. The exact solution for this equation can be written

$$y = \tan^{-1} (\sinh(e^x - 1)) \quad (12.9.3)$$

The solution (12.9.3) can be obtained by use of MATLAB's **dsolve** if one desires. The plot of the exact solution (12.9.3) is



⁴³Example 12.9.1 is one taken from the textbook, Polking, John C., and David Arnold, *Ordinary Differential Equations using MATLAB*, Third Edition, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2004.

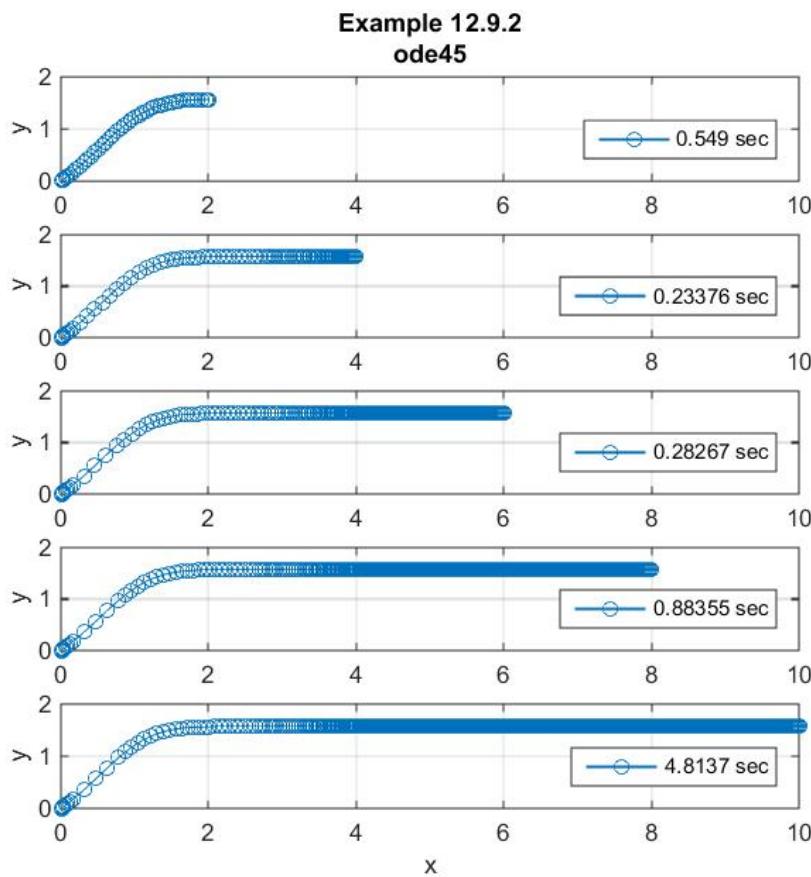
The differential equation (12.9.2) is stiff. In order to illustrate this characteristic, we shall first generate an approximate solution utilizing **ode45**. In order to illustrate that the resulting solution takes a large amount of time, we shall generate five solutions. Each will start at $x = 0$ and extend for increasing values of x . In particular we shall solve (12.9.2) in the intervals $(0,2), (0,4), (0,6), (0,8)$ and $(0,10)$. The differential equation (12.9.2) is defined by the function m-file with the script

```
function dydx=f1292(x,y)
dydx=exp(x)*cos(y);
```

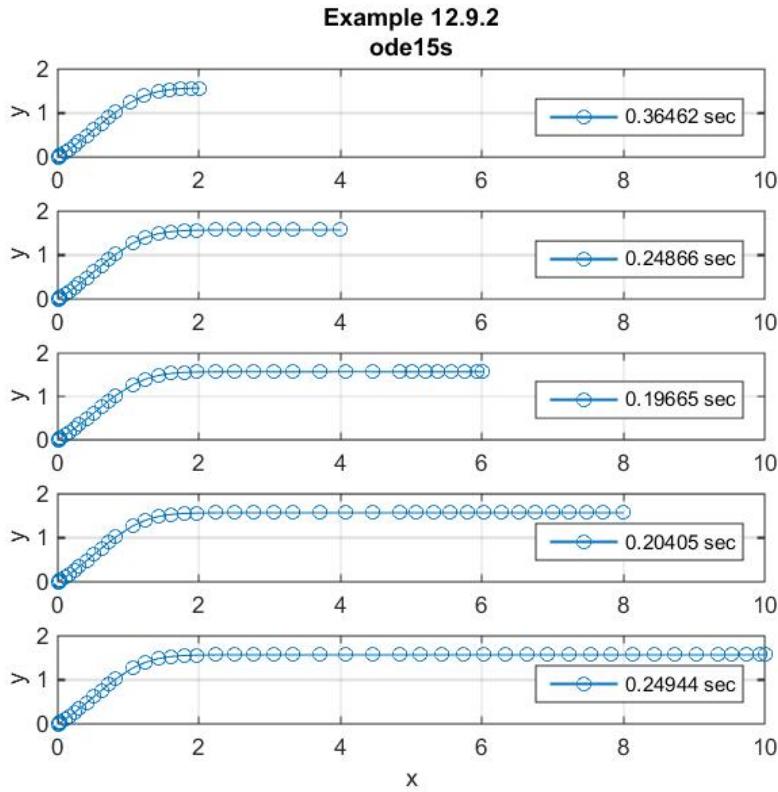
By utilization of the **tic-toc** script we shall show that the solution actually slows down as x increases. The MATLAB script that generates these solutions is

```
clc
clear
%ode45 Solution
for n=1:5
    h(n)=subplot(5,1,n)
    xspan=[0,2*n];
    y0=0;
    tic
    ode45(@f1292,xspan,y0);
    ylabel(h(n),'y')
    timer(n)=toc
    axis([0,10,0,2])
    str=strcat(num2str(toc),' sec')
    legend(h(n),str,'Location','SouthEast')
    grid on
end
title(h(1),'Example 12.9.2')
xlabel(h(5),'x')
```

The resulting figure is



The first figure shows that it took 0.549 sec for MATLAB to generate the solution in the interval $(0, 2)$. The second figure shows that MATLAB took 0.23376 sec for the interval $(0, 4)$. The remaining three figures show that the solution utilizing **ode45** actually slows down. It takes 4.8137 sec to generate the solution in the interval $(0, 10)$. These figures illustrate how the data points become more concentrated as the adaptive solver **ode45** struggles to meet its tolerance requirements. This kind of behavior suggests that one should perform the same calculation except use the stiff solver **ode15s**. The figure that replaces the one above is



This figure reveals the advantage of **ode15s** for a stiff ordinary differential equation. The solution did not slow down as the solution was generated in the interval $(0, 10)$. Also, the number of data points required to meet the solver **ode15s**'s tolerance requirements is substantially less than for **ode45**.

Example 12.9.3: Another example of a stiff ordinary differential equation is

$$\frac{dx}{dt} = x^2 - x^3 \quad (12.9.4)$$

subject to the initial condition

$$x(0) = 10^{-4} \quad (12.9.5)$$

In order to illustrate the stiff feature of this initial value problem, we shall utilize **ode45** and **ode15s** to generate approximate solutions over the interval $(0, 20,000)$.⁴⁴ The MATLAB script

⁴⁴ This differential equation is discussed in Section 7.9 of the textbook, Moler, Cleve, Numerical Computing with MATLAB, SIAM, Philadelphia, 2004. The electronic edition is at <http://www.mathworks.com/moler>. Moler attributes this example to Larry Shampine, who is one of the authors of the MATLAB ordinary differential suite. As explained by Moler, equation (12.9.4) has its origin in the study of flame propagation.

```
function dxdt=f1293(t,x)
dxdt=x^2-x^3;
```

creates the function m-file **f1293.m** that defines the ordinary differential equation (12.9.4). Given this file, the MATLAB script⁴⁵

```
clc
clear
x0=0.0001
tspan=[0,20000]
%ode45 Solution
subplot(1,2,1)
tic
ode45(@f1293,tspan,x0);
toc
ylabel('x(t)')
xlabel('t')
str=strcat(num2str(toc), ' sec')
legend(str,'Location','NorthWest')
grid on
title('ode45')
%ode15s Solution
subplot(1,2,2)
```

⁴⁵ The script given for Exercise 12.9.3 contains the line **subtitle('Example 2.9.3')**. This line creates the common title for the two figures. This command was utilized earlier in Example 11.11.2. As mentioned in Section 11.11, if the online resources for MATLAB are examined, one finds there are several virtually equivalent ways to create a common title. The approach used here is explained at <http://www.mathworks.com/matlabcentral/answers/100459-how-can-i-insert-a-title-over-a-group-of-subplots>. This link gives the script for the function m-file **subtitle.m**. In our case, this script has been modified slightly in order to position the common title. In any case, for us the script is

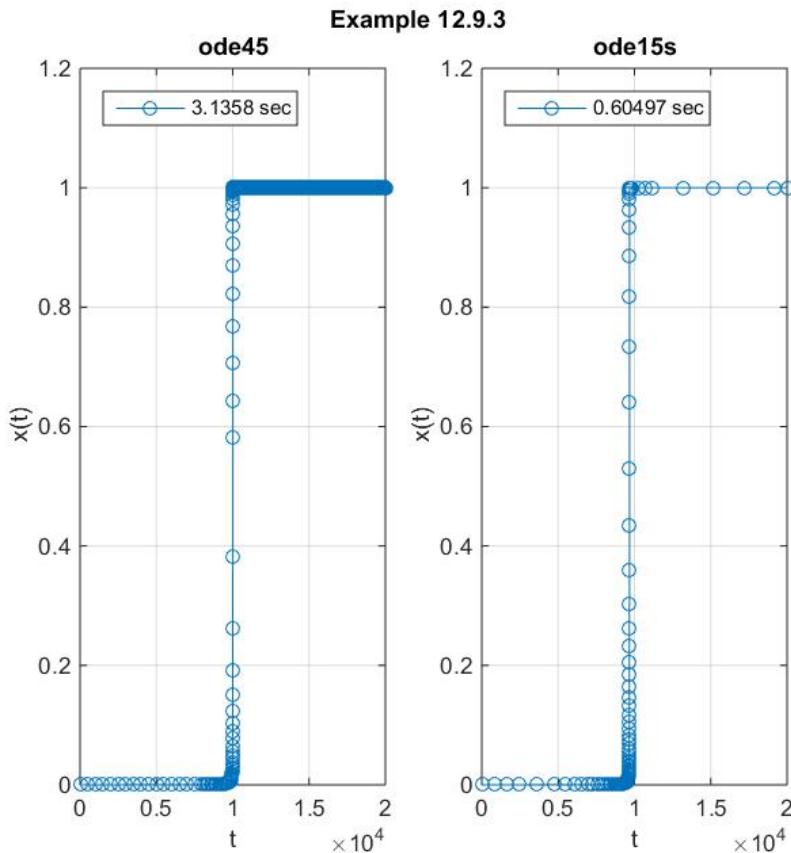
```
function [ax,h]=subtitle(text)
%
%Centers a title over a group of subplots.
%Returns a handle to the title and the handle to an axis.
% [ax,h]=subtitle(text)
%           returns handles to both the axis and the title.
% ax=subtitle(text)
%           returns a handle to the axis only.
ax=axes('Units','Normal',...
    'Position',[.075 .075 .85 .88],...
    'Visible','off');
set(get(ax,'Title'),'Visible','on')
title(text);
if (nargout < 2)
    return
end
h=get(ax,'Title');
```

```

tic
ode15s(@f1293,tspan,x0);
toc
ylabel('x(t)')
xlabel('t')
str=strcat(num2str(toc), ' sec')
legend(str,'Location','NorthWest')
grid on
title('ode15s')
subtitle('Example 2.9.3')

```

creates two plots. The first plot solves the initial value problem (12.9.4) and (12.9.5) utilizing **ode45** and the second on utilizing **ode15s**. In each case, the time of the computation is shown. The result of executing the above script is



The stiffness of (12.9.4) shows up in the **ode45** solution by the number of calculation points required to produce the constant solution after $t = 10^4$. It also is revealed by the excessive time required to generate the solution relative to that of **ode15s**.

Exercises

12.9.1: A second order ordinary differential equation that can exhibit stiff behavior is the *van der Pol equation*.⁴⁶

$$\frac{d^2y}{dt^2} + \mu(y^2 - 1) \frac{dy}{dt} + y = 0 \quad (12.9.6)$$

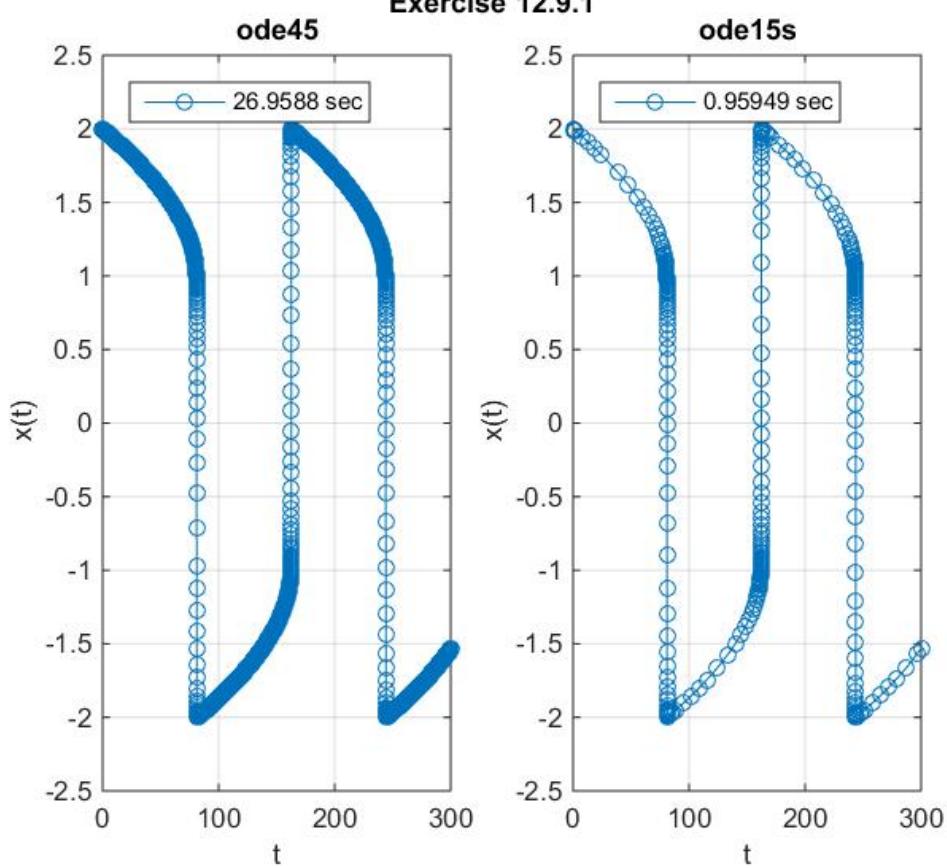
The parameter μ in (12.9.6) is a positive constant that must be specified. This equation arises in the modeling of electrical circuits with a triode whose resistance changes with the current. It also arises in the study of certain kinds of chemical reactions, and the study of certain kinds of wind induced motions of structures.

It turns out that whether or not (12.9.6) is stiff depends upon the numerical value of μ . Our objective with this exercise is to solve (12.9.6) in the interval $0 \leq t \leq 300$ subject to the initial condition

$$y(0) = 1 \quad \text{and} \quad \frac{dy(0)}{dt} = 0 \quad (12.9.7)$$

for the value $\mu = 100$. Display the stiff nature of this problem by solving it first with ode45 and then with ode15s. As with Example 12.9.3, these two solutions can be displayed in a single figure. A possible figure is

⁴⁶ Information about the Dutch physicist Balthasar van der Pol can be found at http://en.wikipedia.org/wiki/Balthasar_van_der_Pol. Examples involving the van der Pol equation that are equivalent to Exercise 12.9.1 are standards when discussing stiff ordinary differential equations. See, for example, <http://www.mathworks.com/help/matlab/ref/ode15s.html> and Chapter 8 of Polking, John C., and David Arnold, Ordinary Differential Equations using MATLAB, Third Edition, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2004.

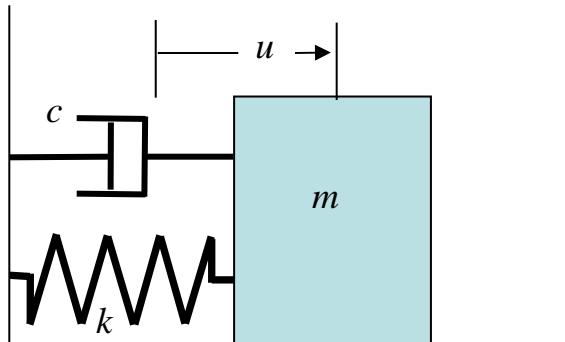
Exercise 12.9.1

Section 12.10. Systems of Linear Ordinary Differential Equations

With the exception of Example 12.5.1, Example 12.7.1, Example 12.8.2, Exercise 12.8.5 and Exercise 12.9.1, the examples discussed thus far involve *first order* ordinary differential equations. In this section, we shall focus on examples that utilize the MATLAB solvers to solve first order systems of ordinary differential equations. Most of the examples will begin with second order ordinary differential equations which will be converted to systems of first order equations as illustrated in Section 12.1.

We shall also illustrate with examples features of the MATLAB solvers that are not illustrated in our earlier sections in this chapter. In particular, we shall see how to utilize *global variables* and how to pass parameters to a function file that defines the differential equation. Global variables were briefly mentioned in Appendix A.

Example 12.10.1: This example looks again at Example 5.6.2 that involved finding the solution to the linear ordinary differential equation governing harmonic motion with damping. The analytical solution was discussed in Example 5.6.2 and again in Example 6.5.2. It is instructive to use **ode45** to proceed directly to the numerical solution without first generating the analytical solution. It is useful to illustrate this single degree of freedom of vibrating system by the following figure.



This example is one of a *free vibrations* problem because there is no *forcing function* applied to the mass m . The usual way such systems are modeled is to assume the spring is a *linear spring* and that the dashpot is a *linear dashpot*. This means that the force, in the spring case, is a *positive* constant, k , times the displacement, and that the force, in the dashpot case, is a *nonnegative* constant, c , times the velocity. With these kinds of assumptions, the initial value problem is

$$m \frac{d^2u}{dt^2} + c \frac{du}{dt} + ku = 0 \text{ and } u(0) = u_0, \frac{du(0)}{dt} = v_0 \quad (12.10.1)$$

where the initial displacement u_0 and the initial velocity v_0 are given.

It is customary when working physical problems to express the differential equation in terms of dimensional and non-dimensional coefficients as follows by defining

$$\omega_0^2 = \frac{k}{m} \quad (12.10.2)$$

and the dimensionless measure of the damping by the symbol

$$\zeta = \frac{c}{2m\omega_0} \quad (12.10.3)$$

The coefficient ω_0^2 has the physical dimension of

$$\frac{\text{Force / Length}}{\text{Mass}} = \frac{(\text{Mass})(\text{Length / Time}^2) / \text{Length}}{\text{Mass}} = \frac{1}{\text{Time}^2}$$

Therefore, it is dimensionally a *frequency*. It is called the *natural frequency*. The damping coefficient $\zeta = \frac{c}{2m\omega_0}$ is easily verified to be dimensionless. In terms of these symbols, the ordinary differential equation (12.10.1) takes the form

$$\frac{d^2u(t)}{dt^2} + 2\zeta\omega_0 \frac{du(t)}{dt} + \omega_0^2 u(t) = 0 \quad (12.10.4)$$

Equation (12.10.4) is the form of the differential equation in our earlier examples 5.6.2 and 6.5.1. As mentioned in our discussion of Example 5.6.2, an important special case is when the differential equation is *under damped*. This means that $1 - \zeta^2 > 0$. In general, there are three cases of importance

- | | |
|----------------------|-------------------|
| a) Under damped | $1 - \zeta^2 > 0$ |
| b) Critically damped | $1 - \zeta^2 = 0$ |
| c) Over damped | $1 - \zeta^2 < 0$ |

The analytical solutions corresponding to the three cases are as follows:

- a) Under damped:

$$u(t) = e^{-\zeta \omega_0 t} \begin{pmatrix} u_0 \cos(\omega_0(\sqrt{1-\zeta^2})t) \\ + \frac{1}{\sqrt{1-\zeta^2}} \left(\frac{v_0}{\omega_0} + \zeta u_0 \right) \sin(\omega_0(\sqrt{1-\zeta^2})t) \end{pmatrix} \quad (12.10.5)$$

b) Critically damped

$$u(t) = e^{-\omega_0 t} (u_0 + (v_0 + \omega_0 u_0) t) \quad (12.10.6)$$

c) Over damped

$$u(t) = e^{-\zeta \omega_0 t} \begin{pmatrix} u_0 \cosh(\omega_0(\sqrt{\zeta^2 - 1})t) \\ + \frac{1}{\sqrt{\zeta^2 - 1}} \left(\frac{v_0}{\omega_0} + \zeta u_0 \right) \sinh(\omega_0(\sqrt{\zeta^2 - 1})t) \end{pmatrix} \quad (12.10.7)$$

Equation (12.10.5) is the result of applying the initial conditions (12.10.1)₂ to the solution given in equation 5.6.44. The solutions (12.10.6) and (12.10.7) can be obtained by similar methods.

Our purpose with this example is to use the initial value problem (12.10.1) as an *example* of how to utilize the MATLAB solvers for second order ordinary differential equations. The earlier examples, 12.5.1, 12.7.1 and 12.8.2 also illustrated second order differential equations. However, in this case we shall add a feature that the other examples did not illustrate. We shall define the differential equation within MATLAB such that we can *pass parameters* from the function file to the numerical solution. Exactly what this means will become clear in the following.

As usual, the *first step* is to write the ordinary differential equation (12.10.1) in *normal form*. We showed in our earlier examples, the first step is to define a column vector $\mathbf{x}(t)$ by

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} u(t) \\ \frac{du(t)}{dt} \end{bmatrix} \quad (12.10.8)$$

The next step is to use (12.10.1)₁ and form

$$\frac{d\mathbf{x}(t)}{dt} = \begin{bmatrix} \frac{du(t)}{dt} \\ \frac{d^2u(t)}{dt^2} \end{bmatrix} = \begin{bmatrix} \frac{du(t)}{dt} \\ -2\zeta\omega_0 \frac{du(t)}{dt} - \omega_0^2 u(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ -2\zeta\omega_0 x_2(t) - \omega_0^2 x_1(t) \end{bmatrix} \quad (12.10.9)$$

Therefore, the initial value problem, restated in terms of the normal form is

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ -2\zeta\omega_0 x_2(t) - \omega_0^2 x_1(t) \end{bmatrix} \text{ and } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} \quad (12.10.10)$$

The *second step* is to create a function m-file that defines the right hand side of the ordinary differential equation written in normal form, i.e., equation (12.10.10). Unlike our earlier examples, we shall not assign in advance numerical values to the material constants in advance. We shall leave the constants ω_0 and ζ unspecified in the function m-file and, in a way to be shown, pass actual values from the script that generates the solution. The function m-file for this example will be denoted by **f12101.m**. It is defined by the script

```
function dxdt=f12101(t,x,w0,zeta)
dxdt=zeros(2,1) %Preallocate
dxdt=[x(2);-w0^2*x(1)-2*zeta*w0*x(2)];
```

As the above script illustrates, we have defined the differential equation within **f12101.m** in such a way that it contains the physical constants ω_0 and ζ . The function file has as one of its inputs the values of ω_0 and ζ . These values are assigned in the script below. The details of the script utilize the structure illustrated in equation (12.8.10), repeated,

$$[x,y] = \text{ode45}(yprime,xspan,y0,[1,p1,p2,\dots,pn]) \quad (12.10.11)$$

Our solution will adopt the default **options** and prescribe two parameters, **p1=w0** and **p2=zeta**. We shall seek the solution over the interval $0 < t < 60$. In addition, we shall adopt the numerical values

$$m = 20 \text{ kg}, k = 20 \text{ kg/sec}^2 \quad \text{and} \quad c = 5 \text{ kg/sec} \quad (12.10.12)$$

and initial conditions

$$\begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (12.10.13)$$

Therefore, with the definitions (12.10.2) and (12.10.3), we see that

$$\omega_0 = \sqrt{\frac{k}{m}} = 1 \quad \text{and} \quad \zeta = \frac{c}{2m\omega_0} = \frac{5}{40} = \frac{1}{8} \quad (12.10.14)$$

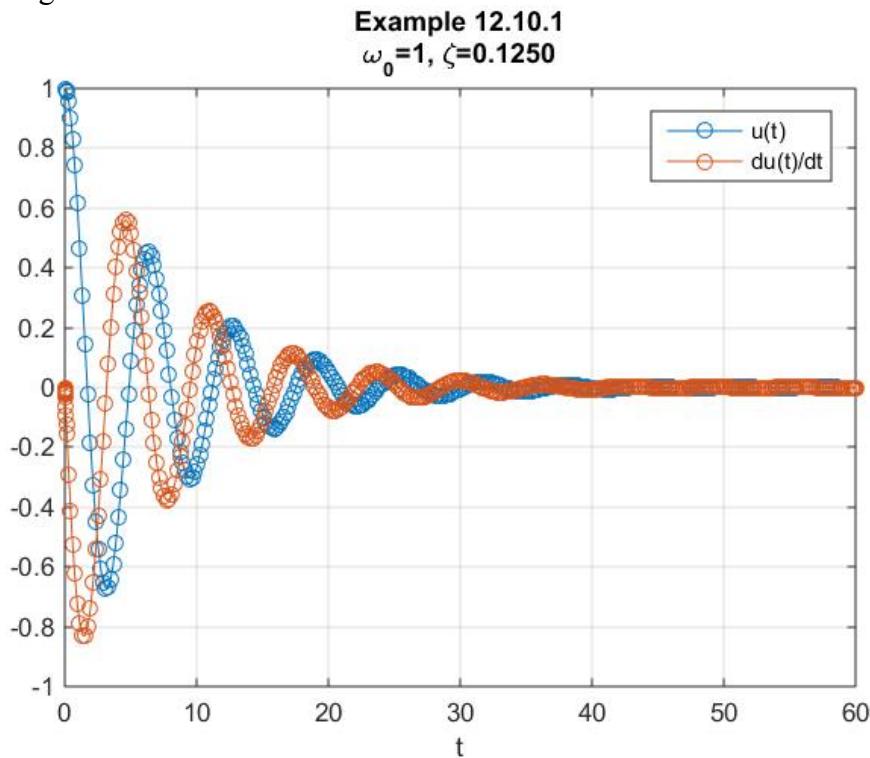
These numerical results show that the problem we have adopted, the solution is under damped. The script that generates the solution is

```

clc
clear
m=20
k=20
c=5
w0=sqrt(k/m)
zeta=c/(2*m*w0)
u0=1
v0=0
x0=[u0,v0]
tspan=[0,60]
ode45(@f12101,tspan,x0,[],w0,zeta)
grid on
xlabel('t')
legend('u(t)', 'du(t)/dt')
title({'Example 12.10.1', '\omega_0=1, \zeta=0.1250'})

```

The result is the figure



This graph shows how the solution $u(t)$ and its derivative $\frac{du(t)}{dt}$ damp out in time. This feature, of course, is displayed by the analytical solution (12.10.5).

The graph of $u(t)$ and $\frac{du(t)}{dt}$ can be made more elaborate by the introduction of a second axis in order to distinguish more clearly between the plot of $u(t)$ and the plot of $\frac{du(t)}{dt}$. The script that will produce the second axis is illustrated in Example 12.5.1.

Finally, the advantage of prescribing material constants as above is that one can easily work a family of problems for different values of the constants without the necessity of modifying **f12101.m** with each choice of the constants.

This point in the discussion is a good one to mention a slightly different approach to a solution like with Example 12.10.1 where it is desired to specify parameters in the ordinary differential equation. The key is the concept of a **global** variable. This concept was briefly explained in Section A.7 of the Appendix. The idea is to make the parameters global variables. This is achieved by two alterations to our script above. The function m-file **f12101.m** is replaced by

```
function dxdt=f12101(t,x)
global w0 zeta
dxdt=zeros(2,1) %Preallocate
dxdt=[x(2);-w0^2*x(1)-2*zeta*w0*x(2)];
```

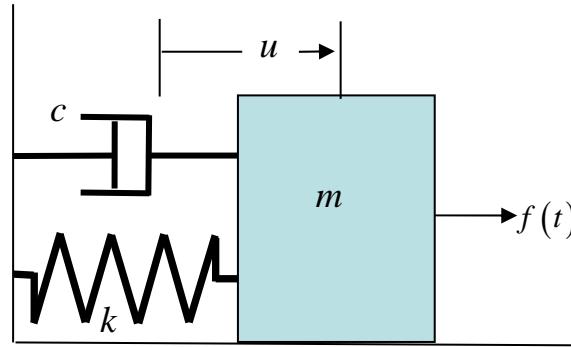
and the script that utilizes the new function m-file is replaced by

```
clc
clear
global w0 zeta
m=20
k=20
c=5
w0=sqrt(k/m)
zeta=c/(2*m*w0)
u0=1
v0=0
x0=[u0,v0]
tspan=[0,60]
ode45(@f12101,tspan,x0)
grid on
xlabel('t')
legend('u(t)', 'du(t)/dt')
title({'Example 12.10.1', '\omega_0=1, \zeta=0.1250'})
```

This different approach requires that the parameters **w0** and **zeta** be declared as global variables in both m-files. In the one immediately above, the numerical values are assigned and, because of their global nature, passed to the function m-file that defines the ordinary differential equation. The solution that is generated by **ode45** no longer needs to have the values of **w0** and **zeta**

specified as additional parameters. While this alternate prescription of the parameters works perfectly well, by convention we shall utilize the first approach outlined in Example 12.10.2.

Example 12.10.2: In this example, we shall again solve for the motion of the simple *linear spring-mass-damper* arrangement of our first example, except that now we shall allow an *external forcing function*. The following figure suggests this case.



The initial value problem that replaces (12.10.1) is

$$m \frac{d^2u}{dt^2} + c \frac{du}{dt} + ku = f(t) \quad \text{and} \quad u(0) = u_0, \frac{du(0)}{dt} = v_0 \quad (12.10.15)$$

The equation that replaces (12.10.4) is

$$\frac{d^2u(t)}{dt^2} + 2\zeta\omega_0 \frac{du(t)}{dt} + \omega_0^2 u(t) = \frac{1}{m} f(t) \quad (12.10.16)$$

The analytical solution, for the under damped case, that replaces (12.10.5) turns out to be

$$u(t) = e^{-\zeta\omega_0 t} \left(u_0 \cos(\sqrt{1-\zeta^2})t + \frac{1}{\sqrt{1-\zeta^2}} \left(\frac{v_0}{\omega_0} + \zeta u_0 \right) \sin(\sqrt{1-\zeta^2})t \right) + \frac{1}{m\omega_0 \sqrt{1-\zeta^2}} \int_{\tau=0}^{t=t} e^{-\zeta\omega_0(t-\tau)} \sin(\omega_0(\sqrt{1-\zeta^2})(t-\tau)) f(\tau) d\tau \quad (12.10.17)$$

The normal form that is equivalent to (12.10.16) and that replaces (12.10.10) is

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ -2\zeta\omega_0 x_2(t) - \omega_0^2 x_1(t) + \frac{f(t)}{m} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} \quad (12.10.18)$$

The new element of this problem is that we shall include a forcing function of the explicit form

$$\frac{f(t)}{m} = F \cos \omega t \quad (12.10.19)$$

where ω is a known forcing frequency and F is a prescribed amplitude. Given (12.10.19), the solution (12.10.17) can be shown to reduce to the sum of a *transient solution* and a *steady state solution*.

$$u(t) = \underbrace{\left(u_0 - F \frac{\omega_0^2 - \omega^2}{(\omega_0^2 - \omega^2)^2 + 4\zeta^2 \omega_0^2 \omega^2} \right) e^{-\zeta \omega_0 t} \cos(\omega_0 (\sqrt{1-\zeta^2}) t) + \left(\frac{1}{\sqrt{1-\zeta^2}} \left(\frac{v_0}{\omega_0} + \zeta u_0 \right) - F \frac{\zeta}{\sqrt{1-\zeta^2}} \frac{\omega_0^2 + \omega^2}{(\omega_0^2 - \omega^2)^2 + 4\zeta^2 \omega_0^2 \omega^2} \right) e^{-\zeta \omega_0 t} \sin(\omega_0 \sqrt{1-\zeta^2} t)}_{\text{Goes to zero as } t \text{ grows} = \text{Transient Solution}} \\ + F \underbrace{\left(\frac{\omega_0^2 - \omega^2}{(\omega_0^2 - \omega^2)^2 + 4\zeta^2 \omega_0^2 \omega^2} \cos(\omega t) + \frac{2\zeta \omega \omega_0}{(\omega_0^2 - \omega^2)^2 + 4\zeta^2 \omega_0^2 \omega^2} \sin(\omega t) \right)}_{\text{Steady State Solution}} \quad (12.10.20)$$

In problems where $u_0 \neq 0$, it is convenient for our numerical calculations to express (12.10.20) in the dimensionless form

$$\begin{aligned}
u(t) = & \underbrace{\left(1 - \frac{F}{u_0 \omega_0^2} \frac{1 - \frac{\omega^2}{\omega_0^2}}{\left(1 - \frac{\omega^2}{\omega_0^2} \right)^2 + 4\zeta^2 \frac{\omega^2}{\omega_0^2}} \right) e^{-\zeta \omega_0 t} \cos(\omega_0 (\sqrt{1-\zeta^2}) t)}_{\text{Goes to zero as } t \text{ grows} = \text{Transient Solution}} \\
& + \underbrace{\left(\frac{1}{\sqrt{1-\zeta^2}} \left(\frac{v_0}{u_0 \omega_0} + \zeta \right) - \frac{F}{u_0 \omega_0^2} \frac{\zeta}{\sqrt{1-\zeta^2}} \frac{1 - \frac{\omega^2}{\omega_0^2}}{\left(1 - \frac{\omega^2}{\omega_0^2} \right)^2 + 4\zeta^2 \frac{\omega^2}{\omega_0^2}} \right) e^{-\zeta \omega_0 t} \sin(\omega_0 \sqrt{1-\zeta^2} t)}_{\text{Steady State Solution}} \\
& + \underbrace{\frac{F}{u_0 \omega_0^2} \left(\frac{1 - \frac{\omega^2}{\omega_0^2}}{\left(1 - \frac{\omega^2}{\omega_0^2} \right)^2 + 4\zeta^2 \frac{\omega^2}{\omega_0^2}} \cos(\omega t) + \frac{2\zeta \frac{\omega}{\omega_0}}{\left(1 - \frac{\omega^2}{\omega_0^2} \right)^2 + 4\zeta^2 \frac{\omega^2}{\omega_0^2}} \sin(\omega t) \right)}_{\text{Steady State Solution}}
\end{aligned} \tag{12.10.21}$$

The analytical solution (12.10.21) shows that the dimensionless displacement $\frac{u(t)}{u_0}$ has two components. The transient decays exponentially with a dimensionless time measured by $\omega_0 t$ and the steady state oscillates with frequency ω , a dimensionless time measured by ωt , a

dimensionless amplitude $\frac{F}{u_0 \omega_0^2} \frac{1}{\sqrt{\left(1 - \frac{\omega^2}{\omega_0^2} \right)^2 + 4\zeta^2 \frac{\omega^2}{\omega_0^2}}}$ and with a phase shift φ defined by ⁴⁷

⁴⁷ A simple rearrangement shows that the steady state solution part of (12.10.21) can be written

$$\frac{F}{u_0 \omega_0^2} \frac{1}{\sqrt{\left(1 - \frac{\omega^2}{\omega_0^2} \right)^2 + 4\zeta^2 \frac{\omega^2}{\omega_0^2}}} \cos(\omega t - \varphi)$$

$$\tan \varphi = \frac{2\zeta \frac{\omega}{\omega_0}}{1 - \frac{\omega^2}{\omega_0^2}} \quad (12.10.22)$$

In this example, we shall take

$$\frac{F}{u_0 \omega_0^2} = 1 \quad \text{and} \quad \zeta = \frac{1}{8} \quad (12.10.23)$$

and plot the solution for a forcing frequency ω that is $\frac{1}{2}$ the natural frequency ω_0 . In other words, we shall take

$$\frac{\omega}{\omega_0} = \frac{1}{2} \quad (12.10.24)$$

The function m-file **f12102.m**, with the script,

```
function dxdt=f12102(t,x,w0,zeta,w,F)
dxdt=zeros(2,1); %Preallocate
dxdt=[x(2);-w0^2*x(1)-2*zeta*w0*x(2)+F*cos(w*t)];
```

is a slight modification of the one we used above, **f12101.m**. The parameters to be passed to **f12102.m** are the natural frequency ω_0 , the dimensionless damping coefficient ζ , the forcing frequency ω and the forcing amplitude F . The objective is to have MATLAB generate a plot of the solution for the dimensionless displacement, $\frac{u(0)}{u_0}$, as a function of t subject to the initial conditions

$$u(0) = u_0 \quad \text{and} \quad \frac{du(0)}{dt} = v_0 = 0 \quad (12.10.25)$$

The MATLAB script

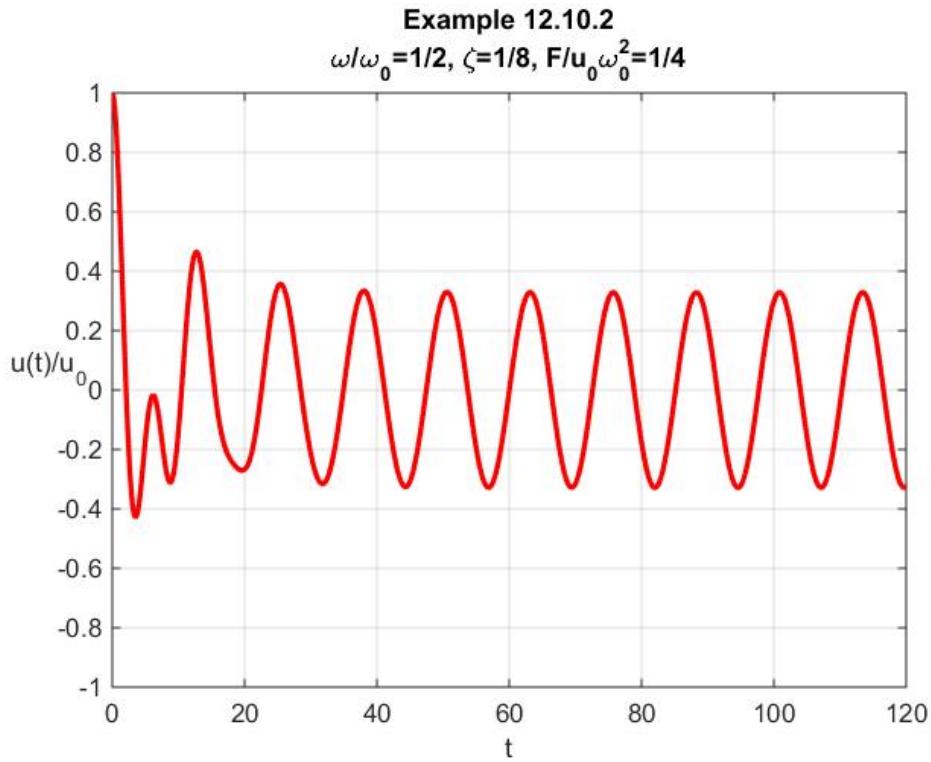
```
clc
clear
w0=1
zeta=1/8
F=1/4
w=1/2
u0=1
v0=0
```

```

x0=[u0,v0]
tspan=[0,120]
[t,x]=ode45(@f12102,tspan,x0,[],w0,zeta,w,F);
plot(t,x(:,1),'r','LineWidth',2)
grid on
axis([0 120 -1 1])
xlabel('t')
ylabel('u(t)/u_0','rotation',0)
title({'Example 12.10.2',...
    '\omega/\omega_0=1/2, \zeta=1/8, F/u_0\omega_0^2=1/4'})

```

produces the figure



Note that the choices **w0=1**, **zeta=1/8**, **F=1/4**, **w=1/2** and **u0=1** in the script achieve the solution displayed in the figure in terms of the dimensionless ratios shown. As reflected in the analytical solution (12.10.21), the above plot shows the steady state solution dominating the solution for large t .

Example 12.10.3: In this example, we shall again look at a one degree of freedom vibrating system with a forcing function. The example will set the damping coefficient ζ to zero and set the initial displacement u_0 and initial velocity v_0 to zero. The forcing function in this case is still given by (12.10.19). These specializations reduce the analytical solution (12.10.20) to

$$\frac{u(t)\omega_0^2}{F} = \frac{1}{1 - \frac{\omega^2}{\omega_0^2}} (\cos(\omega t) - \cos(\omega_0 t)) = \frac{2}{1 - \frac{\omega^2}{\omega_0^2}} \sin\left(\frac{\omega_0 + \omega}{2}t\right) \sin\left(\frac{\omega_0 - \omega}{2}t\right) \quad (12.10.26)$$

The second form of the analytical solution, equation (12.10.26)₂, shows that the solution is

composed of two components, one that oscillates with a frequency $\frac{\omega_0 + \omega}{2}$ and one that oscillates

with a frequency $\frac{\omega_0 - \omega}{2}$. This behavior relates to a phenomena known as “beats” where, for

forcing frequencies near the natural frequency, the displacement displays the superposition of the

“fast” frequency $\frac{\omega_0 + \omega}{2}$ and the “slow” frequency $\frac{\omega_0 - \omega}{2}$ in a special way. We can modify the

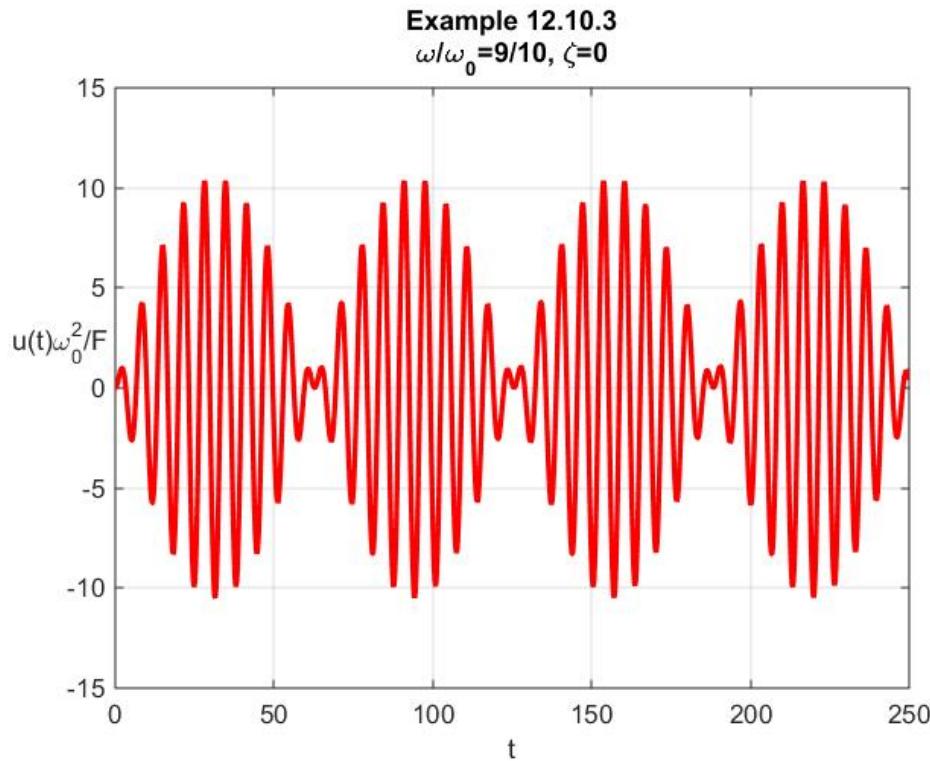
script utilized for Example 12.10.2 for this special case to the result

```

clc
clear
w0=1
zeta=0
F=1
w=9/10
u0=0
v0=0
x0=[u0,v0]
tspan=[0,250]
[t,x]=ode45(@f12102,tspan,x0,[],w0,zeta,w,F);
plot(t,x(:,1),'r','LineWidth',2)
grid on
axis([0 250 -15 15])
xlabel('t')
ylabel('u(t)\omega_0^2/F','rotation',0)
% legend('u(t)/u_0')
title({'Example 12.10.3',...
    '\omega/\omega_0=9/10, \zeta=0'})

```

The following figure displays the beats phenomena



The evolution of this figure through a family of solutions for ω ranging from zero to a value near ω_0 is informative. This kind of information is effectively displayed by utilization of MATLAB's animation capabilities. The script

```

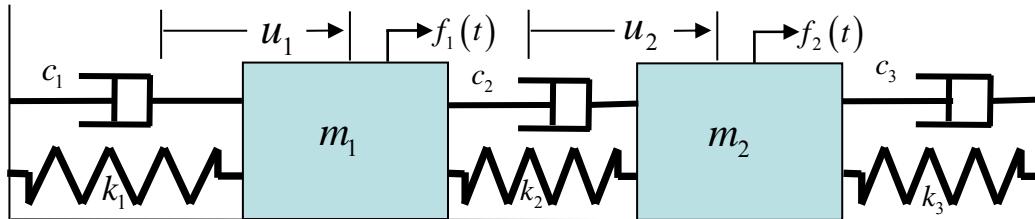
clc
clear
w0=1
zeta=0
F=1
w=[0:.01:.9]
u0=0
v0=0
x0=[u0,v0]
tspan=[0,250]
for j=1:length(w)
    [t,x]=ode45(@f12102,tspan,x0,[],w0,zeta,w(j),F);
    plot(t,x(:,1),'r','linewidth',2)
    grid on
    axis([0,250,-15,15])
    xlabel('t')
    ylabel('u(t)\omega_0^2/F','Rotation',0,...)
        'Position',[-17.3549 1 -1])
    title('Exercise 12.10.3')
    text(110,-10,['\omega/\omega_0 = ' num2str(w(j))])
    M(j) = getframe(gca,[-70 -50 620 498])
end

```

```
end
```

will produce this animation.⁴⁸

Example 12.10.4: In Example 5.6.6, we discussed the analytical solution of a two degree of vibration problem based, essentially, upon the figure



where m_1 and m_2 are masses, k_1, k_2 and k_3 are spring constants, c_1, c_2 and c_3 are damping constants and u_1 and u_2 are displacements. Finally, the functions $f_1(t)$ and $f_2(t)$ are forcing functions. The constants $m_1, m_2, k_1, k_2, k_3, c_1, c_2$ and c_3 are positive constants. The system of second order ordinary differential equations which govern the motion of this system can be written

$$M\ddot{\mathbf{u}}(t) + C\dot{\mathbf{u}}(t) + K\mathbf{u}(t) = \mathbf{f}(t) \quad (12.10.27)$$

where \mathbf{u} is the column vector of displacements defined by

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}, \quad (12.10.28)$$

M is a symmetric positive definite (diagonal) matrix of masses defined by

$$M = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}, \quad (12.10.29)$$

⁴⁸ The additional script appended to that above

```
vid=VideoWriter('Example12103.mp4','MPEG-4')
vid.FrameRate = 5
open(vid)
writeVideo(vid,M)
close(vid)
```

will produce a video file **Example12103.mp4** of the animation. This video can be viewed from the electronic version of Appendix B of this work.

C is a symmetric positive semidefinite matrix of damping coefficients defined by

$$C = \begin{bmatrix} c_1 + c_2 & -c_2 \\ -c_2 & c_2 + c_3 \end{bmatrix}, \quad (12.10.30)$$

K is a symmetric positive definite matrix of spring constants defined by

$$K = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{bmatrix} \quad (12.10.31)$$

and $\mathbf{f}(t)$ is a column matrix of forcing functions defined by

$$\mathbf{f}(t) = \begin{bmatrix} f_1(t) \\ f_2(t) \end{bmatrix} \quad (12.10.32)$$

The *normal form* of (12.10.27) is

$$\begin{aligned} \frac{d}{dt} \begin{bmatrix} \mathbf{u}(t) \\ \dot{\mathbf{u}}(t) \end{bmatrix} &= \begin{bmatrix} I & \mathbf{0} \\ \mathbf{0} & M^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{0} & I \\ -K & -C \end{bmatrix} \begin{bmatrix} \mathbf{u}(t) \\ \dot{\mathbf{u}}(t) \end{bmatrix} + \begin{bmatrix} I & \mathbf{0} \\ \mathbf{0} & M^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{0} \\ \mathbf{f}(t) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{0} & I \\ -M^{-1}K & -M^{-1}C \end{bmatrix} \begin{bmatrix} \mathbf{u}(t) \\ \dot{\mathbf{u}}(t) \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ M^{-1}\mathbf{f}(t) \end{bmatrix} \end{aligned} \quad (12.10.33)$$

where $\dot{\mathbf{u}} = \frac{d\mathbf{u}}{dt}$. This first order coupled system of four ordinary differential equations can be written in the form

$$\frac{d\mathbf{x}(t)}{dt} = A\mathbf{x}(t) + \mathbf{g}(t) \quad (12.10.34)$$

where $\mathbf{x}(t)$ is defined by

$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{u}(t) \\ \dot{\mathbf{u}}(t) \end{bmatrix} = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \dot{u}_1(t) \\ \dot{u}_2(t) \end{bmatrix} \quad (12.10.35)$$

and

$$(12.10.36)$$

$$A = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -M^{-1}K & -M^{-1}C \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{(k_1+k_2)}{m_1} & \frac{k_2}{m_1} & -\frac{(c_1+c_2)}{m_1} & \frac{c_2}{m_1} \\ \frac{k_2}{m_2} & -\frac{k_2+k_3}{m_2} & \frac{c_2}{m_2} & -\frac{c_2+c_3}{m_2} \end{bmatrix} \quad (12.10.37)$$

and

$$\mathbf{g}(t) = \begin{bmatrix} \mathbf{0} \\ M^{-1}\mathbf{f}(t) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \frac{f_1(t)}{m_1} \\ \frac{f_2(t)}{m_2} \end{bmatrix} \quad (12.10.38)$$

In this example, we wish to set up the structure for finding an approximate numerical solution to an initial value problem based upon the system (12.10.27). We shall denote by **f12104.m** the function m-file with the script

```
function dxdt=f12104(t,x,K,C,M,g)
n=length(M)
A=zeros(n),eye(n);-inv(M)*K,-inv(M)*C
dxdt=zeros(4,1); %Preallocate
dxdt=A*x+g(t);
%g(t) an anonymous function
```

Next, we shall illustrate the solution in the special case where ⁴⁹

$$M = \begin{bmatrix} 20 & 0 \\ 0 & 20 \end{bmatrix} \text{ kg} \quad (12.10.39)$$

$$K = \begin{bmatrix} 40 & -20 \\ -20 & 40 \end{bmatrix} \text{ kg/sec}^2 \quad (12.10.40)$$

$$C = \begin{bmatrix} .7 & -.4 \\ -.4 & .8 \end{bmatrix} \text{ kg/sec} \quad (12.10.41)$$

Therefore,

⁴⁹ The choices (12.10.39), (12.10.40) and (12.10.41) produce the homogeneous solution given by equation (5.6.71)

$$\begin{aligned}
 A &= \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{k_1+k_2}{m_1} & \frac{k_2}{m_1} & -\frac{c_1+c_2}{m_1} & \frac{c_2}{m_1} \\ \frac{k_2}{m_2} & -\frac{k_2+k_3}{m_2} & \frac{c_2}{m_2} & -\frac{c_2+c_3}{m_2} \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 & 1 & -0.350 & 0.0200 \\ 1 & -2 & 0.0200 & -0.0400 \end{bmatrix}
 \end{aligned} \tag{12.10.42}$$

In addition, we shall take

$$\mathbf{f}(t) = \begin{bmatrix} f_1(t) \\ f_2(t) \end{bmatrix} = \begin{bmatrix} 100 \cos\left(\frac{t}{2}\right) \\ 0 \end{bmatrix} \tag{12.10.43}$$

and

$$\mathbf{u}(0) = \begin{bmatrix} 6 \\ -6 \\ 0 \\ 0 \end{bmatrix} \tag{12.10.44}$$

The following MATLAB script generates the approximate solution over the interval $0 < t < 400$:

```

clc
clear
M=[20,0,0,20]
K=[40,-20;-20,40]
C=[.7,-.4;-.4,.8]
%Define the forcing function g(t) as an anonymous function
g=@(t)([eye(2),zeros(2);zeros(2),inv(M)]*[0;0;100*cos(t/2);0])
x0=[6,-6,0,0]
tspan=[0,400]
[t,x]=ode45(@f12104,tspan,x0,[],K,C,M,g);

subplot(2,1,1)
plot(t,x(:,1),'b','LineWidth',1)
grid on
axis([0 400 -10 10])

```

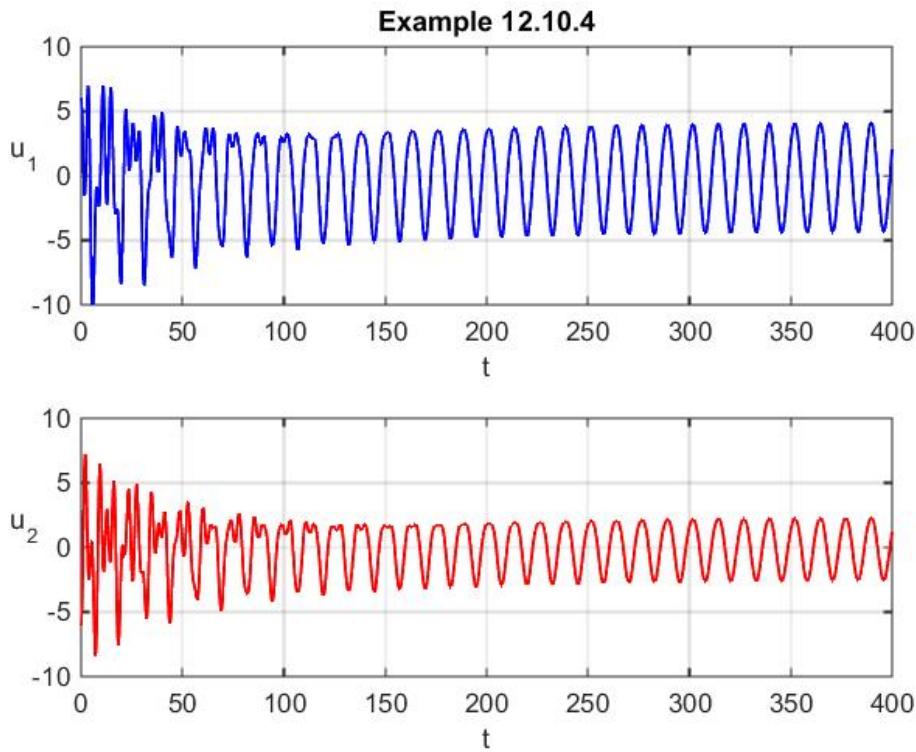
```

xlabel('t')
ylabel('u_1(t)', 'Rotation', 0)
title('Example 12.10.4')

subplot(2,1,2)
plot(t,x(:,3), 'r', 'linewidth', 1)
grid on
axis([0 400 -10 10])
xlabel('t')
ylabel('u_2(t)', 'Rotation', 0)

```

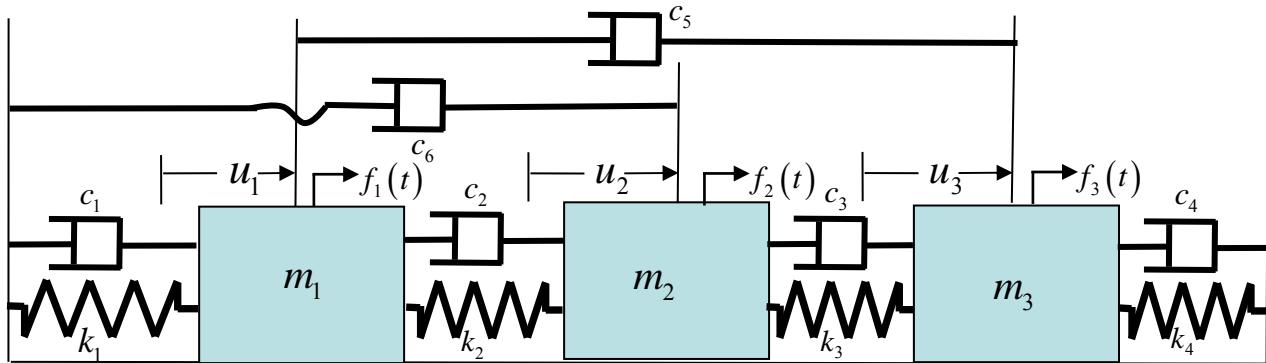
The result of this script are the two figures



While the result is more complicated than the damped forced vibration problem discussed in Example 12.10.2, this figure displays the transient solution decaying into a steady state solution. A word of caution about linear systems as illustrated by Example 12.10.4 is that they can result in stiff ordinary differential equations. This feature can be identified in a practical way when the solution takes an excessive amount of time to complete. A more fundamental way to identify the problem is to recall from Section 12.9 is that a stiff system arises when at least one eigenvalue of the system is negative and large compared to the others in the system. For the numbers adopted in Example 12.10.2, the four eigenvalues of the matrix (12.10.42)₂ turn out to be $-0.0288 \pm 1.7318i$ and $-0.0087 \pm 1.000i$ and the stiffness of the system did not surface as an issue.

Note that the script for Example 12.10.4 is structured such that it can easily be adapted for more complicated vibrating systems including more complicated forcing functions. One simply has to substitute the appropriate matrices K, C and M and the appropriate forcing function $g(t)$.

Exercise 12.10.1: In our discussion of Example 5.6.7, we looked at the following configuration of a *three degree of freedom* system with linear springs, linear damping and forcing functions.



In matrix form, the equations of motion for this damped three degree of freedom system are given by (12.10.27), where

$$M = \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix} \quad (12.10.45)$$

$$C = \begin{bmatrix} c_1 + c_2 + c_5 & -c_2 & -c_5 \\ -c_2 & c_2 + c_3 + c_6 & -c_3 \\ -c_5 & -c_3 & c_3 + c_4 + c_5 \end{bmatrix} \quad (12.10.46)$$

$$K = \begin{bmatrix} k_1 + k_2 & -k_2 & 0 \\ -k_2 & k_2 + k_3 & -k_3 \\ 0 & -k_3 & k_3 + k_4 \end{bmatrix} \quad (12.10.47)$$

$$\mathbf{f}(t) = \begin{bmatrix} f_1(t) \\ f_2(t) \\ f_3(t) \end{bmatrix} \quad (12.10.48)$$

and

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \end{bmatrix} \quad (12.10.49)$$

The matrix A still takes the form (12.10.37)₁ and is given by

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -\frac{k_1+k_2}{m_1} & \frac{k_2}{m_1} & 0 & -\frac{c_1+c_2+c_5}{m_1} & \frac{c_2}{m_1} & \frac{c_5}{m_1} \\ \frac{k_2}{m_2} & -\frac{k_2+k_3}{m_2} & \frac{k_3}{m_2} & \frac{c_2}{m_2} & -\frac{c_2+c_3+c_6}{m_2} & \frac{c_3}{m_2} \\ 0 & \frac{k_3}{m_3} & -\frac{k_3+k_4}{m_3} & \frac{c_5}{m_3} & \frac{c_3}{m_3} & -\frac{c_3+c_4+c_5}{m_3} \end{bmatrix} \quad (12.10.50)$$

and the matrix $\mathbf{g}(t)$ is given by

$$\mathbf{g}(t) = \begin{bmatrix} \mathbf{0} \\ M^{-1}\mathbf{f}(t) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{f_1(t)}{m_1} \\ \frac{f_2(t)}{m_2} \\ \frac{f_3(t)}{m_3} \end{bmatrix} \quad (12.10.51)$$

For the purposes of this exercise, adopt the numerical values introduced in Example 5.6.7, namely,

$$m_1 = m_2 = 1, m_3 = 2 \quad (12.10.52)$$

$$k_1 = k_2 = k_3 = 1, k_4 = 2, \quad (12.10.53)$$

$$c_1 = .3, c_2 = .4, c_3 = .4, c_4 = .06, c_5 = .06, c_6 = .02 \quad (12.10.54)$$

Show that these numbers reduce the matrix (12.10.50) to

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -2 & 1 & 0 & -0.76 & 0.4 & 0.06 \\ 1 & -2 & 1 & 0.4 & -0.82 & 0.4 \\ 0 & 0.5 & -1.5 & 0.03 & 0.2 & -0.26 \end{bmatrix} \quad (12.10.55)$$

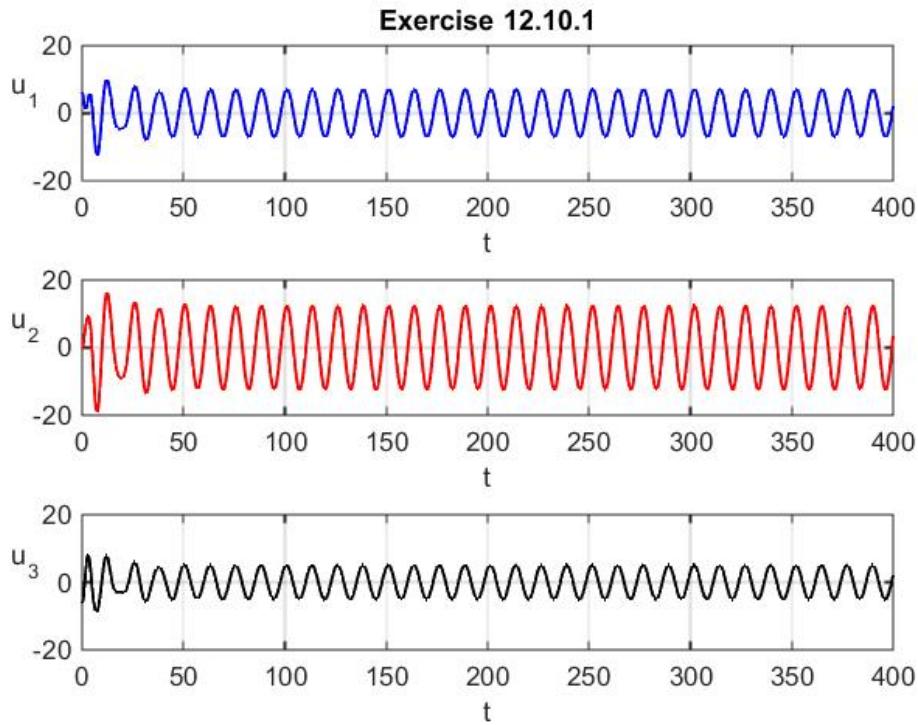
In addition, assume the forcing function $\mathbf{f}(t)$ is given by

$$\mathbf{f}(t) = \begin{bmatrix} 0 \\ 10 \cos\left(\frac{t}{2}\right) \\ 0 \end{bmatrix} \quad (12.10.56)$$

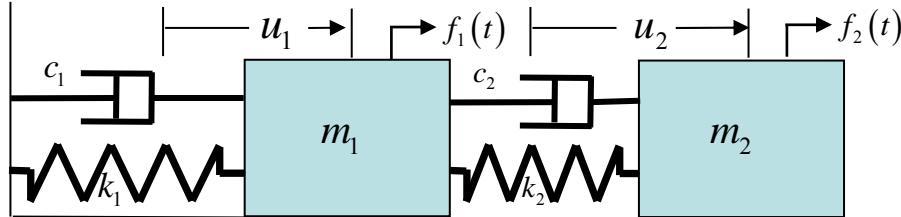
and the initial conditions are given by

$$\mathbf{x}(0) = \begin{bmatrix} \mathbf{u}(0) \\ \dot{\mathbf{u}}(0) \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \\ -6 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (12.10.57)$$

Follow the solution procedure used with Example 12.10.4 and show that a plot of the three displacements produces the figures



12.10.2: Consider a coupled spring mass system



The equations of motion are as follows:

$$\begin{aligned} m_1 \frac{d^2 u_1}{dt^2} &= -c_1 \frac{du_1}{dt} - k_1 u_1 + c_2 \left(\frac{du_2}{dt} - \frac{du_1}{dt} \right) + k_2 (u_2 - u_1) + f_1(t) \\ m_2 \frac{d^2 u_2}{dt^2} &= -c_2 \left(\frac{du_2}{dt} - \frac{du_1}{dt} \right) - k_2 (u_2 - u_1) + f_2(t) \end{aligned} \quad (12.10.58)$$

You are given the material constants as

$$\begin{aligned} m_1 = m_2 &= 1 \\ k_1 = k_2 &= 1 \\ c_1 &= .3 \\ c_2 &= .4 \end{aligned} \quad (12.10.59)$$

The forcing functions are given by

$$\begin{aligned} f_1(t) &= \cos(1.5t) \\ f_2(t) &= \sin(.6t) \end{aligned} \quad (12.10.60)$$

Assume the initial conditions are

$$u_1(0) = \frac{du_1(0)}{dt} = u_2(0) = \frac{du_2(0)}{dt} = 0 \quad (12.10.61)$$

Utilize MATLAB to generate an approximate solution of this initial value problem and plot the two displacements for the interval $t \in [0, 120]$

12.10.3: Consider the ordinary differential equation ⁵⁰

$$t(1-t)\frac{d^2x(t)}{dt^2} + (c - (a+b+1)t)\frac{dx(t)}{dt} - abx(t) = 0 \quad (12.10.62)$$

where a, b and c are constants. Utilize **ode45** to obtain an approximate solution of this ordinary differential equation with the choices

$$a = \frac{1}{2}, b = 1 \quad \text{and} \quad c = 1 \quad (12.10.63)$$

in the interval $[.01, .99]$ subject to the initial conditions

$$x(.01) = 1 \quad \text{and} \quad \frac{dx(.01)}{dt} = 0 \quad (12.10.64)$$

⁵⁰ The differential equation (12.10.62) is one that has been studied extensively. It is called the *hypergeometric equation*. See <http://www.efunda.com/math/hypergeometric/hypergeometric.cfm>. The analytical solution is obtained by a power series method. The answer is called a *Hypergeometric Function*.

Section 12.11. Systems of Nonlinear Ordinary Differential Equations

In this section, we shall focus the discussion on examples that involve finding approximate numerical solutions of systems of *nonlinear* ordinary differential equations. Our previous examples 12.5.1, 12.7.1 and 12.9.1 also concerned nonlinear systems. This section builds upon those earlier formulations. Among the examples will be those that illustrate the use of `odeset` that was briefly discussed in Section 12.8.

Example 12.11.1: Our earlier examples, Example 5.6.2, 6.5.2 and 12.10.1, concerned single degree of freedom free vibrations problems for linear springs and linear dashpots. This example removes the feature that the dashpot is liner. In particular, a model of nonlinear damping in a single degree of freedom free vibrations problem is defined by the following initial value problem

$$m \frac{d^2u}{dt^2} + \left(c + e \underbrace{\left(\frac{du}{dt} \right)^2}_{\text{Nonlinear}} \right) \frac{du}{dt} + ku = 0 \quad \text{and} \quad u(0) = u_0, \frac{du(0)}{dt} = v_0 \quad (12.11.1)$$

where m is the mass, k is the spring constant and $c + e \left(\frac{du}{dt} \right)^2$ is, in effect, the nonlinear damping constant. This model of damping has the nonlinear damping determined by two constants c and e . For our purposes, these constants are nonnegative. As a nonlinear ordinary differential equation, we no longer have an exact solution like (12.10.5) through (12.10.7). Never the less, we can utilize the MATLAB solvers to predict the mechanical response.

For simplicity, we shall use the definition (12.10.2) and write the differential equation (12.11.1) in the form

$$\frac{d^2u}{dt^2} + \left(\frac{c}{m} + \frac{e}{m} \left(\frac{du}{dt} \right)^2 \right) \frac{du}{dt} + \omega_0^2 u = 0 \quad (12.11.2)$$

The normal form of (12.11.2) and the initial conditions (12.11.1)₂ are given by

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ -\left(\frac{c}{m} + \frac{e}{m} x_2(t)^2 \right) x_2(t) - \omega_0^2 x_1(t) \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} \quad (12.11.3)$$

Equation (12.11.3) is the nonlinear generalization of our earlier result (12.10.10). This differential equation is defined be the function m-file **f12111.m** whose script is

```
function dxdt=f12111(t,x,k,c,e,m)
```

```
dxdt=zeros(2,1); %Preallocate
dxdt=[x(2);-k/m*x(1)-c/m*x(2)-e/m*x(2)^3];
```

For simplicity, the function m-file f12111.m has been structured so that the four constants **k**, **c**, **e** and **m** are individually prescribed. Other approaches are possible. For the purposes of this example, we shall adopt the same numerical values utilized in Example 12.10.1, namely,

$$m = 20 \text{ kg}, k = 20 \text{ kg/sec}^2 \quad \text{and} \quad c = 5 \text{ kg/sec} \quad (12.11.4)$$

along with the value

$$e = 25 \text{ kg sec/m}^2 \quad (12.11.5)$$

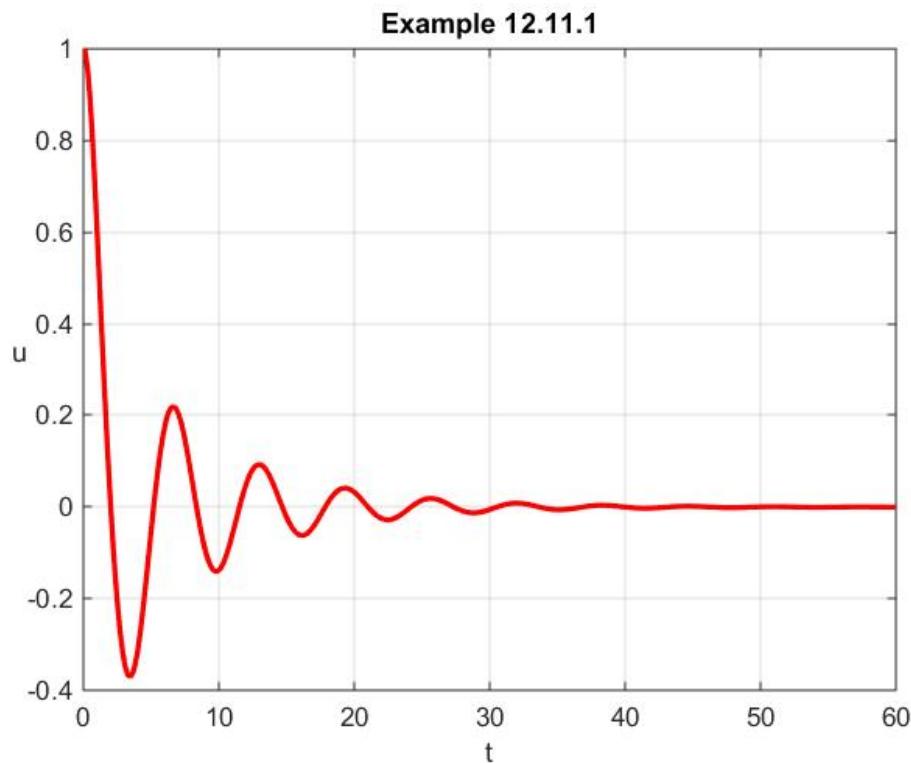
for the coefficient of the nonlinear term. In addition, we shall adopt the initial condition also utilized in Example 12.10.1

$$\begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (12.11.6)$$

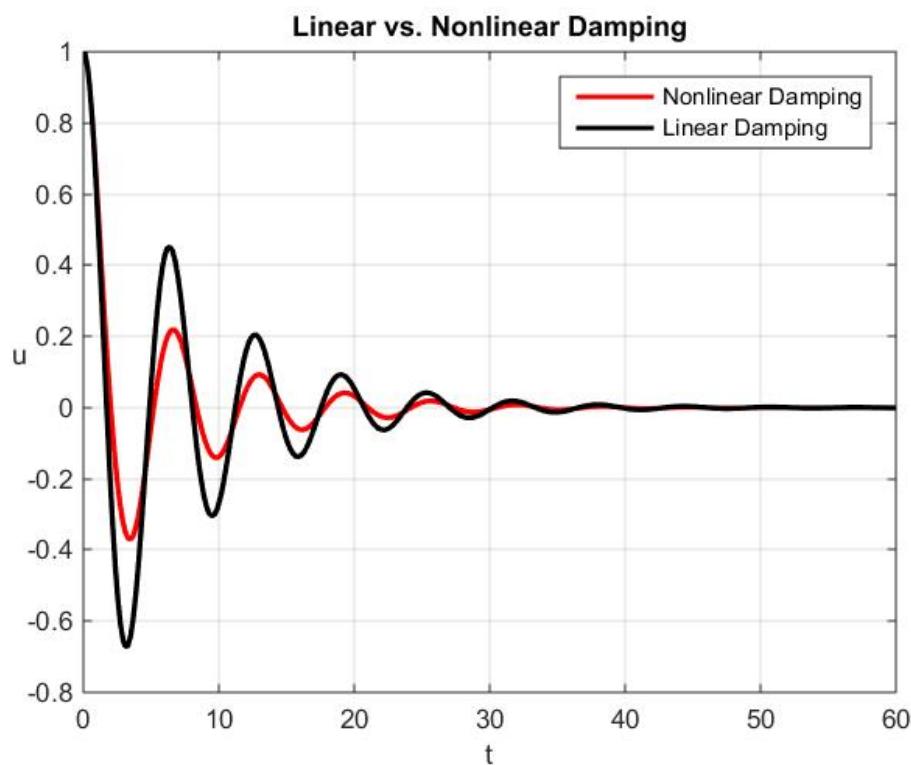
The script

```
clc
clear
m=20
k=20
c=5
e=25
u0=1
v0=0
x0=[u0,v0];
tspan=[0,60];
[t,x]=ode45(@f12111,tspan,x0,[],k,c,e,m);
plot(t,x(:,1),'r','LineWidth',2)
grid on
xlabel('t')
ylabel('u','Rotation',0)
title('Example 12.11.1')
axis([0,60,-.4,1])
```

generates the figure



It is interesting to plot the solution for the linear dashpot obtained in Example 12.10.1 superimposed upon the one just obtained. This linear case can be achieved by placing $\epsilon = 0$ in the above script



Thus, the nonlinear dashpot causes greater damping than does the linear one for the case being discussed.

Example 12.11.2: This example is a nonlinear differential equation known as Duffing's Equation.^{51,52} It takes the form

$$m \frac{d^2u(t)}{dt^2} + c \frac{du(t)}{dt} + ku(t) + qu(t)^3 = f(t) \text{ and } u(0) = u_0, \frac{du(0)}{dt} = v_0 \quad (12.11.7)$$

This ordinary differential equation models a forced vibration problem with *linear damping* and where the spring is a *nonlinear spring* whose force-displacement relationship is

$$F_{\text{spring}} = -k \left(1 + \frac{q}{k} u^2 \right) u \quad (12.11.8)$$

The parameter q is an *extra material property* that must be prescribed. If $q > 0$ the spring is *hard*, and if $q < 0$ the spring is *soft*.

The *normal form* of (12.11.7) is

$$\frac{d}{dt} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{k}{m}x_1 - \frac{q}{m}x_1^3 - \frac{c}{m}x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m}f(t) \end{bmatrix} \text{ and } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} \quad (12.11.9)$$

If we set the parameter q to zero, then this model becomes the linear one discussed in Example 12.10.2. In this case, we shall adopt the numerical values

$$m = 20 \text{ kg}, k = 320 \text{ kg/sec}^2 \text{ and } c = 20 \text{ kg/sec} \quad (12.11.10)$$

along with the value

$$q = 20 \text{ kg/sec}^2 \text{ m}^2 \quad (12.11.11)$$

For the forcing function, we shall take

$$f(t) = 800 \cos(6t) \quad (12.11.12)$$

This differential equation (12.11.9)₁ is defined by the function m-file **f12112.m** whose script is

⁵¹ G. Duffing, "Erzwungene Schwingungen bei veränderlicher Eigenfrequenz und ihre technische Bedeutung" , Vieweg (1918)

⁵² Information about the German engineer and inventor Georg Duffing can be found at <http://www.atomosyd.net/spip.php?article97>.

```

function dxdt=f12112(t,x,k,q,c,m,g)
dxdt=zeros(2,1); %Preallocate
dxdt=[x(2);-k/m*x(1)-c/m*x(2)-q/m*x(1)^3]+g(t);
%g an anonymous function

```

We shall actually generate the solution for Duffing's equation for several different initial conditions. For our first case, we shall assume the initial condition is

$$\begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (12.11.13)$$

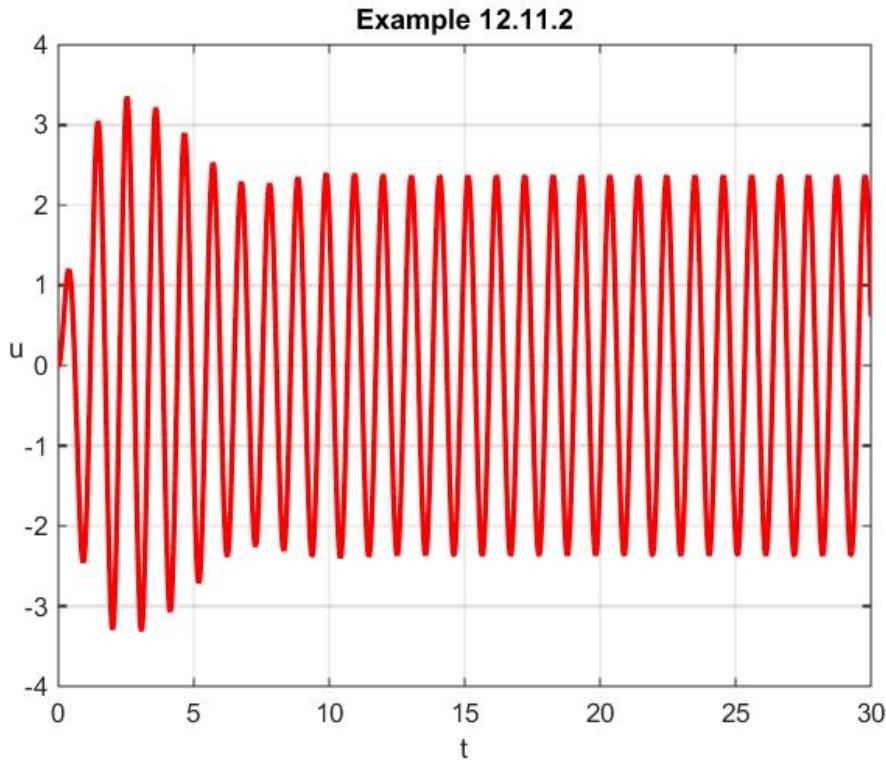
The script that will generate a plot of the solution for $u(t) = x_1(t)$ is

```

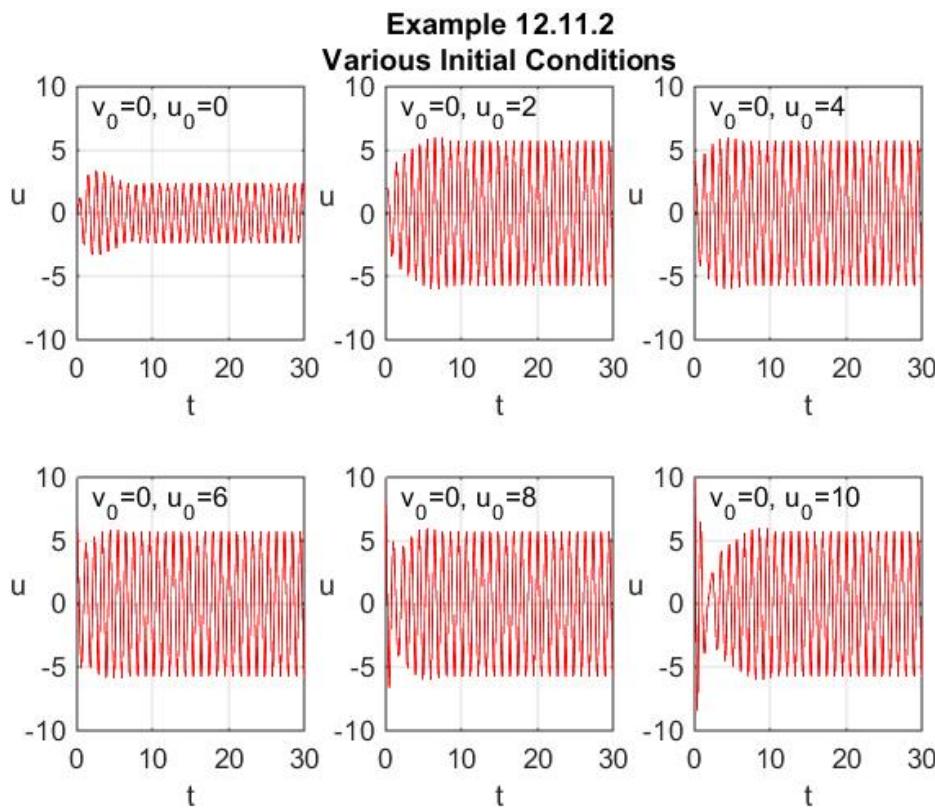
clc
clear
m=20
k=320
c=20
q=20
w=6
g=@(t)([0;800*cos(w*t)/m]);
u0=0
v0=0
x0=[u0,v0];
tspan=[0,30];
[t,x]=ode45(@f12112,tspan,x0,[],k,q,c,m,g);
plot(t,x(:,1),'r','LineWidth',2)
grid on
xlabel('t')
ylabel('u','Rotation',0)
title('Example 12.11.2')

```

The resulting plot is



An interesting aspect of the *linear* forced vibrations problem such as illustrated in Example 12.10.2, is that the steady state solution, i.e. the long time solution, does not depend upon the initial conditions. This property can be seen from the analytical solution (12.10.21). It turns out that this feature is slightly more complicated for the kind of nonlinear forced vibrations problem illustrated by (12.11.9). If we continue to adopt (12.11.10), (12.11.11) and (12.11.12) and generate approximate solutions of (12.11.9) for the initial conditions $u_0 = 0, 2, 4, 6, 8$ and 10 while holding $v_0 = 0$ the results are



Interestingly, for these six initial conditions, the steady state solution appear to be essentially the same *except* for the one starting from $(u_0, v_0) = (0, 0)$. If one experiments with various initial conditions, it appears that the initial conditions roughly in the range $0 \leq u_0 \leq .84$ produce a steady state solution like the first one shown in the figures above. For an initial condition of approximately $u_0 = .85$ and larger, the steady state solution looks like the other figures above. Of course, the details above also depend upon the choices (12.11.10), (12.11.11) and (12.11.12).

Exercises

12.11.1: You are given the ordinary differential equation

$$\frac{d^3u}{dt^3} + 2\frac{d^2u}{dt^2} + \left(\frac{du}{dt}\right)^2 + 3u = 0 \quad (12.11.14)$$

Utilize MatLab to solve this ordinary differential equation in the interval $5 \leq t \leq 12$ subject to the initial conditions

$$u(5) = 1, \frac{du(5)}{dt} = 0 \quad \text{and} \quad \frac{d^2u(5)}{dt^2} = 0 \quad (12.11.15)$$

12.11.2: You are given the ordinary differential equation

$$\frac{d^2u}{dt^2} + 5u \frac{du}{dt} + (u + 7)\sin(t) = 0 \quad (12.11.16)$$

Utilize MatLab to solve this ordinary differential equation in the interval $0 \leq t \leq 15$ subject to the initial conditions

$$u(0) = 6 \quad \text{and} \quad \frac{du(0)}{dt} = 1.5 \quad (12.11.17)$$

12.11.3: A certain model of an object in freefall subject to drag results in the following coupled nonlinear ordinary differential equations:

$$m \frac{d^2x}{dt^2} = -c_d \frac{dx}{dt} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} \quad (12.11.18)$$

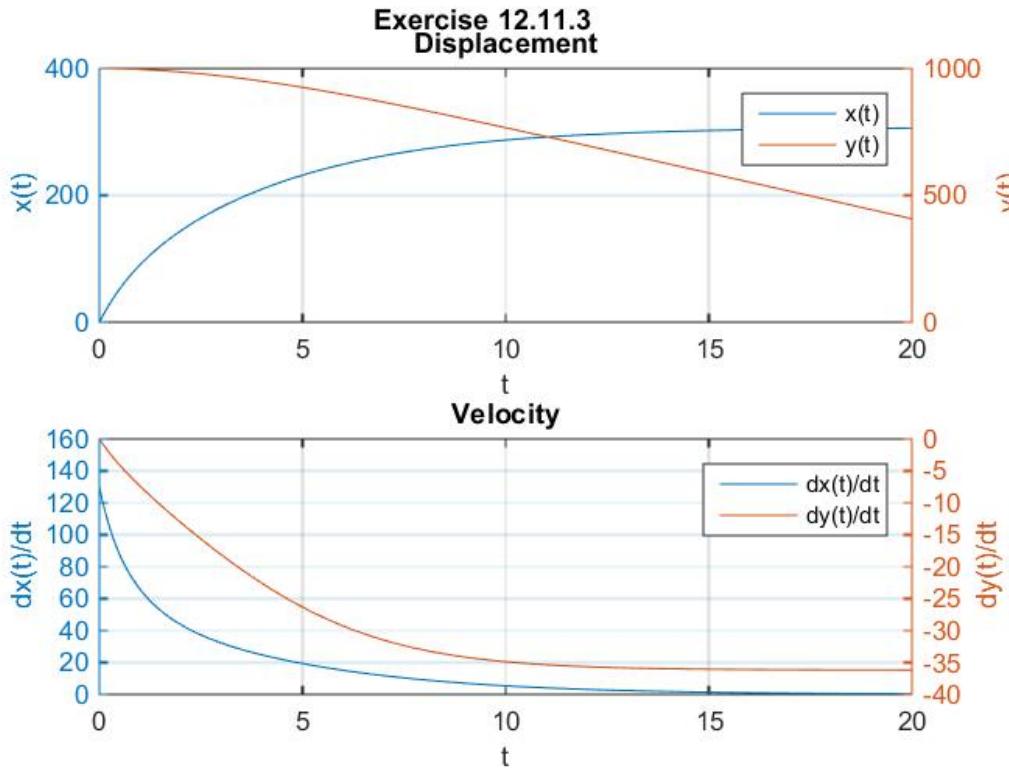
and

$$m \frac{d^2y}{dt^2} = -c_d \frac{dy}{dt} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} - mg \quad (12.11.19)$$

If m , the mass of the object, is 80 kg, c_d , the drag coefficient, is $6.0 \text{ Nsec}^2/\text{m}^2$ and g , the gravitational constant, is 9.81 m/sec^2 , utilize MATLAB to find an approximate solution in the interval $t \in [0, 10]$ subject to the initial conditions

$$\begin{aligned} x(0) &= 0 & \frac{dx(0)}{dt} &= 130 \text{ m/sec} \\ y(0) &= 1000 \text{ m} & \frac{dy(0)}{dt} &= 0 \end{aligned} \quad (12.11.20)$$

The correct solution should produce curves like the following



12.11.4: The model of Exercise 12.11.2 is modified to include the presence of a bungee cord of length L and linear spring constant k which tethers the object to its initial position. The bungee cord is accommodated by modifying (12.11.18) and (12.11.19), respectively to

$$m \frac{d^2x}{dt^2} = \begin{cases} -c_d \frac{dx}{dt} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} & \text{for } \sqrt{(x-x(0))^2 + (y-y(0))^2} \leq L \\ -c_d \frac{dx}{dt} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} \\ \quad -k \left(\sqrt{(x-x(0))^2 + (y-y(0))^2} - L \right) \frac{x-x(0)}{\sqrt{(x-x(0))^2 + (y-y(0))^2}} & \text{for } \sqrt{(x-x(0))^2 + (y-y(0))^2} > L \end{cases} \quad (12.11.21)$$

and

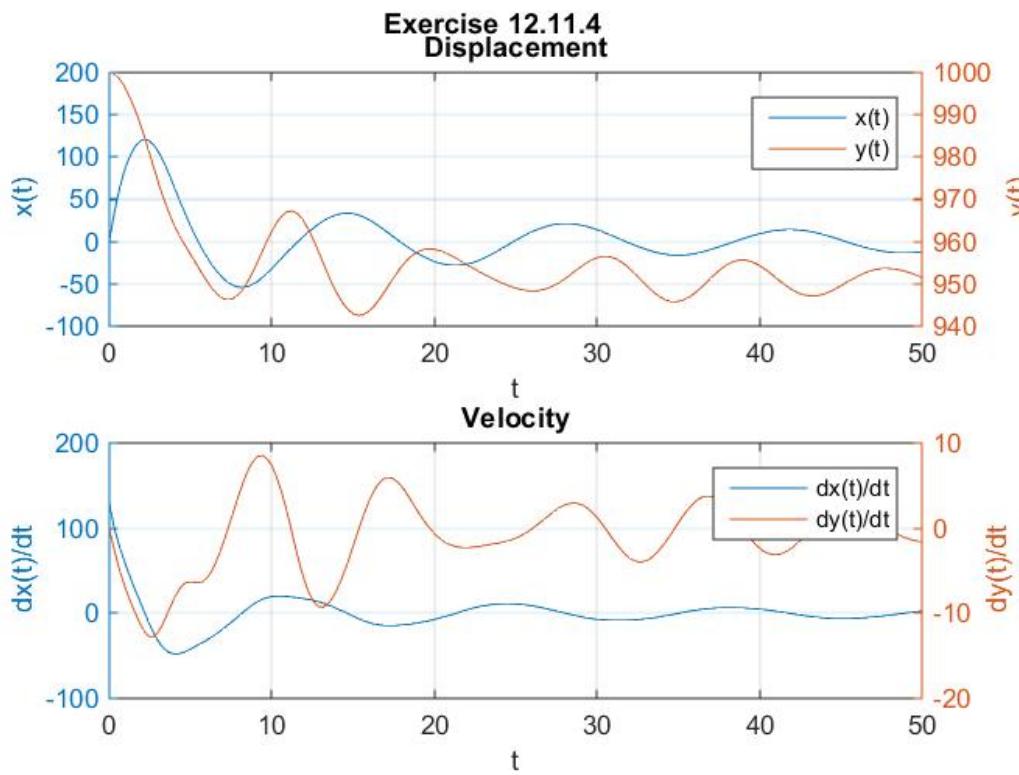
$$m \frac{d^2y}{dt^2} = \begin{cases} -c_d \frac{dy}{dt} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} - mg & \text{for } \sqrt{(x-x(0))^2 + (y-y(0))^2} \leq L \\ -c_d \frac{dy}{dt} \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} - mg \\ -k \left(\sqrt{(x-x(0))^2 + (y-y(0))^2} - L \right) \frac{y - y(0)}{\sqrt{(x-x(0))^2 + (y-y(0))^2}} & \text{for } \sqrt{(x-x(0))^2 + (y-y(0))^2} > L \end{cases} \quad (12.11.22)$$

If m , the mass of the object, is 80 kg, c_d , the drag coefficient, is 6.0 Nsec²/m², L , the bungee length, is 30 m, k , the bungee spring constant, is 40 N/m and g , the gravitational constant, is 9.81 m/sec², utilize MATLAB to find an approximate solution in the interval $t \in [0,10]$ subject to the initial conditions

$$\begin{aligned} x(0) &= 0 & \frac{dx(0)}{dt} &= 130 \text{ m/sec} \\ y(0) &= 1000 \text{ m} & \frac{dy(0)}{dt} &= 0 \end{aligned} \quad (12.11.23)$$

This problem is best worked if one utilizes in the file defining the system of differential equations the logical expression `<` and the **if-else-end** construct. Both of these concepts are discussed in Section A.8 of Appendix A.

The correct solution should produce curves like the following

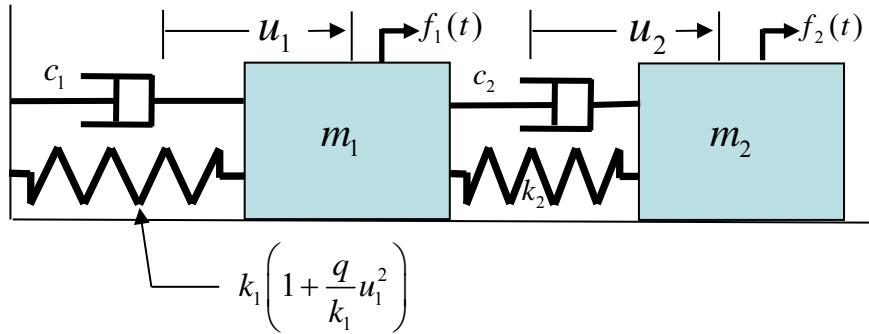


12.11.5: A projectile is launched vertically from the earth's surface. On the assumption that the only force acting on the projectile is gravity, the equation of motion can be shown to be

$$\frac{d^2y}{dt^2} = -g \frac{R^2}{(R+y)^2} \quad (12.11.24)$$

where $\frac{dy}{dt}$ is the vertical velocity (m/s), t is the time (s), y is the altitude (m) measured upwards from the earth's surface, g is the gravitational acceleration at the earth's surface (9.81m/s^2), and R =the earth's radius (6.37×10^6 m). Use one of the MATLAB numerical solvers and determine the maximum height that the projectile will reach if launched with an initial velocity of 1400 m/s

12.11.6: The figure for Exercise 12.1.2 shows a coupled two degree of freedom nonlinear vibrating system. The figure of that exercise is repeated as follows:



The first spring is nonlinear with the force-displacement relationship shown. The equations of motion for this system are given by equation (12.1.31), repeated,

$$\begin{aligned} m_1 \ddot{u}_1 &= -c_1 \dot{u}_1 - k_1 \left(1 + \frac{q}{k_1} u_1^2 \right) u_1 + c_2 (\dot{u}_2 - \dot{u}_1) + k_2 (u_2 - u_1) + f_1(t) \\ m_2 \ddot{u}_2 &= -c_2 (\dot{u}_2 - \dot{u}_1) - k_2 (u_2 - u_1) + f_2(t) \end{aligned} \quad (12.11.25)$$

where \$m_1\$ and \$m_2\$ are the two masses, \$k_1\$, \$k_2\$ and \$q\$ are spring constants, \$c_1\$ and \$c_2\$ are damping constants and \$f_1(t)\$ and \$f_2(t)\$ are forcing functions.

Adopt the following values for the above constants and forcing functions:

$$\begin{aligned} m_1 = m_2 &= 1, k_1 = k_2 = 1, q = .1, c_1 = c_2 = .1 \\ f_1(t) &= 0, f_2(t) = 20 \cos(2t) \end{aligned} \quad (12.11.26)$$

and solve the ordinary differential equations (12.11.25) over the interval \$[0, 50]\$ subject to the initial conditions \$u_1(0) = 2\$, \$\frac{du_1(0)}{dt} = 0\$, \$u_2(0) = -2\$ and \$\frac{du_2(0)}{dt} = 0\$. Display your solution in the form of a plot of the two displacements as a function of time.

Section 12.12. Forced Vibrations of Nonlinear Pendulum with Damping

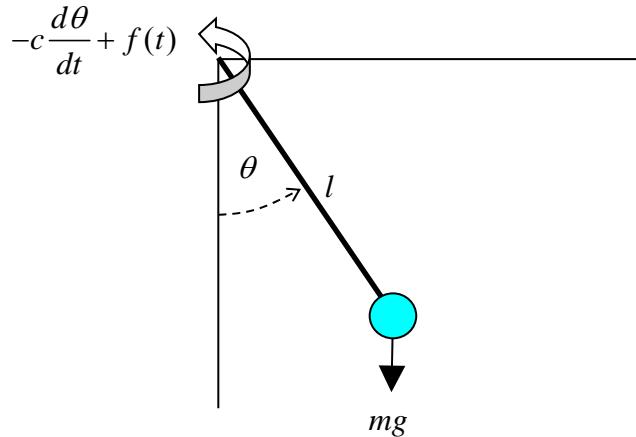
An interesting and important nonlinear ordinary differential equation is

$$m \frac{d^2 u}{dt^2} + c \frac{du}{dt} + k \sin u = f(t) \quad (12.12.1)$$

subject to the usual initial conditions

$$u(0) = u_0, \frac{du(0)}{dt} = v_0 \quad (12.12.2)$$

The physical situation where this equation and the associated initial value problem occurs is the modeling of a *damped pendulum* subjected to a *forcing function*. The usual geometric arrangement for the motion of a pendulum is as follows:



where $-c \frac{d\theta}{dt}$ is a *damping moment* applied at the axis of rotation proportional to the angular velocity, $\frac{d\theta}{dt}$, $f(t)$ is an *external moment* applied at the axis of rotation and mg is the weight of the pendulum. Elementary dynamics tells us that the equation of motion in this case is

$$ml^2 \frac{d^2 \theta}{dt^2} = -mgl \sin \theta - c \frac{d\theta}{dt} + f(t) \quad (12.12.3)$$

It is convenient to rearrange (12.12.3) into the form

$$\frac{d^2\theta}{dt^2} + \frac{c}{ml^2} \frac{d\theta}{dt} + \frac{g}{l} \sin \theta = \frac{1}{ml^2} f(t) \quad (12.12.4)$$

Frequently, equation (12.12.4) is solved by adopting the assumption that the angle θ is small. In this case, the approximation $\sin(\theta) \approx \theta$ is adopted and (12.12.4) becomes a linear ordinary differential equation. When this approximation is adopted, solutions of (12.12.4) behave like the single degree of freedom forced vibrations problem discussed in Example 12.10.2. We shall see in this section that the nonlinear equation (12.12.4) displays features that are substantially more complicated than one finds for the linear case.⁵³

As usual, we need to express this initial value problem in its equivalent normal form. The result is

$$\frac{d}{dt} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{g}{l} \sin x_1 - \frac{c}{ml^2} x_2 + \frac{1}{ml^2} f(t) \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} \quad (12.12.5)$$

where

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \theta \\ \frac{d\theta}{dt} \end{bmatrix} \quad (12.12.6)$$

In order to implement a MATLAB solution, we will use the function m-file **f12121.m** with the script

```
function dxdt=f12121(t,x,m,l,c,g,f)
dxdt=zeros(2,1); %Preallocate
dxdt=[x(2);-g/l*sin(x(1))-c/m/l^2*x(2)]+[0;f(t)]/m/l^2;
%f an anonymous function
```

to define the differential equation (12.12.5)₁.

Example 12.12.1: As our first numerical example, consider the unforced case where⁵⁴

⁵³ The differential equation (12.12.4) is discussed in the textbook, Ordinary Differential Equations using MATLAB by John C. Polking and David Arnold, published by Pearson Prentice Hall. As explained by Polking and Arnold, additional discussion of this example can be found in Borrelli, R., C. S. Coleman, Computers, Lies, and the Fishing Season, College Math Journal, Vol. 25, No. 5, pp.401-412, 1994 and Hubbard, J. C., The Forced Damped Pendulum: Chaos, Complication and Control, The American Mathematical Monthly, Vol. 106, No. 8, pp. 741-758, 1999. Hubbard points out that solutions of (12.12.4) can exhibit complicated and unstable behavior that at first glance appears contradictory.

⁵⁴ The numbers selected for m, l, c and the gravitational constant g result in the left side of the differential equation (12.12.4) reducing to

$$\begin{aligned}
 f(t) &= 0 \\
 m &= 20 \text{ kg} \\
 l &= 9.81 \text{ m} \\
 c &= 2(9.81)^2 \text{ kg/sec}^2 \\
 g &= 9.81 \text{ m/sec}^2
 \end{aligned} \tag{12.12.7}$$

Also, we shall begin the discussion by adopting the initial condition

$$\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} .9\pi \\ 0 \end{bmatrix} \tag{12.12.8}$$

In this case the pendulum is held with the mass almost vertical and released with zero angular velocity. We shall generate and plot the solution over the time interval $0 < t < 50$. The script sufficient to graph the displacement vs. time for the initial conditions (12.12.8) and the values (12.12.7) is as follows:

```

clc
clear
m=20
l=9.81
c=2*(9.81)^2
g=9.81
f=@(t)0
tspan=[0,50]
u0=9*pi/10
v0=0
x0=[u0,v0]
[t,x]=ode45(@f12121,tspan,x0,[],m,l,c,g,f);
plot(t,x(:,1),'r','LineWidth',2)
hold on
axis([0, 50,-pi,pi])
grid on
xlabel('t')
ylabel('\theta (t)', 'Rotation', 0)
set(gca,'YTick',[-pi:pi/3:pi],...
'YTickLabel',{'-\pi','-2\pi/3',...

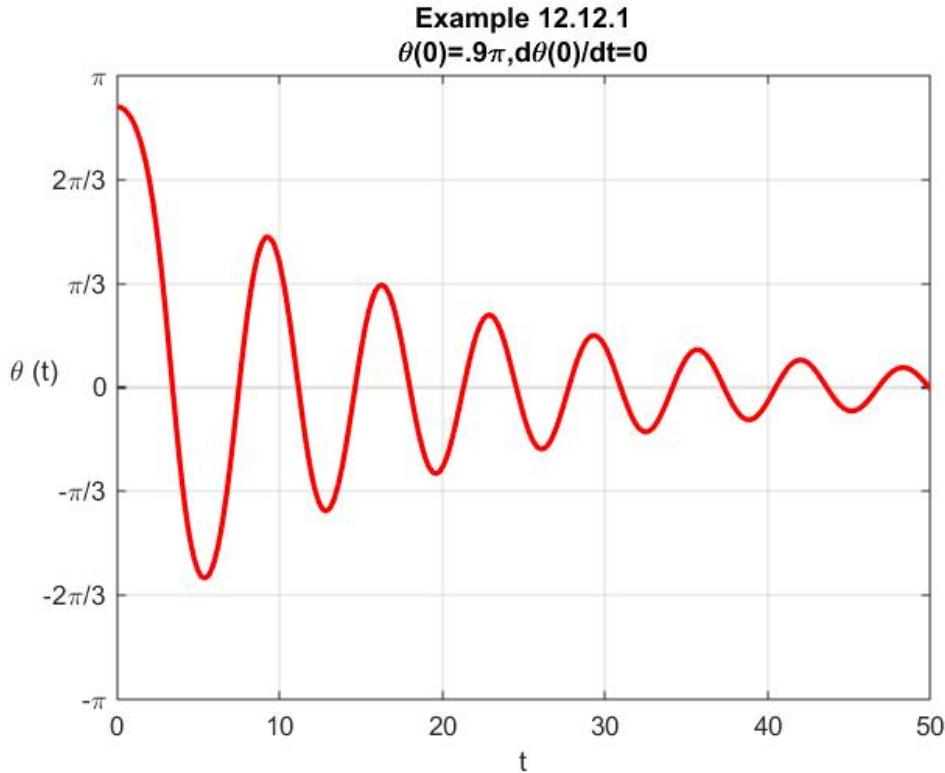
```

$$\frac{d^2\theta}{dt^2} + \frac{1}{10} \frac{d\theta}{dt} + \sin \theta = \frac{1}{ml^2} f(t)$$

Later, when we make the choice $\frac{1}{ml^2} f(t) = \cos(t)$, our numbers will reduce our discussion to the same differential equation discussed in the references given in the last footnote.

```
'-\pi/3','0','\pi/3','2\pi/3','\pi'})  
title({'Example 2.12.1','\theta(0)=.9\pi,d\theta(0)/dt=0'})
```

The result turns out to be the damped oscillation about the angle $\theta = 0$ one would more or less expect



If one desires an animation that shows the evolution of the above figure along with the corresponding motion of the pendulum, the following script will create the two figures and the animation⁵⁵

```
clc  
clear  
m=20  
l=9.81
```

⁵⁵ The additional script appended to that above

```
vid=VideoWriter('Example12121.mp4','MPEG-4')  
open(vid)  
writeVideo(vid,F)  
close(vid)
```

will produce a video file **Example12121.mp4** of the animation. This video can be viewed from the electronic version of Appendix B of this work.

```
c=2*(9.81)^2
g=9.81
c/m/l^2
f=@(t)0
tspan=[0,50]
u0=9*pi/10
v0=0
x0=[u0,v0]
[t x] = ode45(@f12121,tspan,x0,[],m,l,c,g,f);

subplot(1,2,1)
hold on
xlabel('t')
ylabel('\theta (t)', 'Rotation', 0)
grid on
title('Example 12.12.1')
set(gca,'YTick',[-pi:pi/3:pi],...
    'YTickLabel',{'-\pi','-2\pi/3',...
    '-\pi/3','0','\pi/3','2\pi/3','\pi'})
axis([0,tspan(2),-pi,pi])

subplot(1,2,2)
hold on
axis equal
axis([-1.1*(l) 1.1*(l) -1.1*(l) 1.1*(l)]);
axis off;
y1=-l*cos(x(1,1));
x1=l*sin(x(1,1));
h1=plot([0 x1],[0 y1],'k','LineWidth',2);
h2=plot(x1,y1,'o','MarkerFaceColor','r',...
    'MarkerEdgeColor','k','MarkerSize',15);
plot(0,0,'^','MarkerFaceColor','b',...
    'MarkerEdgeColor','k','MarkerSize',5)

for i=1:length(t)
    subplot(1,2,1)
    plot(t(1:i),x(1:i,1),'r','LineWidth',2);
    hold on;
    subplot(1,2,2)
    delete(h1)
    delete(h2)
    y1=-l*cos(x(i,1)); x1=l*sin(x(i,1));
    h1=plot([0 x1],[0 y1],'k','LineWidth',2);
    h2=plot(x1,y1,'o','MarkerFaceColor','r',...
        'MarkerEdgeColor','k','MarkerSize',15);
    title(['t = ' num2str(floor(t(i)))])
```

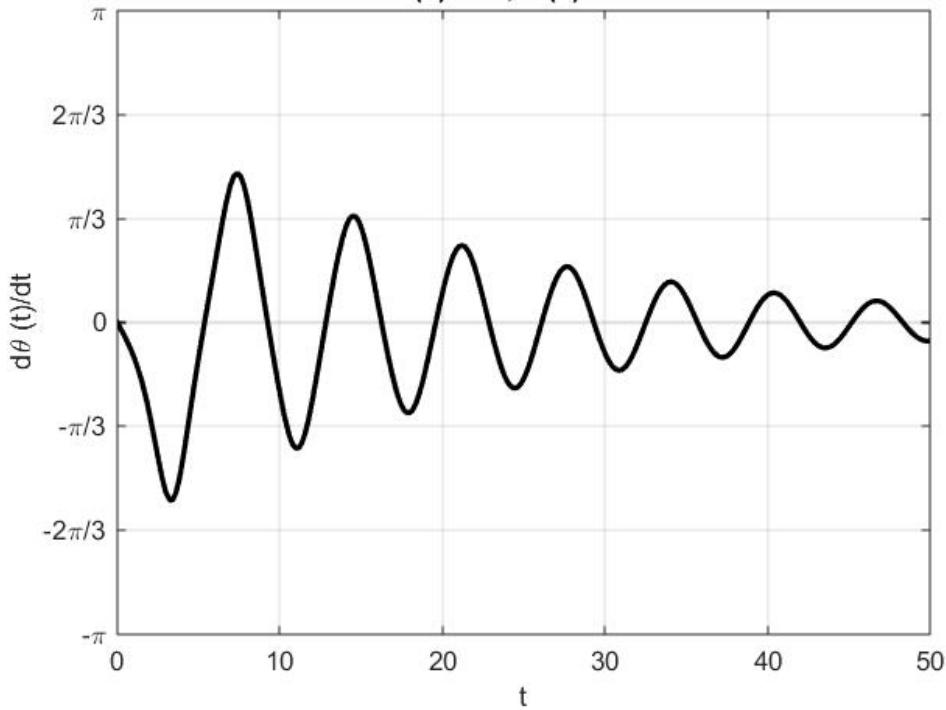
```
F(i)=getframe(gcf)
end
```

If, for this example, one were interested in a plot of the angular velocity, $\frac{d\theta}{dt}$, one simply inserts the script

```
plot(t,x(:,2),'k','linewidth',2)
```

in place of the line `plot(t,x(:,1),'r','linewidth',2)` and makes the obvious change in the y axis label. The result is

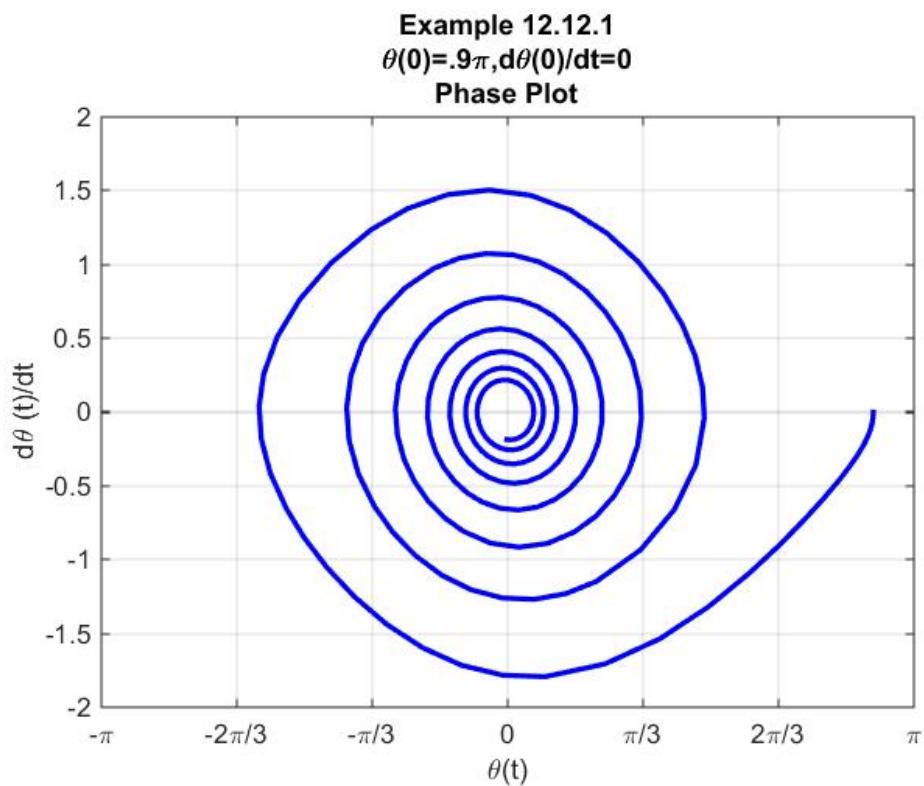
Example 12.12.1
 $\theta(0) = .9\pi, d\theta(0)/dt = 0$



In certain applications, important information can be extracted from the *phase plot*. In our example, the phase plot is a plot of $\frac{d\theta}{dt}$ vs. θ . The independent variable t is an implicit variable in the plot. This kind of plot is obtained by creating a plot of the second column of the output **x** of the script `[t x] = ode45(@f12121,tspan,x0,[],m,l,c,g,f)` vs. its first column. In other words, generate the plot from the script

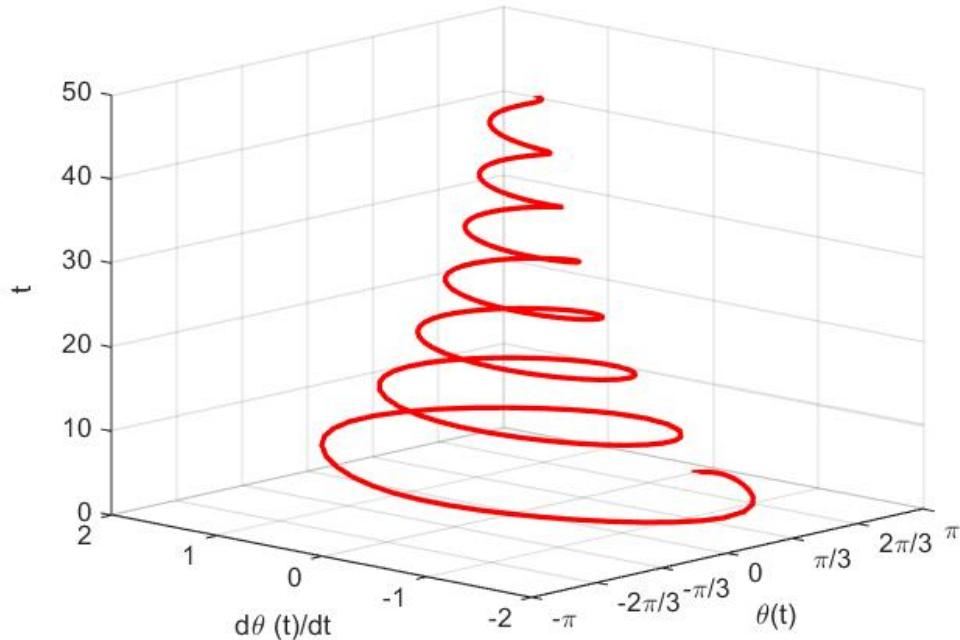
```
plot(x(:,1),x(:,2),'b','linewidth',2)
```

It should not be difficult to show that the resulting phase plot is



The above plots can be combined into a three dimensional plot where the axes are $\theta(t), \frac{d\theta(t)}{dt}$ and t . The resulting figure in this case turns out to be

Example 12.12.1
 $\theta(0)=0.9\pi, d\theta(0)/dt=0$
Three Dimensional Phase Plot



The script that produces this result is

```

clc
clear
m=20
l=9.81
c=2*(9.81)^2
g=9.81
c/m/l^2
f=@(t)
tspan=[0,50]
u0=9*pi/10
v0=0
x0=[u0,v0]
[t,x]=ode45(@f12121,tspan,x0,[],m,l,c,g,f);
plot3(x(:,1),x(:,2),t,'r','LineWidth',2)
view(-47,16)
axis([-pi, pi,-2,2,0,50])
grid on
xlabel('\theta(t)')
ylabel('d\theta (t)/dt')
zlabel('t')
set(gca,'XTick',[-pi:pi/3:pi],...
    'XTickLabel',{'-\pi','-2\pi/3',...
    '-\pi/3','0','\pi/3','2\pi/3','\pi'})

```

```
title({'Example 12.12.1',...
    '\theta(0)=.9\pi,d\theta(0)/dt=0',...
    'Three Dimensional Phase Plot'})
```

As the script indicates, the key command that creates the plot is

```
plot3(x(:,1),x(:,2),t,'r','linewidth',2)
```

Example 12.12.2: In this example, we shall modify the last one by the introduction of a forcing function such that

$$\frac{1}{ml^2} f(t) = \cos \omega t \quad (12.12.9)$$

where ω is a given forcing frequency. This case reduces the differential equation (12.12.4) to

$$\frac{d^2\theta}{dt^2} + \frac{c}{ml^2} \frac{d\theta}{dt} + \frac{g}{l} \sin \theta = \cos \omega t \quad (12.12.10)$$

It is the differential equation (12.12.10) that exhibits the interesting contradictory behavior mentioned in the footnote above. It will aid our discussion if we observe the *equilibrium solutions* for the damped pendulum defined by (12.12.10). An equilibrium solution, of course, is a solution $\theta(t) = \text{constant}$ that satisfies the zero external force version of (12.12.10). It should be evident that the equilibrium solutions are

$$\theta(t) = 0, \pm 2\pi, \pm 4\pi, \dots \quad (12.12.11)$$

and

$$\theta(t) = 0, \pm \pi, \pm 3\pi, \dots \quad (12.12.12)$$

The equilibrium solutions (12.12.11) correspond to the pendulum hanging vertical and are stable in the sense that a small perturbation from these solutions will return to the equilibrium values. Likewise, the equilibrium solutions (12.12.12) correspond to the pendulum pointing upward and are unstable in the sense that a small perturbation from these solutions will produce a large rotation.

Finally, the choice of the forcing frequency is important to our discussion. If our pendulum had been linear, its natural frequency would be the number $\sqrt{\frac{g}{l}}$. In this linear case, we would induce resonance with the choice $\omega = \sqrt{\frac{g}{l}}$.

If we adopt the material constants used in Example 12.12.1, namely,

$$\begin{aligned} m &= 20 \text{ kg} \\ l &= 9.81 \text{ m} \\ c &= 2(9.81)^2 \text{ kg/sec}^2 \\ g &= 9.81 \text{ m/sec}^2 \end{aligned} \tag{12.12.13}$$

the differential equation (12.12.10) reduces to

$$\frac{d^2\theta}{dt^2} + \frac{1}{10} \frac{d\theta}{dt} + \sin \theta = \cos \omega t \tag{12.12.14}$$

Also, we shall adopt the initial condition

$$\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \tag{12.12.15}$$

The results for a nonlinear differential equation, for example, such as (12.12.10) will depend upon the material constants, $\frac{g}{l}$ and $\frac{c}{ml^2}$, the frequency of the forcing function, ω , and the initial conditions u_0 and v_0 . The material constants and the initial conditions in this example correspond to those utilized in the above references. Our purpose in this example is to attempt to generate approximate solutions of the initial value problem defined by (12.12.14) and (12.12.15) for several different forcing frequencies. A point that this example will illustrate is that the default value of the parameter **RelTol** = 10^{-3} , discussed in Section 12.8, does not result in reliable approximate answers. This conclusion will be illustrated in a brute force fashion by simply building the approximate solutions for various values of **RelTol** and the forcing frequency ω .

At the risk of trying to put too much information on our figures, we shall approach the problem by utilizing the solver **ode45** to generate approximate solutions to the initial value problem defined by (12.12.14) and (12.12.15) for three different forcing frequencies. Each approximate solution will be shown on a plot of $\theta(t)$ vs. t for the interval $0 < t < 150$. On each plot there will be four curves, each corresponding to a different choice of **RelTol**. This construction will be repeated three times for a total of nine forcing frequencies. The MATLAB script that will generate these sets of three plots is⁵⁶

```
clc
clear
m=20
l=9.81
```

⁵⁶ The script utilizes the MATLAB function **cellfun**. Information about this function can be found at <http://www.mathworks.com/help/matlab/ref/cellfun.html>.

```

c=2*(9.81)^2
g=9.81
tspan=[0,150]
u0=0
v0=2
x0=[u0,v0]
%List of values of RelTol to be utilized.
%Note 2.3*10^(-14) is near the minimum allowed RelTol
for n=1:11
    options(n)=odeset('RelTol',10^(-(n+2)));
end
options(12)=odeset('RelTol',2.3*10^(-(14)));
%List of values of forcing frequency to be utilized
W=[.4,.5,.6,.8,.9,1,1.1,1.2,1.3];

%Define linestyles and colors to be used in subplots
lines=char('--','---','-.',':');
color='rbgk'
%Create labels for y axis of subplots
CellLabels=zeros(1,42); %Preallocate
for s=1:42
    CellLabels(s)=-38+2*(s-1);
end
CellLabels=cellfun(@num2str, num2cell(CellLabels),...
    'UniformOutput', false)
CellLabels=strcat(CellLabels,{'\pi'});
CellLabels(20)={'0'}

%Select three frequencies and four values RelTol
W1=[W(1),W(2),W(3)]
Options1=[options(1),options(2),options(3),options(4)]

for k=1:3
    w=W1(k)
    f=@(t)(cos(w*t)*m*l^2);
    for j=1:4
        subplot(1,3,k)
        [t,x]=ode45(@f12121,tspan,x0,Options1(j),m,l,c,g,f);
        plot(t, x(:,1),'color',color(j),...
            'linewidth',2,....
            'linestyle',lines(j,:))
        hold on
        grid on
        ylabel('\theta (t)')
    end
    %Assign frequency labels to titles of subplots
    title( strcat('\omega = ',num2str(W1(k))))

```

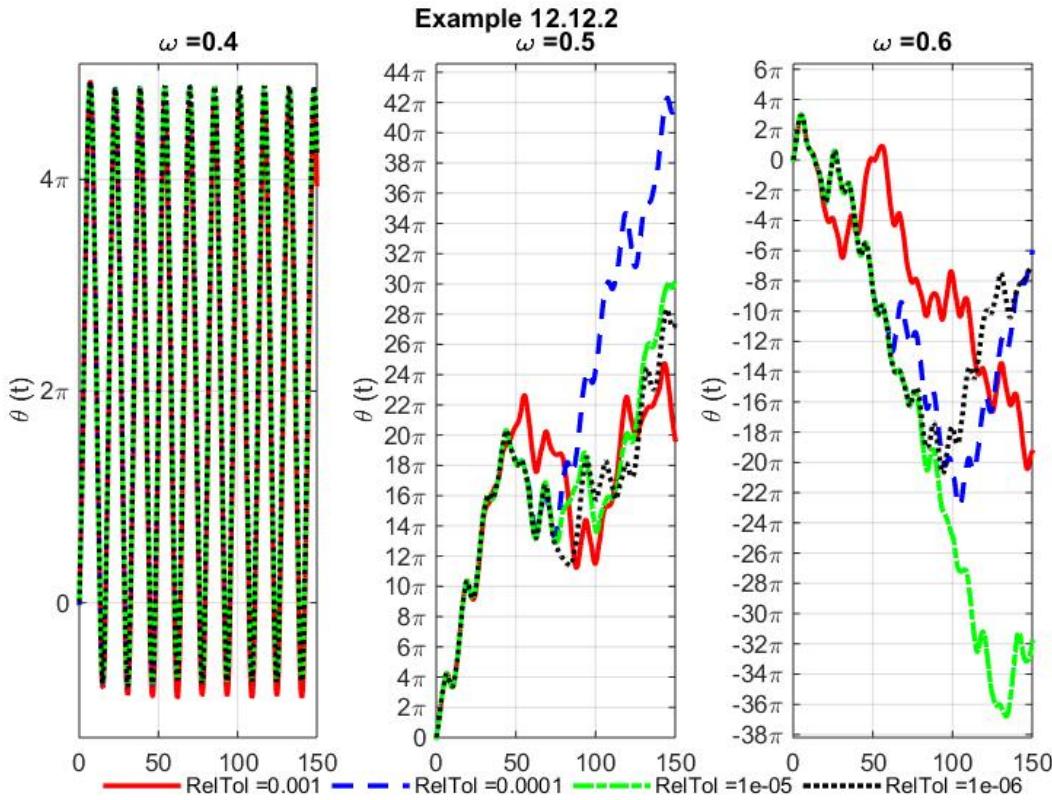
```

%Assign the YTicks and YTickLabels for each subplot.
set(gca,'YTick',[-38*pi:2*pi:44*pi],...
    'YTickLabel',CellLabels)
end
%hL is handle for the legend for the third figure
hL=legend( strcat('RelTol = ',num2str(Options1(1).RelTol)),...
    strcat('RelTol = ',num2str(Options1(2).RelTol)),...
    strcat('RelTol = ',num2str(Options1(3).RelTol)),...
    strcat('RelTol = ',num2str(Options1(4).RelTol)),...
    'Orientation','horizontal');
%The legend is placed with its box off below the three
%subplots.
newPosition = [0.48 0.0 0.1 0.1];
newUnits = 'normalized';
set(hL,'Position', newPosition,'Units',...
newUnits,'Box','off');
%Assign a title to three plots
subtitle('Example 12.12.2')

```

The comments inserted in the script attempt to explain how the script creates the figures. It is especially important to see how the options line assigns the values of **RelTol** and how the selected **RelTol** is utilized in calling the solver **ode45**.

The script above is structured to adopt the values $\text{RelTol} = 10^{-3}, 10^{-4}, 10^{-5}$ and 10^{-6} , and the forcing frequencies $\omega = .4, .5$ and $.6$. The following is the resulting figure



The first curve, the one corresponding to $\omega = .4$ displays a solution that does not appear to be improved by reducing **RelTol**. It does display a pendulum motion where the pendulum spins counter clockwise slightly more than two revolutions, turns around and spins clockwise the same two revolutions and then goes into a steady state that repeats the motion in what appears to be a period of approximately 15 seconds. Things get more complicated when the forcing frequency is increased to $\omega = .5$. For this frequency, the solutions for the four different values of **RelTol** agree for approximately 40 seconds and then diverge. During this approximately 40 seconds the pendulum spins counter clockwise for about 10 revolutions. After the solutions diverge, the four solutions do not even remain close. Thus, more numerical experimentation needs to be performed before one can have any confidence in the solution for $\omega = .5$. When the forcing frequency is increased to $\omega = .6$ we get solutions that show the pendulum rotating in opposite direction from the $\omega = .5$ case. In addition, after roughly 20 seconds the solutions for the four different values of **RelTol** diverge.

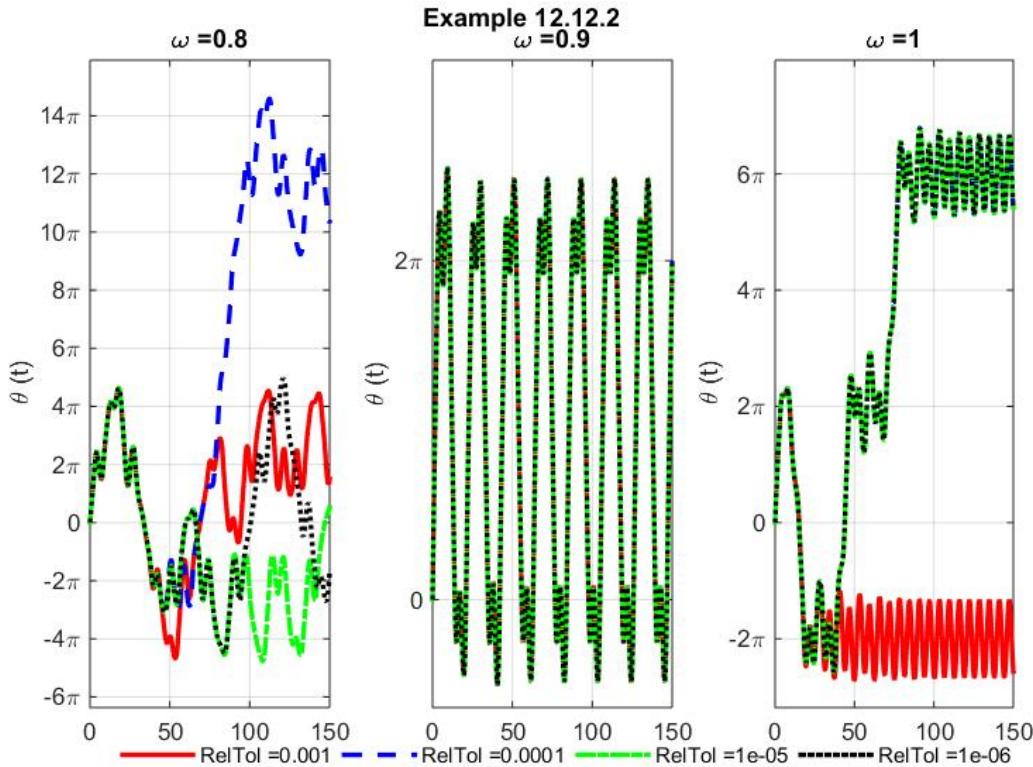
Next, we wish to continue with the four values of $\text{RelTol} = 10^{-3}, 10^{-4}, 10^{-5}$ and 10^{-6} , and generate the solutions for the three forcing frequencies $\omega = .8, .9$ and 1.0 . This new set of figures is achieved by replacing the script

```
W1=[W(1),W(2),W(3)]
```

with the script

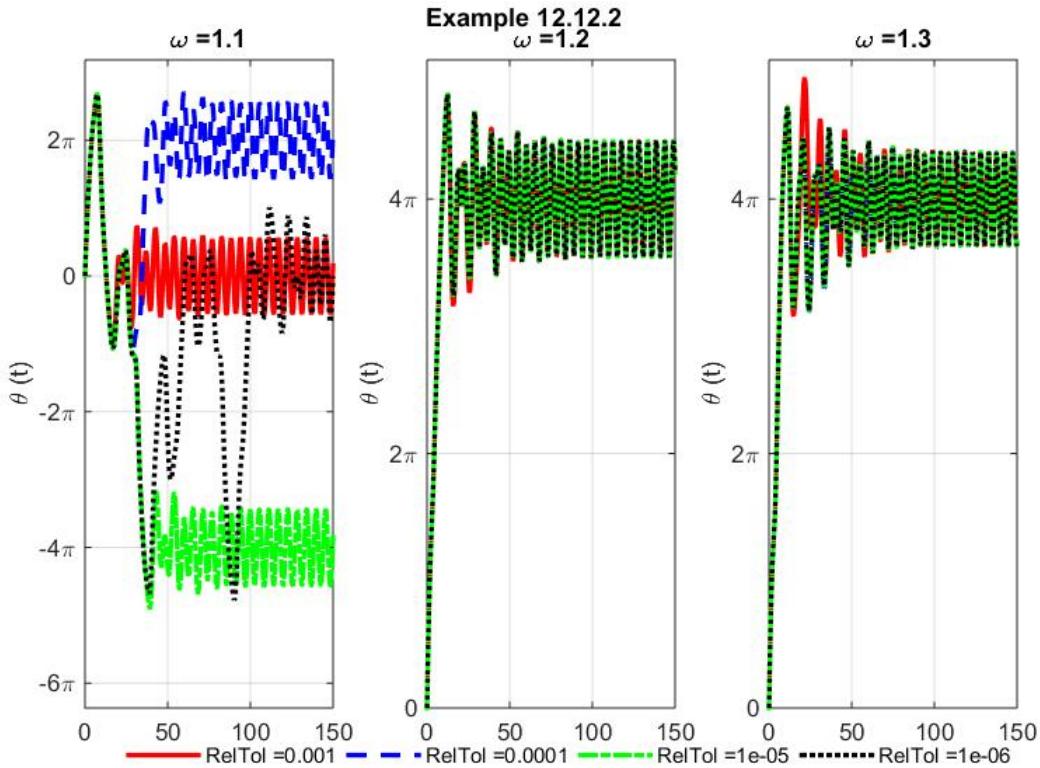
W1=[W(4),W(5),W(6)]

The results are



The first curve, the one corresponding to $\omega = .8$ starts to produce different results for different values of **RelTol** after roughly 40 seconds. Surprisingly, the curve for $\omega = .9$ displays a solution that does not appear to be improved by reducing **RelTol**. It does display a pendulum motion where the pendulum spins counter clockwise for slightly more than one revolution reverses itself and spins clockwise and then repeats the motion in what appears to be a steady state motion with a period of approximately 30 seconds. Things continue to be complicated when the forcing frequency is increased to $\omega = 1.0$. For this frequency, the solutions for the four different values of **RelTol** agree for approximately 20 seconds and then the case **RelTol = 0.001** diverges from the solutions for the other three. The solution for **RelTol = 0.001** goes into a steady state oscillation about $\theta = -2\pi$ while the others eventually go into a steady state oscillation about $\theta = 6\pi$.

Finally, if we continue with the four values of **RelTol = $10^{-3}, 10^{-4}, 10^{-5}$ and 10^{-6}** the three forcing frequencies $\omega = 1.1, 1.2$ and 1.3 yield the plots



Again, we see an irregular pattern of solutions. It is evident that the solutions are strongly dependent on the forcing frequencies. A small change in ω typically produces a large change in the solution. In addition, one would hope that as **RelTol** was decreased the solutions generated would be the same.

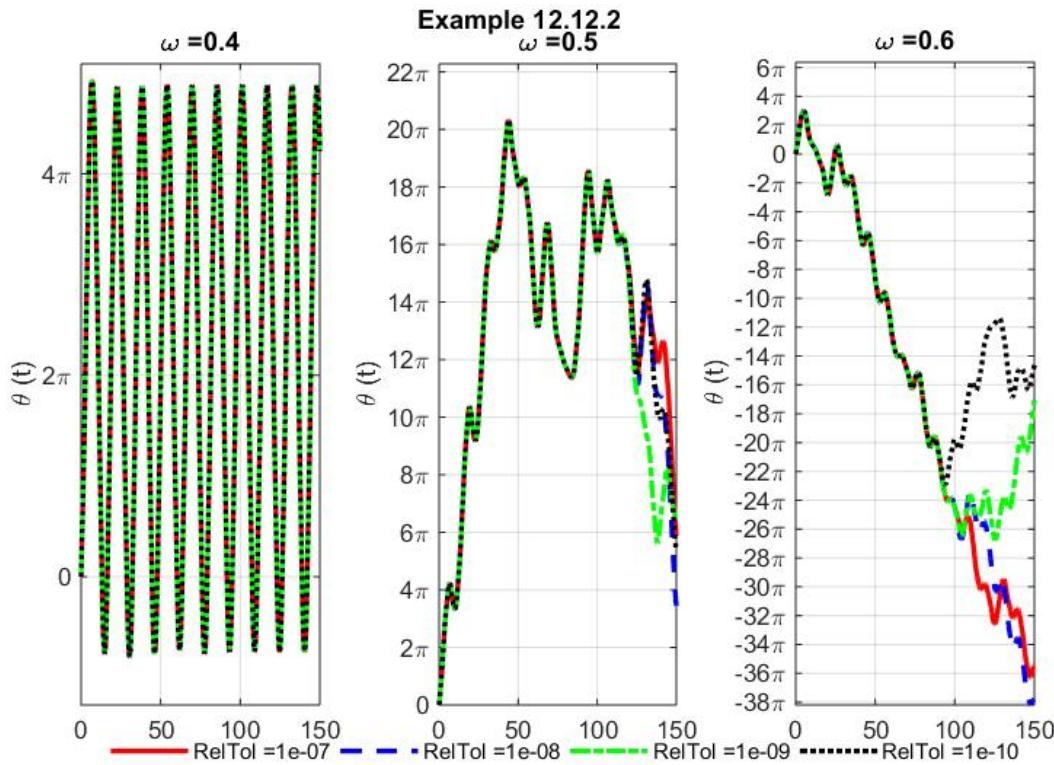
It is evident that this example requires additional numerical work. Our objective will be to continue decreasing the values of **RelTol** and hope that we, at least, gain a single solution for each forcing frequency. In order to generate the solutions for $\text{RelTol} = 10^{-7}, 10^{-8}, 10^{-9}$ and 10^{-10} and $\omega = .4, .5$ and $.6$, one must replace the script

```
Options1=[options(1),options(2),options(3),options(4)]
```

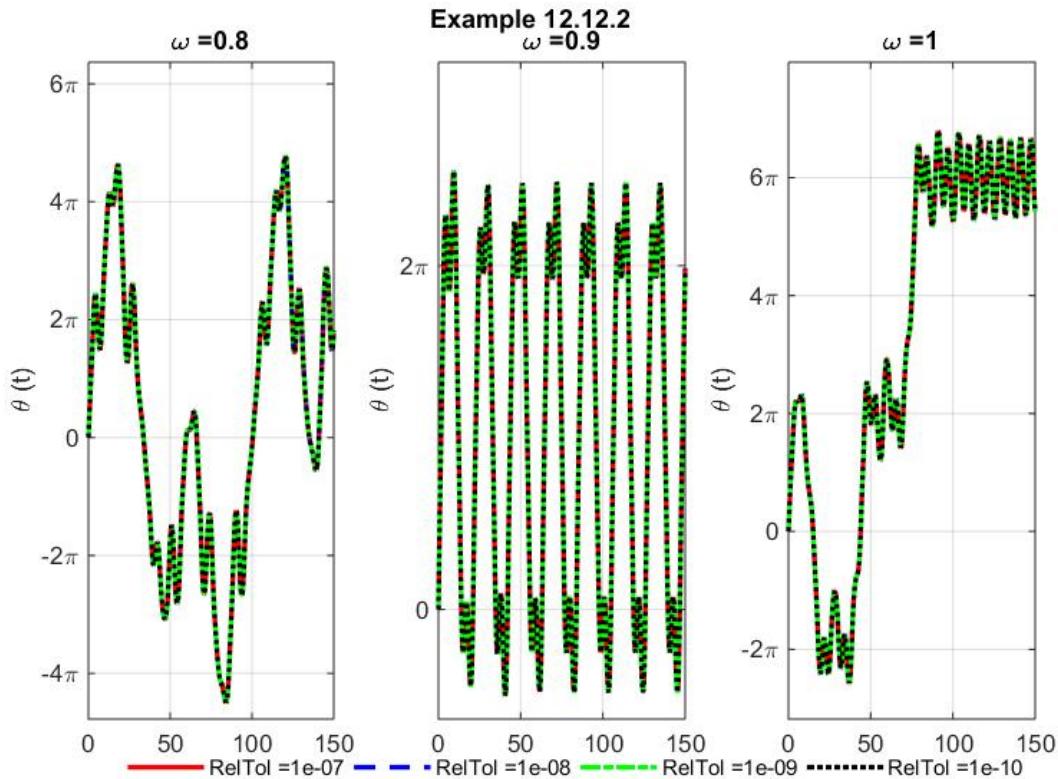
with

```
Options1=[options(5),options(6),options(7),options(8)]
```

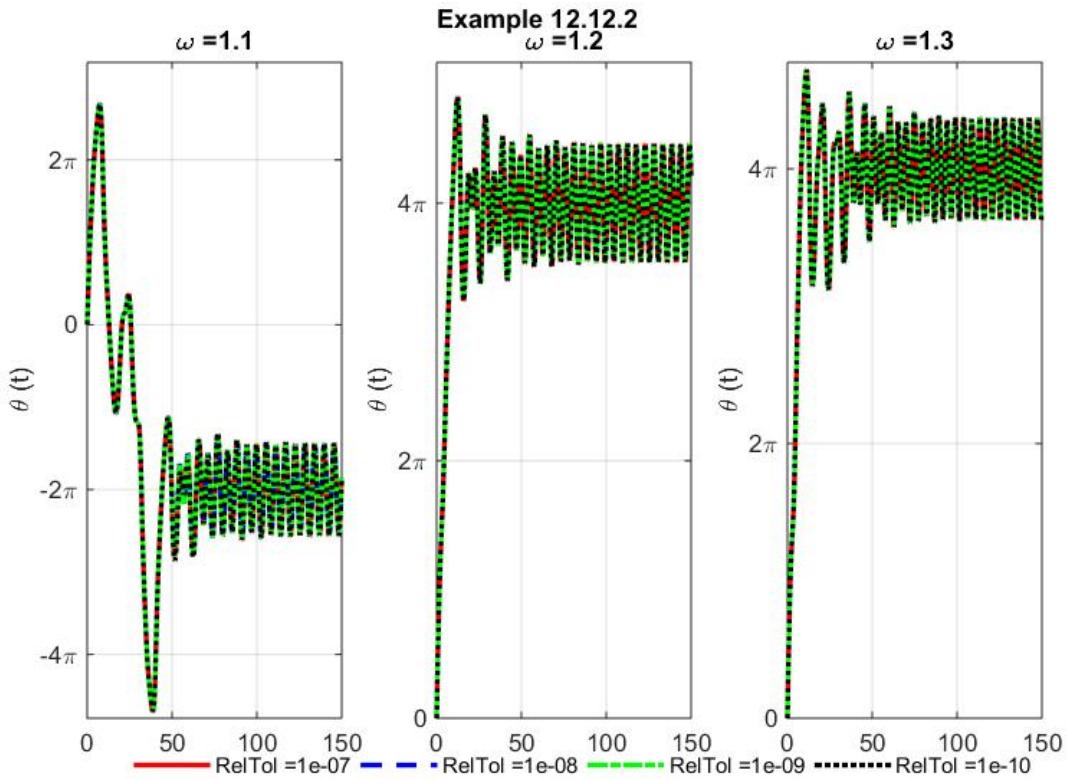
The results are



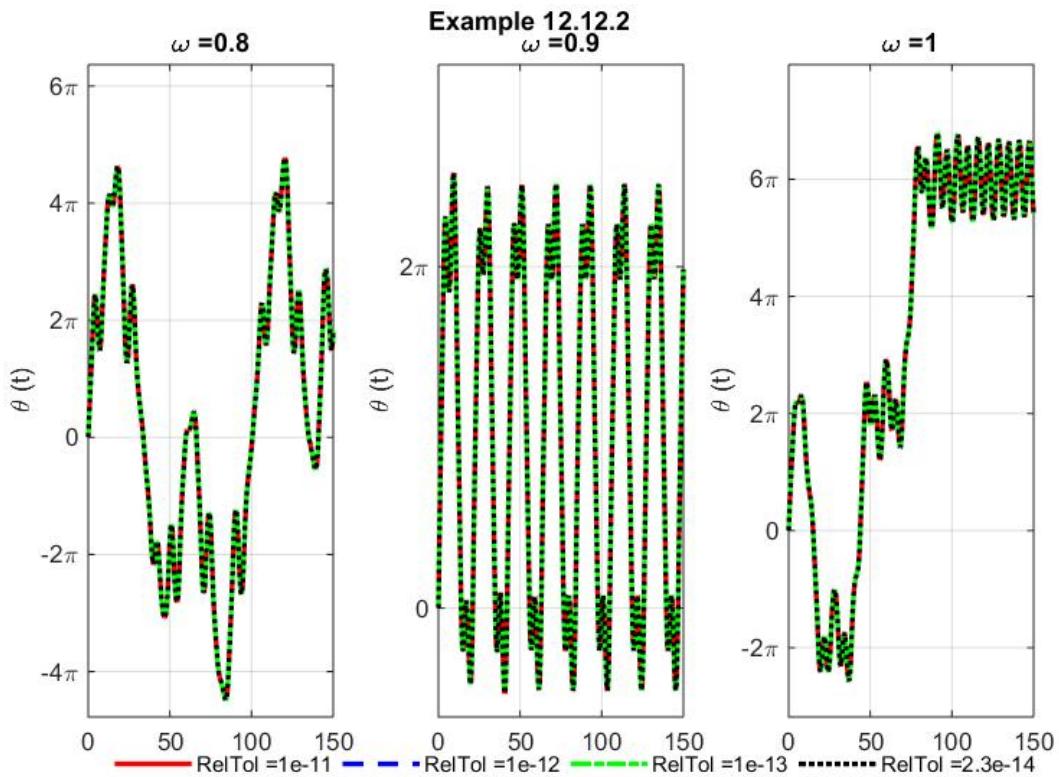
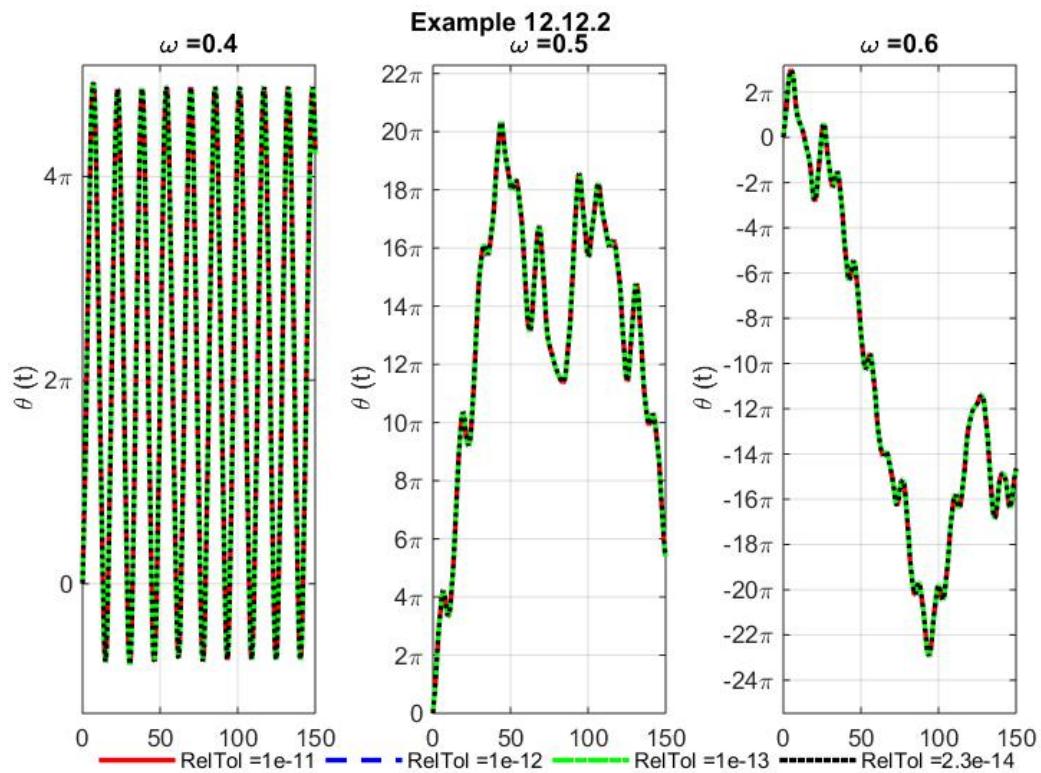
The same choices $\text{RelTol} = 10^{-7}, 10^{-8}, 10^{-9}$ and 10^{-10} produce, for $\omega = .8, .9$ and 1.0



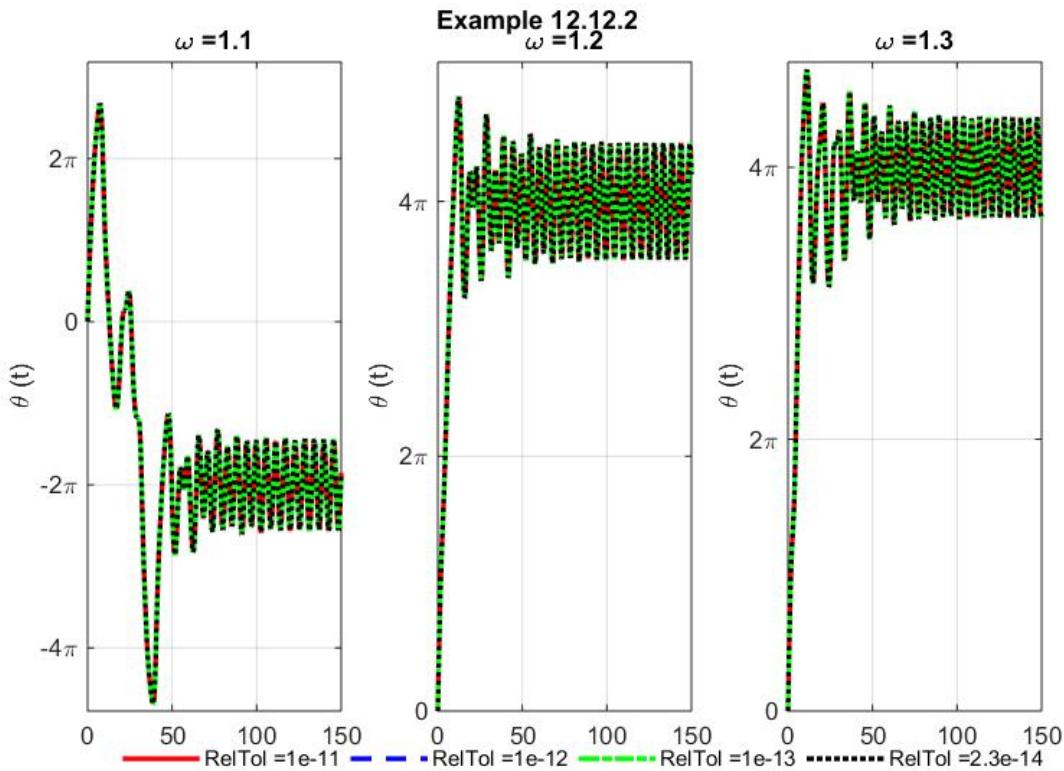
The same choices $\text{RelTol} = 10^{-7}, 10^{-8}, 10^{-9}$ and 10^{-10} produce, for $\omega = 1.1, 1.2$ and 1.3



With the exception of the curves for $\omega = .5$ and $\omega = .6$, the above set of nine curves suggest that a $\text{RelTol} = 10^{-7}$ is sufficient to generate an accurate solution. Next, we shall reduce RelTol to the values $\text{RelTol} = 10^{-11}, 10^{-12}, 10^{-13}$ and $2.3(10)^{-14}$. The last value $2.3(10)^{-14}$, is near the minimum allowed by MATLAB when AbsTol is assigned the default value given in equation (12.8.5). In any case, the resulting set of nine curves are



and



These curves produce the same curve for each of the four values of **RelTol** and for each of the nine forcing frequencies selected. With the exception of the curves for $\omega = .5$ and $\omega = .6$, we learned above that a **RelTol** = 10^{-7} is sufficient to generate an accurate solution. Our last set of curves shows that if we reduce **RelTol** to **RelTol** = 10^{-11} we appear to get an accurate solution for each of the selected forcing frequencies.

It should be evident that the instabilities in the solutions of (12.12.14) subject to the initial conditions (12.12.15) are not fully revealed by the solutions we have generated. For example, the fact that the solution for $\omega = .5$ is fundamentally different than the one for $\omega = .6$, suggests that forcing frequencies in the range $.5 < \omega < .6$ are deserving of additional attention.

Perhaps, the moral to Example 12.12.2 is that one should always test the numerical solutions by conducting numerical experiments for various values of the error parameters. The instabilities in the solutions above showed for large values of the time variable. It seems to be a reality that it is when the independent variable grows large that the default error parameters can be inadequate.

Additional insight into the motion of the damped pendulum can be obtained if one creates animations of the motions as explained in Example 12.12.1. MATLAB script that animates the solution in the case $\omega = 1$ and **RelTol** = 10^{-4} is⁵⁷

⁵⁷ The additional script appended to that above

```

clc
clear
m=20
l=9.81
c=2*(9.81)^2
g=9.81
tspan=[0,150]
u0=0
v0=2
x0=[u0,v0]
%List of possible values of RelTol.
%Note 2.3*10^(-14) is near the minimum allowed RelTol
for n=1:11
    options(n)=odeset('RelTol',10^(-(n+2)));
end
options(12)=odeset('RelTol',2.3*10^(-(14)));
%List of possible values of frequency to be utilized
W=[.4,.5,.6,.8,.9,1,1.1,1.2,1.3];
%Create labels for y axis of subplot
CellLabels=zeros(1,42);
    for s=1:42
        CellLabels(s)=-38+2*(s-1);
    end
CellLabels=cellfun(@num2str, num2cell(CellLabels),...
    'UniformOutput', false)
CellLabels=strcat(CellLabels,{'\pi'});

%Select frequency and RelTol for solution to animate
w=[W(6)]
Options1=[options(2)]
f=@(t)(cos(w*t)*m*l^2);
[t x] = ode45(@f12121,tspan,x0,Options1,m,l,c,g,f);

subplot(1,2,1)
hold on
xlabel('t')
ylabel('\theta (t)', 'Rotation', 0)



---


vid=VideoWriter('Example12122.mp4','MPEG-4')
open(vid)
writeVideo(vid,F(1:5:length(t)))
close(vid)

```

will produce a video file **Example12122.mp4** of the animation. This video can be viewed from the electronic version of Appendix B of this work.

```

grid on
title({'Example 12.12.2',...
    strcat('\omega = ',num2str(w) , ' and',...
        ' RelTol = ',num2str(Options1(1).RelTol))})

set(gca,'YTick',[-38*pi:2*pi:44*pi],...
    'YTickLabel',CellLabels)
axis([0,tspan(2),min(x(:,1)),max(x(:,1))])

subplot(1,2,2)
hold on
axis equal
axis([-1.1*(1) 1.1*(1) -1.1*(1) 1.1*(1)]);
axis off;
y1=-l*cos(x(1,1));
x1=l*sin(x(1,1));
h1=plot([0 x1],[0 y1],'k','LineWidth',2);
h2=plot(x1,y1,'o','MarkerFaceColor','r',...
    'MarkerEdgeColor','k','MarkerSize',15);
plot(0,0,'^','MarkerFaceColor','b',...
    'MarkerEdgeColor','k','MarkerSize',5)

for i=1:5:length(t)
    subplot(1,2,1)
    plot(t(1:i),x(1:i,1),'r','LineWidth',2);
    hold on;
    subplot(1,2,2)
    delete(h1)
    delete(h2)
    y1=-l*cos(x(i,1)); x1=l*sin(x(i,1));
    h1=plot([0 x1],[0 y1],'k','LineWidth',2);
    h2=plot(x1,y1,'o','MarkerFaceColor','r',...
        'MarkerEdgeColor','k','MarkerSize',15);
    title(['t = ' num2str(floor(t(i)))])
    F(i)=getframe(gcf)
end

```

The other solutions can be obtained by replacing the two lines of script

```
w=[W(6)]
Options1=options(2)
```

with the other choices of frequency and `RelTol`.

Exercises

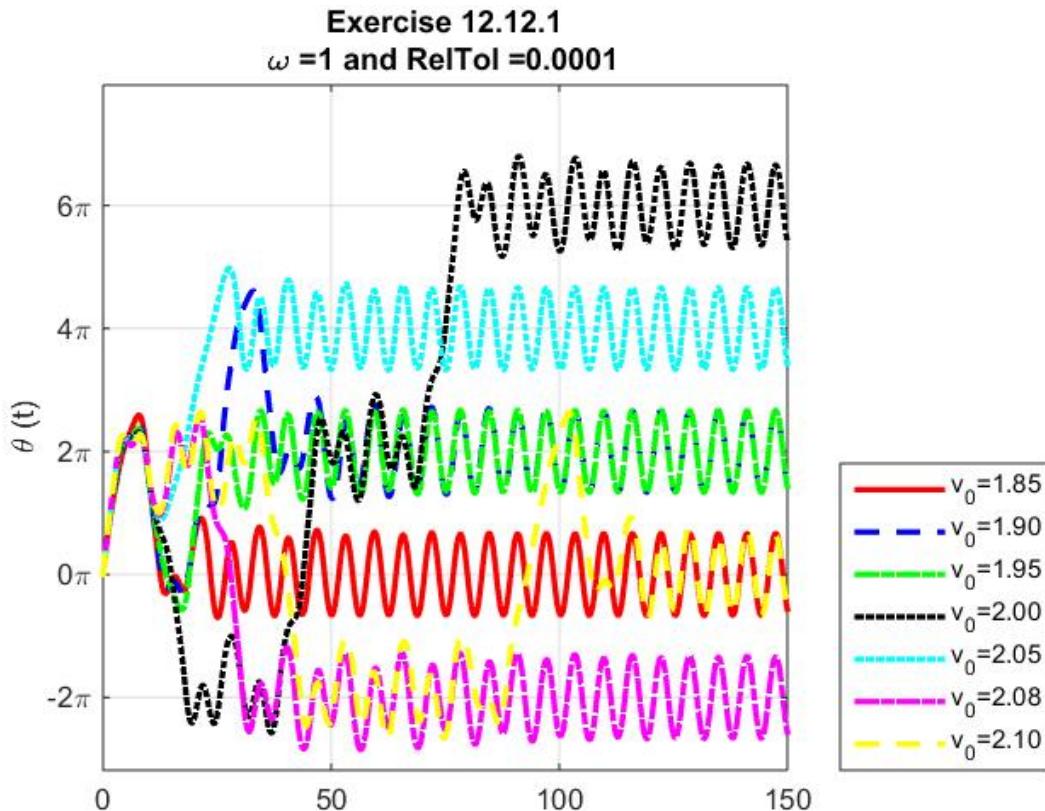
12.12.1:⁵⁸ During all of the solutions generated in Example 12.12.2, the initial conditions were those given by,

$$\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \quad (12.12.16)$$

One can get a sense for the dependence of the solution on initial conditions by working a family of problems each for slightly different initial conditions. Generate a family of seven curves corresponding to the seven initial conditions

$\begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1.85 \end{bmatrix}, \begin{bmatrix} 0 \\ 1.90 \end{bmatrix}, \begin{bmatrix} 0 \\ 1.95 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 2.05 \end{bmatrix}, \begin{bmatrix} 0 \\ 2.08 \end{bmatrix}, \begin{bmatrix} 0 \\ 2.10 \end{bmatrix}$. In each case take $\omega = 1$ and

RelTol = 10^{-4} . The plot of these seven curves should look like

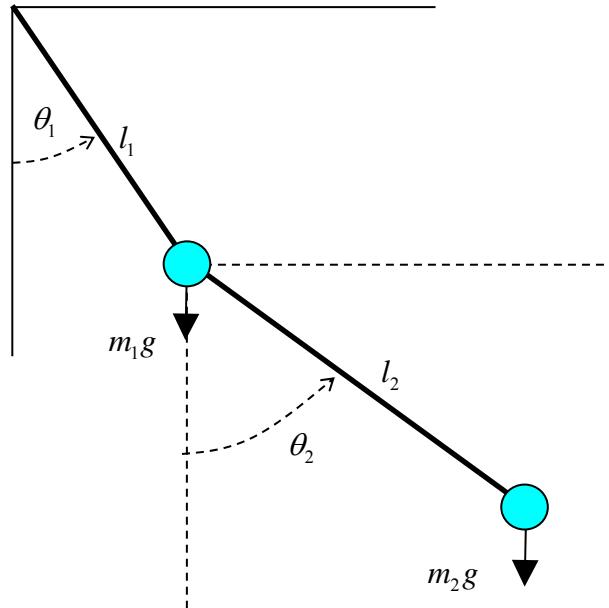


These curves show that after sufficient passage of time, there are five stable solutions oscillating about the equilibrium positions $-2\pi, 0, 2\pi, 4\pi$ and 6π .

⁵⁸ This exercise is suggested by the discussion in Section 3 of Hubbard, J. C., The Forced Damped Pendulum: Chaos, Complication and Control, The American Mathematical Monthly, Vol. 106, No. 8, pp. 741-758, 1999.

Section 12.13. Other Pendulum Examples

In this section, we shall continue to illustrate applications of the MATLAB ode solvers. Most of our examples will involve various generalizations of the kinds of pendulum vibrations problems introduced in Section 12.12. The first example we shall discuss is that of the free undamped vibrations of a *double pendulum*. The following figure illustrates the two linked pendulums of length l_1 and l_2 , respectively.



The *equations of motion* of the double pendulum can be shown to be⁵⁹

$$(m_1 + m_2)l_1^2 \frac{d^2\theta_1}{dt^2} + m_2l_1l_2 \cos(\theta_1 - \theta_2) \frac{d^2\theta_2}{dt^2} + m_2l_1l_2 \left(\frac{d\theta_2}{dt} \right)^2 \sin(\theta_1 - \theta_2) + (m_1 + m_2)gl_1 \sin\theta_1 = 0 \quad (12.13.1)$$

$$m_2l_2^2 \frac{d^2\theta_2}{dt^2} + m_2l_1l_2 \cos(\theta_1 - \theta_2) \frac{d^2\theta_1}{dt^2} - m_2l_1l_2 \left(\frac{d\theta_1}{dt} \right)^2 \sin(\theta_1 - \theta_2) + m_2gl_2 \sin\theta_2 = 0$$

⁵⁹ The derivation of these equations can be found in many places. One example is the textbook, Rosenberg, Reinhardt M., Analytical Dynamics of Discrete Systems, Plenum Press, 1-414, 1977. There are also numerous discussions of the double pendulum on the web. An example is at <http://www.math24.net/double-pendulum.html>.

Because the system governed by (12.13.1) is free of dissipation, the total energy defined by the sum of the kinetic energy and the potential energy is a constant. It is possible to show that this constant is given by

$$E = \frac{1}{2}m_1l_1^2\left(\frac{d\theta_1}{dt}\right)^2 + \frac{1}{2}m_2\left(l_1^2\left(\frac{d\theta_1}{dt}\right)^2 + l_2^2\left(\frac{d\theta_2}{dt}\right)^2 + 2l_1l_2\left(\frac{d\theta_2}{dt}\right)\left(\frac{d\theta_1}{dt}\right)\cos(\theta_1 - \theta_2)\right) + m_1gl_1(1 - \cos\theta_1) + m_2gl_2(1 - \cos\theta_2) + m_2gl_1(1 - \cos\theta_1) \quad (12.13.2)$$

An equivalent form of (12.13.1) is the matrix equation

$$\begin{bmatrix} (m_1 + m_2)l_1^2 & m_2l_1l_2\cos(\theta_1 - \theta_2) \\ m_2l_1l_2\cos(\theta_1 - \theta_2) & m_2l_2^2 \end{bmatrix} \begin{bmatrix} \frac{d^2\theta_1}{dt^2} \\ \frac{d^2\theta_2}{dt^2} \end{bmatrix} + \begin{bmatrix} m_2l_1l_2\left(\frac{d\theta_2}{dt}\right)^2\sin(\theta_1 - \theta_2) + (m_1 + m_2)gl_1\sin\theta_1 \\ -m_2l_1l_2\left(\frac{d\theta_1}{dt}\right)^2\sin(\theta_1 - \theta_2) + m_2gl_2\sin\theta_2 \end{bmatrix} = 0 \quad (12.13.3)$$

As usual, the first step involves expressing the given system of nonlinear ordinary differential equations in normal form. If we define

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} \theta_1(t) \\ \theta_2(t) \\ \frac{d\theta_1(t)}{dt} \\ \frac{d\theta_2(t)}{dt} \end{bmatrix} \quad (12.13.4)$$

then it follows from (12.13.3) that

$$\begin{aligned}
 & \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & (m_1 + m_2)l_1^2 \\ 0 & 0 & m_2l_1l_2 \cos(\theta_1 - \theta_2) \end{array} \right] \frac{d}{dt} \begin{bmatrix} \theta_1(t) \\ \theta_2(t) \\ \frac{d\theta_1(t)}{dt} \\ \frac{d\theta_2(t)}{dt} \end{bmatrix} \\
 &= \begin{bmatrix} \frac{d\theta_1(t)}{dt} \\ \frac{d\theta_2(t)}{dt} \\ -m_2l_1l_2 \left(\frac{d\theta_2}{dt} \right)^2 \sin(\theta_1 - \theta_2) - (m_1 + m_2)gl_1 \sin \theta_1 \\ m_2l_1l_2 \left(\frac{d\theta_1}{dt} \right)^2 \sin(\theta_1 - \theta_2) - m_2gl_2 \sin \theta_2 \end{bmatrix} \tag{12.13.5}
 \end{aligned}$$

If (12.13.4) is used, (12.13.5) becomes

$$\begin{aligned}
 & \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & (m_1 + m_2)l_1^2 \\ 0 & 0 & m_2l_1l_2 \cos(x_1(t) - x_2(t)) \end{array} \right] \frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} \\
 &= \begin{bmatrix} x_3(t) \\ x_4(t) \\ -m_2l_1l_2x_4(t)^2 \sin(x_1(t) - x_2(t)) - (m_1 + m_2)gl_1 \sin x_1(t) \\ m_2l_1l_2x_3(t)^2 \sin(x_1(t) - x_2(t)) - m_2gl_2 \sin x_2(t) \end{bmatrix} \tag{12.13.6}
 \end{aligned}$$

Equation (12.13.6) is in the form of the more general normal form (12.1.28). The matrix of coefficients on the left side of (12.13.6) can be shown to have the inverse

$$\begin{aligned}
& \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (m_1 + m_2)l_1^2 & m_2 l_1 l_2 \cos(x_1(t) - x_2(t)) \\ 0 & 0 & m_2 l_1 l_2 \cos(x_1(t) - x_2(t)) & m_2 l_2^2 \end{bmatrix}^{-1} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{l_1^2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} & -\frac{\cos(x_1(t) - x_2(t))}{l_1 l_2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} \\ 0 & 0 & -\frac{\cos(x_1(t) - x_2(t))}{l_1 l_2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} & \frac{m_1 + m_2}{l_2^2 m_2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} \end{bmatrix} \quad (12.13.7)
\end{aligned}$$

Given (12.13.7), equation (12.13.6) reduces to the normal form

$$\begin{aligned}
\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{l_1^2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} & -\frac{\cos(x_1(t) - x_2(t))}{l_1 l_2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} \\ 0 & 0 & -\frac{\cos(x_1(t) - x_2(t))}{l_1 l_2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} & \frac{m_1 + m_2}{l_2^2 m_2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} \end{bmatrix} \times \\
&\quad \begin{bmatrix} x_3(t) \\ x_4(t) \\ -m_2 l_1 l_2 x_4(t)^2 \sin(x_1(t) - x_2(t)) - (m_1 + m_2) g l_1 \sin x_1(t) \\ m_2 l_1 l_2 x_3(t)^2 \sin(x_1(t) - x_2(t)) - m_2 g l_2 \sin x_2(t) \end{bmatrix} \quad (12.13.8)
\end{aligned}$$

If the matrix multiplication in equation (12.13.8) is carried out, it reduces to ⁶⁰

⁶⁰ In the case where $\theta_1(t)$ and $\theta_2(t)$ are small, equation (12.13.9) can be replaced by the linear equation

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} x_3(t) \\ x_4(t) \\ \frac{-m_2 l_2 x_4(t)^2 \sin(x_1(t) - x_2(t)) - (m_1 + m_2) g \sin x_1(t)}{l_1(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} \\ \frac{-m_2 \cos(x_1(t) - x_2(t))(l_1 x_3(t)^2 \sin(x_1(t) - x_2(t)) - g \sin x_2(t))}{l_1(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} \\ \frac{\cos(x_1(t) - x_2(t))(m_2 l_2 x_4(t)^2 \sin(x_1(t) - x_2(t)) + (m_1 + m_2) g \sin x_1(t))}{l_2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} \\ + \frac{(m_1 + m_2)(l_1 x_3(t)^2 \sin(x_1(t) - x_2(t)) - g \sin x_2(t))}{l_2(m_1 + m_2 - m_2 \cos^2(x_1(t) - x_2(t)))} \end{bmatrix} \quad (12.13.9)$$

Equations (12.13.6), (12.13.8) and (12.13.9) represent three possibilities for creating a function m-file to define the system of ordinary differential equations (12.13.1). If, for example we were to use (12.13.6), it can be defined by

```
function dxdt=f12131a(t,x,m1,m2,L1,L2,g)
dxdt=zeros(4,1), %Preallocate
M=[1,0,0,0;...
    0,1,0,0;...
    0,0,(m1+m2)*L1^2,m2*L1*L2*cos(x(1)-x(2));...
    0,0,m2*L1*L2*cos(x(1)-x(2)),m2*L2^2]
dxdt=inv(M)*[x(3);x(4);...
    -m2*L1*L2*x(4)^2*sin(x(1)-x(2))-...
    (m1+m2)*g*L1*sin(x(1));...
    m2*L1*L2*x(3)^2*sin(x(1)-x(2))-...
    m2*g*L2*sin(x(2))]
```

The advantage of **f1213a.m** is that the script for the symmetric matrix of coefficients **M** is elementary to enter. The disadvantage is that MATLAB must invert **M** as a part of the numerical

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{(m_1 + m_2)}{m_1 l_1} g & \frac{m_2}{m_1 l_1} g & 0 & 0 \\ \frac{(m_1 + m_2)}{m_1 l_2} g & -\frac{(m_1 + m_2)}{m_1 l_2} g & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix}$$

This equation can be analyzed by use of the methods discussed in Section 5.5.

solution. While not an issue for the examples we shall work, in cases where \mathbf{M} is ill conditioned numerical errors could be created by the inversion.

If we were to use (12.13.8), this form of the system of ordinary differential equations can be defined by

```
function dxdt=f12131b(t,x,m1,m2,L1,L2,g)
dxdt=zeros(4,1), %Preallocate
invM=[1,0,0,0;0,1,0,0;...
0,0,1/L1^2/(m1+m2-m2*cos(x(1)-x(2))^2),...
-cos(x(1)-x(2))/L1/L2./(m1+m2-m2*cos(x(1)-x(2))^2);...
0,0,...
-cos(x(1)-x(2))/L1/L2/(m1+m2-m2*cos(x(1)-x(2))^2),...
(m1+m2)/L2^2/m2/(m1+m2-m2*cos(x(1)-x(2))^2)];
dxdt=invM*[x(3);x(4);...
-m2*L1*L2*x(4)^2*sin(x(1)-x(2))-...
(m1+m2)*g*L1*sin(x(1));...
m2*L1*L2*x(3)^2*sin(x(1)-x(2))-...
m2*g*L2*sin(x(2))]
```

The advantage of **f12131b.m** is that it utilizes the analytical form of the inverse given by (12.13.7).

Finally, we can adopt equation (12.13.9) as the normal form and define it by the function m-file

```
function dxdt=f12131c(t,x,m1,m2,L1,L2,g)
dxdt=zeros(4,1); %Preallocate
dxdt=[x(3);x(4);...
(-(m2*L2*x(4)^2*sin(x(1)-x(2)))-...
(m1+m2)*g*sin(x(1))-...
m2*cos(x(1)-x(2))*(L1*x(3)^2*sin(x(1)-x(2))-...
g*sin(x(2)))/L1/(m1+m2-m2*(cos(x(1)-x(2)))^2);...
(cos(x(1)-x(2))*(m2*L2*x(4)^2*sin(x(1)-x(2))+...
(m1+m2)*g*sin(x(1)))+...
(m1+m2)*(L1*x(3)^2*sin(x(1)-x(2))-...
g*sin(x(2)))/L2/(m1+m2-m2*(cos(x(1)-x(2)))^2)];
```

Not only does **f12131c.m** adopt the analytical form of the inverse (12.13.7), it avoids the necessity of the matrix multiplication shown in (12.13.8). A disadvantage is the necessity to confront the entry of complex script without error.⁶¹ For the most part, our examples will utilize **f12131c.m**. However, one can easily show that the other defining function files yield

⁶¹ The complex script can be minimized by the kinds of definitions and rearrangements shown in the class notes <http://www.math.tamu.edu/~mpilant/math308/Matlab/Project3/Project3.pdf>.

equivalent results for our examples. An exception to our use of **f12131c.m** is given below when we show not MATLAB can be utilized when the generalized normal form (12.1.28) is used.

Example 12.13.1: Given our discussion in Section 12.12, we need to be concerned whether or not the default **ReaTol** values are sufficient for the double pendulum. For this example, we shall adopt the numerical values

$$\begin{aligned} m_1 &= 2 \text{ slug} \\ m_2 &= 1 \text{ slug} \\ g &= 32.2 \text{ ft/sec}^2 \\ L_1 &= 1 \text{ ft} \\ L_2 &= 2 \text{ ft} \end{aligned} \tag{12.13.10}$$

the initial conditions

$$\begin{bmatrix} x_1(0) \\ x_2(0) \\ x_3(0) \\ x_4(0) \end{bmatrix} = \begin{bmatrix} \theta_1(0) \\ \theta_2(0) \\ \frac{d\theta_1(0)}{dt} \\ \frac{d\theta_2(0)}{dt} \end{bmatrix} = \begin{bmatrix} \frac{\pi}{2} \\ \pi \\ 0 \\ 0 \end{bmatrix} \tag{12.13.11}$$

and the time interval

$$\mathbf{tspan} = [0, 10] \tag{12.13.12}$$

We shall generate the approximate solutions for the default **RelTol** = 10^{-3} and for **RelTol** = 10^{-4} , **RelTol** = 10^{-5} , **RelTol** = 10^{-6} and **RelTol** = 10^{-7} . What we shall learn is that, for the given properties (12.13.10), the initial conditions (12.13.11) and the time interval (12.13.12), the values **RelTol** = 10^{-3} , **RelTol** = 10^{-4} and **RelTol** = 10^{-5} do not yield the same solutions as does **RelTol** = 10^{-6} and **RelTol** = 10^{-7} . The script that plots the two angles, $\theta_1(t)$ and $\theta_2(t)$, for the given values of **RelTol** is

```
clc
clear
m1 = 2;
m2 = 1;
g = 32.2;
L1 = 1;
L2 = 2;
x0 = [pi/2, pi, 0, 0];
```

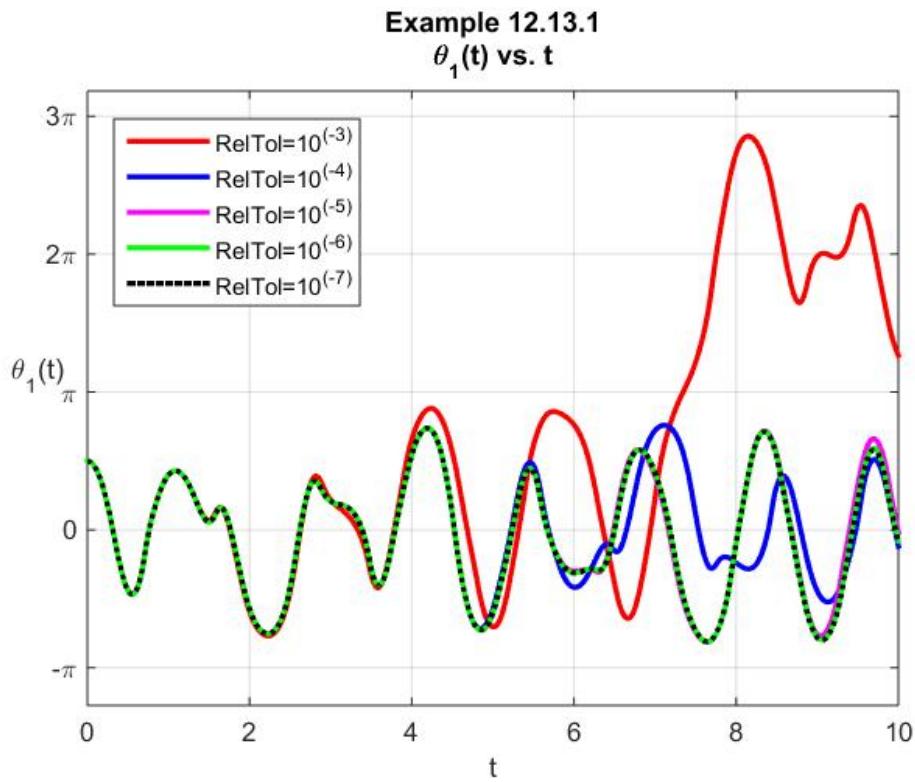
```

tspan=[0,10];
% RelTol=10^(-3), default value
[t1 x1] = ode45(@f12131c,tspan,x0,[],m1,m2,L1,L2,g);
% RelTol=10^(-4)
options=odeset('RelTol',10^(-4));
[t2 x2] = ode45(@f12131c,tspan,x0,options,m1,m2,L1,L2,g);
% RelTol=10^(-5)
options=odeset('RelTol',10^(-5));
[t3 x3] = ode45(@f12131c,tspan,x0,options,m1,m2,L1,L2,g);
options=odeset('RelTol',10^(-6));
[t4 x4] = ode45(@f12131c,tspan,x0,options,m1,m2,L1,L2,g);
options=odeset('RelTol',10^(-7));
[t5 x5] = ode45(@f12131c,tspan,x0,options,m1,m2,L1,L2,g);
figure
plot(t1,x1(:,1),'r','LineWidth',2)
hold on
plot(t2,x2(:,1),'b','LineWidth',2)
plot(t3,x3(:,1),'m','LineWidth',2)
plot(t4,x4(:,1),'g','LineWidth',2)
plot(t5,x5(:,1),'k','LineWidth',2)
grid on
xlabel('t')
ylabel('\theta_1(t)', 'Rotation', 0)
legend('RelTol=10^{(-3)}', 'RelTol=10^{(-4)}',...
    'RelTol=10^{(-5)}', 'RelTol=10^{(-6)}',...
    'RelTol=10^{(-7)}', 'Location', 'NorthWest')
title({'Example 12.13.1', '\theta_1(t) vs. t'})
set(gca,'YTick',[-pi:pi:3*pi],...
    'YTickLabel',{'-\pi','0','\pi','2\pi','3\pi'})

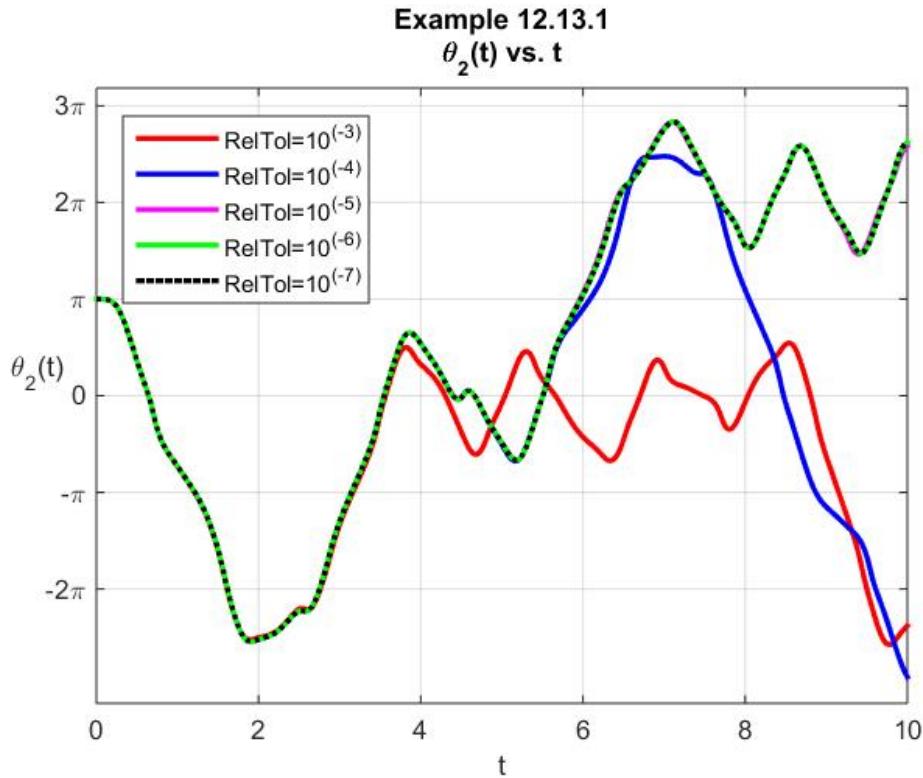
figure
plot(t1,x1(:,2),'r','LineWidth',2)
hold on
plot(t2,x2(:,2),'b','LineWidth',2)
plot(t3,x3(:,2),'m','LineWidth',2)
plot(t4,x4(:,2),'g','LineWidth',2)
plot(t5,x5(:,2),'k','LineWidth',2)
grid on
xlabel('t')
ylabel('\theta_2(t)', 'Rotation', 0)
legend('RelTol=10^{(-3)}', 'RelTol=10^{(-4)}',...
    'RelTol=10^{(-5)}', 'RelTol=10^{(-6)}',...
    'RelTol=10^{(-7)}', 'Location', 'NorthWest')
title({'Example 12.13.1', '\theta_2(t) vs. t'})
set(gca,'YTick',[-2*pi:pi:3*pi],...
    'YTickLabel',...
    {'-2\pi','-\pi','0','\pi','2\pi','3\pi'})

```

The results from executing this script are



and



The first plot, the one for $\theta_1(t)$, shows that the solutions for $\text{RelTol} = 10^{-3}$ and $\text{RelTol} = 10^{-4}$ the solutions are not close to the others. The solutions for $\text{RelTol} = 10^{-6}$ and $\text{RelTol} = 10^{-7}$ appear to be identical to each other. The solution for $\text{RelTol} = 10^{-5}$ is close to those for $\text{RelTol} = 10^{-6}$ and $\text{RelTol} = 10^{-7}$. The second plot, the one for $\theta_2(t)$, shows that the solutions for $\text{RelTol} = 10^{-5}$, $\text{RelTol} = 10^{-6}$ and $\text{RelTol} = 10^{-7}$ appear to be identical to each other. Given this information, in the following we shall adopt $\text{RelTol} = 10^{-6}$.

It probably does need to be explained that the calculation just completed does not constitute a rigorous proof. For different numerical values (12.13.10), different initial conditions (12.13.11) and different time intervals (12.13.12), one might get different results. In any case, as mentioned, in the following examples we shall use $\text{RelTol} = 10^{-6}$.

Example 12.13.2: In our earlier examples, such as Example 12.4.2 and Example 12.11.2, and the Exercise 12.12.1, we explained that nonlinear ordinary differential equations sometimes have the feature that small changes in the initial conditions produce large changes in the solution. Thus, it is reasonable to explore this possibility for the double pendulum. In this exercise, we shall compare the solution generated in Example 12.13.1 for the initial condition (12.13.11), repeated,

$$\begin{bmatrix} x_1(0) \\ x_2(0) \\ x_3(0) \\ x_4(0) \end{bmatrix} = \begin{bmatrix} \theta_1(0) \\ \theta_2(0) \\ \frac{d\theta_1(0)}{dt} \\ \frac{d\theta_2(0)}{dt} \end{bmatrix} = \begin{bmatrix} \frac{\pi}{2} \\ \pi \\ 0 \\ 0 \end{bmatrix} \quad (12.13.13)$$

to the solution generated for the initial condition

$$\begin{bmatrix} x_1(0) \\ x_2(0) \\ x_3(0) \\ x_4(0) \end{bmatrix} = \begin{bmatrix} \theta_1(0) \\ \theta_2(0) \\ \frac{d\theta_1(0)}{dt} \\ \frac{d\theta_2(0)}{dt} \end{bmatrix} = \begin{bmatrix} 1.57 \\ 3.14 \\ 0 \\ 0 \end{bmatrix} \quad (12.13.14)$$

As explained above, in this example and later ones we shall make the choice **RelTol = 10⁻⁶**. The MATLAB script

```

clc
clear
m1 = 2;
m2 = 1;
g = 32.2;
L1 = 1;
L2 = 2;
x0_1 = [pi/2,pi,0,0];
x0_2 = [1.57,3.14,0,0];
tspan=[0,10];
% RelTol=10^(-6)
options=odeset('RelTol',10^(-6));
[t1 x1] = ode45(@f12131c,tspan,x0_1,options,m1,m2,L1,L2,g);
[t2 x2] = ode45(@f12131c,tspan,x0_2,options,m1,m2,L1,L2,g);
figure
plot(t1,x1(:,1),'r','LineWidth',2)
hold on
plot(t2,x2(:,1), '--b','LineWidth',2)
grid on
xlabel('t')
ylabel('\theta_1(t)', 'Rotation', 0)
legend('\theta_1(t)=\pi/2', '\theta_1(t)=1.57',...
    'Location','SouthWest')
title({'Example 12.13.2', '\theta_1(t) vs. t'})

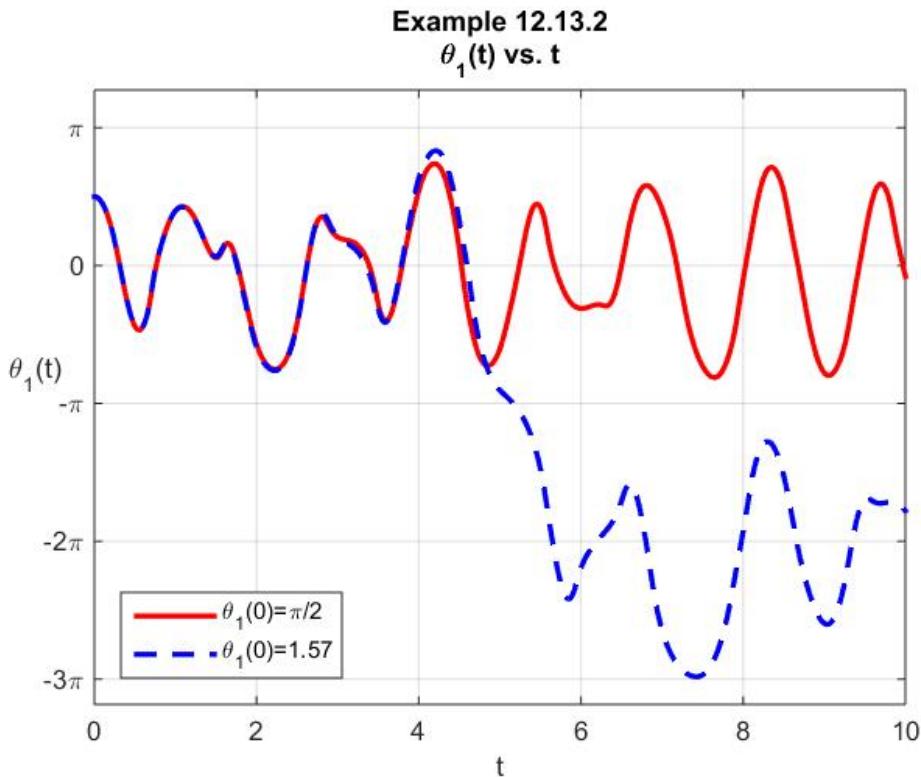
```

```

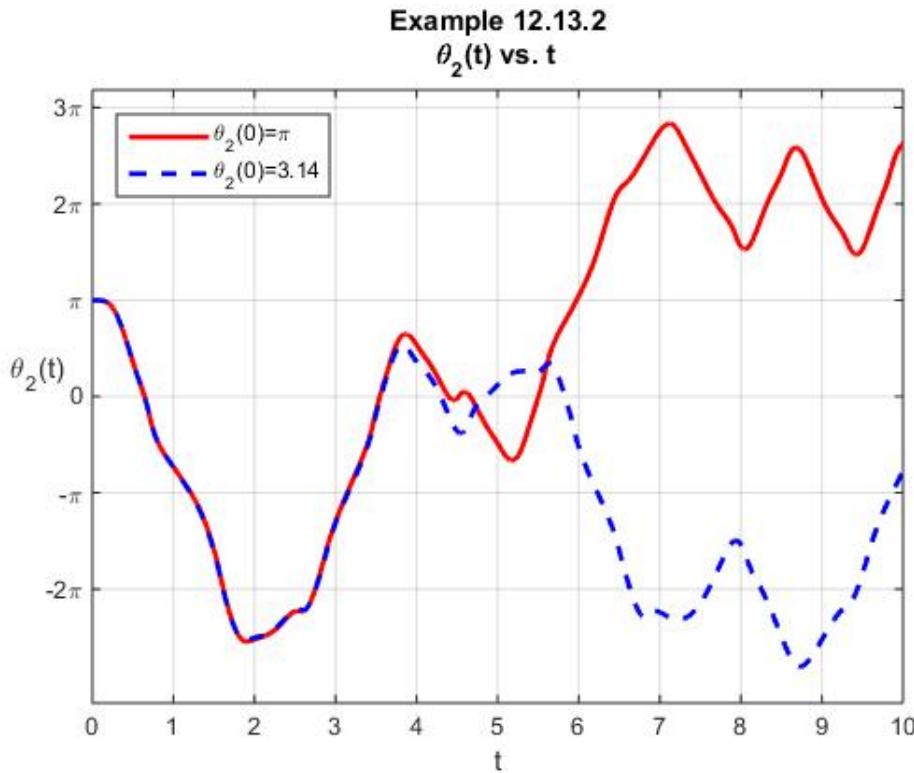
set(gca,'YTick',[-3*pi:pi:3*pi],...
    'YTickLabel',...
    {'-3\pi','-2\pi','-\pi','0','\pi','2\pi','3\pi'})
figure
plot(t1,x1(:,2),'r','LineWidth',2)
hold on
plot(t2,x2(:,2),'--b','LineWidth',2)
grid on
xlabel('t')
ylabel('\theta_2(t)', 'Rotation',0)
legend ('\theta_2(t)=\pi','\theta_2(t)=3.14',...
    'Location','NorthWest')
title({ 'Example 12.13.2', '\theta_2(t) vs. t' })
set(gca,'YTick',[-2*pi:pi:3*pi],...
    'YTickLabel',...
    {'-2\pi','-\pi','0','\pi','2\pi','3\pi'})

```

The results from executing this script are

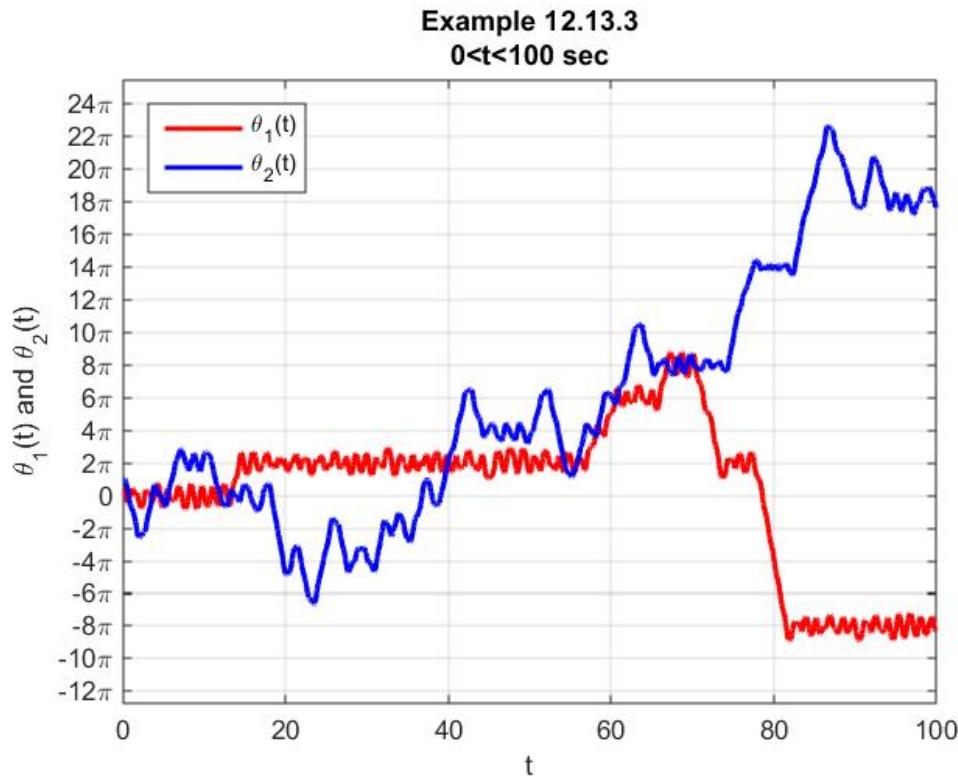


for $\theta_1(t)$ and

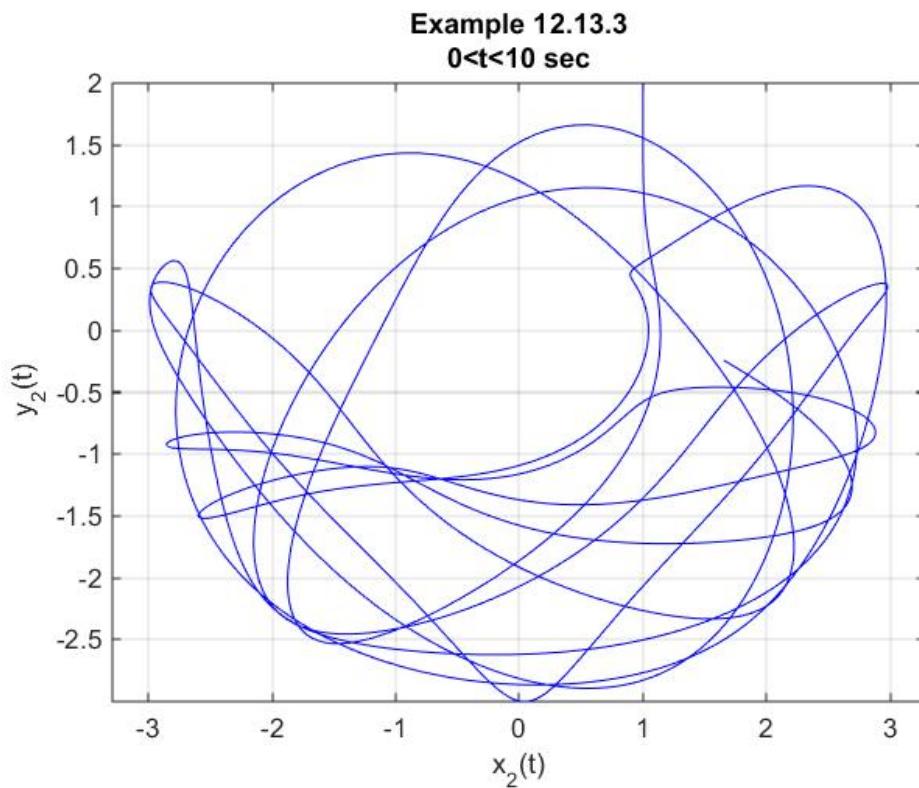


for $\theta_2(t)$. These figures indicate that for the small difference in the two sets of initial conditions (12.13.13) and (12.13.14), the two solutions are close for roughly 4 seconds and then begin to differ in a significant way. At least for the two sets of initial conditions chosen, small changes in initial conditions can produce large changes in the pendulum motions.

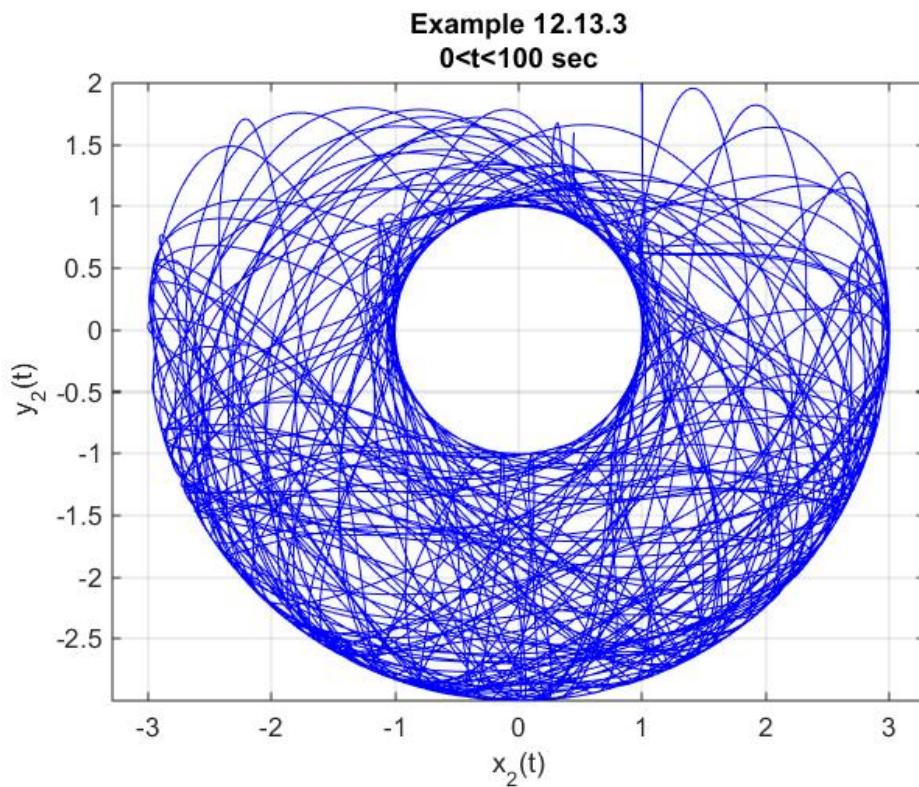
Exercise 12.13.3: The discovery that small differences in initial conditions can result in large differences in the solution makes it difficult to identify the true solution. When, for example, a rounding error can produce a large change in the answer, it is difficult to be confident that the correct solution has been found. The study of initial condition sensitive differential equations is a topic of study known as *chaos theory*. The systems of ordinary differential equations we are studying are entirely deterministic. Their behavior is theoretically determined by their initial conditions. Unfortunately, the practicalities of numerical methods can undermine the theory. In this example, we shall look more closely at the solution for the problem posed in Exercise 12.13.1. We shall make the choice **RelTol = 10^{-6}** . The red curves in the last two figures give the evolution of $\theta_1(t)$ and $\theta_2(t)$ for the interval $0 < t < 10$. If we extend the interval to $0 < t < 100$, script like used above produces the following figure.



It is also interesting to create a parametric plot that tracks the position of the second mass in time. This plot reflects the chaotic nature of the motion of the second mass as it proceeds in time. The result for $0 < t < 10$ sec is



and, for the longer period $0 < t < 100 \text{ sec}$, is



It is instructive to utilize MATLAB to create an animation of the evolution of the plots of pendulum angles vs. time and to show a figure of the pendulum evolving in time. The following MATLAB script creates this animation.^{62,63}

```

clc
clear
m1 = 2;
m2 = 1;
g = 32.2;
L1 = 1;
L2 = 2;
x0 = [pi/2,pi,0,0];
tspan=[0,10];
options=odeset('RelTol',10^(-6))
[t x] = ode45(@f12131c,tspan,x0,options,m1,m2,L1,L2,g);
%Create labels for y axis of subplot
CellLabels=zeros(1,11);
for s=1:11
    CellLabels(s)=-5+(s-1);
end
CellLabels=cellfun(@num2str, num2cell(CellLabels),...
    'UniformOutput', false);
CellLabels=strcat(CellLabels,{'\pi'});
CellLabels(6)={'0'};
subplot(1,2,1)
hold on
xlabel('t')
ylabel('\theta_1(t) and \theta_2(t)')
grid on
title({'Example 12.13.3 Animation',...
    '\theta_1(0)=\pi/2 and \theta_2(0)=\pi'})
set(gca,'YTick',[-38*pi:pi:38*pi],...
    'YTickLabel',CellLabels)
axis([0,tspan(2),min(min(x(:,1:2))),max(max(x(:,1:2)))])
```

⁶² As with the earlier animations, MATLAB will create an mp4 video file of the animation. The additional script appended to that above

```

vid=VideoWriter('Example12133.mp4','MPEG-4')
open(vid)
writeVideo(vid,F([1:20:20*floor(length(t)/20),length(t)]))
close(vid)
```

will produce a video file **Example12133.mp4** of the animation. This video can be viewed from the electronic version of Appendix B of this work.

⁶³ There are several double pendulum animations on line that are instructive to view. An example can be found at <http://www.math24.net/double-pendulum.html>.

```

subplot(1,2,2)
hold on
axis equal
axis([- (L1+L2) (L1+L2) -(L1+L2) (L1+L2)]);
xlabel('x_2')
ylabel('y_2','Rotation',0)
grid on
y1=-L1*cos(x(1,1));
x1=L1*sin(x(1,1));
y2=y1-L2*cos(x(1,2));
x2=x1+L2*sin(x(1,2));
h1=plot([0 x1 x2],[0 y1 y2],'k','LineWidth',2);
h2=plot(x1,y1,'o','MarkerFaceColor','r',...
    'MarkerEdgeColor','k','MarkerSize',15);
h3=plot(x2,y2,'o','MarkerFaceColor','b',...
    'MarkerEdgeColor','k','MarkerSize',15);
plot(0,0,'^','MarkerFaceColor','b',...
    'MarkerEdgeColor','k','MarkerSize',5);

for i=[1:20:20*floor(length(t)/20),length(t)]
    subplot(1,2,1)
    plot(t(1:i),x(1:i,1),'r','linewidth',2)
    hold on
    plot(t(1:i),x(1:i,2),'b','linewidth',2)

legend('\theta_1(t)', '\theta_2(t)', 'Location', 'NorthWest')
    subplot(1,2,2)
    y1=-L1*cos(x(i,1)); x1=L1*sin(x(i,1));
    y2=y1-L2*cos(x(i,2)); x2=x1+L2*sin(x(i,2));
    delete(h1),delete(h2),delete(h3)
    h1=plot([0 x1 x2],[0 y1 y2],'k','LineWidth',2);
    h2=plot(x1,y1,'o','MarkerFaceColor','r',...
        'MarkerEdgeColor','k','MarkerSize',15);
    h3=plot(x2,y2,'o','MarkerFaceColor','b',...
        'MarkerEdgeColor','k','MarkerSize',15);
    x2 = L1*sin(x(1:i,1))+L2*sin(x(1:i,2));
    y2 = -L1*cos(x(1:i,1))-L2*cos(x(1:i,2)) ;
    h4=plot(x2,y2,'b');
    title(['t = ' num2str(floor(t(i)))])
    F(i)=getframe(gcf);
end

```

In our discussion of **options** in Section 12.8, it was explained that if the command **odeset** is executed the list of options, namely, was displayed. The list, repeated from Section 12.8 is

```

AbsTol: [ positive scalar or vector {1e-6} ]
RelTol: [ positive scalar {1e-3} ]
NormControl: [ on | {off} ]
NonNegative: [ vector of integers ]
OutputFcn: [ function_handle ]
OutputSel: [ vector of integers ]
Refine: [ positive integer ]
Stats: [ on | {off} ]
InitialStep: [ positive scalar ]
MaxStep: [ positive scalar ]
BDF: [ on | {off} ]
MaxOrder: [ 1 | 2 | 3 | 4 | {5} ]
Jacobian: [ matrix | function_handle ]
JPattern: [ sparse matrix ]
Vectorized: [ on | {off} ]
Mass: [ matrix | function_handle ]
MStateDependence: [ none | {weak} | strong ]
MvPattern: [ sparse matrix ]
MassSingular: [ yes | no | {maybe} ]
InitialSlope: [ vector ]

```

In Section 12.8, we discussed the options **AbsTol** and **RelTol**. Example 12.12.2 and the examples of this section illustrated the importance and benefits of adjusting **RelTol**. In our next example, we shall illustrate how one can utilize the two options **Mass** and **MStateDependence**. These two options are important in cases where the generalized normal form, given by equation (12.1.28), repeated (with a slight change of notation)

$$\mathbf{M}(t, \mathbf{x}) \frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x}) \quad (12.13.15)$$

must be used. This possibility occurs when the matrix $\mathbf{M}(t, \mathbf{x})$ is singular or ill conditioned for some values of its arguments. Our example is artificial in a sense because it will be based upon the equations of the double pendulum written in the form (12.13.6), repeated,

$$\begin{aligned}
 & \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (m_1 + m_2)l_1^2 & m_2 l_1 l_2 \cos(x_1(t) - x_2(t)) \\ 0 & 0 & m_2 l_1 l_2 \cos(x_1(t) - x_2(t)) & m_2 l_2^2 \end{array} \right] \frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} \\
 &= \begin{bmatrix} x_3(t) \\ x_4(t) \\ -m_2 l_1 l_2 x_4(t)^2 \sin(x_1(t) - x_2(t)) - (m_1 + m_2) g l_1 \sin x_1(t) \\ m_2 l_1 l_2 x_3(t)^2 \sin(x_1(t) - x_2(t)) - m_2 g l_2 \sin x_2(t) \end{bmatrix} \tag{12.13.16}
 \end{aligned}$$

and, as we have seen with our examples of this section, the coefficient matrix is nonsingular. Never the less, it is instructive to use this example to illustrate how one sets up a solution based upon (12.13.15).⁶⁴

The MATLAB documentation provides the following information about the *mass matrix* and *differential-algebraic equation* (DAE) properties.⁶⁵

Mass Matrix and DAE Properties

This section describes *mass matrix* and *differential-algebraic equation* (DAE) properties, which apply to all the solvers except **ode15i**. These properties are not applicable to **ode15i** and their settings do not affect its behavior.

The solvers of the ODE suite can solve ODEs of the form $\mathbf{M}(t, \mathbf{x}) \frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x})$ with a mass matrix $\mathbf{M}(t, \mathbf{x})$ that can be sparse.⁶⁶

When $\mathbf{M}(t, \mathbf{x})$ is nonsingular, equation (12.13.15) is equivalent to $\frac{d\mathbf{x}}{dt} = \mathbf{M}(t, \mathbf{x})^{-1} \mathbf{f}(t, \mathbf{x})$ and the ODE has a solution for any initial values \mathbf{x}_0 at t_0 . The more general form, equation (12.13.15) is convenient when you express a model naturally in terms of a mass matrix. For large, sparse $\mathbf{M}(t, \mathbf{x})$, solving (12.13.15) directly reduces the storage and run-time needed to solve the problem.

When $\mathbf{M}(t, \mathbf{x})$ is singular, then $\mathbf{M}(t, \mathbf{x}) \frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x})$ is a differential algebraic equation (DAE). A DAE has a solution only when \mathbf{x}_0 is consistent; that is, there exists an initial

⁶⁴ A good discussion of solving ordinary differential equations with a mass matrix can be found in Shampine, L. F., I. Gladwell and S. Thompson, Solving ODEs with MATLAB, Cambridge University Press, 2003.

⁶⁵ <http://www.mathworks.com/help/matlab/ref/odeset.html>.

⁶⁶ A sparse matrix is a matrix with the property that most of its elements are zero.

slope $\left.\frac{d\mathbf{x}}{dt}\right|_0$ such that $\mathbf{M}(t_0, \mathbf{x}_0) \left.\frac{d\mathbf{x}}{dt}\right|_0 = \mathbf{f}(t_0, \mathbf{x}_0)$. If \mathbf{x}_0 and $\left.\frac{d\mathbf{x}}{dt}\right|_0$ are not consistent, the

solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE.

The **ode15s** and **ode23t** solvers can solve DAEs of index 1. For examples of DAE problems, see *Example: Differential-Algebraic Problem*, in the MATLAB Mathematics documentation, and the examples *amp1dae* and *hb1dae*.^{67,68,69}

The following table describes the mass matrix and DAE properties. Further information on each property is given following the table.

Mass Matrix and DAE Properties (Solvers Other Than **ode15i**)

Property	Value	Description
Mass	Matrix function handle	Mass matrix or a function that evaluates the mass matrix $\mathbf{M}(t, \mathbf{x})$
MStateDependence	none {weak} strong	Dependence of the mass matrix on \mathbf{x}
MyPattern	Sparse matrix	$\frac{\partial \mathbf{M}(t, \mathbf{x}) \mathbf{v}}{\partial \mathbf{x}}$ sparsity pattern
MassSingular	yes no {maybe}	Indicates whether the mass matrix is singular.
InitialSlope	Vector {zero vector}	Vector representing the consistent initial slope $\left.\frac{d\mathbf{x}}{dt}\right _0$

Description of Mass Matrix and DAE Properties

Mass: For problems of the form $\mathbf{M}(t) \frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x})$, set '**Mass**' to a mass matrix \mathbf{M} . For problems of the form $\mathbf{M}(t, \mathbf{x}) \frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x})$, set '**Mass**' to a function handle **@Mfun**, where

⁶⁷ <http://www.mathworks.com/help/matlab/math/ordinary-differential-equations.html#f1-670396>.

⁶⁸ http://www.mathworks.com/help/matlab/ref/rmvd_matlablink_2d2b72c4cbbb2a89f29bf7f220cf4bbf.html.

⁶⁹ http://www.mathworks.com/help/matlab/ref/rmvd_matlablink_32da45517b9c58cab12366de6c2acbcd.html.

Mfun(t,x) evaluates the mass matrix $\mathbf{M}(t, \mathbf{x})$. The **ode23s** solver can only solve problems with a constant mass matrix **M**. When solving DAEs, using **ode15s** or **ode23t**, it is advantageous to formulate the problem so that **M** is a diagonal matrix (a semiexplicit DAE).

For example problems, see *Finite Element Discretization* in the MATLAB Mathematics documentation, or the examples **fem2ode** or **batonode**.⁷⁰

MStateDependence: Set this property to **none** for problems $\mathbf{M}(t) \frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x})$.

Both **weak** and **strong** indicate $\mathbf{M}(t, \mathbf{x})$, but **weak** results in implicit solvers using approximations when solving algebraic equations.

MvPattern: Set this property to a sparse matrix S with $S_{ij} = 1$ if, for any k , the (i, k) component of $\mathbf{M}(t, \mathbf{x})$ depends on component j of \mathbf{x} , and 0 otherwise. For use with the **ode15s**, **ode23t**, and **ode23tb** solvers when **MStateDependence** is strong. See *burgersode* as an example.⁷¹

MassSingular: Set this property to **no** if the mass matrix is not singular and you are using either the **ode15s** or **ode23t** solver. The default value of **maybe** causes the solver to test whether the problem is a DAE, by testing whether $\mathbf{M}(t_0, \mathbf{x}_0)$ is singular.

Initialslope: Vector representing the consistent initial slope $\left. \frac{d\mathbf{x}}{dt} \right|_0$, where $\left. \frac{d\mathbf{x}}{dt} \right|_0$ satisfies $\mathbf{M}(t_0, \mathbf{x}_0) \left. \frac{d\mathbf{x}}{dt} \right|_0 = \mathbf{f}(t_0, \mathbf{x}_0)$. The default is the zero vector.

This property is for use with the **ode15s** and **ode23t** solvers when solving DAEs.

Example 12.13.4: As an illustration of how to setup a problem where the ordinary differential equation is in the generalized form (12.13.15), we shall again generate an approximate solution for the double pendulum. We shall continue to adopt the properties (12.13.10), repeated,

$$\begin{aligned} m_1 &= 2 \text{ slug} \\ m_2 &= 1 \text{ slug} \\ g &= 32.2 \text{ ft/sec}^2 \\ L_1 &= 1 \text{ ft} \\ L_2 &= 2 \text{ ft} \end{aligned} \tag{12.13.17}$$

the initial conditions (12.13.11), repeated,

⁷⁰ <http://www.mathworks.com/help/matlab/math/ordinary-differential-equations.html#f1-669854>.

⁷¹ http://www.mathworks.com/help/matlab/ref/rmvd_matlablink_58e882913cdd34b1a288b56667c780e4.html.

$$\begin{bmatrix} x_1(0) \\ x_2(0) \\ x_3(0) \\ x_4(0) \end{bmatrix} = \begin{bmatrix} \theta_1(0) \\ \theta_2(0) \\ \frac{d\theta_1(0)}{dt} \\ \frac{d\theta_2(0)}{dt} \end{bmatrix} = \begin{bmatrix} \frac{\pi}{2} \\ \pi \\ 0 \\ 0 \end{bmatrix} \quad (12.13.18)$$

and the time interval

$$\text{tspan} = [0, 10] \quad (12.13.19)$$

It follows from (12.13.16) that the mass matrix is given by

$$M(t, \mathbf{x}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & (m_1 + m_2)l_1^2 & m_2 l_1 l_2 \cos(x_1(t) - x_2(t)) \\ 0 & 0 & m_2 l_1 l_2 \cos(x_1(t) - x_2(t)) & m_2 l_2^2 \end{bmatrix} \quad (12.13.20)$$

As explained above, this matrix is nonsingular. It is defined by a function m-file we shall call `M.m` and that has the following script:

```
function mass=M(t,x,m1,m2,L1,L2,g)
mass=[1,0,0,0;...
      0,1,0,0;...
      0,0,(m1+m2)*L1^2,m2*L1*L2*cos(x(1)-x(2));...
      0,0,m2*L1*L2*cos(x(1)-x(2)),m2*L2^2];
```

Based upon our conclusions in Example 12.13.1, we shall continue to take `RelTol = 10^-6`. Next, the mass matrix (12.13.20) depends upon `x`. The instructions above about `MStateDependence` tell us that this option is either `strong` or `weak`. Our choice will be `strong` because we are not going to utilize an implicit solver. As a result of this discussion, our script that will attempt to find the approximate solution will include the script

```
options=odeset('Mass',@M,...
              'MStateDependence','strong',...
              'RelTol',10^(-6))
```

Finally, we need to create a function m-file that defines the right side of the differential equation (12.13.16). The following script defines the function m-file `f12134.m` that serves our purposes.

```

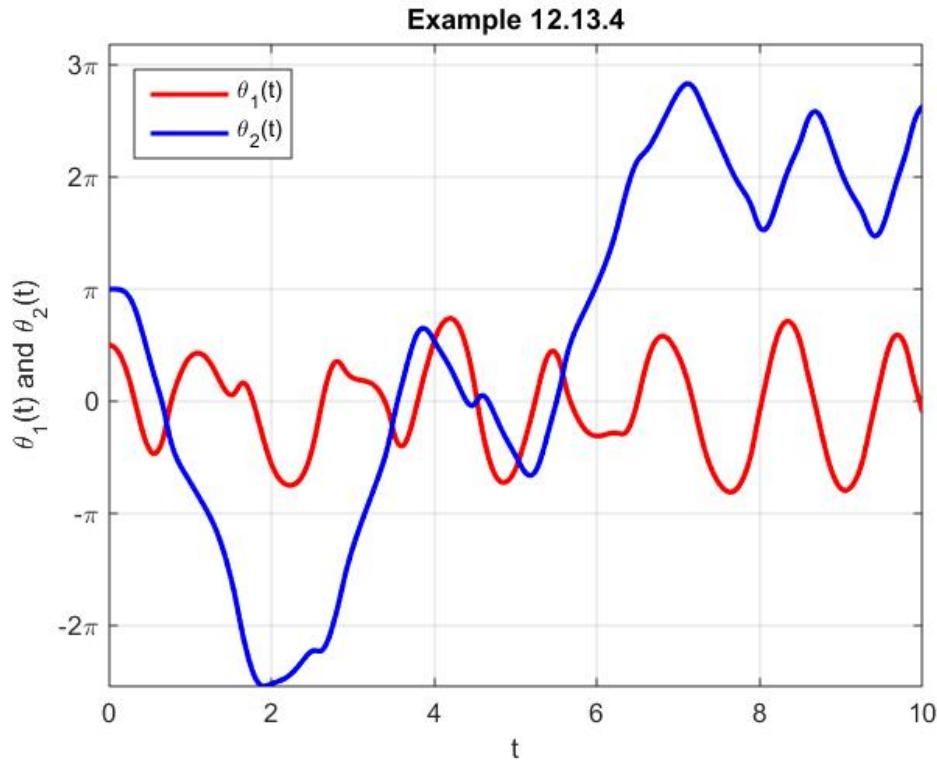
function dxdt=f12134(t,x,m1,m2,L1,L2,g)
dxdt=zeros(4,1), %Preallocate
dxdt=[x(3);x(4);...
-m2*L1*L2*x(4)^2*sin(x(1)-x(2))-...
(m1+m2)*g*L1*sin(x(1));...
m2*L1*L2*x(3)^2*sin(x(1)-x(2))-...
m2*g*L2*sin(x(2))]
```

Given the two function m-files, **M.m** and **f12134.m**, the approximate solution to the initial value problem (12.13.18) with the properties (12.13.17) over the time interval (12.13.19) is generated by the script

```

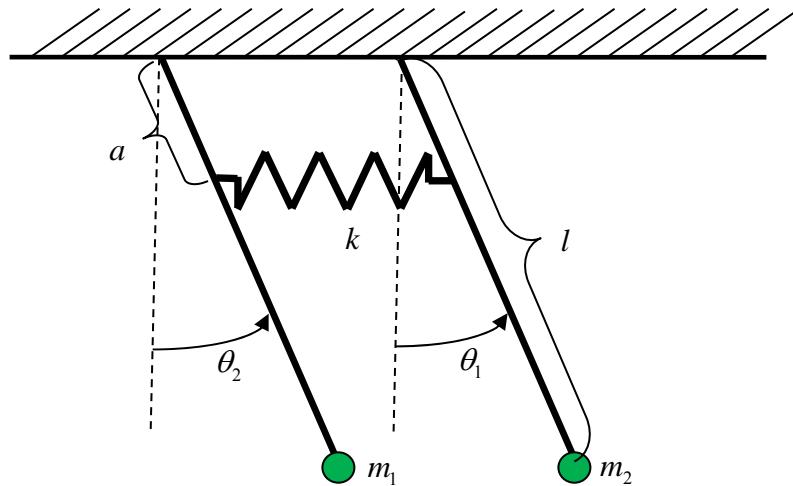
clc
clear
m1 = 2;
m2 = 1;
g = 32.2;
L1 = 1;
L2 = 2;
x0 = [pi/2,pi,0,0];
tspan=[0,10];
options=odeset('Mass',@M,...
'MStateDependence','strong',...
'RelTol',10^(-6))
[t x] = ode45(@f12134,tspan,x0,options,m1,m2,L1,L2,g);
figure
plot(t,x(:,1),'r','LineWidth',2)
hold on
plot(t,x(:,2),'b','LineWidth',2)
grid on
xlabel('t')
ylabel('\theta_1(t) and \theta_2(t)')
legend ('\theta_1(t)', '\theta_2(t)',...
'Location','Northwest')
set(gca,'YTick',[-3*pi:pi:3*pi],...
'YTickLabel',...
{ '-3\pi', '-2\pi', '-\pi', '0', ...
'\pi', '2\pi', '3\pi' })
title('Example 12.13.4')
```

If this script is executed, the result is the figure



These curves are the same as the corresponding curves in Example 12.13.2.

Example 12.13.5: A *coupled pendulum* is the two pendulum device shown in the following figure



It consists of two simple pendulums of equal length connected by a linear spring at the point shown. The equations of motion for a coupled pendulum are

$$\begin{aligned} m_1 l^2 \frac{d^2 \theta_1}{dt^2} &= -m_1 g l \sin \theta_1 - k a^2 (\theta_1 - \theta_2) \\ m_2 l^2 \frac{d^2 \theta_2}{dt^2} &= -m_2 g l \sin \theta_2 + k a^2 (\theta_1 - \theta_2) \end{aligned} \quad (12.13.21)$$

In this example, we shall adopt the following numerical values for the properties that appear in (12.13.21).

$$\begin{aligned} m_1 = m_2 &= 20 \text{ kg} \\ k &= 40 \text{ N/m} \\ l &= 2 \text{ m} \\ a &= \frac{1}{2} \text{ m} \\ g &= 9.81 \text{ m/sec}^2 \end{aligned} \quad (12.13.22)$$

Also, we shall adopt the initial conditions

$$\theta_1(0) = \frac{d\theta_1(0)}{dt} = \frac{d\theta_2(0)}{dt} = 0, \quad \theta_2(0) = \frac{\pi}{4} \quad (12.13.23)$$

We are interested in finding an approximate solution to the system (12.13.21) subject to the initial conditions (12.13.23) over the interval $0 < t < 130$.

The normal form of (12.13.21) is obtained in the usual way. If we define

$$\begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \frac{d\theta_1}{dt} \\ \frac{d\theta_2}{dt} \end{bmatrix} \quad (12.13.24)$$

Then the usual calculation yields ⁷²

⁷² In the case where $\theta_1(t)$ and $\theta_2(t)$ are small, equation (12.13.25) can be replaced by the linear equation

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} \frac{d\theta_1}{dt} \\ \frac{d\theta_2}{dt} \\ \frac{d^2\theta_1}{dt^2} \\ \frac{d^2\theta_2}{dt^2} \end{bmatrix} = \begin{bmatrix} \frac{d\theta_1}{dt} \\ \frac{d\theta_2}{dt} \\ -\frac{g}{l} \sin \theta_1 - \frac{ka^2}{m_1 l^2} (\theta_1 - \theta_2) \\ -\frac{g}{l} \sin \theta_2 + \frac{ka^2}{m_2 l^2} (\theta_1 - \theta_2) \end{bmatrix} = \begin{bmatrix} x_3 \\ x_4 \\ -\frac{g}{l} \sin x_1 - \frac{ka^2}{m_1 l^2} (x_1 - x_2) \\ -\frac{g}{l} \sin x_2 + \frac{ka^2}{m_2 l^2} (x_1 - x_2) \end{bmatrix} \quad (12.13.25)$$

The function m-file that defines this ordinary differential equation is given the name **f12135.m** and is defined by the script

```
function dxdt=f12135(t,x,m1,m2,L,a,k,g)
dxdt=zeros(4,1), %Preallocate
dxdt=[x(3);x(4);-g/L*sin(x(1))-k*a^2/m1/L^2*(x(1)-x(2));...
-g/L*sin(x(2))+k*a^2/m2/L^2*(x(1)-x(2))]
```

The script that generates the desired solution and plot is

```
clc
clear
m1=40
m2=40
L=6
a=.9*L
k=7/4
g=9.81
x0=[0,pi/4,0,0]
tspan=[0,300]
[t,x]=ode45(@f12135,tspan,x0,[],m1,m2,L,a,k,g)
plot(t,x(:,1),'r','LineWidth',1)
hold on
```

$$\frac{d}{dt} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\left(\frac{g}{l} + \frac{ka^2}{m_1 l^2}\right) & \frac{ka^2}{m_1 l^2} & 0 & 0 \\ \frac{ka^2}{m_2 l^2} & -\left(\frac{g}{l} + \frac{ka^2}{m_2 l^2}\right) & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix}$$

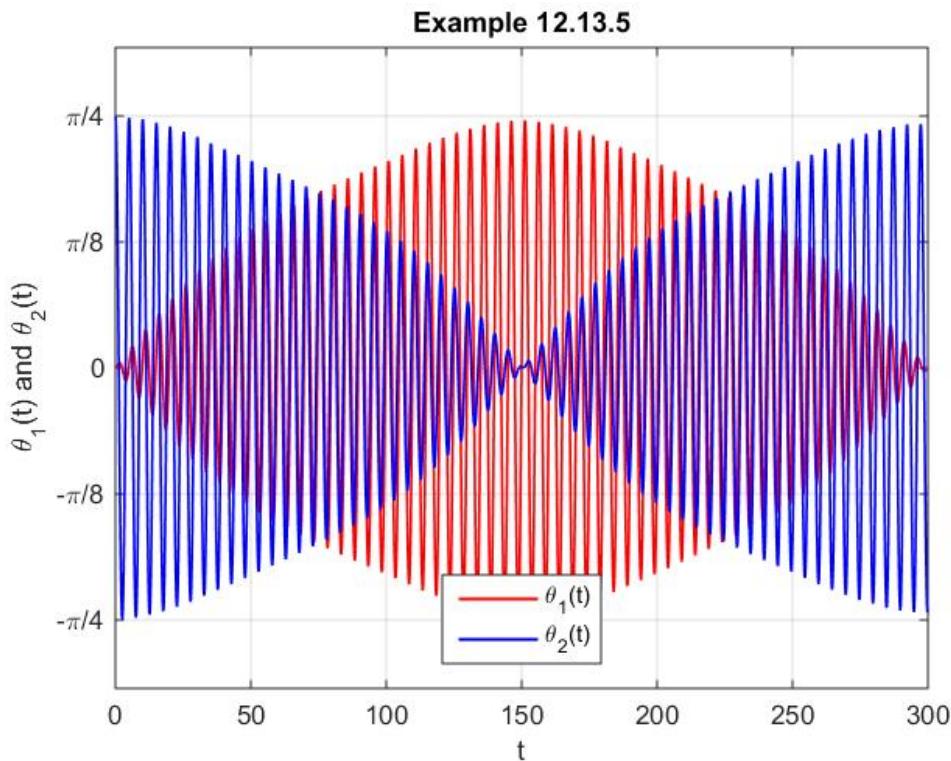
This equation can be analyzed by use of the methods discussed in Section 5.5.

```

plot(t,x(:,2),'b','LineWidth',1)
grid on
xlabel('t')
ylabel('\theta_1(t) and \theta_2(t)')
axis([0,tspan(2),-1,1])
legend ('\theta_1(t)', '\theta_2(t)', 'Location', 'South')
set(gca,'YTick',[-pi/4:pi/8:pi/4],...
    'YTickLabel',...
    {'-\pi/4', '-\pi/8', '0', ...
    '\pi/8', '\pi/4'})
title('Example 12.13.5')

```

Note that the above script adopts the default value of **RelTol**. One can confirm this choice by generating the solution for smaller values. If the above script is executed, the resulting figure is



This figure shows that the coupling between the two pendulums causes the right pendulum to transfer its energy, thus its motion, to the pendulum on the left in approximately 150 seconds. At that approximate time, the motion is transferred back to the right pendulum. This periodic motion continues in time because the coupled pendulum model does not contain dissipation.

Like our other examples, one can use MATLAB to create an animation of the motion of the coupled pendulum. The following MATLAB script creates this animation.^{73,74}

```

clc
clear
m1=40;
m2=40;
L=6;
a=.9*L;
k=7/4;
g=9.81;
x0=[0,pi/4,0,0];
tspan=[0,300];
[t,x]=ode45(@f12135,tspan,x0,[],m1,m2,L,a,k,g);
subplot(1,2,1)
hold on
xlabel('t')
ylabel('\theta_1(t) and \theta_2(t)')
grid on
title({'Example 12.13.5 Animation',...
    '\theta_1(0)=0 and \theta_2(0)=\pi/4'});
set(gca,'YTick',[ -pi/4:pi/8:pi/4],...
    'YTickLabel',...
    { '-\pi/4', '-\pi/8', '0',...
    '\pi/8', '\pi/4'});
axis([0,tspan(2),min(min(x(:,1:2))),max(max(x(:,1:2)))] )

subplot(1,2,2)
axis([-8 8 -8 1]);
axis off;
hold on
d1=-3;
d2=3;
x1=d1+L*sin(x(1,1));
y1=-L*cos(x(1,1));
x2=d2+L*sin(x(1,2));

```

⁷³ As with the earlier animations, MATLAB will create an mp4 video file of the animation. The additional script appended to that above

```

vid=VideoWriter('Example12135.mp4','MPEG-4')
vid.FrameRate = 10
open(vid)
writeVideo(vid,F([1:5:5*floor(length(t)/5),length(t)]))
close(vid)

```

will produce a video file **Example12133.mp4** of the animation. This video can be viewed from the electronic version of Appendix B of this work.

⁷⁴ There are several coupled pendulum animations online that are instructive to view.

```
y2=-L*cos(x(1,2));
x1a=d1+a*sin(x(1,1));
y1a=-a*cos(x(1,1));
x2a=d2+a*cos(x(1,2));
y2a=-a*sin(x(1,2));
ne=7;a1=2;ro=.4;
[xs,ys]=spring(x1a,y1a,x2a,y2a,ne,a1,ro);
% Note: spring.m simulates a two dimensional
% spring. This file can be downloaded from
% http://www.mathworks.com/matlabcentral/fileexchange
% /33168-springpendulum/content/SpringPendulum/Spring.m
hs=plot(xs,ys,'k','LineWidth',2);
h1=plot([d1 x1],[0 y1],'k','LineWidth',2);
h2=plot(x1,y1,'o','MarkerFaceColor','r',...
    'MarkerEdgeColor','k',...
    'MarkerSize',15);
h3=plot([d2 x2],[0 y2],'k','LineWidth',2);
h4=plot(x2,y2,'o','MarkerFaceColor','b',...
    'MarkerEdgeColor','k',...
    'MarkerSize',15);

plot(d1,0,'^','MarkerFaceColor','b',...
    'MarkerEdgeColor','k','MarkerSize',5);
plot(d2,0,'^','MarkerFaceColor','b',...
    'MarkerEdgeColor','k','MarkerSize',5);

for i=[1:5:5*floor(length(t)/5),length(t)];
    subplot(1,2,1)
    plot(t(1:i),x(1:i,1),'r','linewidth',1/2)
    hold on
    plot(t(1:i),x(1:i,2),'b','linewidth',1/2)
    legend('\theta_1(t)', '\theta_2(t)', 'Location', 'South')
    subplot(1,2,2)
    y1=-L*cos(x(i,1)); x1=d1+L*sin(x(i,1));
    y2=-L*cos(x(i,2)); x2=d2+L*sin(x(i,2));
    y1a=-a*cos(x(i,1));x1a=d1+a*sin(x(i,1));
    y2a=-a*cos(x(i,2));x2a=d2+a*sin(x(i,2));
    [xs,ys]=spring(x1a,y1a,x2a,y2a,ne,a,ro);
    delete(h1),delete(h2),delete(h3),delete(h4),delete(hs)
    hs=plot(xs,ys,'k','LineWidth',2);
    h1=plot([d1 x1],[0 y1],'k','LineWidth',2);
    h2=plot(x1,y1,'o','MarkerFaceColor','r',...
        'MarkerEdgeColor','k','MarkerSize',15);
    h3=plot([d2 x2],[0 y2],'k','LineWidth',2);
    h4=plot(x2,y2,'o','MarkerFaceColor','b',...
        'MarkerEdgeColor','k','MarkerSize',15);
```

```
title( [ 't = ' num2str(floor(t(i)))] )
F(i)=getframe(gcf);
end
```

Appendix A

INTRODUCTION TO MATLAB¹

This Appendix is intended to provide a brief and somewhat superficial introduction to the technical computing program MATLAB. This Appendix does not attempt to cover all of the MATLAB features utilized in this textbook. The textbook usually does attempt to explain the aspects of MATLAB not discussed here.²

MATLAB is a powerful but rather easy to use numerical tool. A major advantage of MATLAB is that it does not require a complicated language such as C and FORTRAN. It is structured around matrix based techniques to solve problems encountered by engineers, mathematicians and scientists. MATLAB integrates computation, programming and graph creation in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. The MATLAB programming language is intuitive and very straightforward to use since almost every data object is assumed to be a matrix array that does not require dimensioning. One of the best ways to learn it is to utilize the abundant online resources provided by MathWorks, the program creator. MATLAB tutorials and learning resources can be found at

http://www.mathworks.com/academia/student_center/tutorials/launchpad.html. A suggested lists of videos that are an invaluable aid to learning MATLAB are as follows:

1. “Interactive MATLAB Tutorial” video at
https://www.mathworks.com/academia/student_center/tutorials/register.html.
2. “Working in the Development Environment” video at
<http://www.mathworks.com/videos/working-in-the-development-environment-69021.html>.
3. “Writing a MATLAB Program” video at <http://www.mathworks.com/videos/writing-a-matlab-program-69023.html>.
4. “Working with Arrays” video at <http://www.mathworks.com/videos/working-with-arrays-in-matlab-69022.html>.

A list of MATLAB videos can be found at

<http://www.mathworks.com/products/matlab/videos.html#>.

¹ This APPENDIX is based in a significant way on notes that were prepared by my colleague, Dr. Waqar Malik, when, as a graduate student, he was aiding me teach an undergraduate course in Numerical Analysis at Texas A&M University.

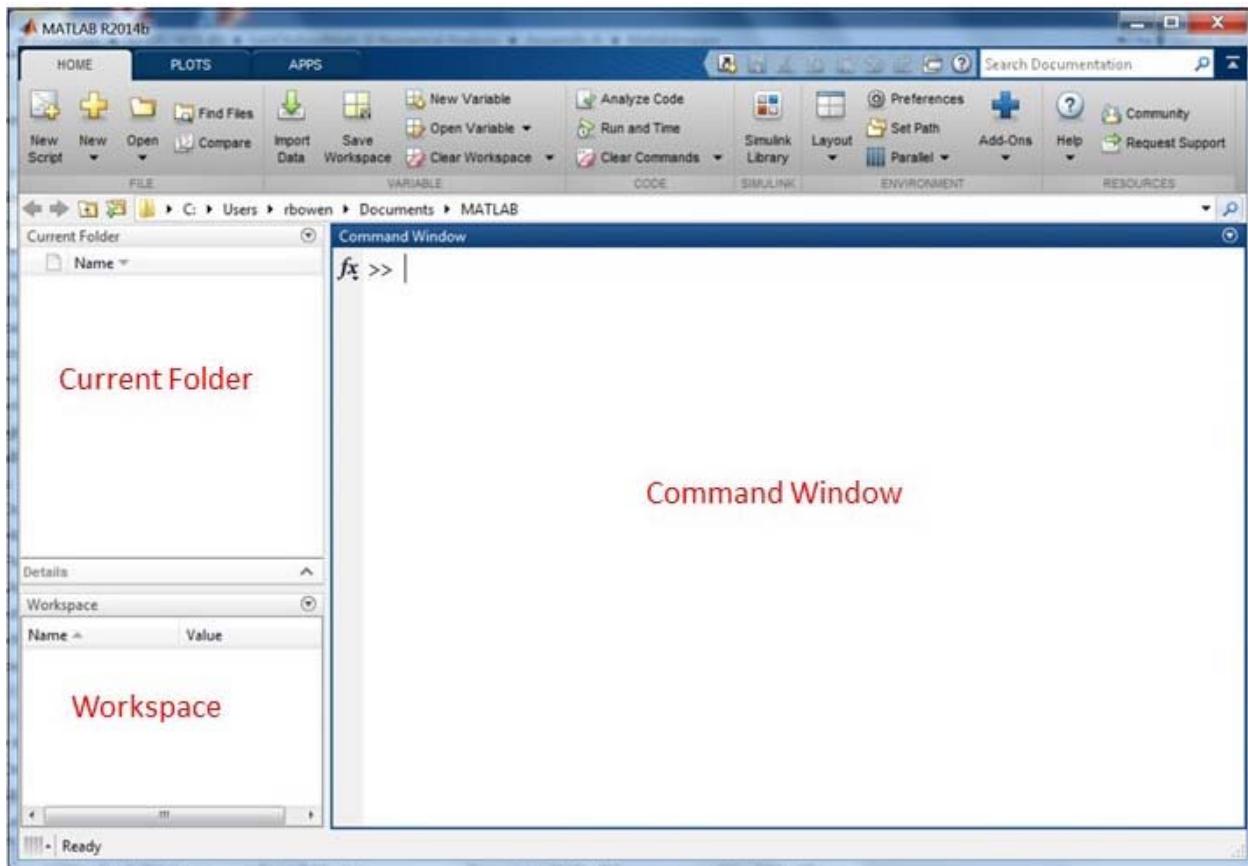
² The discussion in this Appendix is focused on MATLAB running on a computer running the Microsoft Windows operating system. MATLAB is also available for computers running the UNIX and the Apple Mac OS X operating systems. It should not be difficult to apply the introductory information in this appendix to the understanding of MATLAB running on computers utilizing these other operating systems.

Section A.1. Components and Features of MATLAB

The MATLAB program has the following components:

1. *Development Environment*, which contains the tools that enable users to develop programs. The Development Environment is also referred to as the MATLAB desktop.
2. *Mathematical Function Library*, which contains algorithms to handle some simple complex mathematical operations. This library contains a vast array of built in functions that are needed in the applications.
3. *MATLAB Language*, which is a highly flexible programming language, offering object-oriented programming features.
4. *Graphics*, which can be used to plot graphs and data visualization in two and three dimensional space. This feature of MATLAB can be of great use to you in your other courses throughout this semester.
5. *Application Programming Interface*, which enables MATLAB to work with programs written in other programming languages. We shall not make use of this MATLAB feature during this course.

MATLAB is started by simply double-clicking the MATLAB shortcut icon,  , on the computer's desktop. When MATLAB starts, the MATLAB desktop (Development Environment) appears, containing tools (graphical user interfaces) for managing files, variables, and applications associated with MATLAB. You can modify the desktop configuration to best meet your needs. The following figure illustrates the default configuration:

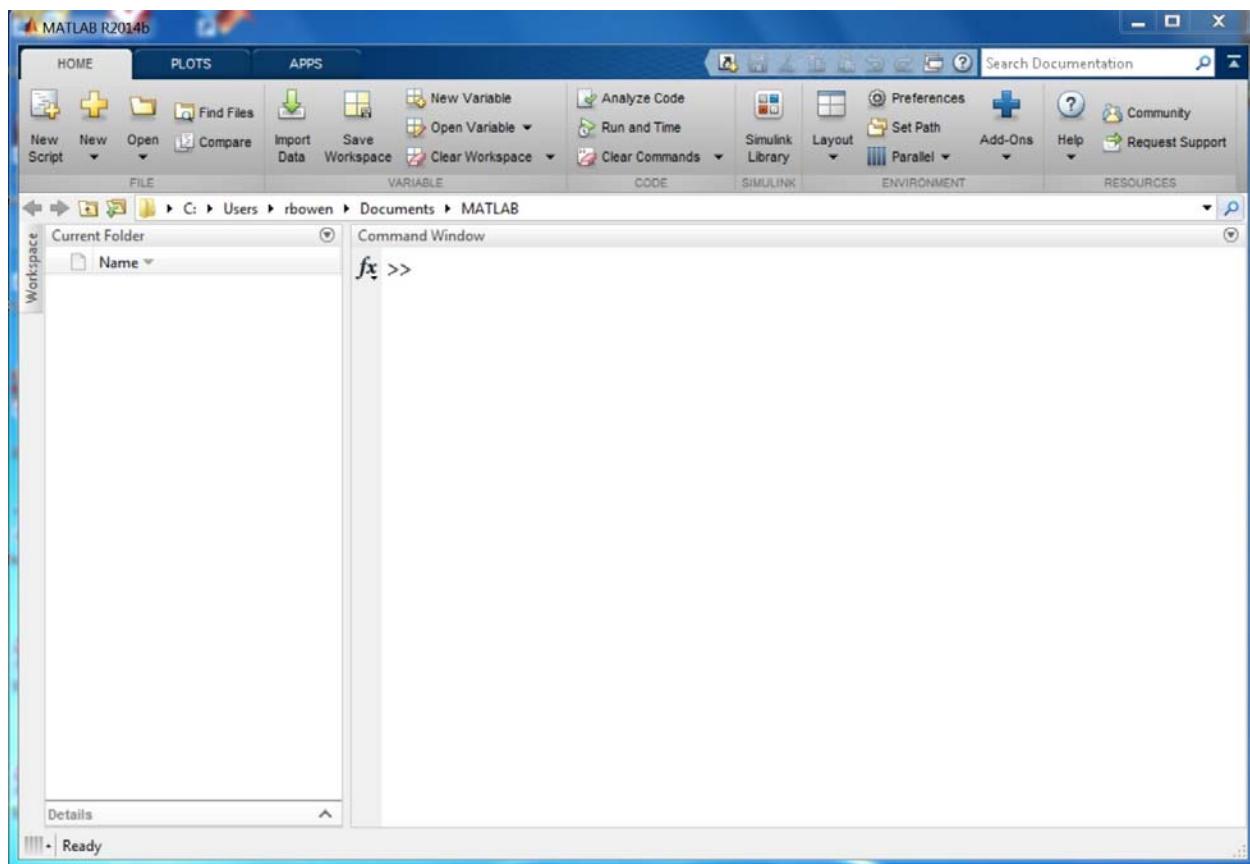


The different tools/parts of the desktop are described below along with their functionalities:³

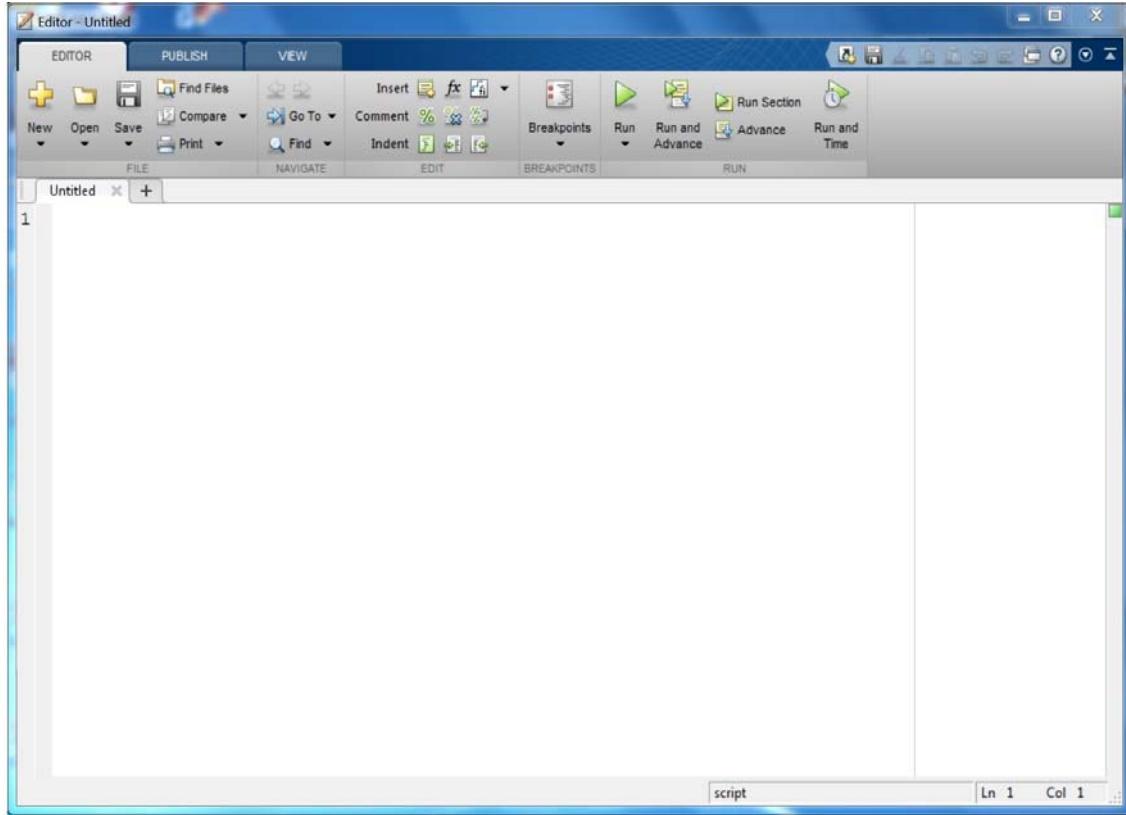
- Command Window:** The command window enables users to interact with MATLAB. It is a command-line prompt, where users can enter data, run MATLAB functions and other m-files (programs), and display results.
- Current Folder:** This window lists all the files in the current working directory. It enables the user to view files, perform file operations such as open, find files and file content, and manage and tune your files. An elementary but important point is that if you place a file in a directory that is not in the current working directory, then MATLAB cannot find it.
- Workspace:** The Workspace contains a list of all the variables in use. It shows the name of each variable, its value, its array size, its size in bytes, and the class. The icon for each variable denotes its class.

The following configuration is often more convenient because it closes the Workspace and leaves a larger Command Window.

³ The *Community* icon in the upper right of the screen connects to MATLAB CENTRAL. MATLAB CENTRAL is but one of the many online resources for MATLAB.



There are other important elements of MATLAB that we will need to utilize. The most important is the *Editor/Debugger*. When opened, the editor is



The editor is a text editor. It creates *pure text files* without the formatting found in word processors. It is similar but more powerful than the text editor Notepad found in Microsoft Windows. It is used to write the programs that cause MATLAB to execute. The files written using the editor are called *m-files*, and they are stored with *.m* extension. The debugger is used to find errors in the program. Errors in syntax are detected by MATLAB, and indicated in the command window. The error location is also specified. A related tool is the *Profiler*. This tool is used for testing the computational complexity of a program or function. It is a tool that shows you where an M-file is spending its time and can be used to identify sections for optimization. In this work, most of the programs are elementary and do not need to be improved with the Profiler.

Section A.2. Methods of Working with MATLAB

MATLAB is often used interactively, i.e. commands are typed in the Command Windows directly at the *command prompt* (`>>` prompt). Often these commands will look like standard arithmetic or function calls similar to many other computer languages.

- Three useful commands are
 - `clc` which clears the command window.
 - `clear` which removes items from the work space, i.e., clears them from the computer memory.
 - `clear x1 x2` clears the workspace variables `x1` and `x2`
- If you make an error while typing a command, you don't have to retype the whole command.
 - Instead of retyping the entire line, press the *up arrow key*. The previously typed line is redisplayed.
 - Repeated use of the up arrow key recalls earlier lines, from the current and previous sessions.
 - Using the up arrow key, you can recall any line maintained in the Command History window. If you happen to move too many commands backwards, you can hit the down arrow to move to the next command. Similarly, specify the first few characters of a line you entered previously and press the up arrow key to recall the previous line.

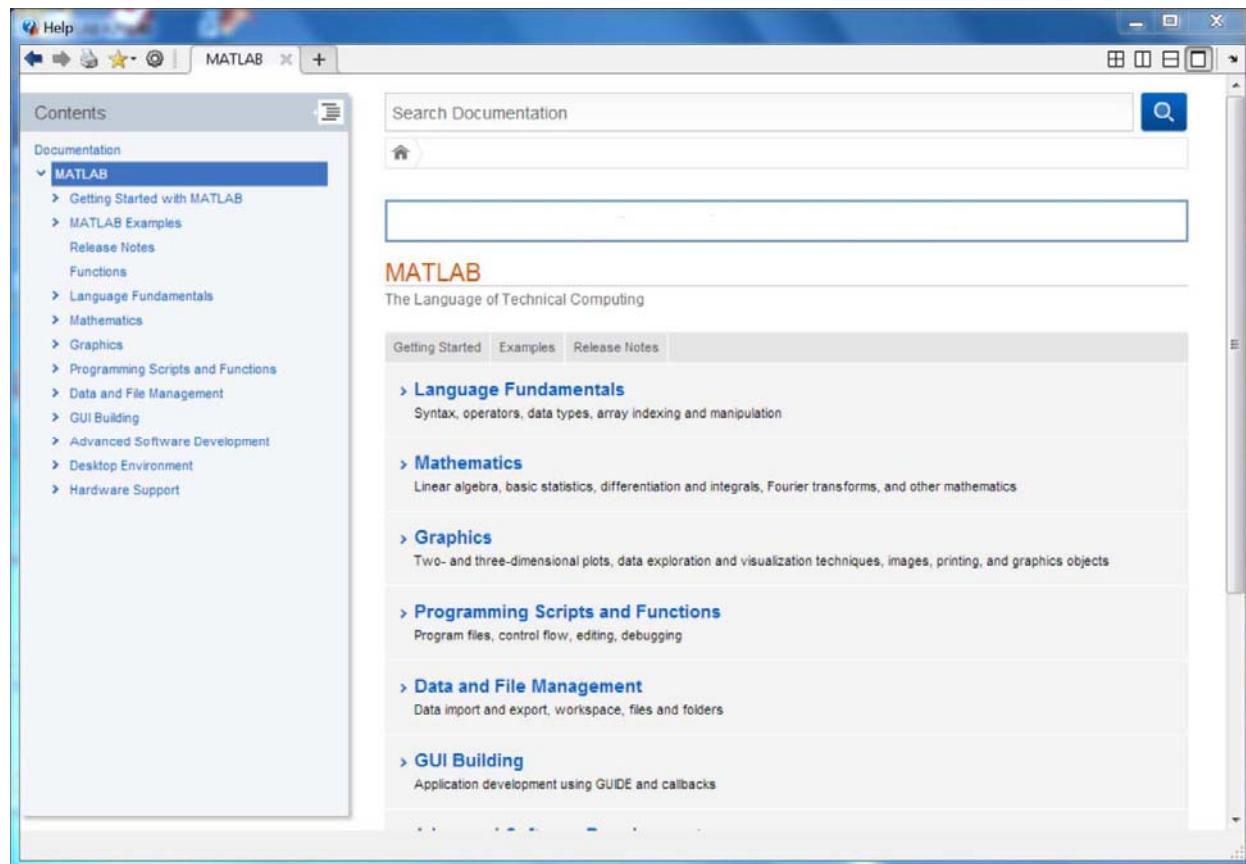
MATLAB is also an *interpreted programming language*. As mentioned above, you can create *m-files* with the editor. These files are text files, created with the editor, containing MATLAB commands. Once created, one can type the name of the name of the m-file at the command prompt and execute the commands in the file a line at a time.

The MATLAB language may be used to write your own functions and procedures which take arguments and return results. These functions must be defined in m-files, not at the command prompt. These files are called *function m-files*. Function m-files are called from the command prompt utilizing their names with arguments the function needs to execute. The name of a function m-file must be the same as the name of the function.

Obtaining help on how to use MATLAB is simple. In many cases, any help you need is provided by an internet search. The information available range from example problems to m-files designed to work specific problems to, as mentioned above, instructional videos. The MATLAB package itself has the majority of the help you will need. MATLAB has two main sources of help.

- The primary means for getting help is the *Help browser*, which provides documentation for all your installed products. Other forms of help are available including m-file help and Technical Support solutions. The Help browser is an excellent place for the study of a MATLAB topic that is new.

The Help browser has the look



- The other form of help is the **help** command entered directly into the Command Window. Typing **help** by itself gives a list of all the help topics. You can then get help on any of these topics by typing **help topicName**.

Section A.3. Vectors and Matrices in MATLAB

MATLAB is a *matrix-based* computing environment. All of the data that you enter into MATLAB is stored in the form of a matrix or a multidimensional array. Even a single numeric value like **100** is stored as a matrix. You will recall from Section 1.1 that a matrix is a rectangular array of numbers. More formally, an M by N *matrix* A is a rectangular array of real or complex numbers A_{ij} arranged in M rows and N columns as follows

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdot & \cdot & \cdot & A_{1N} \\ A_{21} & A_{22} & \cdot & \cdot & \cdot & A_{2N} \\ \cdot & & & & & \\ \cdot & & & & & \\ A_{M1} & A_{M2} & \cdot & \cdot & \cdot & A_{MN} \end{bmatrix} \quad (\text{A.3.1})$$

The MATLAB syntax way of writing (A.3.1) is

$$\mathbf{A} = [\mathbf{A}_{11} \ \mathbf{A}_{12} \ \dots \mathbf{A}_{1N}; \mathbf{A}_{21} \ \mathbf{A}_{22} \ \dots \mathbf{A}_{2N}; \dots; \mathbf{A}_{M1} \ \mathbf{A}_{M2} \ \dots \mathbf{A}_{MN}] \quad (\text{A.3.2})$$

An *equivalent* form of (A.3.2) is

$$\mathbf{A} = [\mathbf{A}_{11}, \mathbf{A}_{12}, \dots, \mathbf{A}_{1N}; \mathbf{A}_{21}, \mathbf{A}_{22}, \dots, \mathbf{A}_{2N}; \dots; \mathbf{A}_{M1}, \mathbf{A}_{M2}, \dots, \mathbf{A}_{MN}] \quad (\text{A.3.3})$$

Example A.3.1: A 3×3 matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad (\text{A.3.4})$$

would be entered into MATLAB by typing

$$\mathbf{A} = [1 \ 2 \ 3; 4 \ 5 \ 6; 7 \ 8 \ 9] \quad (\text{A.3.5})$$

at the command prompt.

The *row matrix* or *row vector* is a $1 \times N$ matrix, e.g.,

$$[A_{11} \ A_{12} \ \cdot \ \cdot \ \cdot \ A_{1N}] \quad (\text{A.3.6})$$

As above, the MATLAB syntax is

$$[\mathbf{A}_{11} \ \mathbf{A}_{12} \ \dots \mathbf{A}_{1N}] \quad (\text{A.3.7})$$

or

$$[\mathbf{A}_{11}, \mathbf{A}_{12}, \dots, \mathbf{A}_{1N}] \quad (\text{A.3.8})$$

The *column matrix* or *column vector* is an $M \times 1$ matrix, e.g.,

$$\begin{bmatrix} A_{11} \\ A_{21} \\ \cdot \\ \cdot \\ \cdot \\ A_{M1} \end{bmatrix} \quad (\text{A.3.9})$$

The MATLAB syntax for (A.3.9) is

$$[\mathbf{A}_{11}; \mathbf{A}_{21}; \dots; \mathbf{A}_{M1}] \quad (\text{A.3.10})$$

Example A.3.2:

$$A = \begin{bmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{bmatrix} \quad (\text{A.3.11})$$

As explained above, the matrix (A.3.11) is created by entering into the command window the string:

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

When enter is hit, the MATLAB output is

```
A =
```

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

If, for some reason, you wanted to suppress this output one can simply end the line that inputs the matrix with a semicolon. In addition to the numerical values, MATLAB also stores the size or dimension of the matrix. This information is obtained by the command

```
>> size(A)

ans =
    4      4
```

which shows the matrix A is a 4×4 matrix

Example A.3.3:

$$a = [-1 \quad 2 \quad -3 \quad 4] \quad (\text{A.3.12})$$

Therefore, enter into MATLAB :

```
>> a = [-1 2 -3 4]

a =
-1      2      -3      4
```

For a column vector

$$b = \begin{bmatrix} -1 \\ 2 \\ -3 \\ 4 \end{bmatrix} \quad (\text{A.3.13})$$

enter into MATLAB

```
>> b = [-1; 2; -3; 4]

b =
-1
```

```

2
-3
4

```

As mentioned above, if you terminate a command with a semi-colon, MATLAB will *suppress the printing* of the variable name and value resulting from the calculation.

```
>> b = [-1; 2; -3; 4];
```

b is formed in the same way, but the result is not displayed because of the semicolon at the end of the line.

To display any workspace variable, just type its name (without a terminating semicolon) :

```
>> a
a =
-1      2      -3      4
```

MATLAB contains many functions that allow special *square* matrices to be created. A few of these are shown in the table below:

zeros	All zeros
ones	All ones
eye	Ones on the diagonal and zeros elsewhere
diag	Diagonal matrix from a vector
rand	Uniformly distributed random elements
randn	Normally distributed random elements

Example A.3.4: As an example of the commands in the above table, consider

```
>> eye(4)
```

```
ans =
```

```

1      0      0      0
0      1      0      0

```

$$\begin{matrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

As explained in Chapter 1, the square matrix with ones down the diagonal and zeros everywhere else is called the *identity matrix*. The **eye** command is MATLAB's way of creating this matrix.

MATLAB contains commands that will provide useful information about matrices that have been created and stored in the command space. Earlier we introduced the **size** command. Additional examples are shown in the table below:

size	size of each dimension
length	size of longest dimension
ndims	number of dimensions
find	indices of nonzero elements

MATLAB does not require any type declarations or dimension statements. When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. Some of the properties of *Variables* are as follows:

- Variables in MATLAB must have names beginning with a letter, followed by any number of letters, digits, or underscores.
- Variables need not be declared prior to use.
- Variable names should be chosen so that they do not conflict with built in function or subroutine names, command names, or names of certain values/constants.
- The equality sign is used to assign values to variables.
- MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. **A** and **a** are not the same variable.

The table below shows some *built-in* variable names and numbers used in MATLAB.

Variable Name	Meaning
ans	value computed in an expression but not stored in a variable name
eps	floating point precision for the computer
i and j	imaginary unit in complex number
pi	$\pi=3.14159\dots$

NaN	“not a number”; crops up in undefined expressions (such as zero divided by zero) and for data gaps
inf	infinity; typically results from division by zero, or arithmetic overflow

The **format** command changes the form of and the number of significant figures of numbers displayed in the workspace.

Example A.3.5: Execute the commands

```
>> x=pi

x =
3.1416

>> format long      %Changes the format of numbers in the
workspace
>> x

x =
3.141592653589793

>> format long e    %Changes the format to long exponential
form
>> x

x =
3.141592653589793e+000

>> format short     %Changes the format back to the default
>> x

x =
3.1416

>>
```

The different formats just change the display. It does not change the number of significant figures stored in the memory.

The last example illustrates a useful feature of MATLAB. If you wish to insert a comment to explain a command or its output, the comment text is preceded by a per cent symbol, %.

Section A.4. Matrix Concatenation and Matrix Addressing in MATLAB

Because the entry of large matrices into the MATLAB command space can be tedious, MATLAB has tools that allow matrices to be constructed from others that have been previously entered. One such tool is call *concatenation*. In the simplest of terms concatenation is the process of joining small matrices to make bigger ones. As we shall see, the small matrices must have certain size compatibilities in order that they can be concatenated. The concatenation process is summarized in the following:

- The pair of square brackets, `[]`, is the *concatenation* operator.
 - The expression `C = [A B]` horizontally concatenates matrices **A** and **B**.
 - The expression `C = [A;B]` vertically concatenates matrices **A** and **B**.

Example A.4.1: If *A* is the matrix, $A = [1, 2, -3, 4]$, then

```
>> C=[A;2*A]
```

C =

$$\begin{matrix} 1 & 2 & -3 & 4 \\ 2 & 4 & -6 & 8 \end{matrix}$$

The following diagram shows two matrices of the same height (i.e., same number of rows) being combined horizontally to form a new matrix.

$$\begin{array}{c|cc} \begin{array}{|c|c|} \hline 7 & 23 \\ \hline 41 & 11 \\ \hline -1 & 90 \\ \hline \end{array} & + & \begin{array}{|c|c|c|c|} \hline 46 & 0 & 13 & -4 \\ \hline 44 & 62 & 31 & 98 \\ \hline 3 & 51 & -9 & 25 \\ \hline \end{array} & = & \begin{array}{|c|c|c|c|c|c|} \hline 7 & 23 & 46 & 0 & 13 & -4 \\ \hline 41 & 11 & 44 & 62 & 31 & 98 \\ \hline -1 & 90 & 3 & 51 & -9 & 25 \\ \hline \end{array} \\ \text{3-by-2} & & \text{3-by-4} & & \text{3-by-6} \end{array}$$

The next diagram illustrates an attempt to horizontally combine two matrices of unequal height. MATLAB does not allow this.

$$\begin{array}{c|cc} \begin{array}{|c|c|} \hline 7 & 23 \\ \hline 41 & 11 \\ \hline -1 & 90 \\ \hline \end{array} & + & \begin{array}{|c|c|c|c|} \hline 46 & 0 & 13 & -4 \\ \hline 44 & 62 & 31 & 98 \\ \hline \end{array} & \neq & \begin{array}{|c|c|c|c|c|c|} \hline 7 & 23 & 46 & 0 & 13 & -4 \\ \hline 41 & 11 & 44 & 62 & 31 & 98 \\ \hline -1 & 90 & & & & \end{array} \\ \text{3-by-2} & & \text{2-by-4} & & \end{array}$$

When utilizing matrices, MATLAB provides a simple method of identifying its elements. As explained in Chapter 1, in matrix algebra, when one is given a matrix A , the element in the i^{th} row and j^{th} column is written A_{ij} . MATLAB utilizes what is essentially the same labeling.

- To reference a particular element in a matrix, specify its row and column number using the following syntax: If \mathbf{A} is the matrix variable. The quantity $\mathbf{A}(\mathbf{i}, \mathbf{j})$ is the ij element of the matrix A .

Example A.4.2: Given the matrix A defined by (A.3.11) and repeated here,

$$A = \begin{bmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{bmatrix} \quad (\text{A.4.1})$$

we can extract the element on the 2nd row and 3rd column and assign it to a new variable \mathbf{z} :

```
>> z = A(2,3)
z =
    11
```

Often, you want MATLAB to generate a column or a row of a matrix. As an extension of the single element address above, you can address a row and a column by the commands

- To refer to *all* of the i^{th} row, use $\mathbf{A}(\mathbf{i}, :)$
- To refer to *all* of the j^{th} column, use $\mathbf{A}(:, \mathbf{j})$

Example A.4.3: As an illustration of the above indexing, calculate $\mathbf{A}(1, :)$ and $\mathbf{A}(2, :)$ for the matrix (A.4.1)

```
>> r1 = A(1,:)
r1 =
    16      3      2      13
>> c2 = A(:,2)
c2 =
```

```

3
10
6
15

```

Another generalization if the above addressing is to treat the indices **i** and **j** in the formula **A(i,j)** as vectors. In this way, we can extract from **A** its submatrices.

Example A.4.4:

```
>> A1 = A(:,[1 3 4])
```

```
A1 =
```

```

16      2      13
 5      11      8
 9      7      12
 4     14      1

```

This assigns to **A1** all (:) by itself means all) rows of **A** and columns 1, 3 and 4.

Example A.4.5:

```
>> A2 = A([2 3],:)
```

```
A2 =
```

```

5      10      11      8
9       6       7      12

```

(rows 2 and 3, and all of the columns)

```
>> A3 = A([2 3],[2 3])
```

```
A3 =
```

```

10      11
 6       7

```

(rows 2 and 3, columns 2 and 3)

We can extend the *colon notation* to specify a *sequence*, e.g. create a row vector **v** which starts at 1, with increments of 2 and stops at 10 :

```
>> v = 1:2:10
v =
1      3      5      7      9
```

If you omit the increment, it defaults to 1 :

```
>> v = 1:10
v =
1      2      3      4      5      6      7      8      9      10
```

Negative increments can be stored. For example

```
>> v=10:-2:1
v =
10      8      6      4      2
```

- The format is **first:step:last** The result is always a row vector, or the empty matrix if **last < first and step>0**.
 - Sometimes the sequence is written with *optional* brackets as

[first:step:last]

Example A.4.6: We can use this vector notation when referring to a sub matrix :

```
>> A4 = A(1:2:3, 2:4)
A4 =
3      2      13
6      7      12
```

(rows 1 and 3 (i.e. start at 1, increment by 2, stop at 3), columns 2,3,4 (i.e. start at 1, stop at 3 – default increment of 1 is used).

The **linspace** command also creates a row matrix. It is implemented as follows:

- linspace(a,b,n)** outputs a row vector of **n** *equally spaced* points starting with **a** and ending with **b**.

- The command **linspace** allows **a>b** or **b>a**.

Example A.4.7:

```
>> linspace(0,1,6)  
  
ans =  
  
    0    0.2000    0.4000    0.6000    0.8000    1.0000
```

If **n** is omitted, **linspace** automatically generates 100 equally spaced points.

Section A.5. Mathematical Operators in MATLAB

The common mathematical operations are implemented in MATLAB as follows:

- + addition
- - subtraction
- * multiplication
- / division
- ^ exponentiation
- \ left division
- ' transpose

Matrix multiplication, as defined in Chapter 1, is also implemented in MATLAB

Example A.5.1: Given the matrices

$$C = \begin{bmatrix} 3 & -2 \\ 2 & 4 \\ 1 & -3 \end{bmatrix} \text{ and } B = \begin{bmatrix} -2 & 1 & 3 \\ 4 & 1 & 6 \end{bmatrix} \quad (\text{A.5.1})$$

The MATLAB implementation of the multiplication CB is

```
>> C=[3,-2;2,4;1,-3]
```

```
C =
```

```
3      -2
2      4
1      -3
```

```
>> B=[-2,1,3;4,1,6]
```

```
B =
```

```
-2      1      3
4      1      6
```

```
>> A*B
```

```
ans =
```

```
-14      1      -3
12      6      30
-14     -2     -15
```

A similar set of steps yields

$$BC = \begin{bmatrix} -1 & -1 \\ 20 & -22 \end{bmatrix} \quad (\text{A.5.2})$$

MATLAB has two different types of arithmetic operations. They are referred to as *matrix* and *array* operations. These operations are defined as follows:

- Matrix
 - Matrix arithmetic operations are defined by the rules of linear algebra.
 - Matrix multiplication as illustrated in Example A.5.1 is an example matrix arithmetic operation
 - Array
 - Array arithmetic operations are carried out *element by element*, and can be used with multidimensional arrays

Example A.5.2: Given the matrices A defined in equation (A.4.1), repeated,

$$A = \begin{bmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{bmatrix} \quad (\text{A.5.3})$$

and the matrix D defined by

$$D = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad (\text{A.5.4})$$

yield a matrix product

$$AD = \begin{bmatrix} 218 & 252 & 286 & 320 \\ 258 & 292 & 326 & 360 \\ 258 & 292 & 326 & 360 \\ 218 & 252 & 286 & 320 \end{bmatrix} \quad (\text{A.5.5})$$

The *array multiplication* result is

$$A.*D = \begin{bmatrix} 16*1 & 3*2 & 2*3 & 13*4 \\ 5*5 & 10*6 & 11*7 & 8*8 \\ 9*9 & 6*10 & 7*11 & 12*12 \\ 4*13 & 15*14 & 14*15 & 1*16 \end{bmatrix} = \begin{bmatrix} 16 & 6 & 6 & 52 \\ 25 & 60 & 77 & 64 \\ 81 & 60 & 77 & 144 \\ 52 & 210 & 210 & 16 \end{bmatrix} \quad (\text{A.5.6})$$

We will utilize array operations of a number of types. The following table summarizes some of the options.

Table of Operations

	Matrix Arithmetic	Array Arithmetic
Addition	A+B	
Subtraction	A-B	
Multiplication	A*B	A.*B
Division	A/B	A./B
Division (Right)	A\B	A.\B
Power	A^B	A.^B
Transpose	A'	A.'

The period character (.) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs .+ and .- are not used.

It is important to point out that the MATLAB implementation of the transpose involves two steps. The first is interchange of rows and columns as explained in Section 1.9 and the second is forming the complex conjugate of each element. The result is the adjoint matrix as discussed in Section 4.9. For the array version of the transpose, the only action is that the rows and columns are interchanged.

MATLAB has a long list of built in elementary functions. If you type

```
>> help elfun
```

the output is the list of built in elementary math functions. The list is

Trigonometric.

sin	- Sine.
sind	- Sine of argument in degrees.
sinh	- Hyperbolic sine.
asin	- Inverse sine.

asind	- Inverse sine, result in degrees.
asinh	- Inverse hyperbolic sine.
cos	- Cosine.
cosd	- Cosine of argument in degrees.
cosh	- Hyperbolic cosine.
acos	- Inverse cosine.
acosd	- Inverse cosine, result in degrees.
acosh	- Inverse hyperbolic cosine.
tan	- Tangent.
tand	- Tangent of argument in degrees.
tanh	- Hyperbolic tangent.
atan	- Inverse tangent.
atand	- Inverse tangent, result in degrees.
atan2	- Four quadrant inverse tangent.
atan2d	- Four quadrant inverse tangent, result in degrees.
atanh	- Inverse hyperbolic tangent.
sec	- Secant.
secd	- Secant of argument in degrees.
sech	- Hyperbolic secant.
asec	- Inverse secant.
asecd	- Inverse secant, result in degrees.
asech	- Inverse hyperbolic secant.
csc	- Cosecant.
cscd	- Cosecant of argument in degrees.
csch	- Hyperbolic cosecant.
acsc	- Inverse cosecant.
acsqd	- Inverse cosecant, result in degrees.
acsch	- Inverse hyperbolic cosecant.
cot	- Cotangent.
cotd	- Cotangent of argument in degrees.
coth	- Hyperbolic cotangent.
acot	- Inverse cotangent.
acotd	- Inverse cotangent, result in degrees.
acoth	- Inverse hyperbolic cotangent.
hypot	- Square root of sum of squares.

Exponential.

exp	- Exponential.
expml	- Compute $\exp(x)-1$ accurately.
log	- Natural logarithm.
log1p	- Compute $\log(1+x)$ accurately.
log10	- Common (base 10) logarithm.
log2	- Base 2 logarithm and dissect floating point number.
pow2	- Base 2 power and scale floating point number.
realpow	- Power that will error out on complex result.
reallog	- Natural logarithm of real number.

realsqrt	- Square root of number greater than or equal to zero.
sqrt	- Square root.
nthroot	- Real n-th root of real numbers.
nextpow2	- Next higher power of 2.

Complex.

abs	- Absolute value.
angle	- Phase angle.
complex	- Construct complex data from real and imaginary parts.
conj	- Complex conjugate.
imag	- Complex imaginary part.
real	- Complex real part.
unwrap	- Unwrap phase angle.
isreal	- True for real array.
cplxpairs	- Sort numbers into complex conjugate pairs.

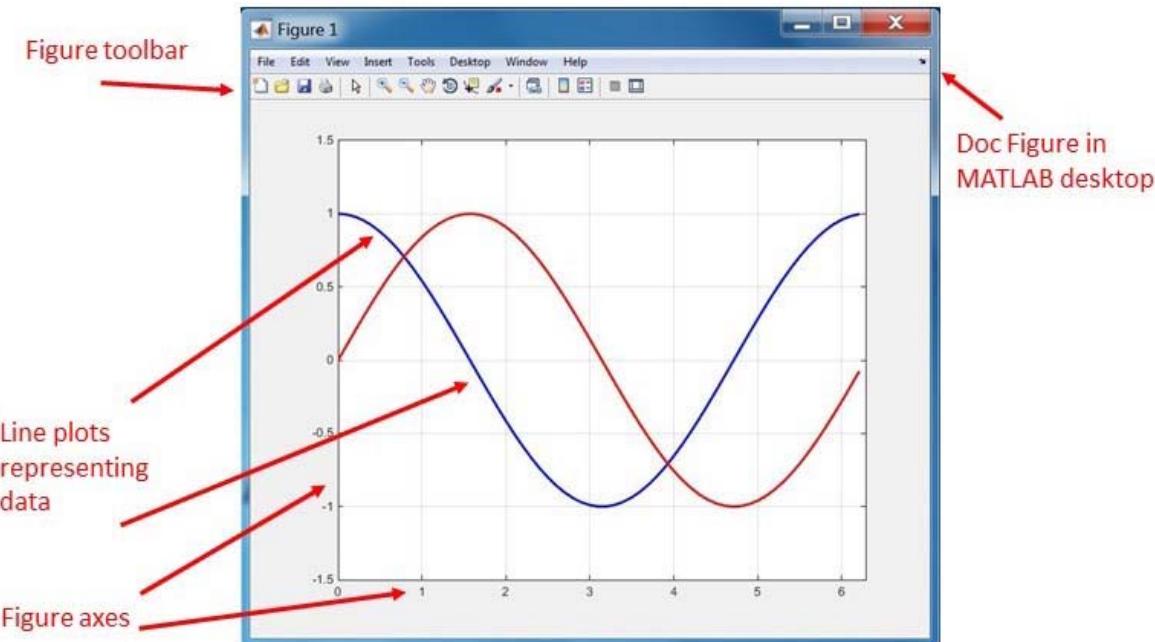
Rounding and remainder.

fix	- Round towards zero.
floor	- Round towards minus infinity.
ceil	- Round towards plus infinity.
round	- Round towards nearest integer.
mod	- Modulus (signed remainder after division).
rem	- Remainder after division.
sign	- Signum.

Section A.6. Creating Plots with MATLAB

One of the major benefits of utilizing MATLAB during the preparation of a textbook is its plotting ability. As this textbook and the many other textbooks that utilize MATLAB illustrate, its ability to display mathematical results graphically is an enormous benefit to the understanding of the subject being discussed. In this appendix, we shall not be able to cover all of the plotting applications that will be found throughout the textbook, but we have tried to provide sufficient information in this appendix, throughout the text and in the online links referenced to enable the reader to understand the applications and to learn how to exploit MATLAB's plotting capabilities. Concerning online links, the link <http://www.mathworks.com/discovery/gallery.html> is to a MATLAB Plot Gallery which provides examples of plots and the associated MATLAB script for a wide variety of plots. It is especially useful as one learns MATLAB and as one adapts MATLAB to specialized graphics problems.

MATLAB plotting functions and tools direct their output to a window that is separate from the Command Window. In MATLAB this window is referred to as a **figure**.



The most fundamental plotting command is **plot**.⁴

⁴ MATLAB has many different plotting commands designed for specific purposes. The website http://www.mathworks.com/discovery/gallery.html?s_v1=47603336_1-15KHMB contains the MATLAB Plot Gallery. It provides many examples of plots that can be created with MATLAB.

- It plots points, given by vectors of **x** and **y** coordinates, with straight lines drawn in between them.
 - The plot function has different forms depending on the input arguments.
 - For example, if **y** is a vector, **plot(y)** produces a linear graph of the elements of **y** versus the index of the elements of **y**.
 - If you specify two vectors as arguments, **plot(x,y)** produces a graph of **y** versus **x**.

Example A.6.1: Perhaps the best way to illustrate MATLAB's graphical features is to construct an elementary plot and then display MATLAB's capabilities by adding features to the plot. The plot we shall select is one utilized in Section 8.1. This plot is a simple two dimensional line plot. The objective is to plot the exponential function

$$y = e^x \quad (\text{A.6.1})$$

in the interval $x \in [-2, 2]$. The elementary syntax required to produce a plot is given at <http://www.mathworks.com/help/matlab/ref/plot.html?refresh=true>. This information explains that the line plot of x versus y is created with the MATLAB script

$$\text{plot}(x, y) \quad (\text{A.6.2})$$

where **x** is a vector (row or column) of x values to be plotted and **y** is a vector of length equal to that of **x** calculated by the rule (A.6.1). For example, if

$$x = [-2 : .1 : 2] \quad (\text{A.6.3})$$

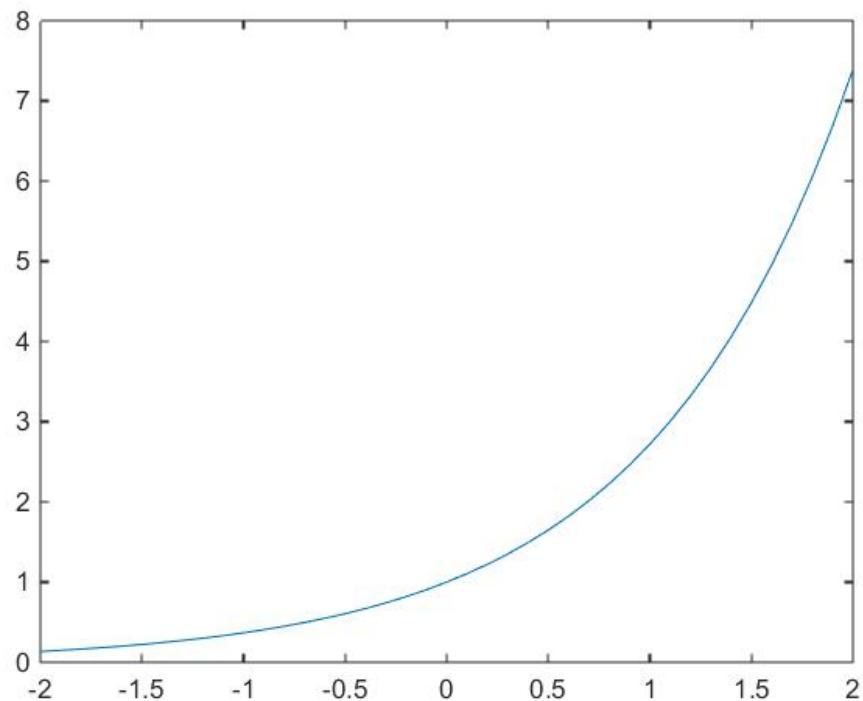
the length of **x** is 41 and the corresponding 41 values of **y** are

$$y = \exp(x) \quad (\text{A.6.4})$$

The MATLAB script

```
x=[ -2 : .1 : 2]
y=exp(x)
plot(x,y)
```

produces the figure

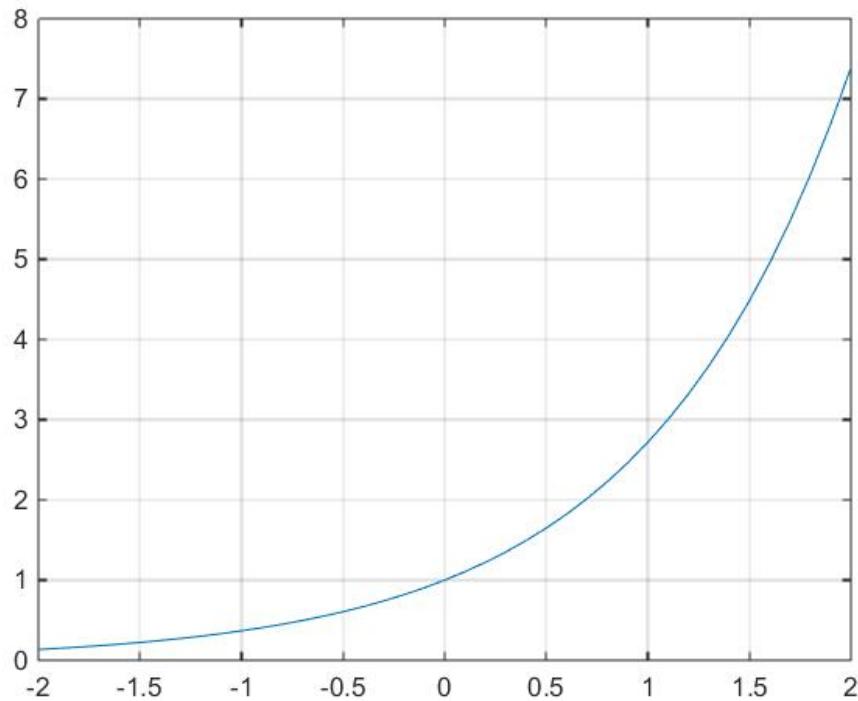


The choice (A.6.3) is made in order to produce a smooth curve. MATLAB automatically selects appropriate axis ranges and tick mark locations.

There are multiple ways this figure can be enhanced. The simple command

grid on (A.6.5)

replaces the above figure with

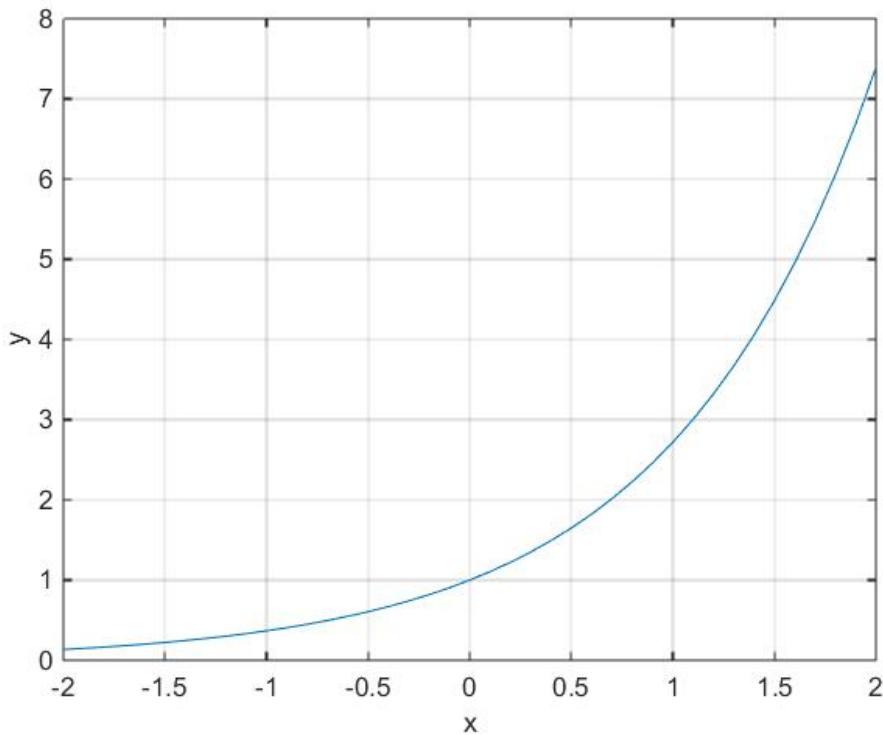


The two lines of script

```
xlabel('x')  
ylabel('y')
```

(A.6.6)

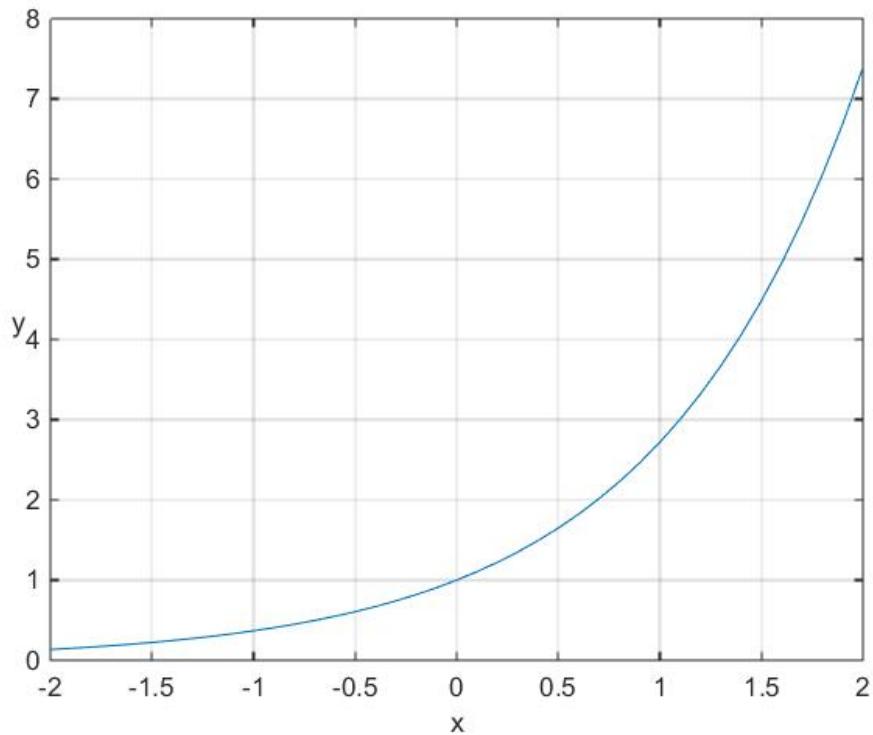
add axis labels to the above figure. The result is



The MATLAB default is to print the y-axis label rotated 90° as shown. The modified command

```
ylabel('y','Rotation',0) (A.6.7)
```

replaces the above figure with

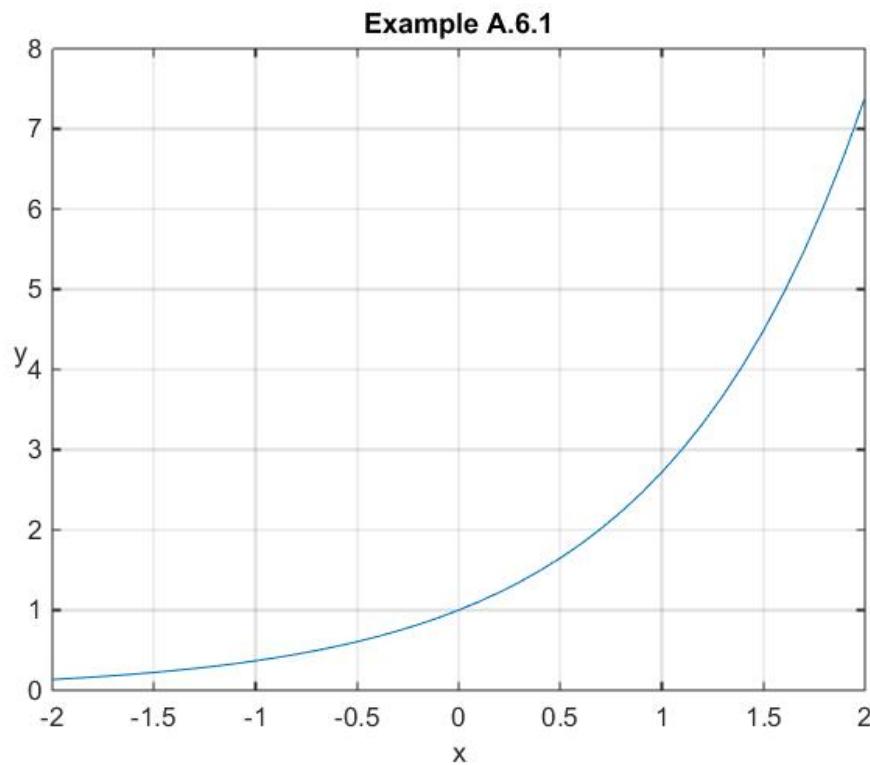


A title, Example A.9.1, can be added to the figure with the script

```
title('Example A.6.1')
```

 (A.6.8)

The result is the plot

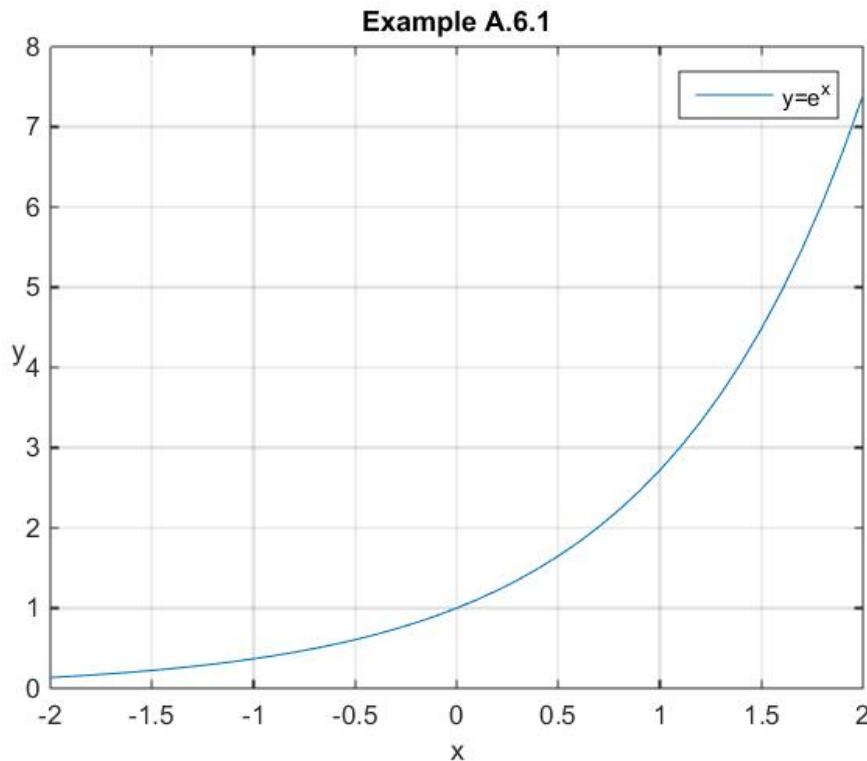


The next enhancement of the figure is to add a legend. The script

```
legend('y = e ^ x')
```

 (A.6.9)

adds a legend to the above figure as



MATLAB has the capability to modify the positions of the legend and the labels. It also has the capability to customize the partition of the x axis and the y axis. These features are illustrated by the many examples in the text. Of course, they are explained in MATLAB's online information.

The MATLAB script that summarizes the various features discussed thus far for this example is

```
x=[-2:.1:2]
y=exp(x)
plot(x,y)
grid on
xlabel('x')
ylabel('y','Rotation',0)
title('Example A.6.1')
legend('y=e^x')
```

The next feature we wish to illustrate is the superposition of additional line plots on the same set of axes. For example, if we wish to superimpose plots of the following three functions⁵

⁵ As explained in Section 8.1, these functions represent Taylor Series expansions of the function $y = e^x$.

$$\begin{aligned}y_1 &= 1 + x \\y_2 &= 1 + x + \frac{1}{2}x^2 \\y_3 &= 1 + x + \frac{1}{2}x^2 + \frac{1}{3!}x^3\end{aligned}\tag{A.6.10}$$

on the axes used above, we first execute the **hold** command with the script

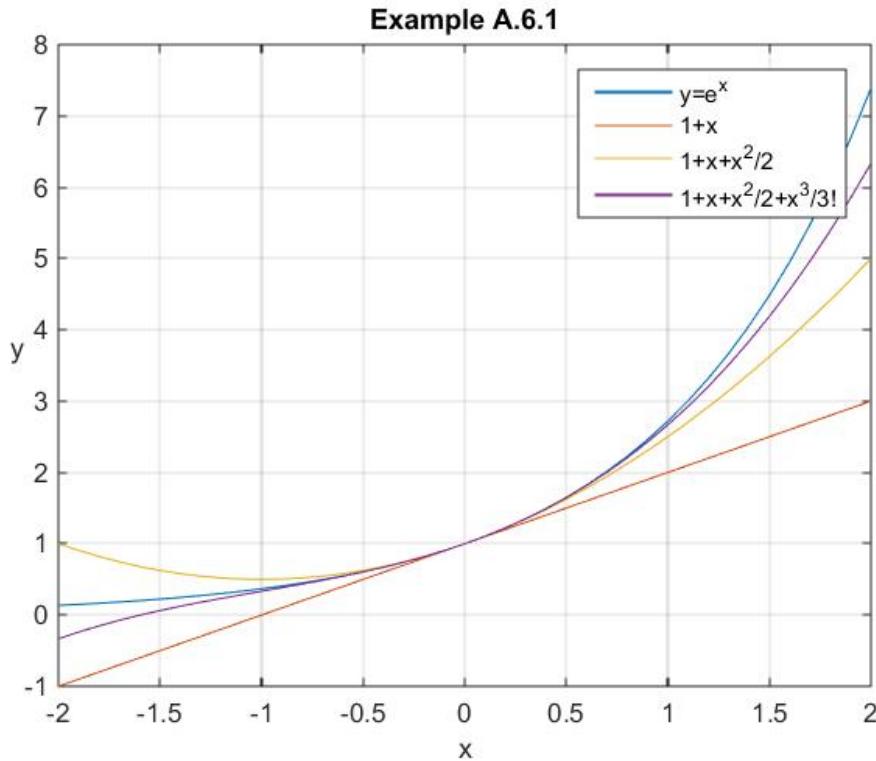
hold on (A.6.11)

The script **hold on** retains plots in the current axes so that new plots added to the axes do not delete the previous ones. These plots use the next line colors and line styles in the MATLAB list of defaults. As with all of MATLAB's default features, they can be changed with appropriate script. Given (A.6.11), if we modify the above script to

```
x=[-2:.1:2]
y=exp(x)
plot(x,y)
grid on
xlabel('x')
ylabel('y','Rotation',0)
title('Example A.6.1')

y1=1+x
y2=1+x+x.^2/2
y3=1+x+x.^2/2+x.^3/factorial(3)
hold on
plot(x,y1)
plot(x,y2)
plot(x,y3)
legend('y=e^x','1+x','1+x+x^2/2','1+x+x^2/2+x^3/3!')
```

the resulting multiline line plot is obtained



This figure reveals a need to place the legend in a position which does not cover all or part of the plot. MATLAB provides a method of placement both inside and outside of the axes. The many features of the **legend** command can be found at

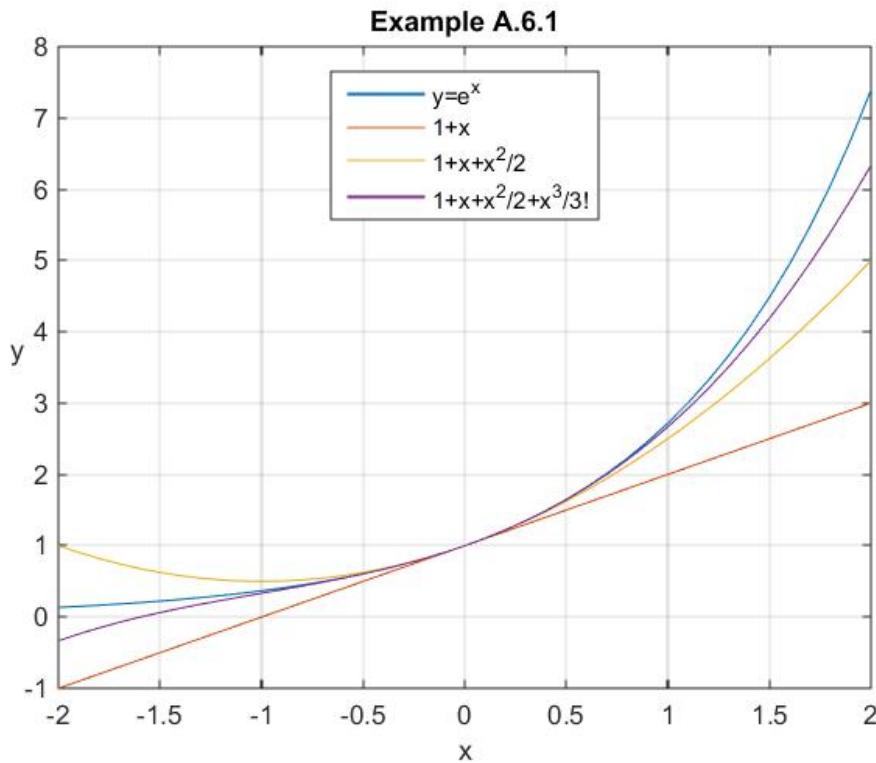
<http://www.mathworks.com/help/matlab/ref/legend.html#bt6r30y>. For our case, if the last line of script above is replaced by⁶

```
legend('y = e^x','1+x','1+x+x^2/2',...
      '1+x+x^2/2+x^3/3!',...
      'Location','North')
```

(A.6.12)

The resulting figure is

⁶ The script in (A.6.12) reveals a convenient MATLAB editing feature. One can break a long line of script at a comma by inserting a new line followed by ... and the remaining script of the particular command.



It is useful to note that when additional lines are added to a figure, circumstances can arise where MATLAB's default axes lengths need to be changed. This kind of change is implemented by the **axis** command. This command is discussed at <http://www.mathworks.com/help/matlab/ref/axis.html>. This command has many features, but for our immediate purposes the syntax

$$\text{axis}([\text{xmin } \text{xmax } \text{ymin } \text{ymax}]) \quad (\text{A.6.13})$$

will change the range of the x axis to $x_{\min} \leq x \leq x_{\max}$ and the range of the y axis to $y_{\min} \leq y \leq y_{\max}$.

There are circumstances where the individual lines need to be a different color, a different thickness or a different style. The details of how these changes are implemented are probably best found online. Our many examples in the text do illustrate how to achieve these refinements. The following table gives a quick and limited view of the basic plotting function and certain of its options:

Function	Operation
Create a plot	<code>>> plot (x1, y1, linestyle1, x2, y2, linestyle2)</code>
Set the current plot	<code>>> figure (1)</code>

Retains plots in the current axes so that new plots added to the axes do not delete existing ones. New plots use the next colors and line styles based on MATLAB defaults. ⁷	<code>>> hold on</code>
Sets the hold state to off so that new plots added to the axes clear existing plots and reset all axes properties.	<code>>> hold off</code>
Change the x or y label of the current plot ⁸	<code>>> xlabel ('New Axis Name')</code>
Change the title of the current plot ⁹	<code>>> title ('New Title')</code>
Change the Axis Limits ¹⁰	<code>>> axis([xmin xmax ymin ymax])</code>
Place a textbox on the graph ¹¹	<code>>> text (x,y,'Text Box Contents')</code>
Turn on/off the grid	<code>>> grid on/off</code>

Plot styles are determined by three properties: **Line Type**, **Point Type**, and **Color**. The following are a few available types.

Line Type	Point Type	Color
'-' Solid	.	r Red
-- Dashed	*	g Green
:' Dotted	o	b Blue
'-. Dash-dot	+	w White
	x	k Black
		i Invisible

⁷ See <http://www.mathworks.com/help/matlab/ref/hold.html>.

⁸ See <http://www.mathworks.com/help/matlab/ref/xlabel.html>.

⁹ See <http://www.mathworks.com/help/matlab/ref/title.html>.

¹⁰ See <http://www.mathworks.com/help/matlab/ref/axis.html>.

¹¹ See <http://www.mathworks.com/help/matlab/ref/text.html>.

At several places in the text, it is convenient to create several plots in a single figure. This construction is achieved by use of MATLAB's **subplot** command. This command is discussed at <http://www.mathworks.com/help/matlab/ref/subplot.html>. The basic syntax of the **subplot** command is

$$\text{subplot}(m,n,p) \quad (\text{A.6.14})$$

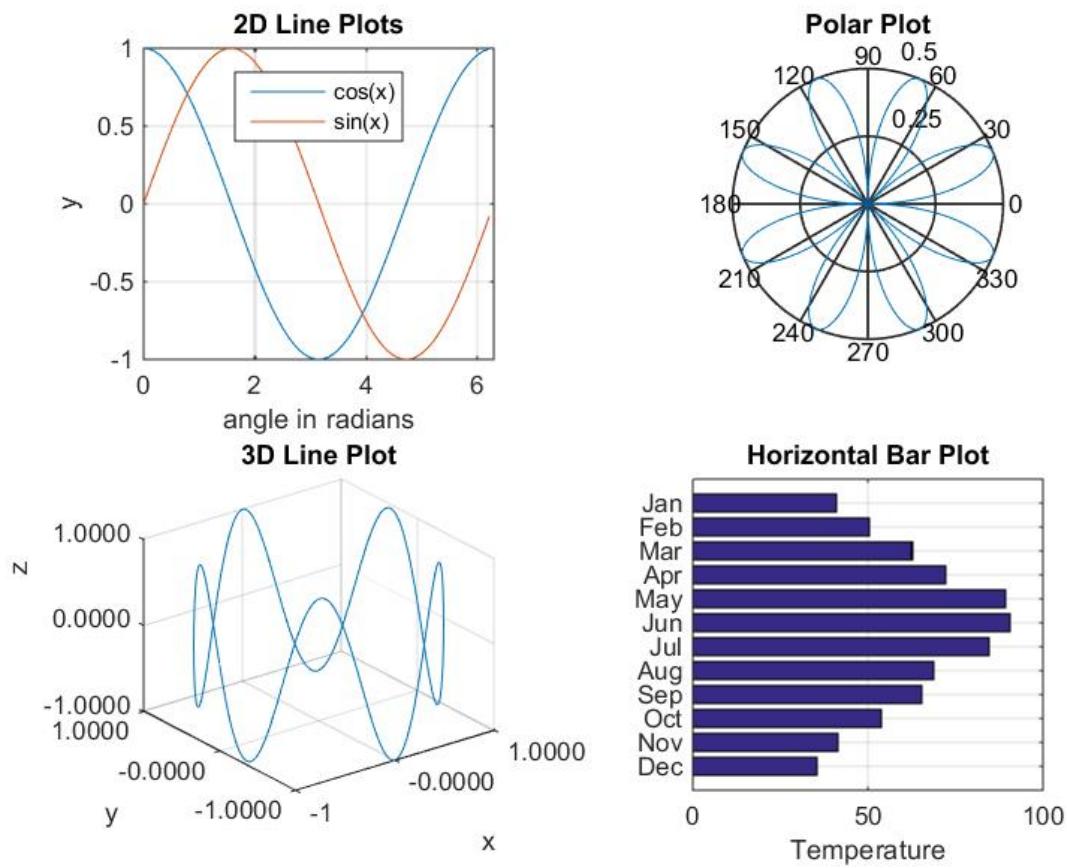
This command divides the figure window into an $m \times n$ grid where the plots are created. The plots are numbered by the integer $p = 1, 2, \dots, mn$. For a particular p , the command (A.6.14) makes the p^{th} subplot current. This implies that the next plot command or commands creates that subplot. The following example illustrates the **subplot** construction.

Example A.6.2: This example creates four different types of graphs and plots them on a single display utilizing the **subplot** construction. A few comments have been inserted to identify the kinds of graphs being created.

```
clc
clear
%Plot four types of graphs
%Divide the screen into a 2 x 2 grid
%First subplot:2D Line Plot
subplot(2,2,1)
x=[0:.1:2*pi]
y1=cos(x)
y2=sin(x)
plot(x,y1)
hold on
plot(x,y2)
grid on
axis([0 2*pi -1 1])
xlabel('angle in radians')
ylabel('y')
legend('cos(x)', 'sin(x)', 'Location', 'North')
title('2D Line Plots')
%Second subplot:Polar Plot
subplot(2,2,2)
t=[0:.01:2*pi]
y3=abs(sin(2*t).*cos(2*t))
polar(t,y3)
title('Polar Plot')
%Third subplot: 3D Line Plot
subplot(2,2,3)
t=[-pi:.01:pi]
x=cos(t)
y=sin(t)
z=sin(5*t)
```

```
plot3(x,y,z)
grid on
xlabel('x')
ylabel('y')
zlabel('z')
title('3D Line Plot')
%Fourth subplot: Horizontal Bar Plot
subplot(2,2,4)
temp=[35.67,41.33,53.96,65.23,68.98,84.74, ...
       90.45,89.18,72.02,62.61,50.62,40.94]
barh(temp)
months = {'Dec', 'Nov', 'Oct', 'Sep', 'Aug', 'Jul', ...
          'Jun', 'May', 'Apr', 'Mar', 'Feb', 'Jan'}
%In the following gca represents the current axis handle.
%In this case the axis is the one for subplot(2,2,4).
set(gca, 'YTick', 1:12)
set(gca, 'YTickLabel', months)
title('Horizontal Bar Plot')
xlabel('Temperature')
grid on
axis([0 100 0 13])
```

The figure with four subplots that results from this script is



Section A.7. Programming with MATLAB

MATLAB is a powerful programming language as well as an interactive computational environment. As mentioned above, files that contain code in the MATLAB language are called *m-files*. You create m-files using the editor mentioned in Section A.1. After an m-file is created, it is executed as you would any other MATLAB function or command. Namely, by entering the name of the file in the command window.

There are *two kinds* of m-files:

- *Scripts*, which do not accept input arguments or return output arguments. They operate on data in the workspace.
- *Functions*, which can accept input arguments and return output arguments. Internal variables are local to the function.

There is no need to compile either type of m-file. You simply type in the name of the m-file in the command window (without the extension) in order to run it.

An important point is that an m-file must be saved *in the path* of MATLAB in order to execute. The path is just a list of directories (folders) in which MATLAB will look for files. The menu sequence **Home** → **Set Path** shows the current paths searched by MATLAB when a command is executed. Another practical point is that MATLAB is picky about how m-files are named. For example, if you put a space in the file name, it will not execute.

Script m-files are the simplest kind of m-file because they have no input or output arguments. They are useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line.

- In the MATLAB command window, select **Home** → **New** → **Script** to open the editor and open a blank m-file.
 - You can put any sequence of commands into a script file and save it.
 - If you type the name of the file at the command line, each of the commands in the script file are executed in the sequence typed. The result is exactly the same result as typing each command in the command window.
 - It is often helpful to insert comments in an m-file. In order to cause MATLAB to not think a comment is a command, you place a per cent sign % as the first character on the line containing the comment. Any text on the line after a per cent sign is ignored
 - While a detail that can be learned later, when you want the % symbol to execute in some fashion, such as a part of a title, you enter it in quotes.

Example A.7.1: The problem is to create a script file (m-file) which we shall call **script.m**. This file will compute the position x as a function of time t of a mass m connected to a spring with spring constant k . The formula connecting these quantities is

$$x = u_0 \cos\left(\sqrt{\frac{k}{m}}t\right) \quad (\text{A.7.1})$$

where u_0 is the initial displacement. The solution involves opening the editor with the sequence **Home** → **New** → **script** as explained above and entering, into the editor, the text

```
clc
clear
k=80;m=20;t=12;u_0=1;
x=u_0*cos(sqrt(k/m*t))
```

Save the file with the name **script.m** by executing **Save**. Next, at the command window type:

```
>> script
```

and MATLAB outputs

```
x =
0.7991
```

The commands in a script are literally interpreted as though they were typed at the prompt.

The second kind of m-file is a function file. These files are the main way to extend the capabilities of MATLAB. Function files provide extensibility to MATLAB by allowing one to create new problem-specific functions having the same status as other built-in MATLAB functions. Such functions are like scripts, but for the purpose of enhancing computational speed, they are compiled into a low-level bytecode when called for the first time. In order to qualify to be a function file, it must have the following properties:

- The first word in the file is **function**.
 - A function can depend upon several arguments or none at all, and return any number of values.
 - The first line of the file specifies the name of the function, along with the number and names of input arguments and output values.

MATLAB contains an m-file template especially designed for function m-files. This template is produced by the command sequence **Home** → **New** → **Function**. In any case, as mentioned, the first line of a function m-file starts with the keyword **function**. It gives the function name and order of arguments. Therefore, each function m-file must start with a line such as

```
function [out1,out2] = myfun(in1,in2,in3)
```

The variables `in1`, `in2`, `in3`, etc. are *input arguments*, and the variables `out1`, `out2`, etc. are *output arguments*. The name of the function, `myfun`, should *match* the name of the m-file. In other words, the m-file in this case should be named `myfun.m`. It is customary to utilize the next several lines of the file as comments explaining the function being created. These lines are displayed in the command window when you type `help myfun`.

Example A.7.2: This example is the creation of a function m-file that will allow the calculation of the surface area and the volume of a right circular cylinder. The script that must be entered into the editor is as follows:

```
function [area,volume] = myfun(r,l)
% myfun: Calculates the area and volume of a right circular
% cylinder
% [area,volume] = myfun(radius, length)
% inputs
% r = radius of cylinder
% l = length of cylinder
area = 2*pi*r*l;
volume = pi*l*r^2;
```

This script needs to be saved in a file named `myfun.m`. From the command window you could enter:

```
>> [a,v] = myfun(3,4)
```

The MATLAB output is

```
a =
75.3982

v =
113.0973
```

The answers are the area and volume for a right circular cylinder with radius of 3 units and length of 4 units.

One of the most important features of a function m-file is its *local workspace*.

- They operate on variables within their *own workspace*
 - *Separate* from the workspace you access at the MATLAB command prompt.
 - Any arguments or other variables created while the function executes are available only to the executing function statements.

- Conversely, variables in the command-line (base) workspace are normally not visible to the function.
 - The values of the input arguments are copies of the original data, so any changes you make to them will not change anything outside the function's scope.
 - In general, the only communication between a function and its caller is through the input and output arguments.
- You can define variables as **global** variables explicitly, allowing more than one workspace context to access them.

Function files can be used for simple mathematical functions and complex ones. They sometimes appear as subprograms of larger more complex programs. MATLAB has provided an alternate ways to introduce functions that are not built in like the elementary functions mentioned in Section A.5 but at the same time are sufficiently simple where a function file is not necessary. One of these alternate ways is by use of what is called an *inline function*. These functions are entered with the MATLAB script that follows the syntax

```
name = inline('math expression typed as a string') (A.7.2)
```

For example, the function $y = \sin(x)$ can be entered as the inline function

```
y = inline('sin(x)') (A.7.3)
```

For x a numerical value, say, $x = 5$, the script **y(x)** yields the value of $\sin(5)$. In a more complicated case like, for example, the function in (A.7.1) where there are four arguments could be entered as the inline function

```
x = inline('cos(sqrt(k/m)*t','k','m','u_0','t') (A.7.4)
```

When (A.7.4) is entered into MATLAB with the script

```
>> x=inline('u_0*cos(sqrt(k/m)*t)','k','m','u_0','t')
```

the output is

```
x =  
Inline function:  
x(k,m,u_0,t) = u_0*cos(sqrt(k/m)*t)
```

While certain of our examples utilize inline functions, the MATLAB documentation indicates that inline functions will be removed in future releases. This documentation recommends the use of *Anonymous Functions*. As explained in the MATLAB documentation, an

anonymous function, like the Inline Function, is a function that is not stored in a program file.¹² It is associated with a variable whose data type is what MATLAB calls a **function_handle**. Like standard functions, anonymous functions can accept inputs and return outputs. The syntax for the example function $y = \sin(x)$ is

$$\mathbf{y} = @(x) \sin(x) \quad (\text{A.7.5})$$

The variable **y** is the function handle and the **@** operator creates the handle. The parentheses **()** after the **@** operator identifies the function arguments. The anonymous function (A.7.5) accepts the single input **x** and produces a single output **y**. The output is an array of the same size as **x**. Anonymous functions can have several inputs. For example, the function given by (A.7.1) can be defined as an anonymous function by

$$\mathbf{x} = @(k,m,u_0,t)(u_0 * \cos(\sqrt{k/m}) * t) \quad (\text{A.7.6})$$

Finally, it is useful to point out that functions can call other functions. This specialized use of function m-files shall be illustrated by examples in the text.

Other useful commands that deserve to be mentioned in this short summary are

- The **input** command is used in m-files to create an *interactive calculation* that asks you for information in a structured fashion.
- The **disp** command allows the output to be labeled.
- The **fprintf** command is a display command that gives you much greater control over how information is displayed.

Information about these commands can be found in MATLAB help. They shall be illustrated in various places in the textbook.

As you work with MATLAB, you will be storing information in the computer memory or, what is called *the MATLAB workspace*. The command

```
>> whos
```

will display the different variables that have been stored along with their sizes and types. There are often reasons to save to a file the information that has been stored in the workspace. For example, you might have stored information that you do not want to lose when you end your MATLAB session. The **save** command allows you to save your information. The syntax of the **save** command is as follows:

- **save filename** saves ALL workspace variables to the file **filename.mat**

¹² See http://www.mathworks.com/help/matlab/matlab_prog/anonymous-functions.html.

- **save filename x1** saves the variable **x1** to the file **filename.mat**
- **save filename x1 x2 x3** saves the variables **x1 x2 and x3** to the file **filename.mat**

If you use the same filename, the contents of the previous file will be overwritten without warning.

To **load** a previously saved file simply enter:

```
»load filename
```

Section A.8. Control Structures

The simple m-files we have utilized thus far execute MATLAB commands sequentially. In order to have more programming flexibility, MATLAB like all programming languages contains *control statements* that allow commands to be executed in a non-sequential fashion. These statements allow for *loops* (or repetition) and for *decision* (or selection) making. As indicated, they allow us to control the order in which statements are executed.

In order to discuss control structures, we need a brief discussion of the *logical expressions* that are a part of MATLAB. The simplest kind of logical expression is `<`, which reads “less than.” If, for example, one enters into MATLAB a mathematically true statement like

```
>> 5<6
```

MATLAB yields the output

```
ans =
1
```

If one enters a mathematically incorrect statement like

```
>> 6<5
```

MATLAB yields the output

```
ans =
0
```

The above is typical. If the logical statement is *correct*, MATLAB yields an answer of 1. If it is *incorrect*, MATLAB returns 0.

A logical expression involves the use of *relational* operators, like `<` in the example above, and *logical* operators for the comparison of variables and matrices of the *same size*. MATLAB has six relational operators. These *compare* corresponding elements of arrays with *equal dimensions*. A table of these six relational operators is as follows:

Relational Operators	
Operator	Description
<code><</code>	less than

>	greater than
<=	less than or equal
>=	greater than or equal
==	Equal
~=	not equal

Relational operators always operate element-by-element. In this example, the resulting matrix shows where an element of **A** is equal to the corresponding element of **B**.

```
>> A = [2 7 6;9 0 5;3 0.5 6];
>> B = [8 7 0;3 2 5;4 -1 7];

>>A == B
ans =
    0     1     0
    0     0     1
    0     0     0
```

For most situations, both operands must be of the same size. An exception is when one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand.

As our examples have illustrated, for the six relational operators

- Locations where the specified relation is *true* receive logical 1.
- Locations where the relation is *false* receive logical 0.

Three of MATLAB's *logical operators* are listed in the following table:

Logical Operators		
Operator	Description	Precedence
~	Not	1
&	and	2
	or	3

The logical operators perform element-wise logical operations on their inputs to produce a like-sized output array. The examples shown in the following table use vector inputs **A** and **B**, where

```
>> A = [0 1 1 0 1];
```

```
>> B = [1 1 0 0 1];
```

Operator	Description	Example
&	Returns 1 for every element location that is true (nonzero) in both arrays, and 0 for all other elements.	A&B = 01001
	Returns 1 for every element location that is true (nonzero) in either one or the other, or both arrays, and 0 for all other elements.	A B = 11101
~	Complements each element of the input array, A , i.e. gives the opposite of A	~A = 10010

We can also combine two logical expressions using so-called logical operators. Note that a logical expression may contain several logical operators in sequence. Logical expressions can be combined by using the logical operands **&**, **|** and **~**.

We can also combine two logical expressions using so-called logical operators. Note that a logical expression may contain several logical operators in sequence. Logical expressions can be combined by using the logical operands **&**, **|** and **~**.

Example A.8.1: One application of the above relational operators is to enable MATLAB to deal, for example, with functions of the type

$$f(x) = \begin{cases} 0 & x < 10 \\ (x-10)^2 & x \geq 10 \end{cases} \quad (\text{A.8.1})$$

can be entered into MATLAB with the syntax

```
>> f=(x>=10).*(x-10).^2
```

Another group of control statements are *decision or selection constructs*. This group of control statements enables you to select at run-time which block of code is executed. The following diagram shows the syntax for three commonly used selection constructs in MATLAB:

if-end construct	if-else-end construct	if-elseif-end construct
<pre>if <condition1>, <program1> end;</pre>	<pre>if <condition1>, <program1> else <program2> end;</pre>	<pre>if <condition1>, <program1> elseif <condition2>, <program2> elseif <condition3>, <program3> else <conditionN> <programN> end;</pre>

In each of these constructs, the block of **statements** `<program1>` will be executed when the logical expression `<condition1>` evaluates to true. Otherwise, the program moves to the next program construction.

- For the **if-end** construct, this means the end of the selection construct.
- For the **if-else-end** construct, the block of **statements** `<program2>` will be executed when `<condition1>` evaluates to false.
- For the **if-elseif-end** construct, MATLAB will systematically evaluate the sequence of logical expressions `<condition1>, condition2>, . . . , <conditionN>` until one evaluates to true.
 - After the corresponding block of `<program>` statements has been executed, the program will jump to the end of the **if-elseif-end** construct.

Example A.8.2: (**if-end** construct): The idea is to build in a notification when a prescribed calculation breaks down in some fashion. The particular example is division by zero, which is not defined. The first step is to construct a function m-file **divide.m**

```
function f = divide(x )
%divide gives the value 1/x when x is not zero
if x==0
    error('You have tried to divide by zero.')
end
f=1/x
```

If one selects an **x** which is not zero, for example, **x=100**, the MATLAB output is

```
>> divide(100)

ans =
0.0100
```

If you select **x=0**, the MATLAB output is

```
>> divide(0)
??? Error using ==> divide at 4
You have tried to divide by zero.
```

Example A.8.3: (if-elseif-end construct) Consider the function defined piecewise by the formula

$$f(x) = \begin{cases} 11t^2 - 5t & 0 \leq t \leq 10 \\ 1100 - 5t & 10 \leq t \leq 20 \\ 50t + 2(t-20)^2 & 20 \leq t \leq 30 \\ 1520e^{-0.2(t-30)} & t > 30 \\ 0 & \text{otherwise} \end{cases} \quad (\text{A.8.2})$$

The MATLAB script

```
function v=vpiece(t)
if t<0
    v=0;
elseif 0<=t&t<=10
    v=11*t^2-5*t;
elseif 10<=t&t<=20
    v=1100-5*t;
elseif 20<=t&t<=30
    v=50*t+2*(t-20)^2;
else
    v=1520*exp(-.2*(t-30))
end
```

uses the **if-elseif-end** construct to define the function m-file **vpiece.m** that enters the function $f(t)$ into MATLAB.

MATLAB provides a number of looping constructs for the efficient computation of similar calculations. The syntax for the **while** and **for** looping constructs is

while-loop construct	for-loop construct
<pre>while <condition1>, <program1> end</pre>	<pre>for i = <array of values> <program1> end</pre>

- In the **while** looping construct, the block of statements **<program1>** will be executed while the logical expression **<condition1>** evaluates to true.
- In the **for** looping construct, the block of statements **<program1>** will be executed for each of the vectors **i** defined by the column elements in **<array>**.

Example A.8.4: This example uses a **for-loop** construction to calculate the binomial coefficient $\frac{n!}{k!(n-k)!}$ of a pair of positive integers where $0 \leq k \leq n$.¹³ We use the following script to create a function file binomial.m:

```
function w=binomial(n,k)
x=1;y=1;z=1
for i=1:n
    x=x*i
end
xout=x
for i=1:k
    y=y*i
end
yout=y
for i=1:(n-k)
    z=z*i
end
zout=z
w=xout/yout/zout
```

At several points within the textbook, the **for-loop** construction just illustrated will be utilized to create matrices of various sizes. It is good programming practice to *preallocate* memory for the matrix by first creating the matrix of zeros with a command such as **zeros(m,n)** followed by the **for-loop** that replaces the zeros with the desired elements. This approach avoids the necessity of MATLAB expanding the size of the array repeatedly as it proceeds through the programming loops. For large matrices, resizing the array can affect the performance of the

¹³ MATLAB has a built in function, called **factorial** that will directly calculate the three factorials in the definition of the binomial coefficient.

program. This because MATLAB must spend time allocating more memory each time the array size is increased. In addition, the newly allocated memory is likely to be noncontiguous, thus slowing down any operations that MATLAB needs to perform on the array.

The preferred method for sizing an array that is expected to grow with subsequent MATLAB steps is to estimate the maximum possible size for the array, and preallocate this amount of memory for it at the time the array is created. In this way, the program performs one memory allocation that reserves one contiguous block.¹⁴

Three other commands that relate to the topic of this section are as follows:

- The **continue** statement passes control to the next iteration of the **for** or **while** loop in which it appears, skipping any remaining statements in the body of the loop.
- The **pause** command will cause MATLAB to wait for a key to be pressed before continuing.
- The **break** statement terminates the execution of a **for** or **while** loop.
 - When a **break** statement is encountered, execution continues with the next statement outside of the loop.

¹⁴ MATLAB's online help, for example at http://www.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html, provides more information about preallocation.

Appendix B

ANIMATIONS

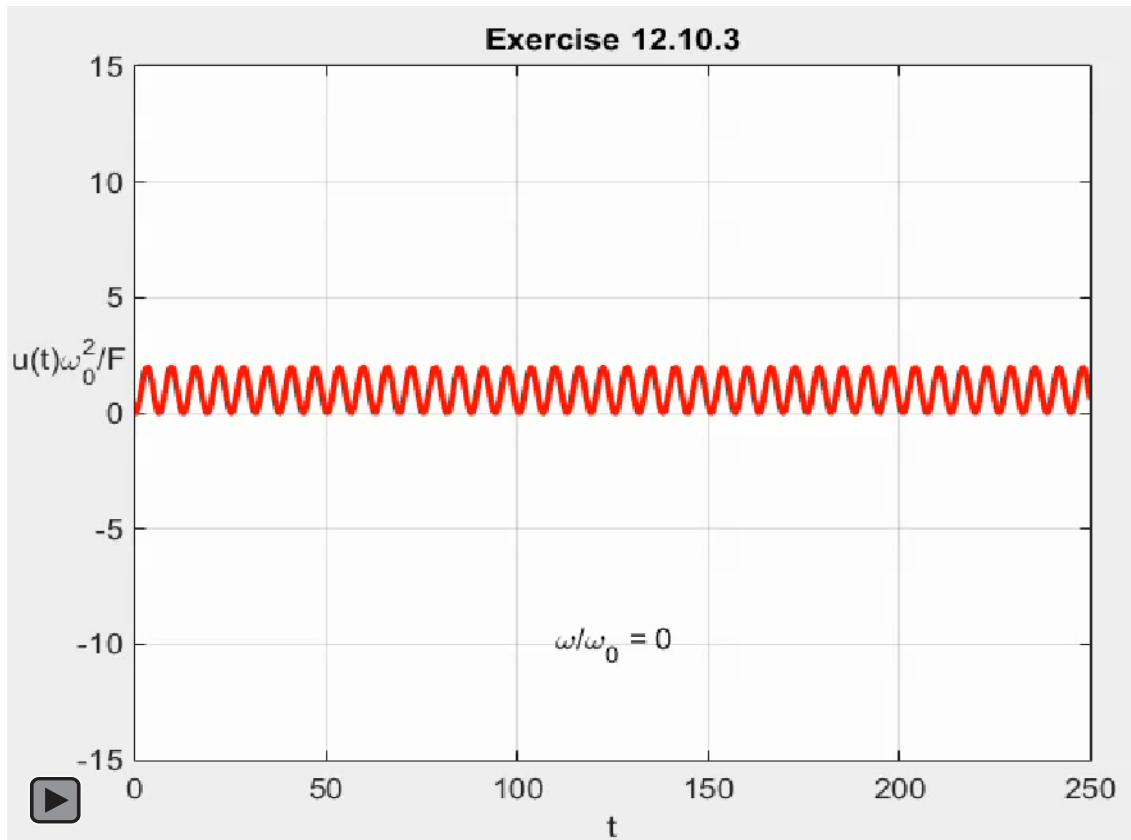
In Chapter 12, we utilized MATLAB to create animations for five of the ordinary differential equations solutions. In this Appendix, we shall collect in one place these animations.

1. Animation of Example 12.10.3

This example concerns a one degree of freedom linear vibrating system with a forcing function. The damping was zero and the two initial conditions were zero. The forcing function was given by equation (12.10.19), repeated

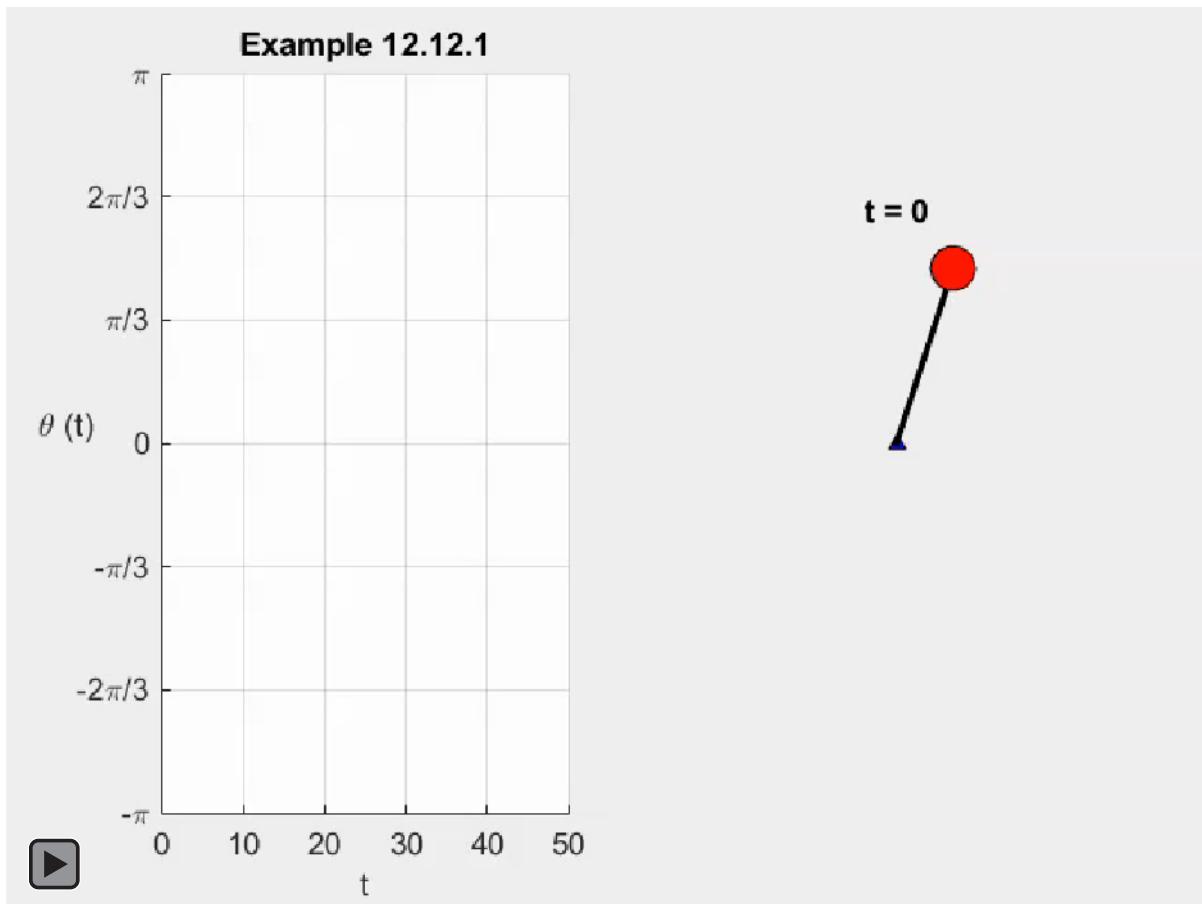
$$\frac{f(t)}{m} = F \cos \omega t \quad (\text{B.1})$$

The animation associated with this example involves plotting the solution for a range of frequencies ω . The particular animation created is



2. Animation of Example 12.12.1

This example concerns the motion of a damped pendulum without a forcing function. The initial conditions are starting the motion from rest with the pendulum held near vertical. The resulting animation is

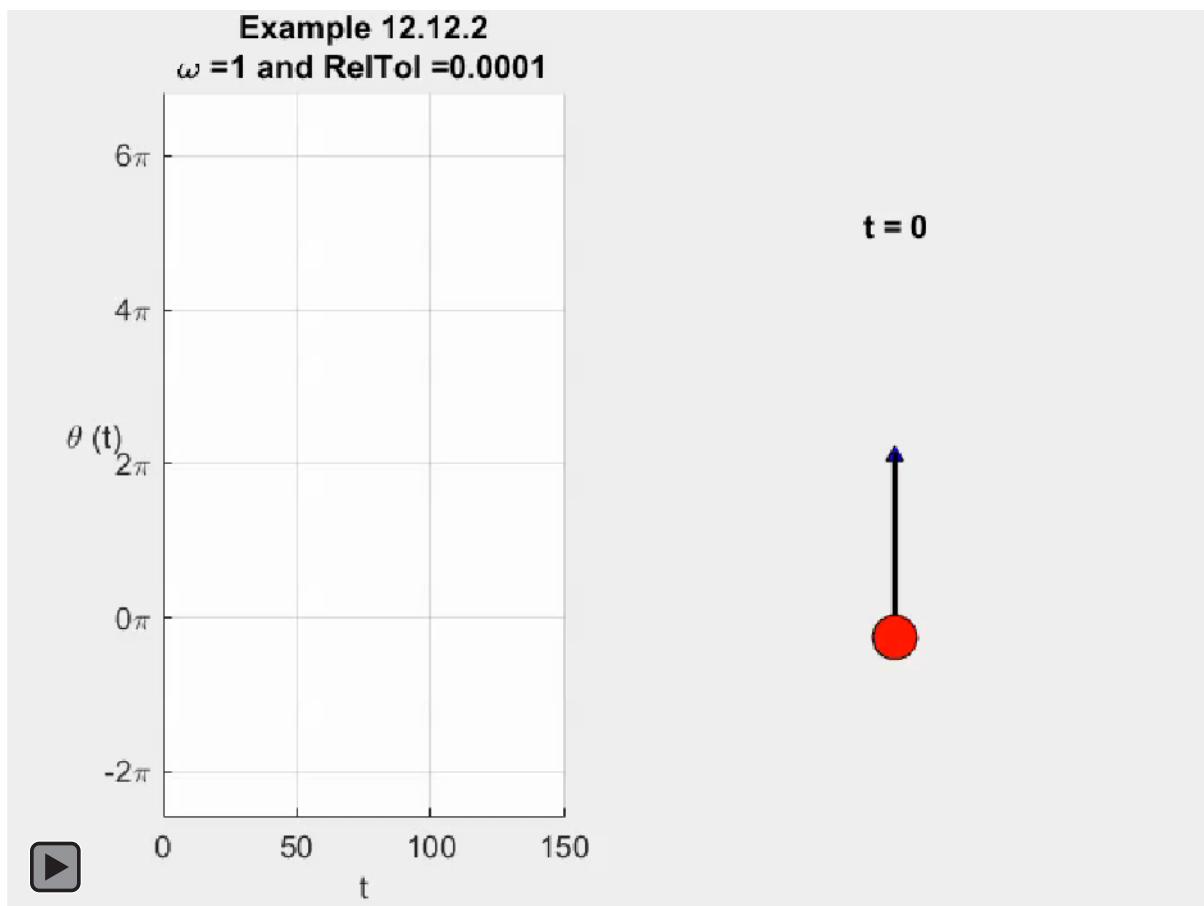


3. Animation of Example 12.12.2

This example adds a forcing function given by equation (12.12.9), repeated,

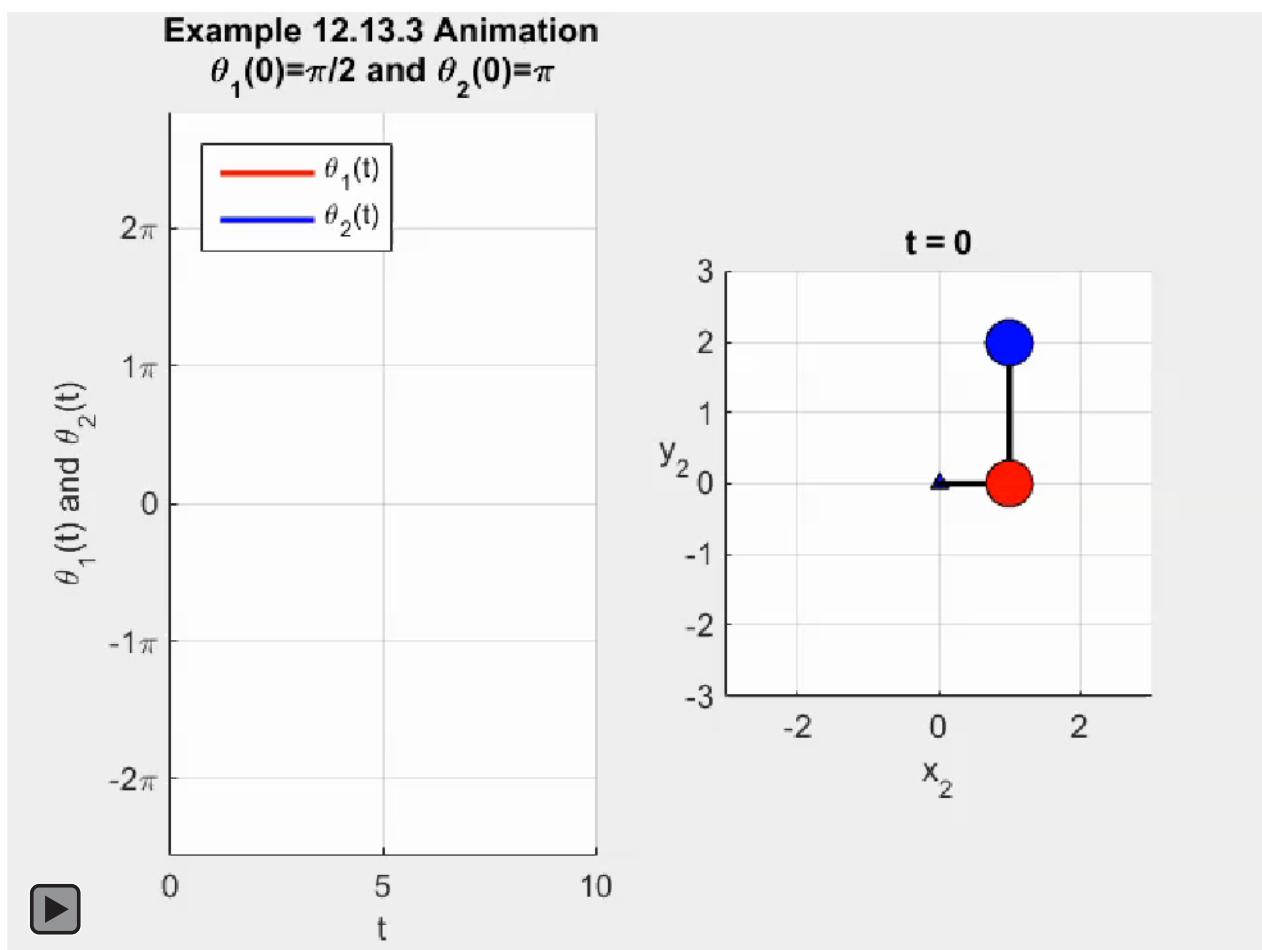
$$\frac{1}{ml^2} f(t) = \cos \omega t \quad (\text{B.2})$$

to the study of the motion of a damped pendulum. The initial conditions are starting the motion from the equilibrium position $\theta = 0$ with an initial velocity of $\frac{d\theta(0)}{dt} = 2$. The resulting animation is



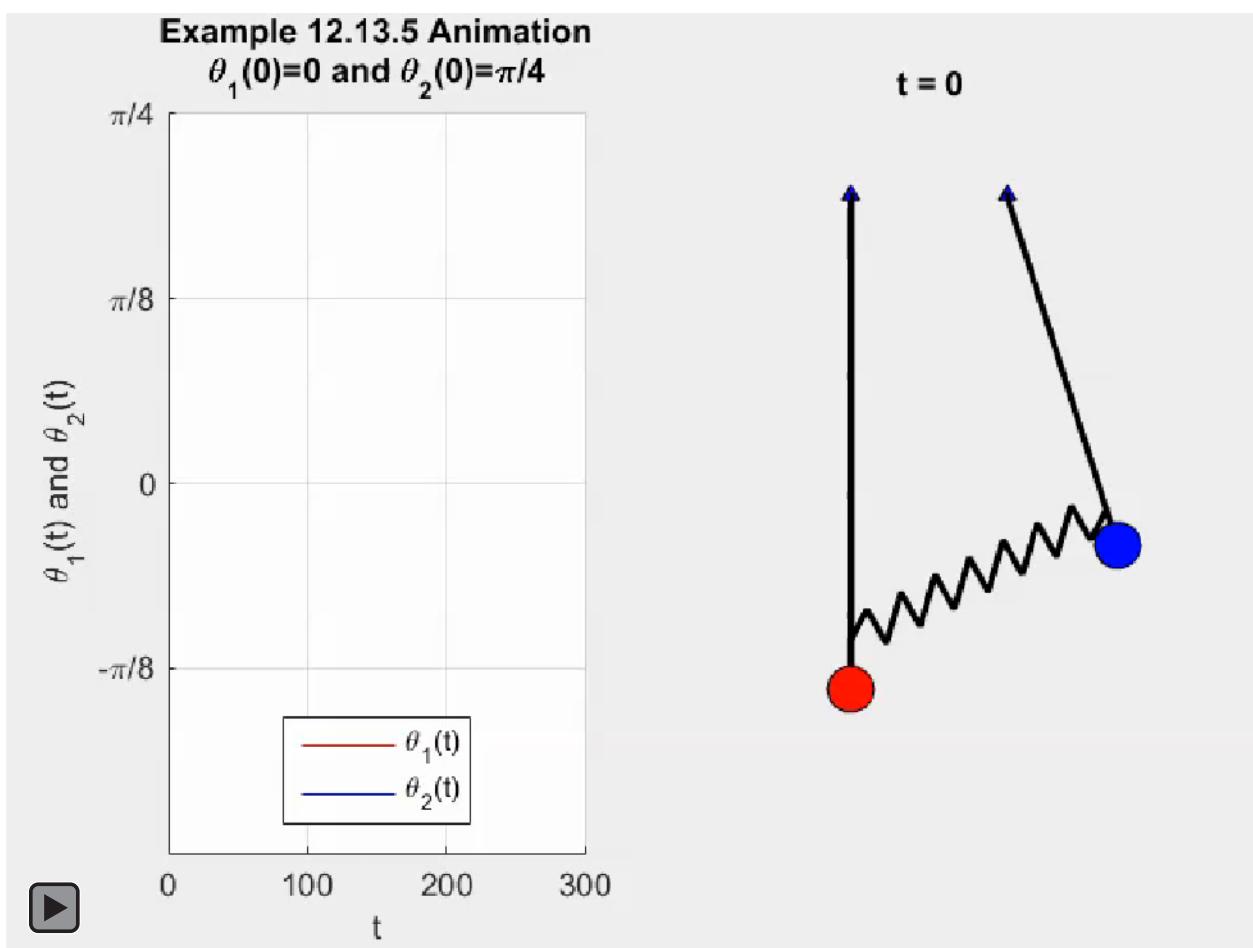
4. Animation of Example 12.13.3

This example concerns the motion of a double pendulum. Both pendulums start from rest, the first from a position of $\theta_1 = \frac{\pi}{2}$ and the second one from a position of $\theta_2 = \pi$. The animation show a plot of y position of the second pendulum versus it's x position as the time evolves. The resulting animation is



5. Animation of Example 12.13.5

This example concerns the motion of a coupled pendulum. The first pendulum starts at $\theta_1 = 0$ with zero angular velocity. The second pendulum starts at $\theta_2 = \frac{\pi}{4}$ with zero angular velocity. The animation displays the exchange of motion transferred by the connecting spring between the two pendulums. The resulting animation is



INDEX

- Absolute Error Tolerance, 984
Adaptive Lobatto Method, 908
Adaptive Solvers, 977, 983
Addition, 588, 632, 652, 668, 680, 693, 702, 729, 771, 804, 829, 865-866, 895, 905, 908, 926, 949, 1012, 1025, 1029, 1034, 1057, 1059, 1107, 1119, 1121, 1152
Adjugate, 587-589, 797
Airy Function, 990, 992
Airy Ordinary Differential Equation, 990
Algebraic Multiplicity, 633
Angle, 725, 728, 1046, 1048, 1123, 1137
Angular Velocity, 1045, 1047, 1050, 1159
Animation, 1021-1022, 1048, 1064, 1082, 1094, 1155-1159
Anonymous Function, 679-680, 692, 699, 907, 937, 950, 953, 969, 983, 1024-1025, 1037, 1046, 1145
Application Programming Interface, 1098
Arithmetic Mean, 744, 765, 859
Array Operations, 1120-1121
Augmented Matrix, 595, 597, 599-601, 801
Back Substitution, 596-597, 599, 627
Balthasar Van Der Pol, 1007
Base 2, 651-652, 1122
Basis, 601, 629-630, 633, 695, 705, 777-778, 791-797, 801, 803-804, 808, 810-813, 815, 819-822, 836-838, 853, 860
Basis Functions, 777
Bernoulli Equation, 993
Bessel Function Of The Second Kind, 912
Bessel Functions, 896-897, 905, 912, 921, 941, 990
Binary, 651, 653, 656-659, 661, 663-664
Bins, 766
Bisection Method, 672, 679, 683-685, 690, 692, 694, 696-697, 700, 703, 731
Bracketing Method, 672, 683, 694
Brook Taylor, 637
Carl David Tolm  Runge, 849, 959
Cartesian Product, 727
Change Of Basis, 794-795, 813, 853, 860
Chaos Theory, 1079
Characteristic Polynomial, 630, 634, 825
Characteristic Subspace, 633
Chebyshev, 604-606
Chopping, 652, 656-657, 668
Cofactors, 587
Colin Maclaurin, 639
Colon Notation, 1115
Column Matrix, 591, 606, 809, 816, 818, 950, 952, 969, 982, 1023, 1106
Column Sum Norm, 614-618
Command Prompt, 983, 1103, 1105, 1143
Command Window, 583, 585, 588, 591, 679, 681, 1099, 1101, 1103-1104, 1106, 1125, 1141-1143
Complex Numbers, 584, 602-603, 672, 731, 1105
Composite Boole's Rule, 894
Composite Simpson 1/3 Rule, 893, 901, 903
Composite Simpson 3/8 Rule, 894, 904
Composite Trapezoidal Rule, 892, 897-898, 901
Concatenation, 1113
Condition Number, 612-614, 616, 856
Consistency Theorem, 735
Control Structures, 1147
Correlation Coefficient, 747, 766, 768
Coupled Pendulum, 1090, 1093-1094, 1159
Current Folder, 1099
Curve Fitting, 771, 773
Damped Forced Vibration, 1026
Damped Pendulum, 1045-1046, 1053, 1063, 1066, 1156-1157
Degree Of Freedom, 923, 1009, 1019, 1027, 1033, 1043, 1046, 1155
Determinant, 586-587, 719, 794, 797
Development Environment, 1097-1098
Diagonal Elements, 621, 826
Diagonal Matrix, 634, 1087, 1108
Difference, 583-584, 648, 805-807, 818-819, 845, 900, 902, 933, 937, 997, 1079
Differential-Algebraic, 1085-1086
Dimension, 601, 633, 718, 729, 791, 797, 813, 828, 839, 907, 919, 1010, 1107, 1109
Direction Field, 927-931, 939-941, 943, 947, 959, 962, 965
Divided Difference, 805-807, 819
Divided Difference Table, 806-807, 819

- Domain, 661
Double Pendulum, 1067, 1073, 1076, 1082, 1084, 1087, 1158
Editor, 1100-1101, 1103, 1141-1143
Eigenvalue, 615, 629, 631, 633-635, 834, 915, 997, 1026
Eigenvector, 629-630, 633
Elementary Symmetric Polynomial, 820
Elimination Method, 621
Émile Picard, 925
Equations Of Motion, 923, 1027, 1030, 1044, 1067, 1090
Equilibrium Solution, 1053
Error Function, 911, 975
Euler'S Method, 933, 935, 937, 939, 941, 943, 945, 950, 957, 961
Exponential Biasing, 661
Exponential Linear Transformation, 634, 836
Exponential Matrix, 634
False Position Method, 672, 694-697, 700, 702-704, 731
Field, 652, 927-931, 939-941, 943, 947, 959, 962, 965
Finite Dimensional, 614, 616, 629, 729, 791
Finite Element Discretization, 1087
First Order System, 950, 985, 991
Floating Point Arithmetic, 652-653, 656
Forcing Frequency, 1016, 1018, 1053-1055, 1057-1059
Forcing Function, 974, 977, 1009, 1015-1016, 1019, 1025, 1027, 1029, 1036, 1045, 1053-1054, 1155-1157
Forward Difference Approximation, 933
Fourth Order Runge-Kutta Method, 964, 967, 978-979
Free Vibrations, 1009, 1033
Frobenius Norm, 614
Function File Within A Function File, 953
Function m-file, 602, 621, 679, 692-693, 698, 702, 711, 813-814, 819, 828-829, 832-833, 838, 840, 855, 898, 901-902, 904, 906, 909, 928, 952, 969-971, 975, 991, 1002, 1005, 1012, 1014, 1018, 1024, 1033-1034, 1036, 1046, 1071-1072, 1088, 1092, 1103, 1142-1143, 1150-1151
Functions, 595, 614, 634, 638, 668, 672-673, 679-680, 683, 702, 715, 719, 721, 729, 749, 777, 871, 873-876, 879-884, 886-887, 891-893, 896-897, 905, 908, 910-912, 921, 924, 941, 987, 990, 1022-1023, 1027, 1031, 1044, 1098-1099, 1103, 1108, 1121, 1125, 1132, 1141-1142, 1144-1145, 1149
Gamma Function, 905, 990
Gauss-Jordan Elimination, 595, 599, 627
Gaussian Elimination, 595-596, 621, 625
Gegenbaur, 604
Geometric Multiplicity, 633
Georg Duffing, 1036
George Biddell Airy, 990
George Boole, 894
Global Truncation Error, 935-936, 967
Global Variable, 1014
Gram-Schmidt, 595, 601-603
Group, 736, 902, 1005, 1149
Handle Graphics, 957
Hermite, 604, 607-609
Hermitian, 615
Heun Method, 961-962, 966, 969-971, 973, 976-977, 979, 986-987, 989
Hilbert Matrix, 616-618
Histogram, 766
Horner Method, 800-801, 806, 848, 856
Hypergeometric Equation, 1031
Hypergeometric Function, 1031
Identity, 584, 603, 621, 659, 795, 802, 811, 823, 896, 1109
Identity matrix, 584, 621, 811, 823, 1109
IEEE 64 bit floating-point, 637, 651, 657, 663
Ill Conditioned, 587, 611, 613-615, 617-619, 813, 818, 831, 845, 853-856, 1072, 1084
Infinite Sequence, 663
Initial Condition, 916-918, 921, 925, 927, 930, 936-937, 943, 946, 950, 952, 971, 973-974, 991, 1004, 1007, 1034, 1037, 1039, 1047, 1054, 1076-1077, 1079
Initial Value Problem, 916-917, 919, 923, 925, 930, 933, 946, 959, 971, 977, 982, 985,

- 994-995, 997, 1001, 1004, 1006, 1009, 1011-1012, 1015, 1024, 1031, 1033, 1045-1046, 1054, 1086, 1089
 Inline Function, 679-680, 907, 950, 969, 983, 1144-1145
 Inner Product, 605, 607-609, 737-738
 Interpolation, 681, 695, 702, 736, 787-850, 852-860, 862-914
 Interpolation Method, 695, 858-859
 Interpolation Nodes, 791
 Interpolation Points, 791
 Inverse, 585-586, 588, 592, 611, 614, 719, 740, 796, 798, 801, 811, 837, 855, 1069, 1072, 1121-1122
 Isaac Newton, 705, 894
 Iteration Condition, 717, 719, 934
 Iterative Relative Error, 692
 Jacobi, 604
 Jacopo Francesco Riccati, 941
 Joseph Raphson, 705
 Joseph-Louis Lagrange, 835
 Kernel, 740, 742
 Lagrange Basis, 792, 810-813, 838
 Lagrange Interpolation, 811, 835, 837, 839, 841, 853, 859, 865, 867, 869, 871, 873, 875, 877, 879, 881, 883, 885, 887
 Lagrange Polynomial, 789, 810-811, 835, 838, 853, 870
 Least Squares, 735-737, 739
 Least Squares Problem, 735
 Left Pseudo Inverse, 740
 Leguerre, 604
 Length, 612, 632-633, 679, 738, 767, 813-814, 828, 831, 839, 875, 912, 942, 950, 970, 978, 988-989, 1010, 1021, 1024, 1041-1042, 1049, 1064-1065, 1067, 1082-1083, 1090, 1094-1095, 1109, 1126, 1143
 Leonhard Paul Euler, 933
 Linear Dashpot, 1009, 1035
 Linear Interpolation, 787-789, 891-892
 Linear Regression, 741-745, 747, 749-750, 755-757, 760, 764, 768, 773-774, 777, 843
 Linear Spring, 1009, 1015, 1041, 1090
 Linear Transformation, 584, 629, 634, 735, 737-738, 740, 836
 Lipschitz Condition, 926
 Local Workspace, 1143
 Logical Expression, 1042, 1147, 1149-1150, 1152
 Logical Operators, 1147-1149
 Lower Triangular, 621, 801
 Lu Decomposition, 621-627
 m-file, 596, 602, 621, 648-649, 669, 679, 690-693, 698, 702, 711, 813-814, 819, 828-829, 832-833, 838, 840, 855, 898, 901-902, 904, 906, 909, 928, 950, 952, 969-971, 975, 983, 991, 1002, 1005, 1012, 1014, 1018, 1024, 1033-1034, 1036, 1046, 1071-1072, 1088, 1092, 1101, 1103-1104, 1141-1143, 1150-1151
 Machine Epsilon, 667-668
 Magnitude, 605-606, 608-609, 654, 662, 669-670, 746, 945, 997
 Mantissa, 656-657, 665
 Martin Wilhelm Kutta, 959
 Mass Matrix, 923, 1085-1088
 Mathematical Function Library, 1098
 Matlab, 582-589, 591-599, 601-607, 609, 611, 614-616, 618, 622-627, 629, 631-635, 637, 640-641, 648-652, 657-658, 665-669, 673, 676, 679-682, 690, 692-693, 697, 699-700, 708, 711-712, 720, 723, 726, 729-735, 740, 742-747, 750, 755-757, 759-760, 763, 765-766, 776, 783, 791, 797, 799-801, 804, 808, 812-819, 821, 823, 825-827, 829-831, 833, 835, 837-841, 843-845, 847, 851, 855-856, 859-860, 862, 868-870, 874-875, 889, 896-898, 900, 902-905, 907-908, 910-913, 915, 928-929, 937, 941, 947, 949, 951, 953-955, 957-958, 969, 971, 973-975, 977-979, 981-987, 989-991, 993-995, 1000-1005, 1007, 1009, 1011, 1018, 1021, 1025, 1031, 1033, 1039-1040, 1042-1043, 1046, 1054, 1061, 1063, 1067, 1071-1073, 1077, 1082, 1085-1087, 1094, 1097-1155
 Matlab Ode Solvers, 981-983, 985, 987, 989, 991, 993, 995, 1067
 Matrix Multiplication, 788, 1070, 1072, 1119-1120
 Mean Value Theorem, 637-638

- Midpoint Method, 962, 966-967
Minimum Polynomial, 634
Minor, 982
Modified Gram-Schmidt, 603
Monomial Basis, 791, 793, 796, 812-813, 815, 837-838, 853
Multilinear Regression, 778
Multiplication, 583, 592, 668-669, 729, 788, 827, 1070, 1072, 1119-1121
Multistep Solver, 981-982
Natural Frequency, 1010, 1018, 1020, 1053
Newton Coates Coefficients, 895
Newton Interpolation, 804, 806, 819, 821, 823, 825, 827-831, 833, 839, 841, 854, 856-858
Newton Polynomial, 789, 803, 805, 819, 828, 830, 832, 838
Newton-Cotes Formulas, 894
Newton-Raphson Method, 673, 705-709, 711, 713, 715, 719, 722-723
Nonlinear Regression, 749, 751
Nonlinear Spring, 1036
Norm, 602, 604, 612-618, 738, 767
Normal Equation, 738-741, 762-763, 766, 773, 780
Normal Form, 916-917, 919-924, 970, 991, 1011-1012, 1015, 1023, 1033, 1036, 1046, 1068-1070, 1072-1073, 1084, 1091
Numerical Integration, 889, 891, 893, 895-897, 899, 901, 903, 905, 907, 909, 911, 913, 965
One Step Method, 934
One To One, 661, 737, 740, 742
Onto, 737
Open Method, 672-673, 705
Orthogonal, 601-604, 739
Orthogonal Matrix, 604
Orthogonal-Triangular Decomposition, 604
Overdetermined, 591, 735, 737, 739
Overdetermined Systems, 591, 735, 737, 739
Overflow, 653, 656, 667-669, 1110
Pafnuty Chebyshev, 604
Partial Pivoting, 596-597, 599, 621, 624-625
Partition, 865, 869, 882, 886, 889-890, 892-895, 933, 950-951, 970, 1132
Permutation, 621, 626
Permutation Matrix, 621
Phase Plot, 1050, 1053
Picard'S Theorem, 925-926
Piecewise Cubic Interpolation, 878, 893
Piecewise Lagrange Interpolation, 865, 867, 869, 871, 873, 875, 877, 879, 881, 883, 885, 887
Piecewise Quadratic Interpolation, 881, 892
Pivot Coefficient, 595-596
Placeholder, 702, 985
Polynomial, 604-609, 630, 634, 672, 729-731, 733, 736-737, 741, 750, 756, 759-770, 773, 778, 787-789, 791-793, 795-801, 803-816, 819-820, 822, 825-826, 828-830, 832-841, 843-845, 847-854, 860, 863-866, 869-871, 881, 884, 889-891, 894, 902
Polynomial Interpolation, 791, 793, 795, 797, 799, 801, 803, 805, 807, 809, 811, 813, 819, 828, 838, 853, 870-871, 889-890
Polynomial Regression, 750, 759, 761, 763, 765, 767, 769, 778, 843
Polynomials, 604-609, 729-731, 733, 759-760, 765, 773, 791, 796, 801, 804-805, 811-813, 819-820, 826, 835-836, 839, 843, 847, 849, 851-854, 860, 865-866
Positive Definite, 1022-1023
Positive Semidefinite, 1023
Preallocate, 587-588, 606, 727, 799-800, 819, 839, 950, 952-953, 970-971, 991, 1012, 1014, 1018, 1024, 1034, 1037, 1046, 1055, 1071-1072, 1089, 1092, 1152-1153
Product, 583, 602, 605, 607-609, 652, 658, 717, 731, 737-738, 822, 827, 839, 1120
Profiler, 1101
Projections, 603
QR Decomposition, 604, 845
Quadratic Convergence, 707
Ralston'S Method, 963
Range, 655-656, 675-676, 847-848, 859, 941, 974, 1039, 1063, 1103, 1135, 1155
Rank, 630, 740
Reciprocal Condition Number, 616, 856

- Reduced Row Echelon Form, 595, 599
 Regression, 735-736, 738, 740-752, 754-757, 759-787, 843
 Rehuel Lobatto, 908
 Relational Operators, 1147-1149
 Relative Error, 612-613, 649, 690, 692-693, 697, 711, 984
 Relative Error Tolerance, 984
 Remainder, 639, 642, 731, 895, 902, 905, 1123
 Residual, 738-739, 746, 773, 779
 Resonance, 1053
 Roger Cotes, 894
 Round Off Error, 603, 645, 648, 652, 668
 Row Matrix, 730-731, 813-815, 828-829, 838-840, 1105, 1116
 Row Operations, 801
 Row Sum Norm, 615-618
 Rudolf Lipschitz, 926
 Runge Function, 849, 852
 Runge-Kutta Methods, 959-961, 963, 965, 967, 969, 971, 973, 975, 977, 979, 984
 Scalar, 583, 629, 672, 679, 729, 983, 1084, 1148
 Scalar Multiplication, 729
 Scripts, 1141-1142
 Second Order Runge-Kutta Method, 962, 966
 Shape Functions, 873-876, 879-884, 886-887, 891-893
 Sign Bit, 658
 Signed Magnitude Method, 654
 Singular Value Decomposition, 634
 Solution, 591-600, 611-613, 631, 637, 668, 671, 684, 702, 728, 735-737, 739-744, 746-747, 750, 752, 760, 763-764, 770, 773, 780, 783-784, 788, 793-794, 797, 802-803, 811, 813, 836, 868, 875, 880, 915-917, 922-923, 925, 927, 930-931, 933-934, 936-938, 940-941, 943-947, 949-953, 959-960, 962, 965, 970-974, 977, 981-987, 991-994, 997-1006, 1009, 1011-1020, 1022, 1024-1026, 1029, 1031, 1033, 1035, 1037-1040, 1042, 1044, 1046-1047, 1053-1054, 1057-1059, 1061, 1063-1064, 1066, 1072, 1076-1077, 1079, 1085, 1087-1089, 1091-1093, 1142, 1155
 Sparse Matrix, 983-984, 1084-1087
 Spectral Norm, 615-618
 Spread, 745-747, 765-767
 Square Matrix, 585-587, 592, 612, 731, 822, 825, 922, 1109
 Standard Deviation, 745-747, 765-767, 859-860
 Standard Error, 746, 766
 Steady State Solution, 1016-1017, 1019, 1026, 1038-1039
 Stem Plot, 988
 Step Size, 889, 900-901, 904, 906, 933, 936, 940, 944-945, 950-951, 969, 972-973, 977, 979, 983-984, 986-989, 997, 1000
 Stiff, 945, 981-982, 997-1005, 1007, 1026
 Stopping Criteria, 692-693, 724
 Subspace, 633
 Sum, 583, 614-618, 648-649, 652, 891, 898, 902, 905, 907, 910, 935, 1016, 1068, 1122
 Symbol, 586, 613, 632, 719, 729, 905, 952, 975, 1010, 1111, 1141
 Symmetric, 616, 740, 820, 910, 1022-1023, 1071
 Symmetric Matrix, 616, 910, 1071
 Systems, 591, 595, 611-612, 627, 652-653, 657, 671-672, 715, 717, 719, 721, 723, 725, 727, 735, 737, 739, 915-916, 923, 969, 974, 981, 1009, 1011, 1013, 1015, 1017, 1019, 1021, 1023, 1025-1027, 1029, 1031, 1033, 1035, 1037, 1039, 1041, 1043, 1067, 1079, 1097
 Taylor'S Series, 639, 642, 645
 Taylor'S Theorem, 637-638, 642, 706, 716-717, 895, 935
 Tensor Product, 602
 Third Order Runge-Kutta Method, 963, 966-967
 Thomas Simpson, 893
 Trace, 583
 Transient Solution, 1016, 1026
 Transition Matrix, 630, 795, 808, 819-822, 826, 834, 837, 839, 860

Transpose, 584, 587, 794, 796, 826, 950, 970, 1119, 1121
Tridiagonal Matrix, 594
Truncation Error, 641, 643, 645-647, 668, 895, 900-901, 904, 906, 935-936, 960, 965-967
Under Damped, 1010, 1012, 1015
Underflow, 653, 656, 668-669
Unitary, 634
Upper Triangular, 604, 621, 627
Van Der Pol Equation, 1007
Vandermonde, 588, 618-619, 794, 796, 813-814, 831, 837, 853
Variables, 668, 749, 771, 859, 862, 907, 941, 950-951, 969, 982, 984, 1001, 1009, 1014, 1098-1099, 1103, 1109, 1141, 1143-1147
Variance, 745-746, 766
Vector, 596, 601, 606, 612, 614, 616, 629, 634, 671, 675-676, 679, 715-718, 724, 729-731, 733, 735-739, 742, 745, 760, 764, 791-792, 814, 825, 843, 860, 907, 915-920, 950-952, 959, 969, 982-984, 991, 1011, 1022, 1084, 1086-1087, 1105-1108, 1115-1116, 1126, 1148
Vector Space, 601, 629, 729, 736, 791-792, 860
Weighting Function, 605-607, 609
William George Horner, 800
Workspace, 583-584, 985, 1099, 1103, 1108, 1110, 1141, 1143-1145
Zero Matrix, 584

INDEX of MATLAB COMMANDS and SCRIPT

AbsTol, 983–985, 1061, 1084
adjugate, 587–589, 797
airy, 941, 990, 992
arrayfun, 928
besselj, 896–897, 900
bessely, 912
bisect, 693–694, 697, 700–702, 985
boole, 894–895, 906, 911
break, 693, 698, 712, 1134, 1153
cellfun, 1054–1055, 1064, 1082
charpoly, 634, 825–826, 833–834
circshift, 823, 827, 829, 832
cond, 587, 616–618, 817–818, 832, 856
conv, 731, 839
Departure, 746–747, 766
det, 586–587, 589, 630, 634, 794–796
diag, 825–826, 833, 1108
double, 586, 631, 633–634, 676–677, 800–801, 825, 834, 908–909, 1067, 1073, 1076, 1082, 1084, 1087, 1098, 1158
draw_dir_field, 928, 941
dsolve, 941, 974, 1001
eig, 631–633
ElementaryGauss, 596, 598–599
elemlu, 621–623
euler357, 949–950, 952–955, 957–958, 969, 971, 974, 976–978, 982–983, 986, 988
expm, 634
eye, 584, 602, 604, 621, 822–823, 826, 832–833, 1024–1025, 1108–1109
factorial, 641, 648–649, 905, 1133, 1152
falsepos, 697, 699–701
flipud, 991–992
floor, 950–951, 970, 1049, 1065, 1082–1083, 1094–1096, 1123
for-end, 810, 823, 937
for-loop, 1152
format, 648–649, 661, 669, 680, 691, 694, 699–702, 756, 799, 978, 1110, 1116
fprintf, 649–650, 900, 937–938, 1145
fsolve, 723–724, 727–728
fzero, 673, 679–682, 691, 702, 708
gamma, 905, 990, 992
get, 611, 645, 664, 685, 702, 726, 773, 941–942, 957, 986, 1005, 1057, 1063, 1066, 1076, 1104
getframe, 1021, 1050, 1065, 1083, 1096
global, 935–936, 967, 1009, 1014, 1144
GmSchmidt, 602–603
GmSchmidtModified, 603
heun357, 969–971, 974, 976, 982–983, 986, 989
if-else-end, 1042, 1149–1150
if-elseif-end, 691, 1149–1151
if-end, 814, 905, 1149–1150
integral, 605, 607, 609, 889, 891, 895–896, 898, 900–909, 911–913, 934, 965–967
integral2, 910
integral3, 910
inv, 585–586, 589, 592–593, 740, 743–744, 755, 783–784, 1024–1025, 1071
isempty, 693, 697, 712, 898, 901, 904, 907, 909
lagrange, 789, 792, 810–813, 835–841, 850, 853–854, 858–859, 865–867, 869–871, 873, 875, 877, 879, 881, 883, 885, 887
linspace, 799, 815, 829, 831, 833, 840, 844, 847, 851–852, 855, 857, 862, 864, 868–869, 874, 897, 902, 904, 907, 910, 928, 937, 942, 950–951, 970, 1116–1117
lu, 621–627

Mass, 923, 984, 1009–1010, 1015, 1030, 1033, 1040, 1042, 1047, 1080, 1084–1089, 1142
mean, 637–638, 744–745, 747, 765–767, 784, 859–860, 862–863
meshgrid, 777, 910, 928
minpoly, 634
mod, 901–902, 904, 907, 909–910, 1123
MStateDependence, 984, 1084, 1086–1089
NaN, 661, 679, 1110
nargin, 602, 604, 693, 697, 712, 814, 829, 839, 898, 901, 904, 907, 909, 950, 970
newton, 673, 705–709, 711, 713, 715, 717, 719, 721–723, 789, 792, 801, 803–806, 810–813, 819–821, 823, 825, 827–833, 838–839, 841, 847–848, 850, 853–854, 856–859, 894–895
newton2, 833
newton3, 833
newtraph, 711–712
norm, 602, 604, 612–618, 738, 767
num2cell, 1055, 1064, 1082
num2str, 875, 900, 1002, 1005–1006, 1021, 1049, 1055–1056, 1064–1065, 1082–1083, 1096
ode15s, 982, 986, 998–1000, 1003–1007, 1086–1087
ode23, 981
ode23s, 982, 1087
ode23t, 982, 1086–1087
ode23tb, 982, 1087
ode45, 981–989, 992–995, 998–1000, 1002–1007, 1009, 1012–1014, 1019–1021, 1025, 1031, 1034, 1037, 1047, 1049–1050, 1052, 1054–1056, 1064, 1074, 1077, 1082, 1089, 1092, 1094
ode113, 981–982
odeset, 983–985, 1033, 1055, 1064, 1074, 1077, 1082–1083, 1085, 1088–1089

options, 634, 681, 702, 983–985, 1012, 1055–1056, 1059, 1064–1065, 1074, 1077, 1082–1084, 1088–1089, 1121, 1135
plot, 640–641, 674–676, 679, 682, 684–685, 698, 700, 706, 708–710, 720, 725–726, 733, 743, 751–752, 764, 766, 768, 770, 776–777, 784–785, 799, 815–817, 830, 840, 844, 847–850, 852, 855, 857, 862–864, 868–870, 873–874, 889, 896–897, 909, 927, 938, 942–944, 946, 953–954, 957, 971–976, 985–995, 999, 1001, 1006, 1014, 1018–1021, 1025–1026, 1029, 1031, 1034–1035, 1037, 1044, 1047, 1049–1051, 1053–1055, 1065–1066, 1074, 1076–1078, 1080, 1083, 1089, 1092–1093, 1095, 1098, 1125–1126, 1130, 1132–1138, 1158
plot3, 776, 1052–1053, 1138
plotyy, 954–957
poly, 634, 731, 825
polyfit, 755–757, 764, 843–845, 850–854, 863–864
polyint, 908
polyval, 731, 733, 799–800, 814, 818, 844–845, 852, 856, 863–864
qr, 604, 845
quad, 907, 912
quad2d, 910
quadl, 907, 912
quiver, 928, 942
realmax, 666–667
realmin, 665, 668
RelTol, 983–985, 1054–1059, 1061, 1063–1065, 1073–1074, 1077, 1082, 1084, 1088–1089, 1093
roots, 630, 634, 671–680, 682–684, 686, 688, 690, 692, 694, 696, 698, 700, 702–708, 710, 712, 714, 716, 718–722, 724–734
rref, 599–601, 607

set, 601, 605, 607–609, 618, 630, 637–638, 646, 652–653, 661, 663, 665, 684, 688, 690–691, 696, 699, 702, 715, 720, 727–729, 737, 741, 744, 749–750, 755, 759, 762, 767–768, 771, 777, 787, 791–792, 796–797, 801, 804, 807, 810, 814–815, 820, 835, 845, 847, 850–851, 861, 865–866, 868, 870, 873, 875, 881, 883–884, 887, 890, 905, 942, 947, 956, 1005, 1019, 1024, 1036, 1047, 1049, 1052, 1056–1057, 1061, 1063, 1065, 1074, 1078, 1082, 1086–1087, 1089, 1093–1094, 1120, 1132, 1135, 1138, 1141
simplify, 589, 607, 927
simpson13, 901
simpson38, 904, 906, 909
simpsonmulti38, 909
size, 587–588, 596–597, 602, 604, 616, 621, 657, 663, 747, 777, 889, 900–901, 904, 906, 933, 936, 940, 944–945, 950–952, 969, 972–973, 977, 979, 983–984, 986–989, 997, 1000, 1099, 1107, 1109, 1113, 1145, 1147–1148, 1152–1153
sprintf, 875, 978
std, 746, 767, 860, 862–863
strcat, 875, 1002, 1005–1006, 1055–1056, 1064–1065, 1082
subplot, 726, 998–1000, 1002, 1005, 1025–1026, 1049, 1055–1056, 1064–1065, 1082–1083, 1094–1095, 1137–1138
subtitle, 902, 1005–1006, 1056
svd, 634
sym, 586–587, 606, 631, 799, 822–823, 826
tic, 833, 945, 999–1000, 1002, 1005
toc, 833, 945, 999–1000, 1002, 1005–1006
trace, 583
trapazoid, 898, 900, 904
trapz, 907
vander, 618, 743
var, 746
varargin, 693, 697–698, 711–712
Videowriter, 1022, 1048, 1064, 1082, 1094
view, 777, 860, 965, 1052, 1082, 1094, 1099, 1135
while-loop, 1152
whos, 1145
writeVideo, 1022, 1048, 1064, 1082, 1094
zero, 584, 587, 595–599, 622–623, 625, 642, 651, 656, 659–660, 665, 679, 681, 684, 696, 707, 724, 729, 742, 876, 881, 897, 905, 911–912, 944, 1016–1017, 1019, 1021, 1036, 1047, 1053, 1085–1087, 1110, 1123, 1150–1151, 1155, 1159