Advanced Mathematics and Mechanics Applications Using MATLAB®

Third Edition

Howard B. Wilson
University of Alabama

Louis H. Turcotte
Rose-Hulman Institute of Technology

David Halpern
University of Alabama
For my dear wife, Emma.

Howard B. Wilson

For my loving wife, Evelyn, our departed cat, Patches, and my parents.

Louis H. Turcotte

© 2003 by Chapman & Hall/CRC
Preface

This book uses MATLAB\textsuperscript{®} to analyze various applications in mathematics and mechanics. The authors hope to encourage engineers and scientists to consider this modern programming environment as an excellent alternative to languages such as FORTRAN or C++. MATLAB\textsuperscript{1} embodies an interactive environment with a high level programming language supporting both numerical and graphical commands for two- and three-dimensional data analysis and presentation. The wealth of intrinsic mathematical commands to handle matrix algebra, Fourier series, differential equations, and complex-valued functions makes simple calculator operations of many tasks previously requiring subroutine libraries with cumbersome argument lists.

We analyze problems, drawn from our teaching and research interests, emphasizing linear and nonlinear differential equation methods. Linear partial differential equations and linear matrix differential equations are analyzed using eigenfunctions and series solutions. Several types of physical problems are considered. Among these are heat conduction, harmonic response of strings, membranes, beams, and trusses, geometrical properties of areas and volumes, flexure and buckling of indeterminate beams, elastostatic stress analysis, and multi-dimensional optimization.

Numerical integration of matrix differential equations is used in several examples illustrating the utility of such methods as well as essential aspects of numerical approximation. Attention is restricted to the Runge-Kutta method which is adequate to handle most situations. Space limitation led us to omit some interesting MATLAB features concerning predictor-corrector methods, stiff systems, and event locations.

This book is not an introductory numerical analysis text. It is most useful as a reference or a supplementary text in computationally oriented courses emphasizing applications. The authors have previously solved many of the examples in FORTRAN. Our MATLAB solutions consume over three hundred pages (over twelve thousand lines). Although few books published recently present this much code, comparable FORTRAN versions would probably be significantly longer. In fact, the conciseness of MATLAB was a primary motivation for writing the book.

The programs contain many comments and are intended for study as separate entities without an additional reference. Consequently, some deliberate redundancy

\textsuperscript{1}MATLAB is a registered trademark of The MathWorks, Inc. For additional information contact:
The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-1500
(508) 647-7000, Fax: (508) 647-7001
Email: info@mathworks.com

© 2003 by Chapman & Hall/CRC
exists between program comments and text discussions. We also list programs in a style we feel will be helpful to most readers. The source listings show line numbers adjacent to the MATLAB code. MATLAB code does not use line numbers or permit goto statements. We have numbered the lines to aid discussions of particular program segments. To conserve space, we often place multiple MATLAB statements on the same line when this does not interrupt the logical flow.

All of the programs presented are designed to operate under the 6.x version of MATLAB and Microsoft Windows. Both the text and graphics windows should be simultaneously visible. A windowed environment is essential for using capabilities like animation and interactive manipulation of three dimensional figures. The source code for all of the programs in the book is available from the CRC Press website at http://www.crcpress.com. The program collection is organized using an independent subdirectory for each of the thirteen chapters.

This third edition incorporates much new material on time dependent solutions of linear partial differential equations. Animation is used whenever seeing the solution evolve in time is helpful. Animation illustrates quite well phenomena like wave propagation in strings and membranes. The interactive zoom and rotation features in MATLAB are also valuable tools for interpreting graphical output.

Most programs in the book are academic examples, but some problem solutions are useful as stand-alone analysis tools. Examples include geometrical property calculation, differentiation or integration of splines, Gauss integration of arbitrary order, and frequency analysis of trusses and membranes.

A chapter on eigenvalue problems presents applications in stress analysis, elastic stability, and linear system dynamics. A chapter on analytic functions shows the efficiency of MATLAB for applying complex valued functions and the Fast Fourier Transform (FFT) to harmonic and biharmonic functions. Finally, the book concludes with a chapter applying multidimensional search to several nonlinear programming problems.

We emphasize that this book is primarily for those concerned with physical applications. A thorough grasp of Euclidean geometry, Newtonian mechanics, and some mathematics beyond calculus is essential to understand most of the topics. Finally, the authors enjoy interacting with students, teachers, and researchers applying advanced mathematics to real world problems. The availability of economical computer hardware and the friendly software interface in MATLAB makes computing increasingly attractive to the entire technical community. If we manage to cultivate interest in MATLAB among engineers who only spend part of their time using computers, our primary goal will have been achieved.

Howard B. Wilson hwilson@bama.ua.edu
Louis H. Turcotte turcotte@rose-hulman.edu
David Halpern david.halpern@ua.edu
Contents

1 Introduction
   1.1 MATLAB: A Tool for Engineering Analysis
   1.2 MATLAB Commands and Related Reference Materials
   1.3 Example Problem on Financial Analysis
   1.4 Computer Code and Results
      1.4.1 Computer Output
      1.4.2 Discussion of the MATLAB Code
      1.4.3 Code for Financial Problem

2 Elementary Aspects of MATLAB Graphics
   2.1 Introduction
   2.2 Overview of Graphics
   2.3 Example Comparing Polynomial and Spline Interpolation
   2.4 Conformal Mapping Example
   2.5 Nonlinear Motion of a Damped Pendulum
   2.6 A Linear Vibration Model
   2.7 Example of Waves in an Elastic String
   2.8 Properties of Curves and Surfaces
      2.8.1 Curve Properties
      2.8.2 Surface Properties
      2.8.3 Program Output and Code

3 Summary of Concepts from Linear Algebra
   3.1 Introduction
   3.2 Vectors, Norms, Linear Independence, and Rank
   3.3 Systems of Linear Equations, Consistency, and Least Squares Approximation
   3.4 Applications of Least Squares Approximation
      3.4.1 A Membrane Deflection Problem
      3.4.2 Mixed Boundary Value Problem for a Function Harmonic Inside a Circular Disk
      3.4.3 Using Rational Functions to Conformally Map a Circular Disk onto a Square
   3.5 Eigenvalue Problems
      3.5.1 Statement of the Problem
      3.5.2 Application to Solution of Matrix Differential Equations

© 2003 by Chapman & Hall/CRC
3.5.3 The Structural Dynamics Equation
3.6 Computing Natural Frequencies for a Rectangular Membrane
3.7 Column Space, Null Space, Orthonormal Bases, and SVD
3.8 Computation Time to Run a MATLAB Program

4 Methods for Interpolation and Numerical Differentiation
4.1 Concepts of Interpolation
4.2 Interpolation, Differentiation, and Integration by Cubic Splines
  4.2.1 Computing the Length and Area Bounded by a Curve
  4.2.2 Example: Length and Enclosed Area for a Spline Curve
  4.2.3 Generalizing the Intrinsic Spline Function in MATLAB
  4.2.4 Example: A Spline Curve with Several Parts and Corners
4.3 Numerical Differentiation Using Finite Differences
  4.3.1 Example: Program to Derive Difference Formulas

5 Gauss Integration with Geometric Property Applications
5.1 Fundamental Concepts and Intrinsic Integration Tools in MATLAB
5.2 Concepts of Gauss Integration
5.3 Comparing Results from Gauss Integration and Function QUADL
5.4 Geometrical Properties of Areas and Volumes
  5.4.1 Area Property Program
  5.4.2 Program Analyzing Volumes of Revolution
5.5 Computing Solid Properties Using Triangular Surface Elements and Using Symbolic Math
5.6 Numerical and Symbolic Results for the Example
5.7 Geometrical Properties of a Polyhedron
5.8 Evaluating Integrals Having Square Root Type Singularities
  5.8.1 Program Listing
5.9 Gauss Integration of a Multiple Integral
  5.9.1 Example: Evaluating a Multiple Integral

6 Fourier Series and the Fast Fourier Transform
6.1 Definitions and Computation of Fourier Coefficients
  6.1.1 Trigonometric Interpolation and the Fast Fourier Transform
6.2 Some Applications
  6.2.1 Using the FFT to Compute Integer Order Bessel Functions
  6.2.2 Dynamic Response of a Mass on an Oscillating Foundation
  6.2.3 General Program to Plot Fourier Expansions

7 Dynamic Response of Linear Second Order Systems
7.1 Solving the Structural Dynamics Equations for Periodic Forces
  7.1.1 Application to Oscillations of a Vertically Suspended Cable
7.2 Direct Integration Methods
  7.2.1 Example on Cable Response by Direct Integration

© 2003 by Chapman & Hall/CRC
8 Integration of Nonlinear Initial Value Problems
  8.1 General Concepts on Numerical Integration of Nonlinear Matrix Differential Equations
  8.2 Runge-Kutta Methods and the ODE45 Integrator Provided in MATLAB
  8.3 Step-size Limits Necessary to Maintain Numerical Stability
  8.4 Discussion of Procedures to Maintain Accuracy by Varying Integration Step-size
  8.5 Example on Forced Oscillations of an Inverted Pendulum
  8.6 Dynamics of a Spinning Top
  8.7 Motion of a Projectile
  8.8 Example on Dynamics of a Chain with Specified End Motion
  8.9 Dynamics of an Elastic Chain

9 Boundary Value Problems for Partial Differential Equations
  9.1 Several Important Partial Differential Equations
  9.2 Solving the Laplace Equation inside a Rectangular Region
  9.3 The Vibrating String
  9.4 Force Moving on an Elastic String
    9.4.1 Computer Analysis
  9.5 Waves in Rectangular or Circular Membranes
    9.5.1 Computer Formulation
    9.5.2 Input Data for Program membwave
  9.6 Wave Propagation in a Beam with an Impact Moment Applied to One End
  9.7 Forced Vibration of a Pile Embedded in an Elastic Medium
  9.8 Transient Heat Conduction in a One-Dimensional Slab
  9.9 Transient Heat Conduction in a Circular Cylinder with Spatially Varying Boundary Temperature
    9.9.1 Problem Formulation
    9.9.2 Computer Formulation
  9.10 Torsional Stresses in a Beam of Rectangular Cross Section

10 Eigenvalue Problems and Applications
  10.1 Introduction
  10.2 Approximation Accuracy in a Simple Eigenvalue Problem
  10.3 Stress Transformation and Principal Coordinates
    10.3.1 Principal Stress Program
    10.3.2 Principal Axes of the Inertia Tensor
  10.4 Vibration of Truss Structures
    10.4.1 Truss Vibration Program
  10.5 Buckling of Axially Loaded Columns
    10.5.1 Example for a Linearly Tapered Circular Cross Section
    10.5.2 Numerical Results
10.6 Accuracy Comparison for Euler Beam Natural Frequencies by Finite Element and Finite Difference Methods
   10.6.1 Mathematical Formulation
   10.6.2 Discussion of the Code
   10.6.3 Numerical Results

10.7 Vibration Modes of an Elliptic Membrane
   10.7.1 Analytical Formulation
   10.7.2 Computer Formulation

11 Bending Analysis of Beams of General Cross Section
   11.1 Introduction
      11.1.1 Analytical Formulation
      11.1.2 Program to Analyze Beams of General Cross Section
      11.1.3 Program Output and Code

12 Applications of Analytic Functions
   12.1 Properties of Analytic Functions
   12.2 Definition of Analyticity
   12.3 Series Expansions
   12.4 Integral Properties
      12.4.1 Cauchy Integral Formula
      12.4.2 Residue Theorem
   12.5 Physical Problems Leading to Analytic Functions
      12.5.1 Steady-State Heat Conduction
      12.5.2 Incompressible Inviscid Fluid Flow
      12.5.3 Torsion and Flexure of Elastic Beams
      12.5.4 Plane Elastostatics
      12.5.5 Electric Field Intensity
   12.6 Branch Points and Multivalued Behavior
   12.7 Conformal Mapping and Harmonic Functions
   12.8 Mapping onto the Exterior or the Interior of an Ellipse
      12.8.1 Program Output and Code
   12.9 Linear Fractional Transformations
      12.9.1 Program Output and Code
   12.10 Schwarz-Christoffel Mapping onto a Square
      12.10.1 Program Output and Code
   12.11 Determining Harmonic Functions in a Circular Disk
      12.11.1 Numerical Results
      12.11.2 Program Output and Code
   12.12 Inviscid Fluid Flow around an Elliptic Cylinder
      12.12.1 Program Output and Code
   12.13 Torsional Stresses in a Beam Mapped onto a Unit Disk
      12.13.1 Program Output and Code
   12.14 Stress Analysis by the Kolosov-Muskhelishvili Method
      12.14.1 Program Output and Code

© 2003 by Chapman & Hall/CRC
12.14.2 Stressed Plate with an Elliptic Hole
12.14.3 Program Output and Code

13 Nonlinear Optimization Applications
13.1 Basic Concepts
13.2 Initial Angle for a Projectile
13.3 Fitting Nonlinear Equations to Data
13.4 Nonlinear Deflections of a Cable
13.5 Quickest Time Descent Curve (the Brachistochrone)
13.6 Determining the Closest Points on Two Surfaces
   13.6.1 Discussion of the Computer Code

A List of MATLAB Routines with Descriptions
B Selected Utility and Application Functions

References
Chapter 1

Introduction

1.1 MATLAB: A Tool for Engineering Analysis

This book presents various MATLAB applications in mechanics and applied mathematics. Our objective is to employ numerical methods in examples emphasizing the appeal of MATLAB as a programming tool. The programs are intended for study as a primary component of the text. The numerical methods used include interpolation, numerical integration, finite differences, linear algebra, Fourier analysis, roots of nonlinear equations, linear differential equations, nonlinear differential equations, linear partial differential equations, analytic functions, and optimization methods. Many intrinsic MATLAB functions are used along with some utility functions developed by the authors. The physical applications vary widely from solution of linear and nonlinear differential equations in mechanical system dynamics to geometrical property calculations for areas and volumes.

For many years FORTRAN has been the favorite programming language for solving mathematical and engineering problems on digital computers. An attractive alternative is MATLAB which facilitates program development with excellent error diagnostics and code tracing capabilities. Matrices are handled efficiently with many intrinsic functions performing familiar linear algebra tasks. Advanced software features such as dynamic memory allocation and interactive error tracing reduce the time to get solutions. The versatile but simple graphics commands in MATLAB also allow easy preparation of publication quality graphs and surface plots for technical papers and books. The authors have found that MATLAB programs are often significantly shorter than corresponding FORTRAN versions. Consequently, more time is available for the primary purpose of computing, namely, to better understand physical system behavior.

The mathematical foundation needed to grasp most topics presented here is covered in an undergraduate engineering curriculum. This should include a grounding in calculus, differential equations, and knowledge of a procedure oriented programming language like FORTRAN. An additional course on advanced engineering mathematics covering linear algebra, matrix differential equations, and eigenfunction solutions of partial differential equations will also be valuable. The MATLAB programs were written primarily to serve as instructional examples in classes traditionally referred to as advanced engineering mathematics and applied numerical methods. The greatest benefit to the reader will probably be derived through study of the programs relat-
ing mainly to physics and engineering applications. Furthermore, we believe that several of the MATLAB functions are useful as general utilities. Typical examples include routines for spline interpolation, differentiation, and integration; area and inertial moments for general plane shapes; and volume and inertial properties of arbitrary polyhedra. We have also included examples demonstrating natural frequency analysis and wave propagation in strings and membranes.

MATLAB is now employed in more than two thousand universities and the user community throughout the world numbers in the thousands. Continued growth will be fueled by decreasing hardware costs and more people familiar with advanced analytical methods. The authors hope that our problem solutions will motivate analysts already comfortable with languages like FORTRAN to learn MATLAB. The rewards of such efforts can be considerable.

1.2 MATLAB Commands and Related Reference Materials

MATLAB has a rich command vocabulary covering most mathematical topics encountered in applications. The current section presents instructions on: a) how to learn MATLAB commands, b) how to examine and understand MATLAB’s lucidly written and easily accessible “demo” programs, and c) how to expand the command language by writing new functions and programs. A comprehensive online help system is included and provides lengthy documentation of all the operators and commands. Additional capabilities are provided by auxiliary toolboxes. The reader is encouraged to study the command summary to get a feeling for the language structure and to have an awareness of powerful operations such as \texttt{null}, \texttt{orth}, \texttt{eig}, and \texttt{fft}.

The manual for The Student Edition of MATLAB should be read thoroughly and kept handy for reference. Other references [47, 97, 103] also provide valuable supplementary information. This book extends the standard MATLAB documentation to include additional examples which we believe are complementary to more basic instructional materials.

Learning to use \texttt{help}, \texttt{type}, \texttt{dbtype}, \texttt{demo}, and \texttt{diary} is important to understanding MATLAB. \texttt{help function\_name} (such as \texttt{help plot}) lists available documentation on a command or function generically called “function name.” MATLAB responds by printing introductory comments in the relevant function (comments are printed until the first blank line or first MATLAB command after the function heading is encountered). This feature allows users to create online help for their own functions by simply inserting appropriate comments at the top of the function. The instruction \texttt{type function\_name} lists the entire source code for any function where source code is available (the code for intrinsic functions stored in compiled binary for computational efficiency cannot be listed). Consider the following list of typical examples
### 1.3 Example Problem on Financial Analysis

Let us next analyze a problem showing several language constructs of MATLAB programming. Most of this book is devoted to solving initial value and boundary value problems for physical systems. For sake of variety we study briefly an elementary example useful in business, namely, asset growth resulting from compounded investment return.

The differential equation

\[ Q'(t) = RQ(t) + S \exp(At) \]

describes growth of investment capital earning a rate of investment return \( R \) and augmented by a saving rate \( S \exp(At) \). The general solution of this first order linear equation is

\[ Q(t) = \exp(Rt) \left[ Q(0) + \int_0^t S \exp((A - R)t) \, dt \right]. \]
A realistic formulation should employ inflation adjusted capital defined by

\[ q(t) = Q(t) \exp(-It) \]

where \( I \) denotes the annual inflation rate. Then a suitable model describing capital accumulation over a saving interval of \( t_1 \) years, followed by a payout period of \( t_2 \) years, is characterized as

\[ q'(t) = rq(t) + [s(t \leq t_1) - p \exp(-at_1)(t > t_1)] \exp(at), \quad q(0) = q_0. \]

The quantity \( t \leq t_1 \) equals one for \( t \leq t_1 \) and is zero otherwise. This equation also uses inflation adjusted parameters \( r = R - I \) and \( a = A - I \). The parameter \( s \) quantifies the initial saving rate and \( p \) is the payout rate starting at \( t = t_1 \).

It is plausible to question whether continuous compounding is a reasonable alternative to a discrete model employing assumptions such as quarterly or yearly compounding. It turns out that results obtained, for example, using discrete monthly compounding over several years differ little from those produced with the continuous model. Since long term rates of investment return and inflation are usually estimated rather than known exactly, the simplified formulas for continuous compounding illustrate reasonably well the benefits of long term investment growth. Integrating the differential equation for the continuous compounding model gives

\[ q(t) = q_0 \exp(rt) + s[h(t) - (t > t_1) \exp(at_1)h(t - t_1)] - p (t > t_1)h(t - t_1) \]

where \( h(t) = \frac{\exp(rt) - \exp(at)}{r - a} \). The limiting case for \( r = a \) is also dealt with appropriately in the program below. At time \( T_2 = t_1 + t_2 \) the final capital \( q_2 = q(T_2) \) is

\[ q_2 = q_0 \exp(rT_2) + \frac{s}{r - a} [\exp(rt_1) - \exp(at_1)] \exp(rt_2) - \frac{p}{r - a} [\exp(rt_2) - \exp(at_2)]. \]

Therefore, for known \( r, a, t_1, t_2 \), the four quantities \( q_2, q_0, s, p \) are linearly related and any particular one of these values can be found in terms of the other three. For instance, when \( q_0 = q_2 = 0 \), the saving factor \( s \) needed to provide a desired payout factor \( p \) can be computed from the useful equation

\[ s = p[1 - \exp((a - r)t_2)]/[\exp(rt_1) - \exp(at_1)] \]

A MATLAB program using the above equations was written to compute and plot \( q(t) \) for general combinations of the nine parameters \( R, A, I, t_1, t_2, q_0, s, p, q_2 \). The program allows data to be passed through the call list of function \texttt{finance}, or the interactive input is activated when no call list data is passed. \texttt{Finance} calls function \texttt{inputv} to read data and the function \texttt{savespnd} to evaluate \( q(t) \). First we will show some numerical results and then discuss selected parts of the code. Consider a case where someone initially starting with $10,000 of capital expects to save for 40 years
and subsequently draw $50,000 annually from savings for 20 years, at which time the remaining capital is to be $100,000. Assume that the investment rate before inflation is \( R = 8 \) while the inflation rate is \( I = 4 \). During the 60 year period, annual savings, as well as the pension payout amount, are to be increased to match inflation, so that \( A = 4 \). The necessary value of \( s \) and a plot of the inflation adjusted assets as a function of time are to be determined. The program output shows that when the unknown value of \( s \) was input as nan (meaning Not-a-Number in IEEE arithmetic), a corrected value of $6417 was computed. This says that, with the assumed rate of investment return, saving at an initial rate of $6417 per year and continually increasing that amount to match inflation will suffice to provide the desired inflation adjusted payout. Furthermore, the inflation adjusted financial capital accumulated at the end of 40 years is $733,272. The related graph of \( q(t) \) duplicates the data listed on the text screen. The reader may find it interesting to repeat the illustrative calculation assuming \( R = 11 \), in which case the saving coefficient is greatly reduced to only $1060.

### 1.4 Computer Code and Results

A computer code which analyzes the above equations and presents both numerical and graphical results appears next. First we show the program output, and then discuss particular aspects of the program.

#### 1.4.1 Computer Output

```matlab
>> finance;

ANALYSIS OF THE SAVE-SPEND PROBLEM BY SOLVING
\[ q'(t) = r*q(t) + [s*(t<=t1) - p*(t>t1) * \exp(-a*t1)] * \exp(a*t) \]
where \( r = R - I \), \( a = A - I \), and \( q(0) = q0 \)

To list parameter definitions enter y
otherwise enter n  ? y

INPUT QUANTITIES:
R - annual percent earnings on assets
I - annual percent inflation rate
A - annual percent increase in savings to offset inflation
r,a - inflation adjusted values of R and I
t1 - saving period (years), 0<t<t1
t2 - payout period (years), t1<t<(t1+t2)
s - saving rate at t=0, ($K). Saving is
```

© 2003 by CRC Press LLC
expressed as $s \cdot \exp(a \cdot t)$, $0 < t < t_1$

$p$ - payout rate at $t = t_1$, ($K$). Payout is expressed as

$-p \cdot \exp(a \cdot (t - t_1))$, $t_1 < t < (t_1 + t_2)$

$q_0$ - initial savings at $t = 0$, ($K$)

$q_2$ - final savings at $t = T_2 = t_1 + t_2$, ($K$)

OUTPUT QUANTITIES:
$q$ - vector of inflation adjusted savings values for $0 \leq t \leq (t_1 + t_2)$

$t$ - vector of times (years) corresponding to the components of $q$

$q_1$ - value of savings at $t = t_1$, when the saving period ends

Press return to continue

Input $R, A, I$ (try 11, 4, 4) ? 8, 4, 4
Input $t_1, t_2$ (try 40, 20) ? 40, 20
Input $q_0, s, p, q_2$ (try 20, 5, nan, 40) ? 20, nan, 50, 100

PROGRAM RESULTS

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>R</th>
<th>A</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.000</td>
<td>20.000</td>
<td>8.000</td>
<td>4.000</td>
<td>4.000</td>
</tr>
</tbody>
</table>

$q_0$  $q_1$  $q_2$  $s$  $p$

| 20.000 | 733.272 | 100.000 | 6.417 | 50.000 |

>>
Figure 1.1: Accumulated Assets versus Time

TOTAL SAVINGS WHEN $T_1 = 40, T_2 = 20, s = 6.4175, p = 50$

- $R = 8.000$
- $I = 4.000$
- $A = 4.000$
- $q_0 = 20.000$
- $q_1 = 733.272$
- $q_2 = 100.000$

© 2003 by CRC Press LLC
1.4.2 Discussion of the MATLAB Code

Let us examine the following program listing. The line numbers, which are not part of the actual code, are helpful for discussing particular parts of the program. A numbered listing can be obtained with the MATLAB command `dbtype`.

<table>
<thead>
<tr>
<th>Line</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>Three dots . . . are used to continue function <code>finance</code> to handle the long argument list. The output list duplicates some input items to handle cases involving interactive input.</td>
</tr>
<tr>
<td>3-16</td>
<td>Comment lines always begin with the % symbol. At the interactive command level in MATLAB, typing <code>help</code> followed by a function name will print documentation in the first unbroken sequence of comments in a function or script file.</td>
</tr>
<tr>
<td>20-25</td>
<td>The output heading is printed. Note that q&quot;(t) is used to print q'(t) because special characters such as ' or % must be repeated.</td>
</tr>
<tr>
<td>29-50</td>
<td>Intrinsic function <code>char</code> is used to store descriptions of program variable in a character matrix.</td>
</tr>
<tr>
<td>59</td>
<td>Function <code>nargin</code> checks whether the number of input variables is zero. If so, data values are read interactively.</td>
</tr>
<tr>
<td>68-69</td>
<td>Function <code>inputv</code> reads several variables on the same line.</td>
</tr>
<tr>
<td>70-78</td>
<td>While 1,...,end code sequence loops repeatedly to check data input. Break exits to line 80 if data are OK.</td>
</tr>
<tr>
<td>85-97</td>
<td>Set multiplier constants to solve for one unknown variable among q0, s, p, q2.</td>
</tr>
<tr>
<td>99-105</td>
<td>Determine time vectors to evaluate the solution. Cases where t1 or t2 are zero require special treatment.</td>
</tr>
<tr>
<td>108-112</td>
<td>Intrinsic function <code>isnan</code> is used to identify the variable which was input as nan.</td>
</tr>
<tr>
<td>115-116</td>
<td>User defined function <code>savespnd</code> is used to evaluate q(t) and q(t1).</td>
</tr>
<tr>
<td>119-127</td>
<td>Program results are printed with a chosen format. The statement <code>b=inline('blanks(j)',j')</code> just shortens the name for intrinsic function <code>blanks</code>.</td>
</tr>
<tr>
<td>130-139</td>
<td>Draw the graph along with a title and axis labels.</td>
</tr>
<tr>
<td>141-153</td>
<td>Create a label containing data values. Position it on the graph.</td>
</tr>
<tr>
<td>154</td>
<td>Turn the grid off and bring the graph to the foreground.</td>
</tr>
<tr>
<td>158-161</td>
<td>Function <code>savespnd</code> evaluates q(t). The formula for r=a results from the limiting form of q(t) as parameter a tends to r.</td>
</tr>
<tr>
<td>180-213</td>
<td>Function <code>inputv</code> generalizes the intrinsic function <code>input</code> to read several variables on the same line. <code>Inputv</code> is used often throughout this text.</td>
</tr>
</tbody>
</table>
1.4.3 Code for Financial Problem

Program finance

function [q,t,R,A,I,t1,t2,s,p,q0,q1,q2]=finance...
    (R,A,I,t1,t2,s,p,q0,q2)
%
% [q,t,R,A,I,t1,t2,s,p,q0,q1,q2]=finance...
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%
% This function solves the SAVE-SPEND PROBLEM
% where funds earning interest are accumulated
% during one period and paid out in a subsequent
% period. The value of assets is adjusted to
% account for inflation. This problem is
% governed by the differential equation
% q’(t)=r*q(t)+[s*(t<=t1)
% -p*(t>t1)*exp(-a*t1)]*exp(a*t) where
% r=R-I, a=A-I and the remaining parameters
% are defined below
%
% User m functions required: inputv, savespnd

disp(''), disp([' ','... 
', 'ANALYSIS OF THE SAVE-SPEND PROBLEM BY SOLVING'])
disp(...
['q'(t)=r*q(t)+[s*(t<=t1)-p*(t>t1)*',
'exp(-a*t1)]*exp(a*t')]), disp(...
'where r=R-I, a=A-I, and q(0)=q0'), disp(' ')

disp( ... 
['r','a - inflation adjusted values of R and I',
't1 - saving period (years), 0<t<t1',
't2 - payout period (years), t1<t<(t1+t2)',
's - saving rate at t=0, ($K). Saving is',
' expressed as s*exp(a*t), 0<t<t1',
'p - payout rate at t=t1, ($K). Payout is',
' expressed as',...

© 2003 by CRC Press LLC
-p*exp(a*(t-t1)), t1<t<(t1+t2),
q0 - initial savings at t=0, ($K),
q2 - final savings at t=T2=t1+t2, ($K),
'OUTPUT QUANTITIES:',
q - vector of inflation adjusted savings',
values for 0 <= t <= (t1+t2),
t - vector of times (years) corresponding',
to the components of q',
q1 - value of savings at t=t1, when the',
saving period ends',

% NOTE: WHEN R,I,A,T1,T2 ARE KNOWN, THEN FIXING
% ANY THREE OF THE VALUES q0,s,p,q2 DETERMINES
% THE UNKNOWN VALUE WHICH SHOULD BE GIVEN AS
% nan IN THE DATA INPUT.

% Read data interactively when input data is not
% passed through the call list
if nargin==0
    disp('To list parameter definitions enter y')
    query=input('otherwise enter n ? ','s');
    if query=='Y' | query=='y'
        disp(explain); disp('Press return to continue')
pause, disp(' ')
end

% Read multiple variables on the same line
[t1,t2]=inputv('Input t1,t2 (try 40,20) ? ');
while 1
    [q0,s,p,q2]=inputv(...
    'Input q0,s,p,q2 (try 20,5,nan,40) ? ');
    if sum(isnan([q0,s,p,q2]))==1, break; end
    fprintf(['\nDATA ERROR. ONE AND ONLY ',
    'ONE VALUE AMONG \n', 'THE PARAMETERS ',
    'q0,s,p,q2 CAN EQUAL nan \n'])
end

nt=101; T2=t1+t2; r=(R-I)/100; a=(A-I)/100;
c0=exp(r*T2);

% q0,s,p,q2 are related by q2=c0*q0+c1*s+c2*p
% Check special case where t1 or t2 are zero
if t1==0

© 2003 by CRC Press LLC
disp(' '), disp('s is set to zero when t1=0')
s=0; c1=0;
else
c1=savespnd(T2,t1,0,R,A,I,1,0);
end
if t2==0
disp(' '), disp('p is set to zero when t2=0')
p=0; c2=0;
else
c2=savespnd(T2,t1,0,R,A,I,0,1);
end
if t1==0 | t2==0
t=linspace(0,T2,nt)';
else
n1=max(2,fix(t1/T2*nt));
n2=max(2,nt-n1)-1;
t=[t1/n1*(0:n1),t1+t2/n2*(1:n2)]';
end
% Solve for the unknown parameter
if isnan(q0), q0=(q2-s*c1-p*c2)/c0;
elseif isnan(s), s=(q2-q0*c0-p*c2)/c1;
elseif isnan(p), p=(q2-q0*c0-s*c1)/c2;
else,
q2=q0*c0+s*c1+p*c2;
end
% Compute results for q(t)
q=savespnd(t,t1,q0,R,A,I,s,p);
q1=savespnd(t1,t1,q0,R,A,I,s,p);
% Print formatted results
b=inline('blanks(j)','j'); B=b(3); d='%8.3f';
u=[d,B,d,B,d,B,d,B,d,'
']; disp(' ') fprintf(u,t1,t2,R,A,I), disp(' ')
% Show results graphically
plot(t,q,'k')
title(['INFLATION ADJUSTED SAVINGS WHEN ',...
'S = ',num2str(s),', AND P = ',num2str(p)]);

titl=

['TOTAL SAVINGS WHEN T1 = ',num2str(t1),...
', T2 = ',num2str(t2),', s = ',num2str(s),...
', p = ',num2str(p)]; title(titl)

xlabel('TIME IN YEARS')
ylabel('TOTAL SAVINGS IN $K')

% Character label showing data parameters
label=char(...
sprintf('R = %8.3f',R),...
sprintf('I = %8.3f',I),...
sprintf('A = %8.3f',A),...
sprintf('q0 = %8.3f',q0),...
sprintf('q1 = %8.3f',q1),...
sprintf('q2 = %8.3f',q2));
w=axis; ymin=w(3); dy=w(4)-w(3);
xmin=w(1); dx=w(2)-w(1);
ytop=ymin+.8*dy; Dy=.065*dy;
xlft=xmin+0.04*dx;
text(xlft,ytop,label)

grid off, shg

%======================================================================

function q=savespnd(t,t1,q0,R,A,I,s,p)
%
% This function determines q(t) satisfying
% q'(t)=r*q+[s*(t<=t1)-p*(t>t1)]*...
% exp(-a*t1)]*exp(a*t), with q(0)=q0,
% r=(R-I)/100; a=(A-I)/100

if r~=a
    q=q0*exp(r*t)+s/c*(exp(r*t)-exp(a*t))...
    -(p+s*exp(a*t1))/c*(T>0).*...
    exp(r*T)-exp(a*T));
else % limiting case as a=>r
    q=q0*exp(r*t)+s*t.*exp(r*t)...
    -(p+s*exp(r*t1)).*T.*(T>0).*exp(r*T);
function varargout=inputv(prompt)

% [a1,a2,...,a_nargout]=inputv(prompt)

% This function reads several values on one line. The items should be separated by commas or blanks.

% prompt - A string preceding the data entry. It is set to ' ? ' if no value of prompt is given.
% a1,a2,...,a_nargout - The output variables that are created. If not enough data values are given following the prompt, the remaining undefined values are set equal to NaN

% A typical function call is:
% [A,B,C,D]=inputv('Enter values of A,B,C,D: ')

if nargin==0, prompt=' ? '; end
u=input(prompt,'s'); v=eval(['[',u,',']]);
i=length(v); no=nargout;
varargout=cell(1,no); k=min(i,no);
for j=1:k, varargout{j}=v{j}; end
if no>i
for j=i+1:no, varargout{j}=nan; end
end
Chapter 2

Elementary Aspects of MATLAB Graphics

2.1 Introduction

MATLAB’s capabilities for plotting curves and surfaces are versatile and easy to understand. In fact, the effort required to learn MATLAB would be rewarding even if it were only used to construct plots, save graphic images, and output publication quality graphs on a laser printer. Numerous help features and well-written demo programs are included with MATLAB. By executing the demo programs and studying the relevant code, users can quickly understand the techniques necessary to implement graphics within their programs. This chapter discusses a few of the graphics commands. These commands are useful in many applications and do not require extensive time to master. This next section provides a quick overview of the basics of using MATLAB’s graphics. The subsequent sections in this chapter present several additional examples (summarized in the table below) involving interesting applications which use these graphics primitives.

<table>
<thead>
<tr>
<th>Example</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial Interpolation</td>
<td>2-D graphics and polynomial interpolation functions</td>
</tr>
<tr>
<td>Conformal Mapping</td>
<td>2-D graphics and some aspects of complex numbers</td>
</tr>
<tr>
<td>Pendulum Motion</td>
<td>2-D graphics animation and ODE solution</td>
</tr>
<tr>
<td>Linear Vibration Model</td>
<td>Animated spring-mass response</td>
</tr>
<tr>
<td>String Vibration</td>
<td>2-D and 3-D graphics for a function of form ( y(x, t) )</td>
</tr>
<tr>
<td>Space Curve Geometry</td>
<td>3-D graphics for a space curve</td>
</tr>
<tr>
<td>Intersecting Surfaces</td>
<td>3-D graphics and combined surface plots</td>
</tr>
</tbody>
</table>
2.2 Overview of Graphics

The following commands should be executed since they will accelerate the understanding of graphics functions, and others, included within MATLAB.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help help</td>
<td>discusses use of help command.</td>
</tr>
<tr>
<td>help</td>
<td>lists categories of help.</td>
</tr>
<tr>
<td>help general</td>
<td>lists various utility commands.</td>
</tr>
<tr>
<td>help more</td>
<td>describes how to control output paging.</td>
</tr>
<tr>
<td>help diary</td>
<td>describes how to save console output to a file.</td>
</tr>
<tr>
<td>help plotxy</td>
<td>describes 2D plot functions.</td>
</tr>
<tr>
<td>help plotxyz</td>
<td>describes 3D plot functions.</td>
</tr>
<tr>
<td>help graphics</td>
<td>describes more general graphics features.</td>
</tr>
<tr>
<td>help demos</td>
<td>lists names of various demo programs.</td>
</tr>
<tr>
<td>intro</td>
<td>executes the intro program showing MATLAB commands including fundamental graphics capabilities.</td>
</tr>
<tr>
<td>help funfun</td>
<td>describes several numerical analysis programs contained in MATLAB.</td>
</tr>
<tr>
<td>type humps</td>
<td>lists a function employed in several of the MATLAB demos.</td>
</tr>
<tr>
<td>fplotdemo</td>
<td>executes program fplotdemo which plots the function named humps.</td>
</tr>
<tr>
<td>help peaks</td>
<td>describes a function peaks used to illustrate surface plots.</td>
</tr>
<tr>
<td>peaks</td>
<td>executes the function peaks to produce an interesting surface plot.</td>
</tr>
<tr>
<td>spline2d</td>
<td>executes a demo program to draw a curve through data input interactively.</td>
</tr>
</tbody>
</table>

The example programs can be studied interactively using the type command to list programs of interest. Library programs can also be inspected and printed using the MATLAB editor, but care should be taken not to accidentally overwrite the original library files with changes. Furthermore, text output in the command window can be captured in several ways. Some of these are: (1) Use the mouse to highlight material of interest. Then use the "Print Selected" on the file menu to send output to the printer; (2) Use CTRL-C to copy outlined text to the clipboard. Then open a new file and use CTRL-V to paste the text into the new file; and (3) Use a diary command such as diary mysave.doc to begin printing subsequent command window output into the chosen file. This printing can be turned off using diary off. Then the file can be edited, modified, or combined with other text using standard editor commands.

More advanced features of MATLAB graphics, including handle graphics, control of shading and light sources, creation of movies, etc., exceed the scope of the present text. Instead we concentrate on using the basic commands listed below and on producing simple animations. The advanced graphics can be mastered by studying the
MATLAB manuals and relevant demo programs. The principal graphing commands discussed here are

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot</td>
<td>draw two-dimensional graphs</td>
</tr>
<tr>
<td>xlabel, ylabel, zlabel</td>
<td>define axis labels</td>
</tr>
<tr>
<td>title</td>
<td>define graph title</td>
</tr>
<tr>
<td>axis</td>
<td>set various axis parameters (min, max, etc.)</td>
</tr>
<tr>
<td>legend</td>
<td>show labels for plot lines</td>
</tr>
<tr>
<td>sing</td>
<td>bring graphics window to foreground</td>
</tr>
<tr>
<td>text</td>
<td>place text at selected locations</td>
</tr>
<tr>
<td>grid</td>
<td>turns grid lines on or off</td>
</tr>
<tr>
<td>mesh</td>
<td>draw surface using colored lines</td>
</tr>
<tr>
<td>surf</td>
<td>draw surface using colored patches</td>
</tr>
<tr>
<td>hold</td>
<td>fix the graph limits between successive plots</td>
</tr>
<tr>
<td>view</td>
<td>change surface viewing position</td>
</tr>
<tr>
<td>drawnow</td>
<td>empty graphics buffer immediately</td>
</tr>
<tr>
<td>zoom</td>
<td>magnify graph or surface plot</td>
</tr>
<tr>
<td>clf</td>
<td>clear graphics window</td>
</tr>
<tr>
<td>contour</td>
<td>draw contour plot</td>
</tr>
<tr>
<td>ginput</td>
<td>read coordinates interactively</td>
</tr>
</tbody>
</table>

All of these commands, along with numerous others, are extensively documented by the help facilities in MATLAB. The user can get an introduction to these capabilities by typing "help plot" and by running the demo programs. The accompanying code for the demo program should be examined since it provides worthwhile insight into how MATLAB graphics is used.

2.3 Example Comparing Polynomial and Spline Interpolation

Many familiar mathematical functions such as \( \tan(x) \), \( \exp(x) \), \( \sin(x) \), etc. can be represented well near \( x = 0 \) by Taylor series expansions. If a series expansion converges rapidly, taking a few terms in the series may produce good polynomial approximations. Assuming such a procedure is plausible, one approach to polynomial approximation is to take some data points, say \( (x_i, y_i), \ 1 \leq i \leq n \) and determine the polynomial of degree \( n - 1 \) passing through those points. It appears reasonable that using evenly spaced data is appropriate and that increasing the number of polynomial terms should improve the accuracy of the approximating function. However, it
has actually been shown that a polynomial through points on a function \( y(x) \), where the \( x \) values are evenly spaced, often gives approximations which are not smooth between the data points and tend to oscillate at the ends of the interpolating interval [20]. Attempting to reduce the oscillation by increasing the polynomial order makes matters worse. Surprisingly, a special set of unevenly spaced points bunching data near the interval ends according to

\[
x_j = \left( \frac{a + b}{2} + \frac{(a - b)}{2} \cos \left( \frac{\pi(j - 1)}{n} \right) \right), \quad 1 \leq j \leq n
\]

for the interval \( a \leq x \leq b \) turns out to be preferable. This formula defines what are called the Chebyshev points optimally chosen in the sense described by Conte and de Boor [20].

The program below employs MATLAB functions `polyfit`, `polyval`, and `spline` to produce interpolated approximations to the known function \( 1/(1+x^2) \). The example illustrates how strongly the spacing of the data points for polynomial interpolation can influence results, and also shows that a spline interpolation can be a better choice than high order polynomials. A least square fit polynomial of degree \( n \) through data points defined by vectors \((x_d, y_d)\) is given by

\[
p(x) = \text{polyval}(\text{polyfit}(x_d, y_d, n), x).
\]

When the polynomial order is one less than the number of data points, the polynomial passes through the data points exactly, but it may still produce unsatisfactory interpolation because of large oscillations between the data points. A preferable approximation is often provided by function `spline` giving a piecewise cubic curve with continuous first and second derivatives. The program passes polynomials of degree ten through a set of evenly spaced points and a set of Chebyshev points lying in the range \(-4 \leq x \leq 4\). A spline curve passed through the equidistant points is constructed in addition to a least square polynomial fit employing 501 points. Two graphs are created which show results for \( x \geq 0 \). Only results for positive \( x \) were plotted to provide more contrast between different interpolation results. Figure 2.1 plots the exact function, the spline curve, and the polynomial through the equidistant data. The polynomial is clearly an unsatisfactory approximation, whereas the spline appears to deviate imperceptibly from the exact function. By using the interactive `zoom` feature in MATLAB graphics, parts of the graph can be magnified so the difference between the spline and exact results is clearly visible. Figure 2.2 compares the exact function with a polynomial employing the Chebyshev points. This result is much better than what is produced with equidistant data. An approximation generated from a least square fit polynomial and 501 data points is also shown. This curve fits the exact function unpredictably and significantly misses the desired values at \( x = 0 \) and \( x = \pm 4 \). While general conclusions about interpolation should not be drawn from this simple example, it certainly implies that high order polynomial interpolation over a large range of the independent variable should be used cautiously.

The graphics functions used in the program include `plot`, `title`, `xlabel`, `ylabel`, and `legend`. Some other features of the program are summarized in the table preceding the code listing.
Figure 2.1: Spline and Polynomial Interpolation Using Equidistant Points
Figure 2.2: Interpolation Using Chebyshev Points and 501 Least Square Points
Program polyplot

1: function polyplot
2: % Example: polyplot
3: % --------------------------------------
4: % This program illustrates polynomial and
5: % spline interpolation methods applied to
6: % approximate the function \(1/(1+x^2)\).
7: %
8: % User inline functions used:
9: %  cbp, Ylsq, yexact
10: %
11: % Function for Chebyshev data points
12: cbp=inline(['(a+b)/2+(a-b)/2*cos(pi/n*','
13:    '(1/2:n))'],'a','b','n');
14: %
15: % Polynomial of degree n to least square fit
16: % data points in vectors xd,yd
17: Ylsq=inline('polyval(polyfit(xd,yd,n),x)',
18:      'xd','yd','n','x');
19: %
20: % Function to be approximated by polynomials
21: yexact=inline('1./(1+abs(x).^p)', 'p', 'x');
22: %
23: % Set data parameters. Functions linspace and
24: % cbp generate data with even and Chebyshev
25: % spacing
26: n=10; nd=n+1; a=-4; b=4; p=2;
27: xeven=linspace(a,b,nd); yeven=yexact(p,xeven);
28: xcbp=cbp(a,b,nd); ycbp=yexact(p,xcbp);
29: nlsq=501; % Number of least square points
30: xlsq=linspace(a,b,nlsq); ylsq=yexact(p,xlsq);
31: % Compute interpolated functions for plotting

<table>
<thead>
<tr>
<th>Line</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,17,21</td>
<td>several inline functions are defined</td>
</tr>
<tr>
<td>27</td>
<td>function linspace generates vector of equidistant points</td>
</tr>
<tr>
<td>27,28,34-37</td>
<td>inline functions called</td>
</tr>
<tr>
<td>38</td>
<td>intrinsic spline function is used</td>
</tr>
<tr>
<td>45,57</td>
<td>graph legends created</td>
</tr>
<tr>
<td>52,64</td>
<td>graph images saved to files</td>
</tr>
</tbody>
</table>
Conformal Mapping Example

This example involves analytic functions and conformal mapping. The complex function \( w(z) \) which maps \(|z| \leq 1\) onto the interior of a square of side length 2 can be written in power series form as

\[
w(z) = \sum_{k=0}^{\infty} b_k z^{4k+1}
\]
where
\[ b_k = c \left( \frac{(-1)^k \frac{1}{2} k}{k!(4k+1)} \right), \quad \sum_{k=0}^{\infty} b_k = 1 \]

and \( c \) is a scaling coefficient chosen to make \( z = 1 \) map to \( w = 1 \) (see reference [75]). Truncating the series after some finite number of terms, say \( m \), produces an approximate square with rounded corners. Increasing \( m \) reduces the corner rounding but convergence is rather slow so that using even a thousand terms still gives perceptible inaccuracy. The purpose of the present exercise is to show how a polar coordinate region characterized by
\[ z = re^{i\theta}, \quad r_1 \leq r \leq r_2, \quad \theta_1 \leq \theta \leq \theta_2 \]
transforms and to exhibit an undistorted plot of the region produced in the \( w \)-plane. The exercise also emphasizes the utility of MATLAB for handling complex arithmetic and complex functions. The program has a short driver \texttt{squarrun} and a function \texttt{squarmap} which computes points in the \( w \) region and coefficients in the series expansion. Salient features of the program are summarized in the table below.

Results produced when \( 0.5 \leq r \leq 1 \) and \( 0 \leq \theta \leq 2\pi \) by a twenty-term series appear in Figure 2.3. The reader may find it interesting to run the program using several hundred terms and take \( 0 \leq \theta \leq \pi/2 \). The corner rounding remains noticeable even when \( m = 1000 \) is used. Later in this book we will visit the mapping problem again to show that a better approximation is obtainable using rational functions.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Line</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>squarrun</td>
<td>20-41</td>
<td>Functions \texttt{input}, \texttt{disp}, \texttt{fprintf}, and \texttt{read} are used to input data interactively. Several different methods of printing were used for purposes of illustration rather than necessity. Function \texttt{squarmap} generates results. Function \texttt{genprint} is a system dependent routine which is used to create plot files for later printing.</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>Functions \texttt{linspace} and \texttt{ones} are used to generate points in the ( z )-plane.</td>
</tr>
<tr>
<td></td>
<td>49</td>
<td>Series coefficients are computed using \texttt{cumprod} and the mapping is evaluated using \texttt{polyval} with a matrix argument.</td>
</tr>
<tr>
<td></td>
<td>31-33</td>
<td>Scale limits are calculated to allow an undistorted plot of the geometry. Use is made of MATLAB functions \texttt{real} and \texttt{imag}.</td>
</tr>
<tr>
<td></td>
<td>43-45</td>
<td>Loops are executed to plot the circumferential lines first and the radial lines second. Function which determines limits for a square or cube shaped region.</td>
</tr>
<tr>
<td></td>
<td>48-51</td>
<td></td>
</tr>
<tr>
<td></td>
<td>57-73</td>
<td></td>
</tr>
</tbody>
</table>

© 2003 by CRC Press LLC
Figure 2.3: Mapping of a Square Using a 20-Term Polynomial
MATLAB Example

Program squarrun

function squarrun
% Example: squarrun
% ~~~~~~~~~~~~~~~~~~~
%
% Driver program to plot the mapping of a
% circular disk onto the interior of a square
% by the Schwarz-Christoffel transformation.
%
% User m functions required:
% squarmap, inputv, cubrange
%
% Illustrate use of the functions input and
% inputv to interactively read one or several
% data items on the same line

fprintf('
CONFORMAL MAPPING OF A SQUARE ')
fprintf('BY USE OF A
TRUNCATED SCHWARZ-CHRISTOFFEL SERIES

')
fprintf('
Enter the number of series terms used 
')
m=input('(try 20)? ');

% Illustrate use of the function disp
disp('')
str=['
Input the inner radius, outer ' ...
'radius and number of increments , ...' 
'\n(try .5,1,8)']
fprintf(str);

% Use function inputv to input several variables
[r1,r2,nr]=inputv;

% Use function fprintf to print more
% complicated heading
str=['
Input the starting value of , ...' 
'theta, the final value of theta \n', ... 
'and the number of theta increments ' ... 
'(the angles are in degrees) ' ...

© 2003 by CRC Press LLC
fprintf(str); [t1,t2,nt]=inputv;

% Call function squarmap to make the plot
hold off; clf;
[w,b]=squarmap(m,r1,r2,nr,t1,t2,nt+1);

% Save the plot
% print -deps squarplt
disp(' '); disp('All Done');

function [w,b]=squarmap(m,r1,r2,nr,t1,t2,nt)
% This function evaluates the conformal mapping
% produced by the Schwarz-Christoffel
% transformation w(z) mapping abs(z)<=1 inside
% a square having a side length of two. The
% transformation is approximated in series form
% which converges very slowly near the corners.
% m - number of series terms used
% r1,r2,nr - abs(z) varies from r1 to r2 in
% nr steps
% t1,t2,nt - arg(z) varies from t1 to t2 in
% nt steps (t1 and t2 are measured
% in degrees)
% w - points approximating the square
% b - coefficients in the truncated
% series expansion which has the
% form
% 
% w(z)=sum({j=1:m},b(j)*z*(4*j-3))
% User m functions called: cubrange

% Generate polar coordinate grid points for the
% map. Function linspace generates vectors
% with equally spaced components.
r=linspace(r1,r2,nr);
t=pi/180*linspace(t1,t2,nt);
z=(r*ones(1,nt)).*(ones(nr,1)*exp(i*t));

% Use high point resolution for the outer contour
touter=pi/180*linspace(t1,t2,10*nt);
zouter=r2*exp(i*touter);

% Compute the series coefficients and evaluate the series
k=1:m-1;
b=cumprod([1, -(k-.75).*(k-.5)./(k.*(k+.25))]);
b=b/sum(b); w=z.*polyval(b(m:-1:1),z.^4);
wouter=zouter.*polyval(b(m:-1:1),zouter.^4);

% Determine square window limits for plotting
uu=real([w(:);wouter(:)]);
vv=imag([w(:);wouter(:)]);
rng=cubrange([uu,vv],1.1);
axis('square'); axis(rng); hold on

% Plot orthogonal grid lines which represent the mapping of circles and radial lines
x=real(w); y=imag(w);
xo=real(wouter); yo=imag(wouter);
plot(x,y,'-k',x(1:end-1,:),y(1:end-1,:)',...
'k',xo,yo,'k')

% Add a title and axis labels
title(['Mapping of a Square Using a ',...
num2str(m),'-term Polynomial'])
xlabel('x axis'); ylabel('y axis')
figure(gcf); hold off;

function range=cubrange(xyz,ovrsiz)
%
% This function determines limits for a square or cube shaped region for plotting data values in the columns of array xyz to an undistorted scale

% xyz - a matrix of the form [x,y] or [x,y,z]
% where x,y,z are vectors of coordinate points
% ovrsiz - a scale factor for increasing the window size. This parameter is set to one if only one input is given.
% range - a vector used by function axis to set window limits to plot x,y,z points undistorted. This vector has the form
% [xmin,xmax,ymin,ymax] when xyz has only two columns or the form
% [xmin,xmax,ymin,ymax,zmin,zmax] when xyz has three columns.
%
% User m functions called: none
%-------------------------------------------------------------
if nargin==1, ovrsiz=1; end
pmin=min(xyz); pmax=max(xyz); pm=(pmin+pmax)/2;
pd=max(ovrsiz/2*(pmax-pmin));
if length(pmin)==2
range=pm([1,1,2,2])+pd*[-1,1,-1,1];
else
range=pm([1 1 2 2 3 3])+pd*[-1,1,-1,1,-1,1];
end
%====================================================================
% function varargout=inputv(prompt)
% See Appendix B

2.5 Nonlinear Motion of a Damped Pendulum

Motion of a simple pendulum is one of the most familiar dynamics examples studied in physics. The governing equation of motion can be satisfactorily linearized for small oscillations about the vertical equilibrium position, whereas nonlinear effects become important for large deflections. For small deflections, the analysis leads to a constant coefficient linear differential equation. Solving the general case requires elliptic functions seldom encountered in routine engineering practice. Nevertheless, the pendulum equation can be handled very well for general cases by numerical integration.
Suppose a bar of negligible weight is hinged at one end and has a particle of mass $m$ attached to the other end. The bar has length $l$ and the deflection from the vertical static equilibrium position is called $\theta$. Assuming that the applied forces consist of the particle weight and a viscous drag force proportional to the particle velocity, the equation of motion is found to be

$$\theta''(\tau) + \frac{c}{m} \theta'(t) + \frac{g}{l} \sin(\theta) = 0$$

where $\tau$ is time, $c$ is a viscous damping coefficient, and $g$ is the gravity constant. Introducing dimensionless time, $t$, such that $\tau = \sqrt{l/g} t$ gives

$$\theta''(t) + 2\varsigma \theta'(t) + \sin(\theta) = 0$$

where $\varsigma = \sqrt{l/g c/(2m)}$ is called the damping factor. When $\theta$ is small enough for $\sin(\theta)$ to be approximated well by $\theta$, then a constant coefficient linear equation solvable by elementary means is obtained. In the general situation, a solution can still be obtained numerically without resorting to higher transcendental functions. If we use $\varsigma = 0.10$ for illustrative purposes, and let

$$z = [\theta(t); \theta'(t)]$$

then the original differential equation expressed in first order matrix form is

$$z'(t) = [z(2); -0.2z(2) - \sin(z(1))].$$

An inline function suitable for use by the ode45 integrator in MATLAB is simply $\text{zdot} = \text{inline}('[z(2); -0.2*z(2)-\sin(z(1))]','t','z')$.

A program was written to integrate the pendulum equation when the angular velocity $\omega_0$ for $\theta = 0$ is specified. For the undamped case, it is not hard to show that a starting angular velocity exceeding 2 is sufficient to push the pendulum over the top, but the pendulum will fall back for values smaller than two. For the amount of viscous damping chosen here, a value of about $\omega_0 = 2.42$ barely pushes the pendulum over the top, whereas the top is not reached for $\omega_0 = 2.41$. These cases vividly illustrate that, for a nonlinear system, small changes in initial conditions can sometimes produce very large changes in the response of the system.

In the computer program that follows, a driver function runpen controls input, calls the differential equation solver ode45, as well as a function animpen which plots $\theta$ versus $t$, and performs animation by drawing successive positions of the pendulum. Because the animation routine is very simple and requires little knowledge of MATLAB graphics, the images and the titles flicker somewhat. This becomes particularly evident unless the graph axes are left off. A better routine using more detailed graphics commands to eliminate the flicker problem is presented in Article 2.7 on wave motion in a string. The current program permits interactive input repeatedly specifying the initial angular velocity, or two illustrative data cases can be run by executing the command runpen(1). The differential equation for the problem is defined as function zdot on lines 26 and 27. This equation is integrated numerically.
by calls to function **ode45** on lines 59, 75, and 80. Integration tolerance values were chosen at line 30, and a time span for the simulation is defined interactively at lines 46 and 47. Function **penanim(t,th,titl,tim)** plots theta versus time and animates the system response by computing the range of (x,y) values, fixing the window size to prevent distortion, and sequentially plotting positions of the pendulum to show the motion history. The output results produced by **runpen(1)** are shown below for reference.
Figure 2.5: Partial Motion Trace for Pendulum Pushed Over the Top
Figure 2.6: Angular Displacement versus Time for Pendulum Almost Pushed Over the Top
ALMOST OVER THE TOP FOR $W_0=2.41$

Figure 2.7: Partial Motion Trace for Pendulum Almost Pushed Over the Top
function pendulum(rundemo)
% pendulum(rundemo)
% This example analyzes damped oscillations of
% a simple pendulum and animates the motion.
% The governing second order differential
% equation is
% \[ \theta''(t) + 0.2\theta'(t) + \sin(\theta) = 0 \]
% Type pendulum with no argument for inter-
% active input. Type pendulum(1) to run two
% example problems
% The equation of motion can be written as
% two first order equations:
% \[ \theta'(t) = w; \quad w'(t) = -0.2w - \sin(\theta) \]
% Letting \( z = [\theta; w] \), then
% \[ z'(t) = [z(2); -0.2z(2) - \sin(z(1))] \]

disp(' ') % DAMPED PENDULUM MOTION DESCRIBED BY
disp(' \theta''(t) + 0.2\theta'(t) + \sin(\theta) = 0')

% Create an inline function defining the
% differential equation in matrix form
zdot=inline( ...
'[z(2);-0.2*z(2)-sin(z(1))]', 't', 'z');

% Set ode45 integration tolerances
ops=odeset('reltol',1e-5,'abstol',1e-5);

% Interactively input angular velocity repeatedly
if nargin==0
  while 1, close, disp(' ') disp('Select the angular velocity at the lowest')
    disp('point. Values of 2.42 or greater push the')
    disp('the pendulum over the top. Input zero to stop.')
    w0=input('w0 = ? > ');

  © 2003 by CRC Press LLC
if isempty(w0) | w0==0
    disp( ' ' ), disp('All Done'), disp(' '), return
end

t=input([ 'Input a vector of time values ',
    '(Try 0:.1:30) > ? ']);

disp( ' ')
titl=input('Input a title for the graphs : ',
    's');
disp( ' ' ), disp(
    'Input 1 to leave images of all positions shown'
    'in the animation, otherwise input 0 > ? ');

% Specify the initial conditions and solve the
% differential equation using ode45
theta0=0; z0=[theta0,w0];
[t,th]=ode45(zdot,t,z0,ops);

% Animate the motion
animpen(t,th(:,1),titl,.05,trac)
end

% Run two typical data cases
else

% Choose time limits for the solution
*tmax=30; n=351; t=linspace(0,tmax,n);

% Specify the initial conditions and solve the
differential equation using ode45
theta0=0; z0=[theta0;w0];
[t,th]=ode45(zdot,t,z0,ops);

% Animate the motion
animpen(t,th(:,1),titl,.05, trac)
end

© 2003 by CRC Press LLC
function animpen(t,th,titl,tim,trac)
%
% This function plots theta versus t and animates
% the pendulum motion
%
% t - time vector for the solution
% th - angular deflection values defining the
% pendulum positions
% titl - a title shown on the graphs
% tim - a time delay between successive steps of
% the animation. This is used to slow down
% the animation on fast computers
% trac - 1 if successive positions plotted in the
% animation are retained on the screen, 0
% if each image is erased after it is
%
if nargin<5, trac=0; end; if nargin<4, tim=.05; end;
if nargin<3, titl=''; end

% Plot the angular deflection
plot(t,180/pi*th(:,1),'k'), xlabel('time'),
ylabel('angular deflection (degrees)'), title(titl)
grid on, shg, disp(' '),
disp('Press return to see the animation'), pause
%
nt=length(th); z=zeros(nt,1);
x=[z,sin(th)]; y=[z,-cos(th)];
hold off, close
if trac
axis([-1,1,-1,1]), axis square, axis off, hold on
end
for j=1:nt
X=x(j,:); Y=y(j,:);
plot(X,Y,'k-',X(2),Y(2),'ko','markersize',12)
if ~trac
axis([-1,1,-1,1]), axis square, axis off
end
title(titl), drawnow, shg
if tim>0, pause(tim), end
% if trac==1, print -deps pentrace, end
pause(1),hold off

2.6 A Linear Vibration Model

Important aspects of linear vibration theory are illustrated by the one-dimensional
motion of a mass subjected to an elastic restoring force, a viscous damping force
proportional to the velocity, and a harmonically varying forcing function. The related
differential equation is
\[ m \ddot{x}(t) + c \dot{x}(t) + k x(t) = f_1 \cos(\omega t) + f_2 \sin(\omega t) = \text{real}((f_1 - i f_2) \exp(i \omega t)) \]
with initial conditions of \( x(0) = x_0 \) and \( x'(0) = v_0 \). The general solution is the
sum of a particular solution to account for the forcing function, and a homogeneous
solution corresponding to a zero right hand side. The initial conditions are applied
to the sum of the two solution components. The particular solution is given by
\[ X(t) = \text{real}(F \exp(i \omega t)) \]
with
\[ F = (f_1 - i f_2)/(k - m \omega^2 + i c \omega). \]
The initial conditions given by this particular solution are
\[ X(0) = \text{real}(F) \]
and
\[ X'(0) = \text{real}(i \omega F). \]
The characteristic equation for the homogeneous equation is
\[ m s^2 + c s + k = 0 \]
which has roots
\[ s_1 = (-c + r)/(2m), \quad s_2 = (-c - r)/(2m), \quad r = \sqrt{c^2 - 4mk}. \]
Then the homogeneous solution has the form
\[ u(t) = d(1) \exp(s_1 t) + d(2) \exp(s_2 t) \]
where
\[ d = [1, 1; s_1, s_2] \\ [x_0 - X(0); v_0 - X'(0)] \]
and the complete solution is
\[ x(t) = u(t) + X(t). \]
A couple of special cases arise. The first corresponds to zero damping and a forcing function matching the undamped natural frequency, i.e.,

\[ c = 0, \quad \omega = \sqrt{k/m}. \]

This case can be avoided by including a tiny amount of damping to make \( c = 2\sqrt{mk}/10^6 \). The second case happens when the characteristic roots are equal. This is remedied by perturbing the value of \( c \) to \((1 + 10^{-6})\) times \( c \). Such small changes in a system model where realistic physical parameters are only known approximately will not affect the final results significantly.

In practice, enough damping often exists in the system to make the homogeneous solution components decay rapidly so the total solution approaches the particular solution with the displacement having the same frequency as the forcing function but out of phase with that force. To illustrate this effect, a program was written to solve the given differential equation, plot \( x(t) \), and show an animation for a block connected to a wall with a spring and sliding on a surface with viscous damping resistance. Applying the oscillating force of varying magnitude on the block helps illustrate how the homogeneous solution dies out and the displacement settles into a constant phase shift relative to the driving force.

The following program either reads data interactively or runs a default data example. The solution procedure described above is implemented in function `smdsolve`. For arbitrary values of the system parameters, \( x(t) \) is plotted and a simple animation scheme is used to plot the block, a spring, and the applied force throughout the time history. Figure 2.8 shows \( x(t) \) for the default data case. The input data values for this case use

\[ [m, c, k, f_1, f_2, w, x_0, v_0, t_{max}, nt] \leftrightarrow [1, 3, 1, 1, 0, 2, 0, 2, 30, 250]. \]

Note that near \( t = 11 \), the transient and forced solution components interact so that the block almost pauses momentarily. However, the solution then quickly approaches the steady state. Figure 2.9 shows the final position of the mass and the applied force at the end of the chosen motion cycle.
Figure 2.8: Plot of $x(t)$ for a Linear Harmonic Oscillator

Figure 2.9: Block Sliding On a Plane with Viscous Damping
Program smdplot

function [t,X,m,c,k,f1,f2,w,x0,v0]= smdplot(example)
  
% This function plots the response and animates the
% motion of a damped linear harmonic oscillator
% characterized by the differential equation
% m*x''+c*x'+k*x=f1*cos(w*t)+f2*sin(w*t)
% with initial conditions x(0)=x0, x'(0)=v0.
% The animation depicts forced motion of a block
% attached to a wall by a spring. The block
% slides on a horizontal plane which provides
% viscous damping.

% example - Omit this parameter for interactive input.
% Use smdplot(1) to run a sample problem.
% t,X      - time vector and displacement response
% m,c,k    - mass, damping coefficient,
%            spring stiffness constant
% f1,f2,w  - force components and forcing frequency
% x0,v0    - initial position and velocity
%
% User m functions called: spring smdsolve inputv
% -----------------------------------------------

pltsave=0; disp(' '), disp(.......'
  SOLUTION OF '), disp(....'
'M*X''+C*X''+K*X=F1*COS(W*T)+F2*SIN(W*T)')
  disp(....'
  WITH ANIMATION OF THE RESPONSE')
  disp(' ')

% Example data used when nargin > 0
if nargin > 0
  m=1; c=.3; k=1; f1=1; f2=0; w=2; x0=0; v0=2;
  tmax=25; nt=250;
else % Interactive data input
  [m,c,k]=inputv(...
    'Input m, c, k (try 1, .3, 1) >> ? ');
  [f1,f2,w]=inputv(...

  
© 2003 by CRC Press LLC
'Input f1, f2, w (try 1, 0, 2) >> ? ');

[x0,v0]=inputv(...
'Input x0, v0 (try 0, 2) >> ? ');

tmax,nt]=inputv(...
'Input tmax, nt (try 30, 250) >> ? ');

end

t=linspace(0,tmax,nt);
X=smdsolve(m,c,k,f1,f2,w,x0,v0,t);

% Plot the displacement versus time
plot(t,X,'k'), xlabel('time')
ylabel('displacement'), title(...
'FORCED RESPONSE OF A DAMPED HARMONIC OSCILLATOR')
grid on, shg, disp(' ')
if pltsave, print -deps smdplotxvst; end
disp('Press return for response animation')
pause

% Add a block and a spring to the displacement
xmx=max(abs(X)); X=X/1.1/xmx;
xb=[0,0,1,1,0,0]/2; yb=[0,-1,-1,1,1,0]/2;

% Make an arrow tip
d=.08; h=.05;
xtip=[0,-d,-d,0]; ytip=[0,0,0,h,-h,0];

% Add a spring and a block to the response
[xs,ys]=spring; nm=length(X); ns=length(xs);
nb=length(xb); x=zeros(nm,ns+nb); y=[ys,yb];
for j=1:nn, x(j,:)=[-1+(1+X(j))*xs,X(j)+xb]; end
xmin=min(x(:)); xmax=max(x(:)); d=xmax-xmin;
xmax=xmin+1.1*d; r=[xmin,xmax,-2,2];
rx=r([1 1 2]); ry=[.5,-.5,-.5]; close;

% Plot the motion
for j=1:nn
  % Compute and scale the applied force
  f=f1*cos(w*t(j))+f2*sin(w*t(j));
  f=.5*f; fa=abs(f); sf=sign(f);
  xj=x(j,:); xmaxj=max(xj);
  if sf>0
    xforc=xmaxj+[0,fa,fa+xtip];
else
  xforc=xmaxj+[fa,0,-xtip];
end

% Plot the spring, block, and force
% plot(xj,y,rx,ry,'k',xforc,ytip,'r')
% plot(xj,y,'k-',rx,ry,'k-',xforc,ytip,'k-')
plot(xj,y,'k-',xforc,ytip,'k-',...
  rx,ry,'k-',',linewidth',1)
title('FORCED MOTION WITH DAMPING')
xlabel('FORCED MOTION WITH DAMPING')
axis(r), axis('off'), drawnow
figure(gcf), pause(.05)
end
if pltsave, print -deps smdplotanim; end
disp(' '), disp('All Done')

%====================================
%function [x,y] = spring(len,ht)
% This function generates a set of points
% defining a spring

if nargin==0, len=1; ht=.125; end
x=[0,.5,linspace(1,12,10),13.5,12]
y=[ones(1,5);-ones(1,5)];
y=[0;0;y(:);0;0]'; y=ht/2/max(y)*y;
x=len/max(x)*x;

%====================================
%function [x,v]=smdsolve(m,c,k,f1,f2,w,x0,v0,t)
% This function solves the differential equation
% m*x''(t)+c*x'(t)+k*x(t)=f1*cos(w*t)+f2*sin(w*t)
% with x(0)=x0 and x'(0)=v0
% m,c,k - mass, damping and stiffness coefficients
% f1,f2 - magnitudes of cosine and sine terms in
% the forcing function
% w - frequency of the forcing function
% t - vector of times to evaluate the solution
% x,v - computed position and velocity vectors
% If the system is undamped and resonance will 
% occur, add a little damping
if c==0 & w==wn; c=ccrit/1e6; end;

% If damping is critical, modify the damping
% very slightly to avoid repeated roots
if c==ccrit; c=c*(1+1e-6); end

% Forced response solution
a=(f1-i*f2)/(k-m*w^2+i*c*w);
X0=real(a); V0=real(i*w*a);
X=real(a*exp(i*w*t)); V=real(i*w*a*exp(i*w*t));

% Homogeneous solution
r=sqrt(c^2-4*m*k);
s1=(-c+r)/(2*m); s2=(-c-r)/(2*m);
p=[1,1;s1,s2]\[x0-X0;v0-V0];

% Total solution satisfying the initial conditions
x=X+real(p(1)*exp(s1*t)+p(2)*exp(s2*t));
v=V+real(p(1)*s1*exp(s1*t)+p(2)*s2*exp(s2*t));

%====================================

% function [a1,a2,...,a_nargout]=inputv(prompt)
% See Appendix B

2.7 Example of Waves in an Elastic String

One-dimensional wave propagation is illustrated well by the response of a tightly stretched string of finite length released from rest with given initial deflection. The transverse deflection $y(x, t)$ satisfies the wave equation

$$a^2 y_{xx} = y_{tt}$$

and the general solution for an infinite length string, released from rest, is given by

$$y(x, t) = [F(x - at) + F(x + at)]/2$$

where $F(x)$ is the initial deflection for $-\infty < x < \infty$. The physical interpretation for this equation is that the initial deflection splits in two parts translating at speed $a$, with one part moving to the right and the other moving to the left. The translating
wave solution can be adapted to handle a string of finite length \( l \) by requiring

\[
y(0, t) = y(l, t) = 0.
\]

These end conditions, along with initial deflection \( f(x) \) (defining \( F(x) \) between 0 and \( l \)), are sufficient to continue the solution outside the original interval. We write the initial condition for the finite length string as

\[
y(x, 0) = f(x), \quad 0 < x < l.
\]

To satisfy the end conditions, \( F(x) \) must be an odd-valued function of period \( 2l \).

Introducing a function \( g(x) \) such that

\[
g(x) = f(x), \quad 0 \leq x \leq l
\]

and

\[
g(x) = -f(2l - x), \quad l < x \leq 2l
\]

leads to

\[
F(x) = \text{sign}(x)g(\text{rem}(\text{abs}(x), 2l))
\]

where the desired periodicity is achieved using the MATLAB remainder function, \text{rem}. This same problem can also be solved using a Fourier sine series (see chapter 9). For the present we concentrate on the solution just obtained.

A program was written to implement the translating wave solution when \( f(x) \) is a piecewise linear function computed using \text{interp1}. The system behavior can be examined from three different aspects. 1) The solution \( y(x, t) \) for a range of \( x \) and \( t \) values describes a surface. 2) The deflection curve at a particular time \( t_0 \) is expressed as \( y(x, t_0), \ 0 < x < l \). 3) The motion history at a particular point \( x_0 \) is \( y(x_0, t), \ t \geq 0 \). The nature of \( F(x) \) implies that the motion has a period of \( 2l/a \). Waves striking the boundary are reflected in inverted form so that for any time \( y(x, t + l/a) = -y(x, t) \). The character of the motion is typified by the default data case the program uses to define a triangular initial deflection pattern where

\[
a = 1, \ l = 1, \ \text{xd} = [0, 0.33, 0.5, 0.67, 1], \ \text{yd} = [0, 0, -1, 0, 0].
\]

The program reads the wave speed, the string length, and data points specifying the initial deflection. The solution is evaluated for a range of \( x, t \) values. The function \text{plot3} was used to create Figure 2.10, which is a three-dimensional plot of traces of the string deflection for a sequence of times. Figure 2.11 shows the string position at \( t = 0.33 \). Figure 2.12 plots the deflection history at position \( x = 0.25 \). Finally, a function to animate the solution over two motion cycles illustrates how the initial deflection splits, translates, and reflects from the boundaries. In an attempt to illustrate successive positions assumed in the animation, traces of the motion for a brief period are shown in Figure 2.13.
Figure 2.10: String Position as a Function of Position and Time

Figure 2.11: String Deflection when $t = 0.33$
Figure 2.12: Motion at Quarterpoint of the String

Figure 2.13: Motion Trace Over a Short Time Interval

© 2003 by CRC Press LLC
MATLAB Example

Program strngrun

function strngrun(rundemo)
    
    % strngrun(rundemo)
    % ~~~~~~~~~~~~~~~~
    % This function illustrates propagation of
    % waves in a tightly stretched string having
    % given initial deflection. Calling strngrun
    % with no input argument causes data to be
    % read interactively. Otherwise, strngrun(1)
    % executes a sample data case.
    %
    % User m functions called: strngwav animate
    
    pltsav=0; % flag to save or not save graphs

    disp('')
    disp('WAVE PROPAGATION IN A STRING'), disp(' ')
    if nargin==0 % Input data interactively
        [a,len]=inputv(['Input wave speed (a) and ',...
            'string length (len) > ? ']);
        disp(' ')
        disp(['Enter the number of interior ',...
            'data points (the fixed')] )
    disp(['Enter ',num2str(n),...
        ' sets of x,y to specify interior
        (one pair per line)'])
        for j=2:n+1,[xd(j),yd(j)]=inputv; end;
        disp('')
        disp(['Input tmax and the number of time steps'])
        [tmax,nt]=inputv('(Try len/a and 40) > ? ');
        disp('')
disp('Specify position \(x=x_0\) where the time history is to be evaluated (try len/4) > ? ');
disp('')
disp('Specify time \(t=t_0\) when the deflection curve is to be plotted > ? ');
disp('')
titl=input('Input a graph title > ?', 's');

else \% Example for triangular initial deflection
    a=1; len=1; tmax=len/a; nt=40;
    xd=[0,.33,.5,.67,1]*len; yd=[0,0,-1,0,0];
\% Different example for a truncated sine curve
\% xd=linspace(0,len,351); yd=sin(3*pi/len*xd);
\% k=find(yd<=0); xd=xd(k); yd=yd(k);
    x0=0.25*len; t0=0.33*len/a;
    titl='TRANSLATING WAVE OVER HALF A PERIOD';
end

nx=80; x=0:len/nx:len; t=0:tmax/nt:tmax;

h=max(abs(yd)); xpplot=linspace(0,len,201);
tplot=linspace(0,max(t),251)';

[Y,X,T]=strngwav(xd,yd,x,t,len,a);
plot3(X',T',Y','k'); xlabel('x axis')
ylabel('time'), zlabel('y(x,t)'), title(titl)
if pltsav, print(gcf,'-deps','strngplot3'); end
drawnow, shg, disp('')
disp('Press return to see the deflection')
disp(['when t = ',num2str(t0)]), pause

[yt0,xx,tt]=strngwav(xd,yd,xpplot,t0,len,a);
close; plot(xx(:,),yt0(:,),'k')
xlabel('x axis'), ylabel('y(x,t0)')
title(['DEFLECTION WHEN T = ',num2str(t0)])
axis([min(xx),max(xx),-h,h])
if pltsav, print(gcf,'-deps','strngyxt0'); end
drawnow, shg

disp(' Press return to see the deflection history')

\© 2003 by CRC Press LLC
disp(['at x = ',num2str(x0)]), pause

yx0=strngwav(xd,yd,x0,tplot,len,a);
plot(tplot,yx0,'k')
xlabel('time'), ylabel('y(x0,t)')
title(['DEFLECTION HISTORY AT X = ',num2str(x0)])
axis([0,max(t),-h,h])
if pltsav, print(gcf,'-deps','strngyx0t'); end
drawnow, shg

disp('Press return to see the animation')
disp('over two periods of motion'), pause
x=linspace(0,len,101); t=linspace(0,4*len/a,121);
[Y,X,T]=strngwav(xd,yd,x,t,len,a);
titl='MOTION OVER TWO PERIODS';
animate(X(1,:),Y',titl,.1), pause(2)
if pltsav, print(gcf,'-deps','strnganim'); end

disp(' '), disp('All Done')

%===============================================
% function [Y,X,T]=strngwav(xd,yd,x,t,len,a)
% [Y,X,T]=strngwav(xd,yd,x,t,len,a)
% This function computes the dynamic response of
% a tightly stretched string released from rest
% with a piecewise linear initial deflection. The
% string ends are fixed.
% xd,yd - data vectors defining the initial
% function. xd values should be increasing
% and lie between 0 and len
% x,t - position and time vectors for which the
% solution is evaluated
% len,a - string length and wave speed
if nargin<6, a=1; end; if nargin <5, len=1; end
xd=xd(:); yd=yd(:); p=2*len;

© 2003 by CRC Press LLC
% If end values are not zero, add these points
if xd(end)~=len, xd=[xd;len]; yd=[yd;0]; end
if xd(1)~=0, xd=[0;xd]; yd=[0;yd]; end
nd=length(xd);

% Eliminate any repeated abscissa values
k=find(diff(xd)==0); tiny=len/1e6;
if length(k)>0, xd(k)=xd(k)+tiny; end

% Extend the data definition for len < x < 2*len
xd=[xd;p-xd(nd-1:-1:1)]; yd=[yd;-yd(nd-1:-1:1)];
[X,T]=meshgrid(x,t); xp=X+a*T; xm=X-a*T;
shape=size(xp); xp=xp(:); xm=xm(:);

% Compute the general solution for a piecewise
% linear initial deflection
Y=(sign(xp).*interp1(xd,yd,rem(abs(xp),p),...'
  linear','extrap')+sign(xm).*interp1(xd,yd,...
  rem(abs(xm),p),'linear','extrap'))/2;
Y=reshape(Y,shape);

function animate(x,y,titl,tim,trace)

% animate(x,y,titl,tim,trace)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function performs animation of a 2D curve
% x,y - arrays with columns containing curve positions
% for successive times. x can also be a single
% vector if x values do not change. The animation
% is done by plotting (x(:,j),y(:,j)) for
% j=1:size(y,2).
% titl - title for the graph
% tim - the time in seconds between successive plots

if nargin<5, trace=0; else, trace=1; end;
if nargin<4, tim=.05; end
if nargin<3, trac=''; end; [np,nt]=size(y);
if min(size(x))==1, j=ones(1,nt); x=x(:);
else, j=1:nt; end; ax=newplot;
if trace, XOR='none'; else, XOR='xor'; end
r=[min(x(:)),max(x(:)),min(y(:)),max(y(:))];
axis('equal') % Needed for an undistorted plot
axis(r), % axis('off')
curve = line('color','k','linestyle','-',... 
'erase',XOR, 'xdata',[],'ydata',[]); 
xlabel('x axis'), ylabel('y axis'), title(titl) 
for k = 1:nt 
    set(curve,'xdata',x(:,j(k)),'ydata',y(:,k)) 
    if tim>0, pause(tim), end, drawnow, shg 
end 

%==================================================================
% function varargout=inputv(prompt) 
% See Appendix B

2.8 Properties of Curves and Surfaces

In this section some properties of space curves and surfaces are studied. Examples illustrating the graphics capabilities of MATLAB to describe three-dimensional geometries are given. Readers should also study the demo examples and intrinsic documentation on functions such as plot3, surf, and mesh to appreciate the wealth of plotting options available.

2.8.1 Curve Properties

A space curve is a one-dimensional region representable in parametric form as

\[ R(t) = i x(t) + j y(t) + k z(t), \quad a < t < b \]

where \( i, j, k \) are Cartesian base vectors, and \( t \) is a scalar parameter such as arc length \( s \) or time. At each point on the curve, differential properties naturally lead to a triad of orthonormal base vectors \( T, N, \) and \( B \) called the tangent, the principal normal, and the binormal. The normal vector points toward the center of curvature and the binormal is defined by \( T \times N \) to complete the triad. Coordinate planes associated with the triad are the normal plane containing \( N \) and \( B \), the tangent plane containing \( T \) and \( B \), and the osculating plane containing \( T \) and \( N \). Two other scalar properties of interest are the curvature \( \kappa \) (the reciprocal of the curvature radius) and the torsion \( \tau \), which quantifies the rate at which the triad twists about the direction of \( T \) as a generic point moves along the curve. When a curve is parameterized in terms of arc length \( s \), the five quantities just mentioned are related by the Frenet formulas [91] which are

\[
\frac{dT}{ds} = \kappa N, \quad \frac{dB}{ds} = -\tau N, \quad \frac{dN}{ds} = -\kappa T + \tau B.
\]

Since most curves are not easily parameterized in terms of arc length, more convenient formulas are needed for computing \( T, N, B, \kappa, \) and \( \tau \). All the desired quantities can be found in terms of \( R'(t), R''(t), \) and \( R'''(t) \). Among the five properties,
only torsion, $\tau$, depends on $R'''(t)$. The pertinent formulas are

\[ \hat{T} = \frac{R'(t)}{|R'(t)|}, \quad \hat{B} = \frac{R'(t) \times R''(t)}{|R'(t) \times R''(t)|} \]

\[ \hat{N} = \hat{B} \times \hat{T}, \quad \kappa = \frac{|R'(t) \times R''(t)|}{|R'(t)|^3} \]

and

\[ \tau = \frac{\hat{B} \cdot R'''(t)}{|R'(t) \times R''(t)|}. \]

When the independent variable $t$ means time we get

\[ V = \text{velocity} = \frac{dR}{dt} = \frac{ds}{dt} \frac{dR}{ds} = v\hat{T} \]

where $v$ is the magnitude of velocity called speed. Differentiating again leads to

\[ \frac{dV}{dt} = \text{acceleration} = \frac{dv}{dt} \hat{T} + \kappa v^2 \hat{N} \]

so the acceleration involves a tangential component with magnitude equal to the time rate of change of speed, and a normal component of magnitude $\kappa v^2$ directed toward the center of curvature. The torsion is only encountered when the time derivative of acceleration is considered. This is seldom of interest in Newtonian mechanics.

A function `crvprp3d` was written to evaluate $\hat{T}$, $\hat{N}$, $\hat{B}$, $\kappa$, and $\tau$ in terms of $R'(t)$, $R''(t)$, and $R'''(t)$. Another function `aspiral` applies `crvprp3d` to the curve described by

\[ R(t) = [(r_o + kt) \cos(t); (r_o + kt) \sin(t); ht] \]

where $t$ is the polar coordinate angle for cylindrical coordinates. Figure 2.14 depicts results generated from the default data set where

\[ r_o = 2\pi, \quad k = 1, \quad h = 2, \quad 2\pi \leq t \leq 8\pi, \]

with 101 data points being used. A cross section normal to the surface would produce a right angle describing the directions of the normal and binormal at a typical point. The spiral itself passes along the apex of the right angle. This surface illustrates how the intrinsic triad of base vectors changes position and direction as a point moves along the curve.

An additional function `crvprpsp` was written to test how well cubic spline interpolation approximates curve properties for the spiral. MATLAB provides function `spline` to connect data points by a piecewise cubic interpolation curve having continuous first and second derivatives [27]. This function utilizes other intrinsic functions\(^1\) such as `unmkpp`, `mkpp`, and `ppval`. Although basic MATLAB does not

\[^1\]These functions are included with MATLAB and are a subset of the more comprehensive Spline Toolbox also available from The MathWorks.
include functions for spline differentiation, this can be remedied by the short function *splined* which computes first and second derivatives of the interpolation curve defined by function *spline*. In our example using spline interpolation, approximation of $\tau$ was not obtained because a cubic spline only has its first two derivatives continuous. Approximations for $R'''(t)$ could have been generated by interpolating the computed values of $R'(t)$ and differentiating the results twice. That idea was not explored. To assess the accuracy of the spline interpolation, values for $\text{norm}(\hat{B} - \hat{B}_{\text{approx}})$ and $|(k - k_{\text{approx}})/k|$ were obtained at 101 sample points along the curve. Results depicted in Figure 2.15 show errors in the third decimal place except near the ends of the interpolation interval where a “not a knot” boundary condition is employed [27].
Figure 2.15: Error Plot
Program Output and Code

Program splinerr

function splinerr
    % Example: splinerr
    % ----------------------
    %
    % This program calculates the binormal and curvature error for a spiral space curve.
    %
    % User m functions called:
    % aspiral, crvprpsp, crvprp3d, cubrange, splined
    %---------------------------------------------
    clear; hold off; clf;
    [R,T,N,B,KAP]=aspiral; m=size(R,2);
    [r,t,n,b,k]=crvprpsp(R,m);
    disp(''); disp(...
    'Press [Enter] to show error curves'); pause
    errv=sqrt(sum((B-b).^2));
    errk=abs((KAP-k)./KAP); hold off; clf;
    semilogy(1:m,errv,'k-',1:m,errk,'k--');
    xlabel('point index'); ylabel('error measure');
    title('Error Plot');
    legend('Binormal error','Curvature error',3);
    figure(gcf); disp(' 
    disp('Press [Enter] to finish'); pause
    disp(' '), disp('All done'), disp(' ')
    %==============================================

function [R,T,N,B,kap,tau,arclen]= ...
    aspiral(r0,k,h,t)
% A figure showing the curve along with the
% osculating plane and the rectifying plane
% at each point is also drawn.

% r0, k, h - parameters which define the spiral
% t - a vector of parameter values at
% which the curve is evaluated from
% the parametric form.
% 
% R - matrix with columns containing
% position vectors for points on the
% curve
% T, N, B - matrices with columns containing the
tangent, normal, and binormal vectors
% kap - vector of curvature values
% tau - vector of torsion values
% arclen - value of arc length approximated as
the sum of chord values between
% successive points
% 
% User m functions called:
% crvprp3d, cubrange
% -----------------------------------------------

if nargin==0
    k=1; h=2; r0=2*pi; t=linspace(2*pi,8*pi,101);
end

% Evaluate R, R'(t), R''(t) and R'''(t) for
% the spiral
% t=t(:)'; s=sin(t); c=cos(t); kc=k*c; ks=k*s;
rk=r0+k*t; rks=rk.*s; rkc=rk.*c; n=length(t);
R=[rkc;rks;h*t]; R1=[kc-rks;ks+rkc;h*ones(1,n)];
R2=[-2*ks-rkc;2*kc-rks;zeros(1,n)];
R3=[-3*kc+rks;-3*ks-rkc;zeros(1,n)];

% Obtain geometrical properties
[T,N,B,kap,tau]=crvprp3d(R1,R2,R3);
arclen=sum(sqrt(sum((R(:,2:n)-R(:,1:n-1)).^2)));
Z=[Rn(3,:);R(3,:);Rb(3,:)];

% Draw the surface
v=cubrange([X(:),Y(:),Z(:)]); hold off; clf; close;
surf(X,Y,Z); axis(v); xlabel('x axis');
ylabel('y axis'); zlabel('z axis');
title(['Spiral Showing Osculating and ', ...
'Rectifying Planes']); grid on; drawnow;
figure(gcf);

%============================================================================

function [T,N,B,kap,tau]=crvprp3d(R1,R2,R3)
% This function computes the primary differential properties of a three-dimensional curve parameterized in the form R(t) where t can be arc length or any other convenient parameter such as time.
% R1 - the matrix with columns containing R'(t)
% R2 - the matrix with columns containing R''(t)
% R3 - the matrix with columns containing R'''(t). This matrix is only needed when torsion is to be computed.
% T - matrix with columns containing the unit tangent
% N - matrix with columns containing the principal normal vector
% B - matrix with columns containing the binormal
% kap - vector of curvature values
% tau - vector of torsion values. This equals [] when R3 is not given
% User m functions called: none
%============================================================================

nr1=sqrt(dot(R1,R1)); T=R1./nr1(ones(3,1),:);
R12=cross(R1,R2); nr12=sqrt(dot(R12,R12));
B=R12./nr12(ones(3,1),:); N=cross(B,T);
\( \text{kap} = \frac{\text{nr12}}{\text{nr1}^3}; \)

\% Compute the torsion only when \( R'''(t) \) is given

if nargin==3, \( \text{tau} = \text{dot}(B, R3)/\text{nr12}; \)
else, \( \text{tau} = [] \); end

\%=======================================================================

function \([R, T, N, B, \text{kappa}] = \text{crvprpsp}(\text{Rd}, n)\)

\% 
\% \([R, T, N, B, \text{kappa}] = \text{crvprpsp}(\text{Rd}, n)\)
\% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\% 
\% This function computes spline interpolated \n\% values for coordinates, base vectors and \n\% curvature obtained by passing a spline curve \n\% through data values given in \text{Rd}. \n\% 
\% \text{Rd} - a matrix containing x,y and z values \n\% in rows 1, 2 and 3. \n\% \text{n} - the number of points at which \n\% properties are to be evaluated along \n\% the curve \n\% 
\% \text{R} - a 3 by n matrix with columns \n\% containing coordinates of interpolated \n\% points on the curve \n\% \text{T,N,B} - matrices of dimension 3 by n with \n\% columns containing components of the \n\% unit tangent, unit normal, and unit \n\% binormal vectors \n\% \text{kappa} - a vector of curvature values \n\% 
\% User m functions called:
\% \text{splined, crvprp3d}
\% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\% Create a spline curve through the data points, \n% and evaluate the derivatives of \( R \).
\( \text{nd} = \text{size}(\text{Rd},2); \text{td} = 0:\text{nd}-1; \text{t} = \text{linspace}(0, \text{nd}-1, \text{n}); \)
\( \text{ud} = \text{Rd}(1,:)+i*\text{Rd}(2,:); \text{u} = \text{spline}(\text{td}, \text{ud}, \text{t}); \)
\( \text{u1} = \text{splined}(\text{td}, \text{ud}, \text{t}); \text{u2} = \text{splined}(\text{td}, \text{ud}, \text{t}, 2); \)
\( \text{ud3} = \text{Rd}(3,:); \text{z} = \text{spline}(\text{td}, \text{ud3}, \text{t}); \)
\( \text{z1} = \text{splined}(\text{td}, \text{ud3}, \text{t}); \text{z2} = \text{splined}(\text{td}, \text{ud3}, \text{t}, 2); \)
\( \text{R} = [\text{real}(	ext{u}); \text{imag}(	ext{u}); \text{z}]; \text{R1} = [\text{real}(	ext{u1}); \text{imag}(	ext{u1}); \text{z1}]; \)
R2=[real(u2);imag(u2);z2];

% Get curve properties from crvprp3d
[T,N,B,kappa]=crvprp3d(R1,R2);

%==============================================

function val=splined(xd,yd,x,if2)

% val=splined(xd,yd,x,if2)
% ~~~~~~~~~~~~~~~~~~~~~~~~

% This function evaluates the first or second
derivative of the piecewise cubic
interpolation curve defined by the intrinsic
curve spline provided in MATLAB. If fewer
than four data points are input, then simple
polynomial interpolation is employed

% xd,yd - data vectors determining the spline
curve produced by function spline
% x - vector of values where the first or
the second derivative are desired
% if2 - a parameter which is input only if
y''(x) is required. Otherwise, y'(x)
is returned.
% val - the first or second derivative values
for the spline

% User m functions called: none

n=length(xd); [b,c]=unmkpp(spline(xd,yd));
if n>3 % Use a cubic spline
    if nargin==3, c=[3*c(:,1),2*c(:,2),c(:,3)];
    else, c=[6*c(:,1),2*c(:,2)]; end
    val=ppval(mkpp(b,c),x);
else % Use a simple polynomial
    c=polyder(polyfit(xd(:),yd(:),n-1));
    if nargin==4, c=polyder(c); end
    val=polyval(c,x);
end

%================================================================

© 2003 by CRC Press LLC
% function range=cubrange(xyz,ovrsiz)
% See Appendix B

2.8.2 Surface Properties

Surfaces are two-dimensional regions described parametrically as

\[ \mathbf{R}(u, v) = \hat{i}x(u, v) + \hat{j}y(u, v) + \hat{k}z(u, v) \]

where \( u \) and \( v \) are scalar parameters. This parametric form is helpful for generating a grid of points on the surface as well as for computing surface tangents and the surface normal. Holding \( v \) fixed while \( u \) varies generates a curve in the surface called a \( u \)-coordinate line. A tangent vector to the \( u \)-line is given by

\[ g_u = \frac{\partial \mathbf{R}}{\partial u} = \hat{i} \frac{\partial x}{\partial u} + \hat{j} \frac{\partial y}{\partial u} + \hat{k} \frac{\partial z}{\partial u}. \]

Similarly, holding \( u \) fixed and varying \( v \) produces a \( v \)-line with tangent vector

\[ g_v = \frac{\partial \mathbf{R}}{\partial v} = \hat{i} \frac{\partial x}{\partial v} + \hat{j} \frac{\partial y}{\partial v} + \hat{k} \frac{\partial z}{\partial v}. \]

Consider the following cross product.

\[ g_u \times g_v \, du \, dv = \hat{n} \, dS. \]

In this equation \( \hat{n} \) is the unit surface normal and \( dS \) is the area of a parallelogram shaped surface element having sides defined by \( g_u \, du \) and \( g_v \, dv \).

The intrinsic functions \texttt{surf(X,Y,Z)} and \texttt{mesh(X,Y,Z)} depict surfaces by showing a grid network and related surface patches characterized when parameters \( u \) and \( v \) are varied over constant limits. Thus, values

\[ (u_i, v_j), \ 1 \leq i \leq n, \ 1 \leq j \leq m \]

lead to matrices

\[ X = [x(u_i, v_j)], \ Y = [y(u_i, v_j)], \ Z = [z(u_i, v_j)] \]

from which surface plots are obtained. Function \texttt{surf} colors the surface patches whereas \texttt{mesh} colors the grid lines.

As a simple example, consider the ellipsoidal surface described parametrically as

\[ x = a \cos \theta \cos \phi, \ y = b \cos \theta \sin \phi, \ z = c \sin \theta \]

where \( -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}, \ -\pi \leq \phi \leq \pi \). The surface equation evidently satisfies the familiar equation

\[ \left( \frac{x}{a} \right)^2 + \left( \frac{y}{b} \right)^2 + \left( \frac{z}{c} \right)^2 = 1 \]

© 2003 by CRC Press LLC
for an ellipsoid. The function `ellipsoid(a,b,c)` called with \( a = 2, b = 1.5, c = 1 \) produces the surface plot in Figure 2.18.

Many types of surfaces can be parameterized in a manner similar to the ellipsoid. We will examine two more problems involving a torus and a conical frustum. Consider a circle of radius \( b \) lying in the \( xz \)-plane with its center at \([a, 0, 0]\). Rotating the circle about the \( z \)-axis produces a torus having the surface equation

\[
x = [a + b \cos \theta] \cos \phi, \quad y = [a + b \cos \theta] \sin \phi, \quad z = b \sin \phi
\]

where \(-\pi \leq \theta \leq \pi, -\pi \leq \phi \leq \pi\).

This type of equation is used below in an example involving several bodies. Let us also produce a surface covering the ends and side of a conical frustum (a cone with the top cut off). The frustum has base radius \( r_b \), top radius \( r_t \), and height \( h \), with the symmetry axis along the \( z \)-axis. The surface can be parameterized using an azimuthal angle \( \theta \) and an arc length parameter relating to the axial direction. The lateral side length is

\[
rs = \sqrt{h^2 + (r_b - r_t)^2}.
\]

Let us take \( 0 \leq s \leq (r_b + r_s + r_t) \) and describe the surface \( R(s, \theta) \) by coordinate functions

\[
x = r(s) \cos \theta, \quad y = r(s) \sin \theta, \quad z = z(s)
\]

where \( 0 \leq \theta \leq 2\pi \) and

\[
r(s) = s, \quad 0 \leq s \leq r_b
\]

\[
r(s) = r_b + \frac{(r_t - r_b)(s - r_b)}{r_s}, \quad z = \frac{h(s - r_b)}{r_s}, \quad r_b \leq s \leq (r_b + r_s)
\]

\[
r(s) = r_b + r_s + r_t - r, \quad z = h, \quad (r_b + r_s) \leq s \leq (r_b + r_s + r_t).
\]

The function `frus` produces a grid of points on the surface in terms of \( r_b, r_t, h \), the number of increments on the base, the number of increments on the side, and the number of increments on the top. Figure 2.16 shows the plot generated by `frus`.

An example called `srfex` employs the ideas just discussed and illustrates how MATLAB represents several interesting surfaces. Points on the surface of an annulus symmetric about the \( z \)-axis are created, and two more annuli are created by interchanging axes. A pyramid with a square base is also created and the combination of four surfaces is plotted by finding a data range to include all points and then plotting each surface in succession using the `hold` instruction (See Figure 2.16). Although the rendering of surface intersections is not perfect, a useful description of a fairly involved geometry results. Combined plotting of several intersecting surfaces is implemented in a general purpose function `surfmany`. The default data case for `surfmany` produces the six-legged geometry shown in Figure 2.17.

This section is concluded with a discussion of how a set of coordinate points can be moved to a new position by translation and rotation of axes. Suppose a vector

\[
r = \hat{i}x + \hat{j}y + \hat{k}z
\]
Spike and Intersecting Toruses

undergoes a coordinate change which moves the initial coordinate origin to \((X_0, Y_0, Z_0)\) and moves the base vectors \(\hat{i}, \hat{j}, \hat{k}\) into \(\hat{e}_1, \hat{e}_2, \hat{e}_3\). Then the endpoint of \(r\) passes to

\[
R = \hat{i}X + \hat{j}Y + \hat{k}Z = R_o + \hat{e}_1x + \hat{e}_2y + \hat{e}_3z
\]

where

\[
R_o = \hat{i}X_o + \hat{j}Y_o + \hat{k}Z_o.
\]

Let us specify the directions of the new base vectors by employing the columns of a matrix \(V\) where we take

\[
\hat{e}_3 = \frac{V(:,1)}{\text{norm}(V(:,1))}.
\]

If \(V(:,2)\) exists we take \(V(:,1) \times V(:,2)\) and unitize this vector to produce \(\hat{e}_2\). The triad is completed by taking \(\hat{e}_1 = \hat{e}_2 \times \hat{e}_3\). In the event that \(V(:,2)\) is not provided, we use \([1; 0; 0]\) and proceed as before. The functions \texttt{rgdbodmo} and \texttt{rotatran} can be used to transform points in the manner described above.
Figure 2.17: Surface With Six Legs
Figure 2.18: Rotated and Translated Ellipsoid Surfaces
Function srfex

1. function [x1,y1,x2,y2,x3,y3,xf,yf,zf]= ... 
2. \hspace{1em} srfex(da,na,df,nf) 
3. \hspace{1em} [x1,y1,x2,y2,x3,y3,xf,yf,zf]= ... 
4. \hspace{1em} srfex(da,na,df,nf) 
5. \hspace{1em} ----------------------------------------------- 
6. \hspace{1em} 
7. \hspace{1em} This graphics example draws three toruses 
8. \hspace{1em} intersecting a spike. 
9. 
10. \hspace{1em} User m functions called: frus, surfmany 
11. 
12. if nargin==0 
13. \hspace{1em} da=[4.0,.45]; na=[42,15]; 
14. \hspace{1em} df=[2.2,0,15]; nf=[43,4]; 
15. \hspace{1em} end 
16. 
17. \hspace{1em} Create a torus with polygonal cross section. 
18. \hspace{1em} Data for the torus is stored in da and na 
19. 
20. \hspace{1em} r0=da(1); r1=da(2); nfaces=na(1); nlat=na(2); 
21. \hspace{1em} t=linspace(0,2*pi,nlat)'; 
22. \hspace{1em} xz=[r0+r1*cos(t),r1*sin(t)]; 
23. \hspace{1em} z1=xz(:,2); z1=z1(:,ones(1,nfaces+1)); 
24. \hspace{1em} th=linspace(0,2*pi,nfaces+1); 
25. \hspace{1em} x1=xz(:,1)*cos(th); y1=xz(:,1)*sin(th); 
26. \hspace{1em} y2=x1; z2=y1; x2=z1; y3=x2; z3=y2; x3=z2; 
27. 
28. \hspace{1em} Create a frustum of a pyramid. Data for the 
29. \hspace{1em} frustum is stored in df and nf 
30. \hspace{1em} rb=df(1); rt=df(2); h=df(3); 
31. \hspace{1em} [xf,yf,zf]=frus(rb,rt,h,nf); zf=zf-.35*h; 
32. 
33. \hspace{1em} Plot four figures combined together 
34. \hspace{1em} hold off; clf; close; 
35. \hspace{1em} surfmany(x1,y1,z1,x2,y2,z2,x3,y3,z3,xf,yf,zf) 
36. \hspace{1em} xlabel('x axis'); ylabel('y axis'); 
37. \hspace{1em} zlabel('z axis'); 
38. \hspace{1em} title('Spike and Intersecting Toruses'); 
39. \hspace{1em} axis equal; axis('off'); 
40. \hspace{1em} colormap([1 1 1]); figure(gcf); hold off;
% print -deps srfex

%============================================= %

function [X,Y,Z]=frus(rb,rt,h,n,noplot)
%
% [X,Y,Z]=frus(rb,rt,h,n,noplot)
% % This function computes points on the surface
% % of a conical frustum which has its axis along
% % the z axis.
% % rb,rt,h - the base radius,top radius and
% % height
% % n - vector of two integers defining the
% % axial and circumferential grid
% % increments on the surface
% % noplot - parameter input when no plot is
% % desired
% % X,Y,Z - points on the surface
% % User m functions called: none
%
if nargin==0
    rb=2; rt=1; h=3; n=[23, 35];
end

th=linspace(0,2*pi,n(2)+1)'-pi/n(2);
sl=sqrt(h^2+(rb-rt)^2); s=sl+rb+rt;
m=ceil(n(1)/s*[rb,sl,rt]);
rbot=linspace(0,rb,m(1));
rside=linspace(rb,rt,m(2));
rtop=linspace(rt,0,m(3));
r=[rbot,rside(2:end),rtop(2:end)];
hbot=zeros(1,m(1));
hside=linspace(0,h,m(2));
htop=h*ones(1,m(3));
H=[hbot,hside(2:end),htop(2:end)];
Z=repmat(H,n(2)+1,1);
xy=exp(i*th)*r; X=real(xy); Y=imag(xy);
if nargin<5
    surf(X,Y,Z); title('Frustum'); xlabel('x axis')
ylabel('y axis'), zlabel('z axis')
grid on, colormap([1 1 1]);
figure(gcf);
end

function surfmany(varargin)
  %function surfmany(x1,y1,z1,x2,y2,z2,...
  %  x3,y3,z3,...,xn,yn,zn)
  % This function plots any number of surfaces
  % on the same set of axes without shape
  % distortion. When no input is given then a
  % six-legged solid composed of spheres and
  % cylinders is shown.
  %
  % User m functions called: none
  %-----------------------------------------------
  if nargin==0
    % Default data for a six-legged solid
    n=10; rs=.25; d=7; rs=2; rc=.75;
    [xs,ys,zs]=sphere; [xc,yc,zc]=cylinder;
    xs=rs*xs; ys=rs*ys; zs=rs*zs;
    xc=rc*xc; yc=rc*yc; zc=2*d*zc-d;
    x1=x1; y1=y1; z1=z1;
    x2=x2+d; y2=y2; z2=x2;
    x3=x3-d; y3=y3; z3=x3;
    x4=x4; y4=z4-d; z4=y4;
    x5=x5; y5=z5+d; z5=y5;
    x6=x6; y6=y6; z6=z6+d;
    x7=x7; y7=y7; z7=z7-d;
    x8=x8; y8=y8; z8=z8;
    x9=x9; y9=x9; z9=y9;
    x10=yc; y10=zc; z10=xc;
    varargin={x1,y1,z1,x2,y2,z2,x3,y3,z3,...
               x4,y4,x5,y5,z5,x6,y6,z6,x7,y7,z7,...
               x8,y8,x8,x9,y9,z9,x10,y10,z10};
  end

  % Find the data range
  n=length(varargin);
  r=realmax*[1,-1,1,-1,1,-1];
  s=inline('min([a;b]','a',','b');
  b=inline('max([a;b]','a',','b');
  for k=1:3:n
x=varargin{k}; y=varargin{k+1};
z=varargin{k+2};
x=x(:); y=y(:); z=z(:);

r(1)=s(r(1),x); r(2)=b(r(2),x);
r(3)=s(r(3),y); r(4)=b(r(4),y);
r(5)=s(r(5),z); r(6)=b(r(6),z);
end

% Plot each surface
hold off, newplot
for k=1:3:n
x=varargin{k}; y=varargin{k+1};
z=varargin{k+2};
surf(x,y,z); axis(r), hold on
end

% Set axes and display the combined plot
axis equal, axis(r), grid on
xlabel('x axis'), ylabel('y axis')
zlabel('z axis')
title('SEVERAL SURFACES COMBINED')

% colormap([127/255 1 212/255]); % aquamarine
colormap([1 1 1]);, figure(gcf), hold off

Function rgdbodmo

function [X,Y,Z]=rgdbodmo(x,y,z,v,R0)
% This function transforms coordinates x,y,z to new coordinates X,Y,Z by rotating and translating the reference frames. When no input is given, an example involving an ellipsoid is run.

% x,y,z - initial coordinate matrices referred to base vectors [1;0;0], [0;1;0] and [0;0;1]. Columns of v are used to create new basis vectors i,j,k such that a typical point [a;b;c] is transformed into [A;B;C] according to the equation

© 2003 by CRC Press LLC
\[ [A;B;C] = R_0(:) + [i,j,k] *[a;b;c] \]

- \( v \) - a matrix having three rows and either one or two columns used to construct the new basis \([i,j,k]\) according to methods employed function rotatran.
- \( R_0 \) - a vector which translates the rotated coordinates when \( R_0 \) is input. Otherwise no translation is imposed.
- \( X,Y,Z \) - matrices containing the transformed coordinates.

% User m functions called: elipsoid, rotatran

```matlab
if nargin==0
    [x,y,z]=elipsoid(1,1,2,[17,33],0);R0=[3;4;5];
v=[[1;1;1],[1;1;0]];
end
[n,m]=size(x); XYZ=[x(:),y(:),z(:)]*rotatran(v)';
X=XYZ(:,1); Y=XYZ(:,2); Z=XYZ(:,3);
if ~isempty(R0)
    X=X+R0(1); Y=Y+R0(2); Z=Z+R0(3);
end
X=reshape(X,n,m); Y=reshape(Y,n,m);
Z=reshape(Z,n,m);
if nargin==0
    close; surf(X,Y,Z), axis equal, grid on
    title('ROTATED AND TRANSLATED ELLIPSOID')
    xlabel('x axis'), ylabel('y axis')
    zlabel('z axis'), colormap([1 1 1]); shg
end

%=======================================================

function [x,y,z]=elipsoid(a,b,c,n,noplot)
% [x,y,z]=elipsoid(a,b,c,n,noplot)
% This function plots an ellipsoid having semi-
% diameters \( a,b,c \)
% \( a,b,c \) - semidiameters of the ellipsoid defined
% by \((x/a)^2+(y/b)^2+(z/c)^2=1\)
% \( n \) - vector \([nth,nph]\) giving the number of
% theta values and phi values used to plot
% the surface
% noplot - omit this parameter if no plot is desired
% x,y,z - matrices of points on the surface
% User m functions called: none
%---------------------------------------------------------------

if nargin==0, a=2; b=1.5; c=1; n=[17,33]; end
nth=n(1); nph=n(2);
th=linspace(-pi/2,pi/2,nth)'; ph=linspace(-pi,pi,nph);
x=a*cos(th)*cos(ph); y=b*cos(th)*sin(ph);
z=c*sin(th)*ones(size(ph));
if nargin<5
    surf(x,y,z); axis equal
    title('ELLIPSOID'), xlabel('x axis')
ylabel('y axis'), zlabel('z axis')
    colormap([1 1 1]); grid on, figure(gcf)
end

%==============================================

function mat=rotatran(v)
% ~~~~~~~~~~~~~~~
% This function creates a rotation matrix based
% on the columns of v.
% v - a matrix having three rows and either
% one or two columns which are used to
% create an orthonormal triad [i,j,k]
% returned in the columns of mat. The
% third base vector k is defined as
% v(:,1)/norm(v(:,1)). If v has two
% columns then, v(:,1) and v(:,2) define
% the xz plane with the direction of j
% defined by cross(v(:,1),v(:,2)). If only
% v(:,1) is input, then v(:,2) is set
% to [1;0;0].
% mat - the matrix having columns containing
% the basis vectors [i,j,k]
% User m functions called: none
%---------------------------------------------------------------
109:  k=v(:,1)/norm(v(:,1));
110:  if size(v,2)==2, p=v(:,2); else, p=[1;0;0]; end
111:  j=cross(k,p); nj=norm(j);
112:  if nj~=0
113:     j=j/nj; mat=[cross(j,k),j,k];
114:  else
115:     mat=[[0;1;0],cross(k,[0;1;0]),k];
116:  end
Chapter 3

Summary of Concepts from Linear Algebra

3.1 Introduction

This chapter briefly reviews important concepts of linear algebra. We assume the reader already has some experience working with matrices, and linear algebra applied to solving simultaneous equations and eigenvalue problems. MATLAB has excellent capabilities to perform matrix operations using the fastest and most accurate algorithms currently available. The books by Strang [96] and Golub and Van Loan [47] give comprehensive treatments of matrix theory and of algorithm developments accounting for effects of finite precision arithmetic. One beautiful aspect of matrix theory is that fairly difficult proofs often lead to remarkably simple results valuable to users not necessarily familiar with all of the theoretical developments. For instance, the property that every real symmetric matrix of order $n$ has real eigenvalues and a set of $n$ orthonormal eigenvectors can be understood and used by someone unfamiliar with the proof. The current chapter summarizes a number of fundamental matrix properties and some of the related MATLAB functions. The intrinsic matrix functions use highly efficient algorithms originally from the LINPACK and EISPACK libraries which have now been superceded by LAPACK. [34, 42, 89]. Dr. Cleve Moler, the Chairman and Chief Scientist at The MathWorks, contributed to development of these systems. He also wrote the first version of MATLAB. Readers should simultaneously study the current chapter and the MATLAB demo program on linear algebra.

3.2 Vectors, Norms, Linear Independence, and Rank

Consider an $n$ by $m$ matrix

$$A = [a_{ij}] , \ 1 \leq i \leq n , \ 1 \leq j \leq m,$$

having real or complex elements. The shape of a matrix is computed by \texttt{size}(A) which returns a vector containing $n$ and $m$. The matrix obtained by conjugating the matrix elements and interchanging columns and rows is called the transpose.
Transposition is accomplished with a $'$ operator, so that

$$A_{\text{transpose}} = A'.$$

Transposition without conjugation of the elements can be performed as $A.'$ or as $\text{conj}(A')$. Of course, whenever $A$ is real, $A'$ is simply the traditional transpose.

The structure of a matrix $A$ is characterized by the matrix rank and sets of basis vectors spanning four fundamental subspaces. The rank $r$ is the maximum number of linearly independent rows or columns in the matrix. We discuss these spaces in the context of real matrices. The basic subspaces are:

1. The column space containing all vectors representable as a linear combination of the columns of $A$. The column space is also referred to as the range or the span.
2. The null space consisting of all vectors perpendicular to every row of $A$.
3. The row space consisting of all vectors which are linear combinations of the rows of $A$.
4. The left null space consisting of all vectors perpendicular to every column of $A$.

MATLAB has intrinsic functions to compute rank and subspace bases

- matrix rank = $\text{rank}(A)$
- column space = $\text{orth}(A)$
- null space = $\text{null}(A)$
- row space = $\text{orth}(A')'$
- left null space = $\text{null}(A')'$

The basis vectors produced by $\text{null}$ and $\text{orth}$ are orthonormal. They are generated using the singular value decomposition algorithm [47]. The MATLAB function to perform this type of computation is named $\text{svd}$.

### 3.3 Systems of Linear Equations, Consistency, and Least Squares Approximation

Let us discuss the problem of solving systems of simultaneous equations. Representing a vector $B$ as a linear combination of the columns of $A$ requires determination of a vector $X$ to satisfy

$$AX = B \iff \sum_{j=1}^{m} A(:,j) x(j) = B$$
where the \( j \)’th column of \( A \) is scaled by the \( j \)’th component of \( X \) to form the linear combination. The desired representation is possible if and only if \( B \) lies in the column space of \( A \). This implies the consistency requirement that \( A \) and \([A, B]\) must have the same rank. Even when a system is consistent, the solution will not be unique unless all columns of \( A \) are independent. When matrix \( A \), with \( n \) rows and \( m \) columns, has rank \( r \) less than \( m \), the general solution of \( AX = B \) is expressible as any particular solution plus an arbitrary linear combination of \( m - r \) vectors forming a basis for the null space. MATLAB gives the solution vector as \( X = A \backslash B \). When \( r \) is less than \( m \), MATLAB produces a least squares solution having as many components as possible set equal to zero.

In instances where the system is inconsistent, regardless of how \( X \) is chosen, the error vector defined by

\[ E = AX - B \]

can never be zero. An approximate solution can be obtained by making \( E \) normal to the columns of \( A \). We get

\[ A'AX = A'B \]

which is known as the system of normal equations. They are also referred to as least squares error equations. It is not difficult to show that the same equations result by requiring \( E \) to have minimum length. The normal equations are always consistent and are uniquely solvable when \( \text{rank}(A) = m \). A comprehensive discussion of least squares approximation and methods for solving overdetermined systems is presented by Lawson and Hanson [62]. It is instructive to examine the results obtained from the normal equations when \( A \) is square and nonsingular. The least squares solution would give

\[ X = (A'A)^{-1}A'B = A^{-1}(A')^{-1}A'B = A^{-1}B. \]

Therefore, the least squares solution simply reduces to the exact solution of \( AX = B \) for a consistent system. MATLAB handles both consistent and inconsistent systems as \( X = A \backslash B \). However, it is only sensible to use the least squares solution of an inconsistent system when \( AX \) produces an acceptable approximation to \( B \). This implies

\[ \text{norm}(AX - B) < tol \ast \text{norm}(B) \]

where \( tol \) is suitably small.

A simple but important application of overdetermined systems arises in curve fitting. An equation of the form

\[ y(x) = \sum_{j=1}^{m} f_j(x)c_j \]

involving known functions \( f_j(x) \), such as \( x^{j-1} \) for polynomials, must approximately match data values \((X_i, Y_i), 1 \leq i \leq n\), with \( n > m \). We simply write an overdetermined system

\[ \sum_{j=1}^{n} f_j(X_i)c_j \approx Y_i, \ 1 \leq i \leq n \]
and obtain the least squares solution. The approximation is acceptable if the error components

$$e_i = \sum_{j=1}^{m} f_j (X_i) e_j - Y_i$$

are small enough and the function $y(x)$ is also acceptably smooth between the data points.

Let us illustrate how well MATLAB handles simultaneous equations by constructing the steady-state solution of the matrix differential equation

$$M \ddot{x} + C \dot{x} + Kx = F_1 \cos(\omega t) + F_2 \sin(\omega t)$$

where $M$, $C$, and $K$ are constant matrices and $F_1$ and $F_2$ are constant vectors. The steady-state solution has the form

$$x = X_1 \cos(\omega t) + X_2 \sin(\omega t)$$

where $X_1$ and $X_2$ are chosen so that the differential equation is satisfied. Evidently

$$\dot{x} = -\omega X_1 \sin(\omega t) + \omega X_2 \cos(\omega t)$$

and

$$\ddot{x} = -\omega^2 x.$$ 

Substituting the assumed form into the differential equation and comparing sine and cosine terms on both sides yields

$$(K - \omega^2 M) X_1 + \omega C X_2 = F_1,$$

$$-\omega C X_1 + (K - \omega^2 M) X_2 = F_2.$$ 

The equivalent partitioned matrix is

$$\begin{bmatrix} (K - \omega^2 M) & \omega C \\ -\omega C & (K - \omega^2 M) \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}.$$ 

A simple MATLAB function to produce $X_1$ and $X_2$ when $M$, $C$, $K$, $F_1$, $F_2$, and $\omega$ are known is

```matlab
function [x1,x2,xmax]=forcresp(m,c,k,f1,f2,w)
  kwm=k-(w*w)*m; wc=w*c;
  x=[kwm,wc;-wc,kwm]
  \f1
  x1=x(1:n); x2=x(n+1:2*n);
  xmax=sqrt(x1.*x1+x2.*x2);
end
```

The vector, $xmax$, defined in the last line of the function above, has components specifying the maximum amplitude of each component of the steady-state solution.
The main computation in this function occurs in the third line, where matrix concatenation is employed to form a system of $2n$ equations with $x$ being the concatenation of $X_1$ and $X_2$. The fourth line uses vector indexing to extract $X_1$ and $X_2$ from $x$. The notational simplicity of MATLAB is elegantly illustrated by these features: a) any required temporary storage is assigned and released dynamically, b) no looping operations are needed, c) matrix concatenation and inversion are accomplished with intrinsic functions using matrices and vectors as sub-elements of other matrices, and d) extraction of sub-vectors is accomplished by use of vector indices. The important differential equation just discussed will be studied further in Article 3.5.3 where eigenvalues and complex arithmetic are used to obtain a general solution satisfying arbitrary initial conditions.

### 3.4 Applications of Least Squares Approximation

The idea of solving an inconsistent system of equations in the least squares sense, so that some required condition is approximately satisfied, has numerous applications. Typically, we are dealing with a large number of equations (several hundred is common) involving a smaller number of parameters used to closely fit some constraint. Linear boundary value problems often require the solution of a differential equation applicable in the interior of a region while the function values are known on the boundary. This type of problem can sometimes be handled by using a series of functions which satisfy the differential equation exactly. Weighting the component solutions to approximately match the remaining boundary condition may lead to useful results. Below, we examine three instances where least squares approximation is helpful.

#### 3.4.1 A Membrane Deflection Problem

Let us illustrate how least squares approximation can be used to compute the transverse deflection of a membrane subjected to uniform pressure. The transverse deflection $u$ for a membrane which has zero deflection on a boundary $L$ satisfies the differential equation

$$
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -\gamma, \quad (x,y) \text{ inside } L
$$

where $\gamma$ is a physical constant. Properties of harmonic functions [18] imply that the differential equation is satisfied by a series of the form

$$
\begin{align*}
  u &= \gamma \left[ -\frac{|z|^2}{4} + \sum_{j=1}^{n} c_j \text{real}(z^{j-1}) \right] \\
\end{align*}
$$

© 2003 by CRC Press LLC
where \( z = x + iy \) and constants \( c_j \) are chosen to make the boundary deflection as small as possible, in the least squares sense. As a specific example, we analyze a membrane consisting of a rectangular part on the left joined with a semicircular part on the right. The surface plot in Figure 3.1 and the contour plot in Figure 3.2 were produced by the function `membran` listed below. This function generates boundary data, solves for the series coefficients, and constructs plots depicting the deflection pattern. The results obtained using a twenty-term series satisfy the boundary conditions quite well.
Figure 3.2: Membrane Surface Contour Lines
MATLAB Example

Function membran

```matlab
function [dfl,cof]=membran(h,np,ns,nx,ny)
% [dfl,cof]=membran(h,np,ns,nx,ny)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function computes the transverse
% deflection of a uniformly tensioned membrane
% which is subjected to uniform pressure. The
% membrane shape is a rectangle of width h and
% height two joined with a semicircle of
% diameter two.
%
% Example use: membran(0.75,100,50,40,40);
%
% h - the width of the rectangular part
% np - the number of least square points
% used to match the boundary
% ns - the number of terms used in the
% approximating series to evaluate
% deflections. The series has the
% form
%
% dfl = abs(z)^2/4 +
% sum({j=1:ns},cof(j)*
% real(z^(j-1)))
%
% nx,ny - the number of x points and y points
% used to compute deflection values
% on a rectangular grid
% dfl - computed array of deflection values
% cof - coefficients in the series approximation
%
% User m functions called: none

if nargin==0
    h=.75; np=100; ns=50; nx=40; ny=40;
end
% Generate boundary points for least square
```

© 2003 by CRC Press LLC
% approximation
z=[exp(i*linspace(0,pi/2,round(1.5*np))),...
  linspace(i,-h+i,np),...
  linspace(-h+i,-h,round(np/2))];
z=z(:); xb=real(z); xb=[xb;xb(end:-1:1)];
yb=imag(z); yb=[yb;-yb(end:-1:1)]; nb=length(xb);

% Form the least square equations and solve
% for series coefficients
a=ones(length(z),ns);
for j=2:ns, a(:,j)=a(:,j-1).*z; end
cof=real(a)
  (z.*conj(z))/4;

% Generate a rectangular grid for evaluation
% of deflections
xv=linspace(-h,1,nx); yv=linspace(-1,1,ny);
[x,y]=meshgrid(xv,yv); z=x+i*y;

% Evaluate the deflection series on the grid
dfl=-z.*conj(z)/4+ ...
  real(polyval(cof(ns:-1:1),z));

% Set values outside the physical region of
% interest to zero
dfl=real(dfl).*((1-((abs(z)>=1)&(real(z)>=0))));

% Make surface and contour plots
hold off; close; surf(x,y,dfl);
xlabel(‘x axis’); ylabel(‘y axis’);
zlabel(‘deflection’); view(-10,30);
title(‘Membrane Deflection’); colormap([1 1 1]);
shg, disp(...
 ‘Press [Enter] to show a contour plot’), pause
% print -deps membdefl;
contour(x,y,dfl,15,’k’); hold on
plot(xb,yb,’k-’); axis(‘equal’), hold off
xlabel(‘x axis’); ylabel(‘y axis’);
title(‘Membrane Surface Contour Lines’), shg
% print -deps membcntr
3.4.2 Mixed Boundary Value Problem for a Function Harmonic Inside a Circular Disk

Problems where a partial differential equation is to be solved inside a region with certain conditions imposed on the boundary occur in many situations. Often the differential equation is solvable exactly in a series form containing arbitrary linear combinations of known functions. An approximation procedure imposing the boundary conditions to compute the series coefficients produces a satisfactory solution if the desired boundary conditions are found to be well satisfied. Consider a mixed boundary value problem in potential theory [73] pertaining to a circular disk of unit radius. We seek \( u(r, \theta) \) where function values are specified on one part of the boundary and normal derivative values are specified on the remaining part. The mathematical formulation is

\[
\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} = 0, \quad 0 \leq r < 1, \quad 0 \leq \theta \leq 2\pi,
\]

\[
u(1, \theta) = f(\theta), \quad -\alpha < \theta < \alpha,
\]

\[
\frac{\partial u}{\partial r}(1, \theta) = g(\theta), \quad \alpha < \theta < 2\pi - \alpha.
\]

The differential equation has a series solution of the form

\[
u(r, \theta) = c_0 + \sum_{n=1}^{\infty} r^n[c_n \cos(n\theta) + d_n \sin(n\theta)]
\]

where the boundary conditions require

\[
c_0 + \sum_{n=1}^{\infty} [c_n \cos(n\theta) + d_n \sin(n\theta)] = f(\theta), \quad -\alpha < \theta < \alpha,
\]

and

\[
\sum_{n=1}^{\infty} n[c_n \cos(n\theta) + d_n \sin(n\theta)] = g(\theta), \quad \alpha < \theta < 2\pi - \alpha.
\]

The series coefficients can be obtained by least squares approximation. Let us explore the utility of this approach by considering a particular problem for a field which is symmetric about the \( x \)-axis. We want to solve

\[
\nabla^2 u = 0, \quad r < 1,
\]

\[
u(1, \theta) = \cos(\theta), \quad |\theta| < \pi/2,
\]

\[
\frac{\partial u}{\partial r}(1, \theta) = 0, \quad \pi/2 < |\theta| \leq \pi.
\]

This problem characterizes steady-state heat conduction in a cylinder with the left half insulated and the right half held at a known temperature. The appropriate series solution is

\[
u = \sum_{n=0}^{\infty} c_n r^n \cos(n\theta)
\]
subject to
\[ \sum_{n=0}^{\infty} c_n \cos(n\theta) = \cos(\theta) \text{ for } |\theta| < \pi/2, \]
and
\[ \sum_{n=0}^{\infty} n c_n \cos(n\theta) = 0 \text{ for } \pi/2 < |\theta| \leq \pi. \]

We solve the problem by truncating the series after a hundred or so terms and forming an overdetermined system derived by imposition of both boundary conditions. The success of this procedure depends on the series converging rapidly enough so that a system of least squares equations having reasonable order and satisfactory numerical condition results. It can be shown by complex variable methods (see Muskhelishvili [73]) that the exact solution of our problem is given by

\[ u = \text{real} \left[ z + z^{-1} + (1 - z^{-1}) \sqrt{z^2 + 1} / 2, \right. \]
\[ \left. |z| \leq 1 \right] \]

where the square root is defined for a branch cut along the right half of the unit circle with the chosen branch being that which equals +1 at \( z = 0 \). Readers familiar with analytic function theory can verify that the boundary values of \( u \) yield

\[ u(1, \theta) = \cos(\theta), \ \ |\theta| \leq \pi/2, \]
\[ u(1, \theta) = \cos(\theta) + \sin(|\theta|/2) \sqrt{2|\cos(\theta)|}, \ \ \pi/2 \leq |\theta| \leq \pi. \]

A least squares solution is presented in function \texttt{mbvp}. Results from a series of 100 terms are shown in Figure 3.3. The series solution is accurate within about one percent error except for points near \( \theta = \pi/2 \). Although the results are not shown here, using 300 terms gives a solution error nowhere exceeding 4 percent. Hence the least squares series solution provides a reasonable method to handle the mixed boundary value problem.
Figure 3.3: Mixed Boundary Value Problem Solution
MATLAB Example

Program mbvprun

function mbvprun(nser,nf,ng,neval)
% Example: mbvprun(nser,nf,ng,neval)
% --------------------------
% Mixed boundary value problem for a function
% harmonic inside a circle.
% % User m functions required:
% % mbvp

disp('Calculating');

% Set data for series term and boundary condition points
if nargin==0
    nser=80; nf=100; ng=100; neval=500;
end

% Compute the series coefficients
[cof,y]=mbvp('cos',pi/2,nser,nf,ng,neval);

% Evaluate the exact solution for comparison
thp=linspace(0,pi,neval)';
y=cos(thp*(0:nser-1))*cof;
ye=cos(thp)+sin(thp/2).*sqrt(2*abs(cos(thp))).*(thp>=pi/2);

% Plot results showing the accuracy of the least square solution
thp=thp*180/pi; plot(thp,y,'-',thp,y-ye,'--'); xlabel('polar angle');
ylabel('function value and error');
title(['Mixed Boundary Value Problem ',...
'Solution for ',int2str(nser),' Terms']); legend('Function value','Solution Error'); figure(gcf); % print -deps mbvp

%==============================================

% mbvp

function [cof,y]=mbvp(func,alp,nser,nf,ng,neval)
This function solves approximately a mixed boundary value problem for a function which is harmonic inside the unit disk, symmetric about the x axis, and has boundary conditions involving function values on one part of the boundary and zero gradient elsewhere.

% func - function specifying the function value between zero and alp
% alp - angle between zero and pi which specifies the point where boundary conditions change from function value to zero gradient
% nser - number of series terms used
% nf - number of function values specified from zero to alp
% ng - number of points from alp to pi where zero normal derivative is specified
% neval - number of boundary points where the solution is evaluated
% cof - coefficients in the series solution
% y - function values for the solution

% Create evenly spaced points to impose boundary conditions
th1=linspace(0,alp,nf);
th2=linspace(alp,pi,ng+1); th2(1)=[];

% Form an overdetermined system based on the boundary conditions
yv=feval(func,th1);
cmat=cos([th1(:);th2(:)]*(0:nser-1));
[nr,nc]=size(cmat);
cmat(nf+1:nr,:)=... (ones(ng,1)*(0:nser-1)).*cmat(nf+1:nr,:);
cof=cmat\[yv(:);zeros(ng,1)];
3.4.3 Using Rational Functions to Conformally Map a Circular Disk onto a Square

Another problem illustrating the value of least squares approximation arises in connection with an example discussed earlier in Section 2.4 where a slowly convergent power series was used to map the interior of a circle onto the interior of a square [75]. It is sometimes possible for slowly convergent power series of the form

$$ w = f(z) = \sum_{j=0}^{N} c_j z^j, \; |z| \leq 1 $$

to be replaceable by a rational function

$$ w = \frac{\sum_{j=0}^{n} a_j z^j}{1 + \sum_{j=1}^{m} b_j z^j} $$

Of course, the polynomial is simply a special rational function form with $m = 0$ and $n = N$. This rational function implies

$$ \sum_{j=0}^{n} a_j z^j - w \sum_{j=1}^{m} b_j z^j = w. $$

Coefficients $a_j$ and $b_j$ can be computed by forming least square equations based on boundary data. In some cases, the resulting equations are rank deficient and it is safer to solve a system of the form $UY = V$ as $Y = \text{pinv}(U) * V$ rather than using $Y = U \backslash V$. The former solution uses the pseudo inverse function \text{pinv} which automatically sets to zero any solution components that are undetermined.

Two functions \texttt{ratcof} and \texttt{raterp} were written to compute rational function coefficients and to evaluate the rational function for general matrix arguments. These functions are useful to examine the conformal mapping of the circular disk $|z| \leq 1$ onto the square defined by $|\text{real}(w)| \leq 1, |\text{imag}(w)| \leq 1$. A polynomial approximation of the mapping function has the form

$$ w/z = \sum_{j=0}^{N} c_j (z^4)^j $$

where $N$ must be quite large in order to avoid excessive corner rounding. If we evaluate $w$ versus $z$ on the boundary for large $N$ (500 or more), and then develop

```
86: % Evaluate the solution on the boundary
87: thp=linspace(0,pi,neval)';
88: y=cos(thp*(0:nser-1))*cof;
```
A rational function fit with \( n = m = 10 \), a reasonably good representation of the square results without requiring a large number of series terms. The following program illustrates the use of functions `ratcof` and `raterp`. It also includes a function `sqmp` to generate coefficients in the Schwarz-Christoffel series. (See Chapter 11 for further discussion.) Figure 3.4 shows the geometry mapping produced near a corner.

**MATLAB Example**

**Program makratsq**

```matlab
1 function [ctop,cbot]=makratsq
2 % Example: [ctop,cbot]=makratsq
3 % ---------------------
4 % Create a rational function map of a unit disk
5 % onto a square.
6 %
7 % User m functions required:
8 %  sqmp, ratcof, raterp
```

© 2003 by CRC Press LLC
disp(' '); disp('RATIONAL FUNCTION MAPPING OF A CIRCULAR DISK ONTO A SQUARE'); disp(' '); disp('Calculating'); disp(' '); % Generate boundary points given by the Schwarz-Christoffel transformation 

nsc=501; np=401; ntop=10; nbot=10; 
z=exp(i*linspace(0,pi/4,np));
w=sqmp(nsc,1,1,1,0,45,np);
w=mean(real(w))+i*imag(w); 
z=[z,conj(z)]; w=[w,conj(w)];
% Compute the series coefficients for a rational function fit to the boundary data
[ctop,cbot]=ratcof(z.^4,w./z,ntop,nbot);
ctop=real(ctop); cbot=real(cbot);
% The above calculations produce the following coefficients
[top,bot]=

% 1.0787  1.4948
% 1.5045  0.1406
% 0.0353  -0.1594
% -0.1458  0.1751
% 0.1910  -0.1513
% -0.1797  0.0253
% 0.0489  0.2516
% 0.2595  0.1069
% 0.0945  0.0102
% 0.0068  0.0001
% Generate a polar coordinate grid to describe the mapping near the corner of the square.
% Then evaluate the mapping function.
r1=.95; r2=1; nr=12;
t1=.9*pi/4; t2=1.1*pi/4; nt=101;
[r,th]=meshgrid(linspace(r1,r2,nr), linspace(t1,t2,nt));
z=r.*exp(i*th); w=z.*raterp(ctop,cbot,z.^4);
% Plot the mapped geometry
close; u=real(w); v=imag(w);
plot(u,v,'k','u',v,'k'), axis equal

© 2003 by CRC Press LLC
%==============================================

function [w,b]=sqmp(m,r1,r2,nr,t1,t2,nt)
% [w,b]=sqmp(m,r1,r2,nr,t1,t2,nt)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function evaluates the conformal
% mapping produced by the Schwarz-Christoffel
% transformation w(z) mapping abs(z)<1 inside
% a square having a side length of two. The
% transformation is approximated in series form
% which converges very slowly near the corners.
% This function is the same as squarmap of
% chapter 2 with no plotting.
%----------------------------------------------

% Generate polar coordinate grid points for the
% map. Function linspace generates vectors with
% equally spaced components.
% r=linspace(r1,r2,nr)';
% t=pi/180*linspace(t1,t2,nt);
% z=(r*ones(1,nt)).*(ones(nr,1)*exp(i*t));
% Compute the series coefficients and evaluate
% the series
% k=1:m-1;
% b=cumprod([1,-(k-.75).*(.5)./(k.*(.25))]);
b = b/sum(b); w = z.*polyval(b(m:-1:1),z.^4);

function [a,b]=ratcof(xdata,ydata,ntop,nbot)
% [a,b]=ratcof(xdata,ydata,ntop,nbot)
% %=================================================================
% % Determine a and b to approximate ydata as
% % a rational function of the variable xdata.
% % The function has the form:
% % y(x) = sum(1=>ntop) ( a(j)*x^(j-1) ) / 
% % ( 1 + sum(1=>nbot) ( b(j)*x^(j)) )
% % xdata,ydata - input data vectors (real or complex)
% % ntop,nbot - number of series terms used in the numerator and the denominator.
% % User m functions called: none
% %=================================================================

ydata=ydata(:); xdata=xdata(:);
m=length(ydata);
if nargin==3, nbot=ntop; end;
x=ones(m,ntop+nbot); x(:,ntop+1)=-ydata.*xdata;
for i=2:ntop, x(:,i)=xdata.*x(:,i-1); end
for i=2:nbot
    x(:,i+ntop)=xdata.*x(:,i+ntop-1);
end
ab=pinv(x)*ydata; a=ab(1:ntop); b=ab(ntop+1:ntop+nbot);

function y=raterp(a,b,x)
% y=raterp(a,b,x)
% % This function interpolates using coefficients from function ratcof.
% %=================================================================

© 2003 by CRC Press LLC
Another important linear algebra problem involves the computation of nonzero vectors $X$ and numbers $\lambda$ such that

$$AX = \lambda X$$

where $A$ is a square matrix of order $n$ having elements which may be real or complex. The number $\lambda$, which can also be real or complex, is called the eigenvalue corresponding to the eigenvector $X$. The eigenvalue equation implies

$$[I\lambda - A]X = 0$$

so that $\lambda$ values must be selected to make $I\lambda - A$ singular. The polynomial

$$f(\lambda) = \det(I\lambda - A) = \lambda^n + c_1\lambda^{n-1} + \ldots + c_n$$

is called the characteristic equation and its roots are the eigenvalues. It can be factored into

$$f(\lambda) = (\lambda - \lambda_1)(\lambda - \lambda_2)\cdots(\lambda - \lambda_n).$$

The eigenvalues are generally complex numbers and some of the roots may be repeated. In the usual situation, distinct roots $\lambda_1, \lambda_2, \ldots, \lambda_n$ yield $n$ linearly independent eigenvectors obtained by solving

$$(A - \lambda_j I)X_j = 0, \quad 1 \leq j \leq n.$$  

The case involving repeated eigenvalues is more complicated. Suppose a particular eigenvalue such as $\lambda_1$ has multiplicity $k$. Then the general solution of

$$(A - \lambda_1 I)X = 0$$

will yield as few as one, or as many as $k$, linearly independent vectors. If fewer than $k$ independent eigenvectors are found for any root of multiplicity $k$, then matrix $A$ is
called defective. Occurrence of a defective matrix is not typical. It usually implies special behavior of the related physical system. The combined set of eigenvectors can be written as

\[ A[X_1, \ldots, X_n] = [X_1 \lambda_1, \ldots, X_n \lambda_n] = [X_1, \ldots, X_n] \text{diag}(\lambda_1, \ldots, \lambda_n) \]

or

\[ AU = U \Lambda \]

where \( U \) has the eigenvectors as columns and \( \Lambda \) is a diagonal matrix with eigenvalues on the diagonal. When the eigenvectors are independent, matrix \( U \), known as the modal matrix, is nonsingular. This allows \( A \) to be expressed as

\[ A = U \Lambda U^{-1} \]

which is convenient for various computational purposes. With repeated eigenvalues, the modal matrix is sometimes singular and the last form of decomposition fails. However, the eigenvectors are always independent when the eigenvalues are distinct. For the important special case of a symmetric matrix, a linearly independent set of eigenvectors always exists, even when some eigenvalues are repeated.

A matrix \( A \) is symmetric if \( A = A' \) where \( A' \) is obtained by interchanging columns and rows, and conjugating all elements. Symmetric matrices always have real eigenvalues and a linearly independent set of eigenvectors which can be orthonormalized. The eigenvectors \( X_j \) and \( X_k \) for any two unequal eigenvalues automatically satisfy an orthogonality condition

\[ X'_j X_k = 0, \ j \neq k. \]

Eigenvectors for the same repeated eigenvalue are not automatically orthogonal. Nevertheless, they can be replaced by an equivalent orthogonal set by applying a process called Gram-Schmidt orthogonalization [47]. In cases we care about here, the symmetric matrix \( A \) always has real elements. Therefore the eigenvalues are real with eigenvectors satisfying \( X'_j X_k = \delta_{ij} \), where \( \delta_{ij} \) is the Kronecker delta symbol. The orthogonality condition is equivalent to the statement that \( U'U = I \), so a real symmetric matrix can be expressed as

\[ A = U \Lambda U' \]

It is important in MATLAB that the symmetry condition \( A' = A \) be satisfied perfectly. This implies a zero value for \( \text{max} \left( \text{max} \left( \text{abs}(A - A') \right) \right) \). Sometimes, results that would be symmetric if roundoff error did not occur may produce unsymmetric results contrary to expectation. For example, \( A = B C B' \) should be symmetric if \( C \) is symmetric. Replacing \( A \) by \( (A + A')/2 \) finally will assure perfect symmetry. The MATLAB function \( \text{eig} \) computes eigenvalues and eigenvectors. When a matrix is symmetric, \( \text{eig} \) generates real eigenvalues and orthonormalized eigenvectors.

An important property of symmetric matrices and the related orthonormal eigenvector set occurs in connection with quadratic forms expressed as

\[ F(Y) = Y'AY \]
where \( Y \) is an arbitrary real vector and \( A \) is real symmetric. The function \( F(Y) \) is a one-by-one matrix; hence, it is a scalar function. The algebraic sign of the form for arbitrary nonzero choices of \( Y \) is important in physical applications. Let us use the eigenvector decomposition of \( A \) to write
\[
F = Y'U\Lambda Y = (U'Y)'\Lambda(U'Y).
\]
Taking \( X = U'Y \) and \( Y = UX \) gives
\[
F = X'AX = \lambda_1 x_1^2 + \lambda_2 x_2^2 + \lambda_3 x_3^2 + \ldots + \lambda_n x_n^2.
\]
This diagonal form makes the algebraic character of \( F \) evident. If all \( \lambda_i \) are positive, then \( F \) is evidently positive whenever \( X \) has at least one nonzero component. Then the quadratic form is called positive definite. If the eigenvalues are all positive or zero, the form is called positive semidefinite since the form cannot assume a negative value but can equal zero without having \( X = 0 \). When both negative and positive eigenvalues occur, the form can change sign and is termed indefinite. When the eigenvalues are all negative, the form is classified as negative definite. Perhaps the most important of these properties is that a necessary and sufficient condition for the form to be positive definite is that all eigenvalues of \( A \) be positive.

An important generalization of the standard eigenvalue problem has the form
\[
AX = \lambda BX
\]
for arbitrary \( A \) and nonsingular \( B \). If \( B \) is well conditioned, then it is computationally attractive to simply solve
\[
B^{-1}AX = \lambda X.
\]
In general, it is safer, but much more time consuming, to call \texttt{eig} as

\[
[EIGVECS,EIGVALS]=eig(A,B)
\]

This returns the eigenvectors as columns of \texttt{EIGVECS} and also gives a diagonal matrix \texttt{EIGVALS} containing the eigenvalues.

### 3.5.2 Application to Solution of Matrix Differential Equations

One of the most familiar applications of eigenvalues concerns the solution of the linear, constant-coefficient matrix differential equation
\[
BY'(t) = AY(t), \quad Y(0) = Y_0.
\]
Component solutions can be written as
\[
Y(t) = X e^{\lambda t}, \quad Y'(t) = \lambda X e^{\lambda t}
\]
where $X$ and $\lambda$ are constant. Substitution into the differential equation gives

$$(A - \lambda B)X e^{\lambda t} = 0.$$ 

Since $e^{\lambda t}$ cannot vanish we need

$$AX = \lambda BX.$$ 

After the eigenvalues and eigenvectors have been computed, a general solution is constructed as a linear combination of component solutions

$$Y = \sum_{j=1}^{n} X_j e^{\lambda_j t} c_j.$$ 

The constants $c_j$ are obtained by imposing the initial condition

$$Y(0) = [X_1, X_2, \ldots, X_n]c.$$ 

Assuming that the eigenvectors are linearly independent we get

$$c = [X_1, \ldots, X_n]^{-1} Y_0.$$ 

### 3.5.3 The Structural Dynamics Equation

Eigenvalues are also useful to solve the important second order matrix differential equation for which a particular solution was constructed earlier using real arithmetic. We will now use complex arithmetic and the versatile matrix notation provided in MATLAB. Structural mechanics applications often lead to the second order matrix differential equation

$$M \ddot{X}(t) + C \dot{X}(t) + KX(t) = F_1 \cos(\omega t) + F_2 \sin(\omega t)$$

where $M, C, K$ are constant matrices of order $n$, and $F_1, F_2$ are constant vectors of length $n$, and $\omega$ is the forcing function frequency. Initial conditions of the form

$$X(0) = X_0, \quad \dot{X}(0) = V_0$$

also apply. Solving this initial value problem involves combining a particular solution and a homogeneous solution. The solution we present below applies subject to the restriction that 1) the eigenvalues of the homogeneous equation should be nonzero and 2) if matrix $C$ is zero, then $i\omega$ must not coincide with an eigenvalue of the homogeneous differential equation. The particular solution is

$$X_p(t) = \text{real}(a e^{i\omega t}), \quad a = [K - M \omega^2 + iC\omega]^{-1} [F_1 - iF_2].$$

where we must assume that the implied matrix inversion exists. The particular solution satisfies initial conditions

$$X_p(0) = \text{real}(a), \quad \dot{X}_p(0) = \text{real}(i \omega a).$$
The particular solution plus the homogeneous solution, \( X_h(t) \), must satisfy the general initial conditions. Let us introduce

\[
Z(t) = [X_h(t) ; \dot{X}_h(t)]
\]

which obeys the homogeneous first order equation

\[
\dot{Z}(t) = AZ(t), \ A = [\text{eye}(n, n), \text{zeros}(n, n); -M \ [K, C]]
\]

and can be determined using the eigenvectors and eigenvalues of \( A \). Denoting the matrix of eigenvectors as \( U \) and the column of eigenvalues as \( \Lambda \), we find that

\[
Z(t) = U \text{diag}(D) \exp(i \Lambda t)
\]

where

\[
D = U \ [X_0 - X_p(0); V_0 - \dot{X}_p(0)]
\]

to satisfy the initial conditions. With \( t \) taken as a row of time values, the homogeneous solution is obtained as the first \( n \) rows of \( Z \), and the total solution is just

\[
X(t) = X_p(t) + X_h(t).
\]

A program was written to solve the structural dynamics equation. Error checks are made for the exceptional cases mentioned above. If the system is undamped (\( C = 0 \)) and \( i\omega \) matches an eigenvalue of \( A \), then program execution terminates. Occurrence of zero or repeated eigenvalues is also avoided. The program consists of a driver named \texttt{strdyneq} which reads data from a function provided by the user. An example function named \texttt{threemass} is included as a model for data preparation. Function \texttt{fhrcmk} constructs the general solution of the equation. Results of the computation can be plotted one component at a time. In addition to plotting, the program outputs the eigenvalues, a matrix of solution components, and vectors showing the lower and upper limits of motion for each degree of freedom in the system. Function \texttt{strdyneq} calls \texttt{fhrcmk} at lines 25 and 34. The name of a function defining the input data is requested. Users can employ function \texttt{threemass} to test the program. \texttt{Threemass} models a configuration of three identical masses sliding on a smooth horizontal plane and connected by four identical springs and viscous dampers. The outer two masses are connected to walls and are subjected to forces having equal magnitude but opposite direction. The middle mass has no driving force. The system is initially at rest with zero deflection when forcing functions are applied which nearly resonate with the fourth eigenvalue of the damped homogeneous system. This example was devised to illustrate how the system response grows rapidly when the forcing function is nearly resonant. Function \texttt{fhrcmk} does most of the computation work which occurs at lines 108-109, 132-134, and 139-140. This example illustrates nicely the power of the intrinsic matrix operators provided in MATLAB. A final caveat about the solution method using eigenvalues is that it is somewhat limited by special cases like repeated eigenvalues or a forcing function resonant with a natural frequency. Numerical integration solvers like \texttt{ode45} are not vulnerable to such difficulties.

© 2003 by CRC Press LLC
MATLAB Example
Output Using Function Threemass

strdyneq;

SOLUTION OF THE DIFFERENTIAL EQUATION
M*Y''+C*Y'+K*Y=F1*COS(W*T)+F2*SIN(W*T)

Give the name of a function to create data values
(Try threemass as an example)
>> threemass

Input coordinate number, tmin and tmax
(only press return to stop execution)>> 1,0,50

The value of i*w is at distance 0.050001
from the eigenvalue -0.05+1.4133i

Input coordinate number, tmin and tmax
(only press return to stop execution)>> 2,0,50

Input coordinate number, tmin and tmax
(only press return to stop execution)>>

The system eigenvalues are:

lam =

-0.0146 - 0.7652i
-0.0146 + 0.7652i
-0.0500 - 1.4133i
-0.0500 + 1.4133i
-0.0854 - 1.8458i
-0.0854 + 1.8458i

Range of solution values for final times is:

maxy =

6.4255 0.0000 6.4935

miny =

© 2003 by CRC Press LLC
Figure 3.5: Motion of Mass 1 in Three-Mass Model
Figure 3.6: Motion of Mass 2 in Three-mass Model
Motion of First and Second Mass

MATLAB Code

```matlab
function [t,y,lambda]=strdyneq

% This program integrates the structural dynamics characterized by a general second order
% matrix differential equation having a harmonic forcing function. Input involves mass, stiffness,
% and damping matrices as well as force magnitudes, forcing frequency, and initial conditions. Data
% parameters for the program are created in a user supplied function provided by the user. (For an
% example, see function threemass shown below.)

title=['
SOLUTION OF THE DIFFERENTIAL EQUATION
','M*Y''''+C*Y''+K*Y=F1*COS(W*T)+F2*SIN(W*T)

'];
fprintf(title);

disp( ... 
'Give the name of a function to create data values')
disp(' (Try threemass as an example)');
name=input('>? ','s');
eval(['[m,c,k,f1,f2,w,nt,y0,v0]=',name,';']); jj=1;
while 1
    fprintf('
Input coordinate number, tmin and tmax')
    fprintf('
(only press return to stop execution)');
    [j,t1,t2]=inputv('>? '); j=
    if isnan(j), break; end; J=int2str(j);
    [t,y,lambda]=fhrmck(m,c,k,f1,f2,w,[j,t1,t2],nt,y0,v0);
    if isnan(t), return, end
    [dif,h]=min(abs(lambda-i*w)); lj=num2str(lambda(h));
    if jj=1, jj=jj+1; disp(' '); disp(['The value of i*w is at distance ',num2str(dif)]))
    disp(['from the eigenvalue ',lj])
end
plot(t,y(:,j),'k-'), xlabel('time')
ylabel(['y(',J,')'])
title(['RESPONSE VARIABLE NUMBER ',J])
grid on, shg, dumy=input(' ','s');
```
fprintf('The system eigenvalues are:\n')
display(lam)
fprintf(...)  
'Range of solution values for final times is:\n')
maxy=max(y); miny=min(y); display(maxy)
display(miny), fprintf('All done\n')

%==============================
function [m,c,k,f1,f2,w,nt,y0,v0]=threemass
  % [m,c,k,f1,f2,w,nt,y0,v0]=threemass
  % This function creates data for a three mass
  % system. The name of the function should be
  % changed to specify different problems. However,
  % the output variable list should remain unchanged
  % for compatibility with the data input program.
  m=eye(3,3); k=[2,-1,0;-1,2,-1;0,-1,2]; c=.05*k;

  % Data to excite the highest mode
  f1=[-1;0;1]; f2=[0;0;0]; w=1.413; nt=1000;

  % Data to excite the lowest mode
  f1=[1;1;1]; f2=[0;0;0]; w=.7652; nt=1000;

  % Homogeneous initial conditions
  y0=[-.5;0;.5]; v0=zeros(3,1); y0=0*y0;

%=================================
function [t,y,lam]=fhrmck(m,c,k,f1,f2,w,tlim,nt,y0,v0)
  % [t,y,lam]=fhrmck(m,c,k,f1,f2,w,tlim,nt,y0,v0)
  % This function uses eigenfunction analysis to solve
  % the matrix differential equation
  % m*y''(t)+c*y'(t)+k*y(t)=f1*cos(w*t)+f2*sin(w*t)
  % with initial conditions of y(0)=y0, y'(0)=v0
  % The solution is general unless 1) a zero or repeated
  % eigenvalue occurs or 2) the system is undamped and
  % the forcing function matches a natural frequency.
  % If either error condition occurs, program execution
  % terminates with t and y set to nan.
% m,c,k - mass, damping, and stiffness matrices
%f1,f2 - amplitude vectors for the sine and cosine forcing function components
% w - frequency of the forcing function
% tlim - a vector containing the minimum and maximum time limits for evaluation of the solution
% nt - the number of times at which the solution is evaluated within the chosen limits
% y0,v0 - initial position and velocity vectors
%
% t - vector of time values for the solution
% y - matrix of solution values where y(i,j) is the value of component j at time t(i)
% lam - the complex natural frequencies arranged in order of increasing absolute value

if nargin==0 % Generate default data using 2 masses
m=eye(2,2); k=[2,-1;-1,1]; c=.3*k;
f1=[0;1]; f2=[0;0]; w=0.6; tlim=[0,100]; nt=400;
end
n=size(m,1); t=linspace(tlim(1),tlim(2),nt);
if nargin<10, y0=zeros(n,1); v0=y0; end

% Determine eigenvalues and eigenvectors for the homogeneous solution
A=[zeros(n,n), eye(n,n); -m\[k, c]
[U,lam]=eig(A); [lam,j]=sort(diag(lam)); U=U(:,j);

% Check for zero or repeated eigenvalues and for undamped resonance
wmin=abs(lam(1)); tol=wmin/1e6;
[dif,J]=min(abs(lam-i*w)); lj=num2str(lam(J));
if wmin==0, disp('The homogeneous equation has a zero eigenvalue which is not allowed.
Execution is terminated'), disp(' Return
elseif any(abs(diff(lam))<tol)
 disp('A repeated eigenvalue occurred.
 disp(' Execution is terminated'), disp(' 
 t=nan; y=nan; return
elseif dif<tol & sum(abs(c(:)))==0
 disp('A repeated eigenvalue occurred.
 disp(' Execution is terminated'), disp(' 
 t=nan; y=nan; return
© 2003 by CRC Press LLC
\[
T(U_{xx} + U_{yy}) = \rho U_{tt}, \quad (x,y) \in R,
\]
\[
U(x,y,t) = 0, \quad (x,y) \in L,
\]

3.6 Computing Natural Frequencies for a Rectangular Membrane

One of the most useful applications of eigenvalue problems occurs in natural frequency calculations for linear systems. Let us examine finite difference approximation for the natural frequencies of a rectangular membrane and how well the approximate results compare with exact values. Consider a tightly stretched elastic membrane occupying a region \( R \) in the \((x,y)\) bounded by a curve \( L \) on which the transverse deflection is zero. The differential equation and boundary conditions governing the transverse motion \( U(x,y,t) \) are

\[
T(U_{xx} + U_{yy}) = \rho U_{tt}, \quad (x,y) \in R,
\]
\[
U(x,y,t) = 0, \quad (x,y) \in L,
\]
where $T$ and $\rho$ denote membrane tension and mass density. The natural vibration modes are motion states where all points of the system simultaneously move with the same frequency, which says $U(x, y, t) = u(x, y) \sin(\Omega t)$. It follows that $u(x, y)$ satisfies

$$u_{xx} + u_{yy} = -\omega^2 u, \quad (x, y) \in \mathbb{R},$$
$$u(x, y) = 0, \quad (x, y) \in \mathbb{L},$$

where $\omega = \sqrt{\frac{T}{\rho}}$. In the simple case of a rectangular membrane lying in the region such that $0 \leq x \leq a$ and $0 \leq y \leq b$, the natural frequencies and mode shapes turn out to be

$$\omega_{nm} = \sqrt{\left(\frac{n\pi}{a}\right)^2 + \left(\frac{m\pi}{b}\right)^2}, \quad u_{nm} = \sin\left(\frac{n\pi x}{a}\right) \sin\left(\frac{m\pi y}{b}\right)$$

where $n$ and $m$ are positive integers. It is interesting to see how closely these values can be reproduced when the partial differential equation is replaced by a second order finite difference approximation defined on a rectangular grid. We introduce grid points expressed as

$$x(i) = (i - 1)\Delta x, \quad i = 1, \ldots, N,$$
$$y(j) = (j - 1)\Delta y, \quad j = 1, \ldots, M,$$

where

$$\Delta x = a/(N - 1), \quad \Delta y = b/(M - 1),$$

and we call $u(i, j)$ the value of $u$ at $x(i), y(j)$. Then the Helmholtz equation is replaced by an algebraic eigenvalue problem of the form

$$\Delta^2_x [u(i - 1, j) - 2u(i, j) + u(i + 1, j)] + \Delta^2_y[u(i, j - 1) - 2u(i, j) + u(i, j + 1)] = \lambda u(i, j)$$

where

$$\lambda = (\Delta x \Delta y \omega)^2$$

and associated homogeneous boundary conditions

$$u(1, j) = u(N, j) = u(i, 1) = u(i, M) = 0.$$ 

This combination of equations can be rearranged into familiar matrix form as

$$Au = \lambda u, \quad Bu = 0.$$ 

The MATLAB function `null` can be used to solve the boundary condition equations. We write as $u = Qz$ where $Q = \text{null}(B)$ has orthonormal columns. Substituting into the eigenvalue equation and multiplying both sides by $Q'$ then yields a standard eigenvalue problem of the form $Cz = \lambda z$ where $C = Q'AQ$. Denoting the eigenvector matrix of $C$ by $V$, the eigenvector matrix of the original problem is obtained as $u = QV$, and the desired eigenvalues are simply those of matrix $C$.
A short function named `recmemfr` was written to form and solve the algebraic equations just discussed. Although the ideas are simple, indexing the double indexed quantities pertaining to the finite difference grid is slightly tedious. Intrinsic functions `ind2sub` and `sub2ind` are helpful to perform the indexing. Lines 32-34 of `recmemfr` compute a subset of the lowest frequency values and sort these in ascending order. Lines 37-45 form the homogeneous boundary conditions, and lines 51-56 construct the discretized Helmholtz equation at interior node points. The main computation work is done in lines 59-61 where `null` and `eig` are used. Finally, the results are sorted, the modal arrays are reshaped, and results are plotted to compare the approximate and exact frequencies. In the graph shown below for the case where $(a,b)=(2,1)$, the frequencies obtained using the finite differences are seen to be consistently low. Furthermore, the 50'th frequency is off by about 14 percent, even though 200 grid points were used. Applications leading to eigenvalue problems occur frequently. The ideas touched on in this simple example will be encountered again in Chapters 9 and 10. Readers may find it interesting to modify this example using a higher order difference approximation to see how much the frequency estimates improve.

![Graph](image)

**Figure 3.7:** Approximate and Exact Frequencies for a Rectangular Membrane

© 2003 by CRC Press LLC
function recmemfr

function [w,wex,modes,x,y,nx,ny,ax,by]=recmemfr(......ax,by,nx,ny,noplt)

% [w,wex,modes,x,y,nx,ny,ax,by]=recmemfr(a,b,nx,ny,noplt)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function employs finite difference methods to
% estimate the natural frequencies and mode shapes of
% a rectangular membrane having fixed edges.
% ax, by - membrane side lengths along the x and y axes
% nx,ny - number of finite difference points taken in
% the x and y directions including the edges
% w - vector of (nx-2)*(ny-2) frequencies obtained
% by finite difference approximation of the
% wave equation. These are arranged in
% increasing order
% wex - vector of exact frequencies
% modes - three dimensional array containing the mode
% shapes for various frequencies. The array
% size is [nx,ny,(nx-2)*(nx-2)] denoting
% the x direction, y direction, and the
% frequency numbers matching components of the
% w vector. The i'th mode shape is obtained
% as reshape(vecs(:,i),n,m)
% x,y - vectors defining the finite difference grid
% noplt - optional parameter included if no plot of
% the approximate and exact frequencies is to
% be made

if nargin==0; ax=2; nx=20; by=1; ny=10; end
dx=ax/(nx-1); dy=by/(ny-1);
na=(1:nx-1)'/ax; nb=(1:ny-1)/by;

% Compute exact frequencies for comparison
wex=pi*sqrt(repmat(na.^2,1,ny-1)+repmat(nb.^2,nx-1,1));
wex=sort(wex(:)'); x=linspace(0,ax,nx); y=linspace(0,by,ny); neig=(nx-2)*(ny-2); nvar=nx*ny;

% Form equations to fix membrane edges
k=0; s=[nx,ny]; c=zeros(2*(nx+ny),nvar);
for j=1:nx
m=sub2ind(s,[j,j],[1,ny]); k=k+1;

© 2003 by CRC Press LLC
3.7 Column Space, Null Space, Orthonormal Bases, and SVD

One remaining advanced topic discussed in this chapter is the factorization known as singular value decomposition, or SVD. We will briefly explain the structure of
SVD and some of its applications. It is known that any real matrix having \( n \) rows, \( m \) columns, and rank \( r \) can be decomposed into the form

\[
A = USV' \]

where

- \( U \) is an orthogonal \( n \) by \( n \) matrix such that \( U'U = I \)
- \( V \) is an orthogonal \( m \) by \( m \) matrix such that \( V'V = I \)
- \( S \) is an \( n \) by \( m \) diagonal matrix of the form

\[
S = \begin{bmatrix}
\sigma_1 & 0 & 0 & 0 & 0 \\
0 & \sigma_2 & 0 & 0 & 0 \\
0 & 0 & \ddots & 0 & 0 \\
0 & 0 & 0 & \sigma_r & 0 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

where \( \sigma_1, \ldots, \sigma_r \) are positive numbers on the main diagonal with \( \sigma_i \geq \sigma_{i+1} \). Constants \( \sigma_j \) are called the singular values with the number of nonzero values being equal to the rank \( r \).

To understand the structure of this decomposition, let us study the case where \( n \geq m \). Direct multiplication gives

\[
A'AV = V \text{diag}(\sigma_1^2, \ldots, \sigma_r^2, \text{zeros}(1, m - r)),
\]

and

\[
AA'U = U \text{diag}(\sigma_1^2, \ldots, \sigma_r^2, \text{zeros}(1, n - r)).
\]

Consequently, the singular values are square roots of the eigenvalues of the symmetric matrix \( A'A \). Matrix \( V \) contains the orthonormalized eigenvectors arranged so that \( \sigma_i \geq \sigma_{i+1} \). Although the eigenvalues of \( A'A \) are obviously real, it may appear that this matrix could have some negative eigenvalues leading to pure imaginary singular values. However, this cannot happen because \( A'AY = \lambda Y \) implies \( \lambda = (AY)'(AY)/(Y'Y) \), which clearly is nonnegative. Once the eigenvectors and eigenvalues of \( A'A \) are computed, columns of matrix \( U \) can be found as orthonormalized solutions of

\[
[A'A - \sigma_j I]U_j = 0, \quad \sigma_j = 0, \quad j > r.
\]

The arguments just presented show that performing singular value decomposition involves solving a symmetric eigenvalue problem. However, SVD requires additional computation beyond solving a symmetric eigenvalue problem. It can be very time consuming for large matrices. The SVD has various uses, such as solving the normal
equations. Suppose an \( n \) by \( m \) matrix \( A \) has \( n > m \) and \( r = m \). Substituting the SVD into
\[
A'AX = A'B
\]
gives
\[
V \text{ diag}(\sigma_1^2, \ldots, \sigma_n^2)V'X = VS'U'B.
\]
Consequently, the solution of the normal equations is
\[
X = V \text{ diag}(\sigma_1^{-2}, \ldots, \sigma_m^{-2})S'U'B.
\]

Another important application of the SVD concerns generation of orthonormal bases for the column space and the row space. The column space has dimension \( r \) and the null space has dimension \( m - r \). Consider a consistent system
\[
AX = B = U(SV'X).
\]
Denote \( SV'X \) as \( Y \) and observe that \( y_j = 0 \) for \( j > r \) since \( \sigma_j = 0 \). Because \( B \) can be any vector in the column space, it follows that the first \( r \) columns of \( U \), which are also orthonormal, are a basis for the column space. Furthermore, the decomposition can be written as
\[
AV = US.
\]
This implies
\[
AV_j = U_j\sigma_j = 0, \ j > r
\]
which shows that the final \( m - r \) columns of \( V \) form an orthonormal basis for the null space. The reader can verify that bases for the row space and left null space follow analogously by considering \( A' = VS'U' \), which simply interchanges the roles of \( U \) and \( V \).

MATLAB provides numerous other useful matrix decompositions such as LU, QR, and Cholesky. Some of these are employed in other sections of this book. The reader will find it instructive to read the built-in help information for MATLAB functions describing these decomposition methods. For instance, the command \texttt{help \} gives extensive documentation on the operation for matrix inversion.

### 3.8 Computation Time to Run a MATLAB Program

MATLAB is designed to perform matrix computation with maximum speed and accuracy. Consequently, most standard operations like matrix multiplication, Gauss reduction, eigenvalue calculation, SVD, etc. are implemented as highly optimized and compiled intrinsic functions. Efficient program execution requires optimal use of the built-in functions. Executing nested loops can take a lot of time, so using coding with nested loops should be avoided when computation time is important. To
illustrate how deeply nested loops can slow down execution speed we will compare slow multiplication of square matrices by a Fortran style triple loop, and fast multiplication using the intrinsic matrix multiply capability. The ratio of the slow time to the fast time is much larger than might initially be expected.

Before proceeding with our example, consider the difficulties of accurately timing a computational process. In the first place, the clock in Intel based systems has a resolution of about 0.06 sec, whereas the time for MATLAB to do a 100 by 100 matrix multiply is about 0.005 secs on a 733 Mhz Pentium 4 computer. This implies that, just to account for the crude clock increment, the matrix multiply has to be repeated at least 1200 times to get a total time accurate within one percent. However, this is not the only timing difficulty. MATLAB continuously performs housekeeping tasks such as memory management. The operating system and other programs running simultaneously in the background also use computer resources and affect recorded times. Hence, any timing of algorithmic processes in MATLAB should be done without having several other programs open. Even then, the authors have found that times recorded for the same computation done repeatedly often vary around five percent.

The following program named mattimer was written to compare slow and fast matrix multiplication. The program input includes the matrix order, the number of seconds a loop is performed to improve timing accuracy, and the number of times the basic timing operation is repeated to show how recorded times vary among successive computations. The program also gives the number of floating operations performed per second (Mflops). An $n$ by $n$ matrix multiply involves $n^2$ dot products each requiring $n$ adds and $n$ multiplies. Hence, the number of floating point operations is $2n^3$. An order 100 matrix multiply done in 0.005 seconds would give 400 megaflops. Function multimer does the matrix multiply repeatedly and reads the elapsed time until the specified total number of seconds is reached. Performing loops and reading the clock takes some time, which is subtracted from the time to do the looping, matrix multiplication, and clock reading. We also perform the intrinsic matrix multiplication in a separate function so that both the fast and slow methods have the same computational overhead associated with a function call. Results are shown for matrices of order 100 and 1000. The fast time for an order 100 matrix multiply only took 0.00503 seconds giving 398 megaflops. By comparison, the slow method took more than eighteen hundred times as long as the fast method. This is comparable to making a one hour task take about two and a half months, working twenty-four hours a day, seven days a week. Evidently, intrinsic MATLAB matrix multiply works very well, but nested looping is slow. Something else worth noting is that a dense matrix of order 1000 does not stretch the capabilities of a modern microcomputer. Storing a million word double-precision array only takes 8 megabytes of RAM, which is a small fraction of the 128 megabytes or more typically provided for scientific work. Furthermore, the high order matrix multiply only took 4.6 seconds, which is roughly 1000 times as long as the order 100 time. It turns out that the time needed for most matrix operations increases like the cube of the order, even though a complicated calculation such as singular value decomposition may take around seventeen times as long as a Gauss elimination of the same order.
>> mattimer(100,10,60);

MATRX MULTIPLY TIMING TEST

Get results for a single timer call

The repeated multiplication of matrices of order 100 may take considerable time.

Fast multiply takes 0.0050238 secs. 
Megaflps = 398.1034

Slow multiply takes 9.0714 secs. 
Megaflps = 398.1034

tslow/tfast = 1805.6723

Get results for several timer calls

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tfast</td>
<td>tslow</td>
</tr>
<tr>
<td>5.0473e-003</td>
<td>8.8899e+000</td>
</tr>
<tr>
<td>5.0248e-003</td>
<td>8.8271e+000</td>
</tr>
<tr>
<td>4.9948e-003</td>
<td>8.9685e+000</td>
</tr>
<tr>
<td>5.0075e-003</td>
<td>8.8742e+000</td>
</tr>
<tr>
<td>5.3775e-003</td>
<td>8.9599e+000</td>
</tr>
<tr>
<td>4.9939e-003</td>
<td>8.8499e+000</td>
</tr>
<tr>
<td>5.0217e-003</td>
<td>8.9842e+000</td>
</tr>
<tr>
<td>5.0182e-003</td>
<td>9.0785e+000</td>
</tr>
<tr>
<td>4.9905e-003</td>
<td>8.9598e+000</td>
</tr>
</tbody>
</table>

Time variation defined by (max(t)-min(t))/mean(t)
Variation for tfast = 0.076656
Variation for tslow = 0.028181

>> mattimer(1000,0,60);

MATRX MULTIPLY TIMING TEST

Get results for a single timer call

The repeated multiplication of matrices of order 1000 may take considerable time.

Fast multiply takes 4.5699 secs.
Megaflops = 437.6421

Slow multiply takes 8882.3899 secs.
Megaflops = 0.22516

tslow/tfast = 1943.654

Program mattimer

function mattimer(norder,ktimes,secs)

\% mattimer(norder,ktimes,secs)
\% --------------------------------
\% if nargin==0
\% norder=100; ktimes=10; secs=30;
\% end
fprintf('
MATRIX MULTIPLY TIMING TEST

')
disp('Get results for a single timer call')
multimer(norder,secs,1); t=zeros(ktimes,3);
secs=max(secs,30); if ktimes==0, return, end
disp('Get results for several timer calls')
for j=1:ktimes
[t(j,3),t(j,1),t(j,2)]=multimer(norder,secs);
end
T=(max(t)-min(t))./mean(t);
disp(...
' tfast tslow ratio')
for j=1:ktimes
fprintf('%13.4e %13.4e %13.4e
',t(j,:))
end
disp(' '), disp(...
'Time variation defined by (max(t)-min(t))/mean(t)'
disp(['Variation for tfast = ',num2str(T(1))])
disp(['Variation for tslow = ',num2str(T(2))])
\%============================================
function [ratio,tfast,tslow]=multimer(...
    norder,secs,doprint)

% This function compares the times to perform a
% matrix multiply using the built-in matrix multiply
% and the slow method employing scalar triple looping.
% The ratio of compute times illustrates how much
% faster compiled and vectorized matrix operations
% can be compared to similar calculations using
% interpreted code with scalar looping.
% norder - order of the test matrices used. The
%    default for norder is 100.
% secs - number of seconds each computation is run
%    to get accurate timing. The default (and
%    minimum value) is thirty seconds.
% doprint - print intermediate results only if this
%    variable is given a value
% ratio - ratio of slow to fast multiply times
% tfast - time in seconds to perform a multiply
%    using the built-in precompiled matrix
% tslow - time in seconds to perform a multiply
%    by triple loop method
% User m functions called: matmultf matmults

% Typical results obtained using a Dell Dimension
% XPS B733r computer with 128MB of RAM gave the
% following values:
% >> mattimer(100,0,60);
% MATRIX MULTIPLY TIMING TEST
% Fast multiply takes 0.0050238 secs.
% Megaflops = 398.1034
% Slow multiply takes 9.0714 secs.
% Megaflops = 398.1034
% tslow/tfast = 1805.6723
% >> mattimer(1000,0,60);
% MATRIX MULTIPLY TIMING TEST 
% Fast multiply takes 4.5699 secs. 
% Megaflops = 437.6421 
% Slow multiply takes 8882.3899 secs.
% Megaflops = 0.22516
% tslow/tfast = 1943.654
% >>

% Find time to make a loop and call the clock
nmax=5e3; nclock=0; tstart=cputime;
while nclock<nmax
    tclock=cputime-tstart; nclock=nclock+1;
end
% Time to do one loop and call the timer
tclock=tclock/nclock;

if nargin<3, doprint=0; else, doprint=1; end
if nargin<2, secs=30; end; secs=max(secs,30);
if nargin==0, norder=100; end
a=rand(norder,norder); b=rand(norder,norder);

if doprint
    disp('')
    disp('The repeated multiplication of matrices')
    disp(['of order ',num2str(norder),...
         ' may take considerable time.'])
    disp('')
end

% Time using intrinsic multiply function
pack; tfast=0; nfast=0; tstart=cputime;
while tfast<secs
    cf=matmultf(a,b); nfast=nfast+1;
    tfast=cputime-tstart;
end
tfast=tfast/nfast-tclock;

% Time using Fortran style, triple for:next looping
pack; tslow=0; nslow=0; tstart=cputime;
while tslow<secs
123:    cs=matmults(a,b); nslow=nslow+1;
124:    tslow=cputime-tstart;
125:    end
126:
127:    tslow=tslow/nslow-tclock; ratio=tslow/tfast;
128:    mflops=inline('num2str(2*n^3/1e6/t)','n','t');
129:    if doprint
130:        disp(['Fast multiply takes ','...
131:        num2str(tfast),' secs.'])
132:        disp(['Megaflops = ',...
133:        mflops(norder,tfast)]), disp(' ')
134:        disp(['Slow multiply takes ','...
135:        num2str(tslow),' secs.'])
136:        disp(['Megaflops = ',...
137:        mflops(norder,tslow)]), disp(' ')
138:        disp(['tslow/tfast = ',...
139:        num2str(tslow/tfast)]), disp(' ')
140:    end
141:
142:    function v=matmultf(a,b)
143:    % v=matmultf(a,b). Matrix multiply using
144:    % precompiled function in MATLAB
145:    v=a*b;
146:    %===========================================
147:
148:    function v=matmults(a,b)
149:    % v=matmults(a,b). Matrix multiply using
150:    % Fortran like triple loop
151:    n=size(a,1); m=size(b,2); K=size(a,2);
152:    v=zeros(n,m);
153:    for i=1:n
154:        for j=1:m
155:            t=0;
156:                for k=1:K
157:                    t=t+a(i,k)*b(k,j);
158:                end
159:            v(i,j)=t;
160:        end
161:    end
162:    %===========================================
163:    %===========================================
164:

© 2003 by CRC Press LLC
Chapter 4

Methods for Interpolation and Numerical Differentiation

4.1 Concepts of Interpolation

Next we study three types of one-dimensional interpolation: polynomial, piecewise linear, and cubic spline. The MATLAB functions implementing these methods are discussed along with some additional software developed by the authors to differentiate and integrate splines. A simple discussion of cubic spline interpolation formulated from the viewpoint of elastic beam flexure is given. The chapter concludes with a program to compute finite difference formulas for derivatives of general order.

Interpolation is a process whereby a function is approximated using data known at a discrete set of points. Typically we have points \((x_i, y_i)\) arranged such that \(x_i < x_{i+1}\). These points are to be connected by a continuous interpolation function influenced by smoothness requirements such as: a) the function should not deviate greatly from the data at points lying between the data values; and b) the function should satisfy a differentiability condition such as continuity of first and second derivatives.

Piecewise linear interpolation simply connects successive points by straight lines. This has the disadvantage of producing a function with piecewise constant slope and finite slope discontinuities. An obvious cure for slope discontinuity is to use a curve such as a polynomial of degree \(n-1\) (through \(n\) points) to produce an interpolation function having all derivatives continuous. However, it was seen in Section 2.3 that a polynomial passing exactly through the data points may be highly irregular at intermediate values. Using polynomial interpolations higher than order five or six often produces disappointing results. An excellent alternative to allowing either slope discontinuities or demanding slope continuity of all orders is to use cubic spline interpolation. This method connects successive points by cubic curves joined such that function continuity as well as continuity of the first two function derivatives is achieved.

The MATLAB function \texttt{polyfit}(xd,yd,n) can be used to obtain coefficients in a polynomial of degree \(n\) which either passes through points in data vectors \((xd,yd)\) or fits the data in the least square sense. Since a polynomial of degree \(n-1\) can pass through \(n\) data points, the computation \(c=polyfit(xd,yd,length(xd)-1)\) would produce coefficients in a polynomial passing through the data values. Evaluating the polyno-
mial for an array argument x is accomplished by \( y = \text{polyval}(c, x) \). Combining the two operations gives \( y = \text{polyval}(\text{polyfit}(x, y, \text{length}(x) - 1), x) \). If the chosen polynomial order is less than \( \text{length}(x) - 1 \), then a polynomial fitting the data in the least square sense is produced. For example, a polynomial of order 4 might be fitted to several hundred points. Of course, how well the least square polynomial actually fits the data should be assessed by examining a plot of the curve and the data. MATLAB also has various utility functions to work with polynomials such as \texttt{polyder}, \texttt{polyint}, \texttt{conv}, and \texttt{deconv} which differentiate, integrate, multiply, and divide.

Function \texttt{interp1}(x, y, x, 'method', 'extrap') is a general purpose interpolation function providing several types of interpolation including linear and spline. The default value for 'method' is 'linear'. If the 'extrap' parameter is omitted, then a value of NaN (not a number) is returned for any input argument not lying between min(x) and max(x). Otherwise, extrapolation is performed using the interpolation functions for the outermost intervals. Readers should be cautious about extrapolating far outside the known data range, because this often leads to unreasonable results.

Engineering applications often use idealized functions which are piecewise linear and have finite jump discontinuities. Since function \texttt{interp1} rejects cases where any successive values in the x vector are equal, we remedy this situation with function \texttt{lintrp}(x, y, x) to search x for any repeated values and separate these values by a small fraction of max(x) - min(x). Then \texttt{interp1} is used to perform the interpolation as indicated below.

Function \texttt{lintrp}

```matlab
function y = lintrp(x, y, x)
%
% y = lintrp(x, y, x)
% --------------------------------
% This function performs piecewise linear
% interpolation through data values stored in
% x, y, where x values are arranged in
% nondecreasing order. The function can handle
% discontinuous functions specified when some
% successive values in x are equal. Then the
% repeated x values are shifted by a small
% amount to remove the discontinuities.
% Interpolation for any points outside the range
% of x is also performed by continuing the line
% segments through the outermost data pairs.
%
% x, y - vectors of interpolation data values
% x - matrix of values where interpolated
```
4.2 Interpolation, Differentiation, and Integration by Cubic Splines

Cubic spline interpolation is a versatile method to pass a smooth curve through a sequence of data points. The technique connects the data values with a curve having its third derivative piecewise constant. The curve is piecewise cubic with \( y(x), y'(x) \) and \( y''(x) \) continuous over the whole data range. The MathWorks markets a Spline Toolbox providing extensive capabilities to work with spline functions. A few functions from that toolbox are included in standard MATLAB. The intrinsic functions \texttt{ spline, ppval, mkpp}, and \texttt{ unmkpp} are extended here to handle differentiation and integration. Spline interpolation, viewed from Euler beam theory, is also discussed to amplify on the basic ideas. This simple formulation easily accommodates various end conditions. Readers wanting more detail on spline theory will find the books by de Boor [27] and by Ahlberg and Nilson [2] to be helpful.

Cubic spline theory is motivated by a mechanical drafting tool consisting of a flexible strip bent over several supports with heights adjustable to fit given data. Euler beam theory [9] shows that the deflection curve has third derivative values which are constant between successive supports. This implies that the curve is piecewise cubic and the third derivative values (relating to internal shear forces in beam analysis) can be determined to make the support deflections have chosen values. This is the basis of cubic spline interpolation. The method is attractive because the interpolation function \( y(x) \) is obtainable analytically as well as \( y'(x), y''(x) \) and \( \int y(x)dx \).

Let us formulate the problem mathematically by taking a piecewise constant form for \( y'''(x) \) and integrating this repeatedly to get \( y(x) \). We assume data points \((x_i, y_i), 1 \leq i \leq n \) with \( x_i < x_{i+1} \). Each successive data pair can be connected by a cubic curve with \( y'(x) \) and \( y''(x) \) required to be continuous at all interior data points. If values of \( y'(x) \) or \( y''(x) \) are known at the curve ends, algebraic conditions to impose those values can be written. Using known values of end slope is appropriate, but specifying good second derivative values when end slopes are not known is usually not obvious. As an alternative, it is customary to apply smoothness conditions requiring continuity of \( y'''(x_2) \) and \( y'''(x_{n-1}) \). Books on spline theory [7, 2] refer to imposition of higher order continuity at interior points as "not-a-knot" conditions.
The piecewise constant third derivative of the interpolation function is described as

\[
y'''(x) = \sum_{j=1}^{n-1} c_j < x - x_j >^0
\]

where \(c_j\) are constants to be determined, and the singularity function

\[
<x - a >^n = (x - a)^n(x > a)
\]

is used. This formula for \(y'''(x)\) is easy to integrate, and making the curve pass through the data points is straightforward. It follows that

\[
y''(x) = y''_1 + \sum_{j=1}^{n-1} c_j < x - x_j >^1,
\]

\[
y'(x) = y'_1 + y''_1(x - x_1) + \frac{1}{2} \sum_{j=1}^{n-1} c_j < x - x_j >^2,
\]

\[
y(x) = y_1 + y'_1(x - x_1) + \frac{1}{2} y''_1(x - x_1)^2 + \frac{1}{6} \sum_{j=1}^{n-1} c_j < x - x_j >^3,
\]

\[
\int_{x_1}^{x} y(x) dx = y_1(x - x_1) + \frac{1}{2} y'_1(x - x_1)^2 + \frac{1}{6} y''_1(x - x_1)^3
\]

\[
+ \frac{1}{24} \sum_{j=1}^{n-1} c_j < x - x_j >^4.
\]

The interpolation function automatically goes through the first data point, and the remaining constants are required to satisfy

\[
y_i - y_1 = y'_1(x_i - x_1) + \frac{1}{2} y''_1(x_i - x_1)^2 + \frac{1}{6} \sum_{j=1}^{n-1} c_j < x_i - x_j >^3, \quad i = 2, 3, \ldots, n.
\]

Since \(n + 1\) unknowns are present in the above system, two more end conditions must be included. Five familiar combinations of end conditions include: 1) the "not-a-knot" condition applied at each end; 2) the slope given at each end; 3) the slope given at the left end and the "not-a-knot" condition at the right end; 4) the "not-a-knot" condition at the left end and the slope given at the right end; and 5) a periodic spline is created by making the first and last points have the same values of \(y, y',\) and \(y''.\)

Spline interpolation involves solution of linear simultaneous equations. A desktop computer solves a system of 200 equations in less than 0.03 seconds; so, the equation solving time is modest unless many points are used. The formulation described above is easy to understand, handles general end conditions, and includes interpolation,
differentiation, and integration. It was implemented in two general purpose functions spterp and spcof used below with function curvprop to compute the length and area bounded by a spline curve. Another function splineg using intrinsic function spline is also discussed at the end of the present article. The spline routines provided here are helpful additions for work with splines since they include spline differentiation and integration which do not come in the standard MATLAB package.

4.2.1 Computing the Length and Area Bounded by a Curve

The ideas just described were implemented in functions spterp and spcof which are called in the following program curvprop. This program computes the length of a spline curve and the area bounded by the curve. The length of a curve parameterized in complex form as

\[ z(t) = x(t) + iy(t), \quad a \leq t \leq b \]

can be computed as

\[ \text{length} = \int_a^b \text{abs}(z'(t)) \, dt. \]

Furthermore, when the curve is closed and is traversed in a counterclockwise direction, the area is given by

\[ \text{area} = \frac{1}{2} \int_a^b \text{imag}(\text{conj}(z(t)) \cdot z'(t)) \, dt. \]

The curve length is meaningful for an open or closed curve, but the bounded area only makes sense for a closed curve. The next chapter discusses area properties for shapes bounded by several spline curves. Our present example assumes a simple geometry. It is worth mentioning that applying the last integral to an open curve gives the area enclosed within the curve combined with a line from the last point to the origin and a line from the origin to the first point. This fact is clarified in the next chapter which treats general areas bounded by several spline curves.

The following program curvprop passes a spline curve through data in vectors x,y. The length, bounded area, and a set of data points on the curve are computed. The curve is assumed to have a smoothly turning tangent. The default data example uses points on an ellipse with semi-diameters of two and one. Readers can verify that approximating the ellipse with a 21 point spline curve gives an area approximation accurate within 0.0055 percent and a boundary length accurate within 0.0068 percent. Of course, better accuracy is achievable with more data points.

4.2.2 Example: Length and Enclosed Area for a Spline Curve

Function curvprop

```matlab
function [area,leng,X,Y,closed]=curvprop(x,y,doplot)
```
%  [area,leng,X,Y,closed]=curvprop(x,y,doplot)
%  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function passes a cubic spline curve through
% a set of data values and computes the enclosed
% area, the curve length, and a set of points on
% the curve.
%  
%  % x,y     - data vectors defining the curve.
%  % doplot  - plot the curve if a value is input for
%             % doplot. Otherwise, no plot is made.
%  % area    - the enclosed area is computed. This
%             % parameter is valid only if the curve
%             % is closed and the boundary is traversed
%             % in counterclockwise. For a curve, the
%             % area agrees with a curve closed using
%             % a line from the last point to the
%             % origin, and a line from the origin to
%             % the first point.
%  % leng    - length of the curve
%  % X,Y     - set of points on the curve. The output
%             % intersperses three points on each segment
%             % between the starting data values.
%  % closed  - equals one for a closed curve. Equals zero
%             % for an open curve.
%  
% For default test data, choose an ellipse with
% semi-diameters of 2 and 1.
%  if nargin==0
%    m=21; th=linspace(0,2*pi,m);
%    x=2*cos(th); y=sin(th); x(m)=2; y(m)=0;
%  end
%
% Use complex data coordinates
%  z=x(:)+i*y(:); n=length(z); t=(1:n)';
%  chord=sum(abs(diff(z))); d=abs(z(n)-z(1));
%
% Use a periodic spline if the curve is closed
%  if d < (chord/1e6)
%    closed=1; z(n)=z(1); endc=5;
%    zp=spterp(t,z,1,t,endc);
%  else
%    % Use the not-a-knot end condition for open curve

47:       closed=0; endc=1; zp=spterp(t,z,1,t,endc);
48:     end
49:
50:     \% Compute length and area
51:     \% plot(abs(zp)),shg,pause
52:     leng=spterp(t,abs(zp),3,n,1);
53:     area=spterp(t,1/2*imag(conj(z).*zp),3,n,1);
54:     Z=spterp(t,z,0,1:1/4:n,endc);
55:     X=real(Z); Y=imag(Z);
56:     if nargin>2
57:         plot(X,Y,'-.',x,y,'.'), axis equal
58:         xlabel('x axis'), ylabel('y axis')
59:         title('SPLINE CURVE'), shg
60:     end
61:
62:     \%============================================
63:     \% function \[v,c\]=spterp(xd,yd,id,x,endv,c)
64:     \%
65:     \% \[v,c\]=spterp(xd,yd,id,x,endv,c)
66:     \% ------------------------------------------
67:     \%
68:     \% This function performs cubic spline interpo-
69:     \% lation. Values of y(x),y'(x),y''(x) or the
70:     \% integral(y(t)*dt, xd(1)..x) are obtained.
71:     \% Five types of end conditions are provided.
72:     \%
73:     \% xd, yd - data vectors with xd arranged in
74:     \%     ascending order.
75:     \% id   - id equals 0,1,2,3 to compute y(x),
76:     \%     y'(x), integral(y(t)*dt,t=xd(1)..x),
77:     \%     respectively.
78:     \% v    - values of the function, first deriva-
79:     \%     tive, second derivative, or integral
80:     \%     from xd(1) to x
81:     \% c    - the coefficients defining the spline
82:     \%     curve. If these values are input from
83:     \%     an earlier computation, then they
84:     \%     are not recomputed.
85:     \% endv  - vector giving the end conditions in
86:     \%     one of the following five forms:
87:     \%     endv=1 or endv omitted makes
88:     \%     c(2) and c(n-1) zero
89:     \%     endv=[2,left_end_slope,...
90:     \%     right_end_slope] to impose slope
% values at each end
% endv=[3,left_end_slope] imposes the
% left end slope value and makes
% c(n-1) zero
% endv=[4,right_end_slope] imposes the
% right end slope value and makes
% c(2) zero
% endv=5 defines a periodic spline by
% making y,y',y'' match at both ends

if nargin<5 | isempty(endv), endv=1; end
n=length(xd); sx=size(x); x=x(:); X=x-xd(1);

if nargin<6, c=spcof(xd,yd,endv); end
C=c(1:n); s1=c(n+1); m1=c(n+2); X=x-xd(1);
if id==0 % y(x)
v=yd(1)+s1*X+m1/2*X.*X+...            
powermat(x,xd,3)*C/6;
elseif id==1 % y'(x)
v=s1+m1*X+powermat(x,xd,2)*C/2;
elseif id==2 % y''(x)
v=m1+powermat(x,xd,1)*C;
else % integral(y(t)*dt, t=xd(1)..x)
v=yd(1)*X+s1/2*X.*X+m1/6*X.^3+...        
powermat(x,xd,4)*C/24;
end
v=reshape(v,sx);

%=================================================================

function c=spcof(x,y,endv)
%
% c=spcof(x,y,endv)
% This function determines spline interpolation
% coefficients consisting of the support
% reactions concatenated with y' and y'', at
% the left end.
% x,y - data vectors of interpolation points.
% Denote n as the length of x.
% endv - vector of data for end conditions
% described in function spterp.

© 2003 by CRC Press LLC
% c - a vector [c(1);...;c(n+2)] where the
% first n components are support
% reactions and the last two are
% values of y'(x(1)) and y''(x(1)).
if nargin<3, endv=1; end
x=x(:); y=y(:); n=length(x); u=x(2:n)-x(1);
a=zeros(n+2,n+2); a(1,1:n)=1;
a(2:n,:)=[powermat(x(2:n),x,3)/6,u,u.*u/2];
b=zeros(n+2,1); b(2:n)=y(2:n)-y(1);
if endv(1)==1 % Force, force condition
  a(n+1,2)=1; a(n+2,n-1)=1;
elseif endv(1)==2 % Slope, slope condition
  b(n+1)=endv(2); a(n+1,n+1)=1;
b(n+2)=endv(3); a(n+2,:)=...
  [((x(n)-x').^2)/2,1,x(n)-x(1)];
elseif endv(1)==3 % Slope, force condition
  b(n+1)=endv(2); a(n+1,n+1)=1; a(n+2,n-1)=1;
elseif endv(1)==4 % Force, slope condition
  a(n+1,2)=1; b(n+2)=endv(2);
a(n+2,:)=[((x(n)-x').^2)/2,1,x(n)-x(1)];
elseif endv(1)==5
  a(n+1,1:n)=x(n)-x'; b(n)=0;
a(n+2,1:n)=1/2*(x(n)-x').^2;
a(n+2,n+2)=x(n)-x(1);
else
  error('Invalid value of endv in function spcof')
end
if endv(1)==1 & n<4, c=pinv(a)*b;
else, c=a\b; end

%==============================================
% % % % % % % % % % % % % % % %

function a=powermat(x,X,p)
% a=powermat(x,X,p)
% This function evaluates various powers of a
% matrix used in cubic spline interpolation.
% x,X - arbitrary vectors of length n and N
% a - an n by M matrix of elements such that
% a(i,j)=(x(i)>X(j))*abs(x(i)-X(j))^p
4.2.3 Generalizing the Intrinsic Spline Function in MATLAB

The intrinsic MATLAB function `spline` employs an auxiliary function `unmk` to create the piecewise polynomial definitions defining the spline. The polynomials can be differentiated or integrated, and then functions `mkpp` and `ppval` can be used to evaluate results. We have employed the ideas from those routines to develop functions `splineg` and `splincof` extending the minimal spline capabilities of MATLAB. The function `splincof(xd,yd,endc)` computes arrays b and c usable by `mkpp` and `ppval`. The data vector endc defines the first four types of end conditions discussed above. The function `splineg(xd,yd,deriv,endc,b,c)` handles the same kind of data as function `spterp`. Sometimes arrays b and c may have been created from a previous call to `splineg` or `spterp`. Whenever these are passed through the call list, they are used by `splineg` without recomputation. Readers wanting more details on spline concepts should consult de Boor’s book [7].

Two examples illustrating spline interpolation are presented next. In the first program called, `sinetri`, a series of equally spaced points between 0 and $2\pi$ is used to approximate $y = \sin(x)$ which satisfies

$$ y'(x) = \cos(x) , \quad y''(x) = -\sin(x) , \quad \int_{0}^{x} y(x) dx = 1 - \cos(x). $$

The approximations for the function, derivatives, and the integral are evaluated using `splineg`. Results shown in Figure 4.1 are quite satisfactory, except for points outside the data interval $[0, 2\pi]$. 

© 2003 by CRC Press LLC
Figure 4.1: Spline Differentiation and Integration of $\sin(x)$
Example: Spline Interpolation Applied to Sin(x)

Program sinetrp

```matlab
function sinetrp
% Example: sinetrp
% ~~~~~~~~~~~~~~~
% This example illustrates cubic spline
% approximation of sin(x), its first two
% derivatives, and its integral.
% 
% User m functions required:
% splineg, splincof

% Create data points on the spline curve
xd=linspace(0,2*pi,21); yd=sin(xd);

% Evaluate function values at a dense
% set of points
x=linspace(-pi/2,5/2*pi,61);
[y,b,c]=splineg(xd,yd,x,0);
yp=splineg(xd,yd,x,1,[],b,c);
ypp=splineg(xd,yd,x,2,[],b,c);
yint=splineg(xd,yd,x,3,[],b,c);

% Plot results
z=x/pi; zd=xd/pi;
plot(z,y,'k-',zd,yd,'ko',z,yp,'k:',...
     z,ypp,'k-.',z,yint,'k+');
title(['Spline Differentiation and ', ...
      'Integration of sin(x)']);
xlabel('x / pi'); ylabel('function values');
legend('y=sin(x)','data','y''(x)','y''''(x)', ...
      '\int y(x) dx',1); grid on
figure(gcf); pause;

function [val,b,c]=splineg(xd,yd,x,deriv,endc,b,c)
% [val,b,c]=splineg(xd,yd,x,deriv,endc,b,c)
% ~==============================================
```

© 2003 by CRC Press LLC
% For a cubic spline curve through data points
% xd,yd, this function evaluates y(x), y'(x),
% y''(x), or integral(y(x)*dx, xd(1) to x(j) )
% for j=1:length(x). The coefficients needed to
% evaluate the spline are also computed.
%
% xd,yd - data vectors defining the cubic
% spline curve
% x - vector of points where curve
% properties are computed.
% deriv - denoting the spline curve as y(x),
% deriv=0 gives a vector for y(x)
% deriv=1 gives a vector for y'(x)
% deriv=2 gives a vector for y''(x)
% deriv=3 gives a vector of values
% for integral(y(z)*dz) from xd(1) to x(j) for j=1:length(x)
% endc - endc=1 makes y'''(x) continuous at
% xd(2) and xd(end-1).
% endc=[2,left_slope,right_slope]
% imposes slope values at both ends.
% endc=[3,left_slope] imposes the left
% end slope and makes the discontinuity
% of y''' at xd(end-1) small.
% endc=[4,right_slope] imposes the right
% end slope and makes the discontinuity
% of y''' at xd(2) small.
% b,c coefficients needed to perform the
% spline interpolation. If these are not
% given, function unmkpp is called to
% generate them.
% val values y(x),y'(x),y''(x) or
% integral(y(z)dz, z=xd(1)..x) for
% deriv=0,1,2, or 3, respectively.

if nargin<5 | isempty(endc), endc=1; end
if nargin<7, [b,c]=splincof(xd,yd,endc); end
n=length(xd); [N,M]=size(c);

switch deriv
  case 0 % Function value
    val=ppval(mkpp(b,c),x);
  case 1 % First derivative
C=[3*c(:,1),2*c(:,2),c(:,3)];
val=ppval(mkpp(b,C),x);

case 2 % Second derivative
C=[6*c(:,1),2*c(:,2)];
val=ppval(mkpp(b,C),x);

case 3 % Integral values from xd(1) to x
k=M:-1:1;
C=[c./k(ones(N,1),:),zeros(N,1)];
dx=xd(2:n)-xd(1:n-1); s=zeros(n-2,1);
for j=1:n-2, s(j)=polyval(C(j,:),dx(j)); end
C(:,5)=[0;cumsum(s)]; val=ppval(mkpp(b,C),x);
end

function [b,c]=splincof(xd,yd,endc)
% [b,c]=splincof(xd,yd,endc)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function determines coefficients for
% cubic spline interpolation allowing four
% different types of end conditions.
% xd,yd - data vectors for the interpolation
% endc - endc=1 makes y'''(x) continuous at
% xd(2) and xd(end-1).
% endc=[2,left_slope,right_slope]
% imposes slope values at both ends.
% endc=[3,left_slope] imposes the left
% end slope and makes the discontinuity
% of y''' at xd(end-1) small.
% endc=[4,right_slope] imposes the right
% end slope and makes the discontinuity
% of y''' at xd(2) small.
% if nargin<3, endc=1; end;
% type=endc(1); xd=xd(:); yd=yd(:);
switch type
  case 1
    % y''''(x) continuous at the xd(2) and xd(end-1)
    [b,c]=unmkpp(spline(xd,yd));
case 2
% Slope given at both ends
[b,c]=unmkpp(spline(xd,[endc(2);yd;endc(3)]));

case 3
% Slope at left end given. Compute right end
% slope.
[b,c]=unmkpp(spline(xd,yd));
c=[3*c(:,1),2*c(:,2),c(:,3)];
sright=ppval(mkpp(b,c),xd(end));
[b,c]=unmkpp(spline(xd,[endc(2);yd;sright]));

case 4
% Slope at right end known. Compute left end
% slope.
[b,c]=unmkpp(spline(xd,yd));
c=[3*c(:,1),2*c(:,2),c(:,3)];
sleft=ppval(mkpp(b,c),xd(1));
[b,c]=unmkpp(spline(xd,[sleft;yd;endc(2)]));

end

4.2.4 Example: A Spline Curve with Several Parts and Corners

The final spline example illustrates interpolation of a two-dimensional curve where y cannot be expressed as a single valued function of x. Then we introduce a parameter \( t \) having its value equal to the index \( j \) for each \( (x_j, y_j) \) used. Interpolating \( x(t) \) and \( y(t) \) as continuous functions of \( t \) produces a smooth curve through the data. Function `matlbdat` creates data points to define the curve and calls function `spcry2d` to compute points on a general plane curve. We also introduce the idea of ‘corner points’ where slope discontinuity allows the curve to make sharp turns needed to describe letters such as the ‘t’ in MATLAB. Each curve segment between successive pairs of corner points is parameterized using function `spline`. Results in Figure 4.2 show clearly that spline interpolation can represent a complicated curve. The related code appears after the figure. The same kind of parameterization used for two dimensions also works well for three dimensional curves.

Example: Spline Curve Drawing the Word MATLAB

Program `matlbdat`

```matlab
function matlbdat
```
A Spline Curve Drawing the Word MATLAB

Figure 4.2: Spline Curve Drawing the Word MATLAB
% Example: matlbdat
% ~~~~~~~~~~~~~~~~~
% This example illustrates the use of splines
% to draw the word MATLAB.
%
% User m functions required: spcurv2d

x=[13 17 17 16 17 19 21 22 21 21 23 26
  25 28 30 32 30 32 35 37 37 38
  41 42 42 45 39 42 42 44 47 48 48
  47 47 48 51 53 57 53 52 53 56 57 57
  58 61 63 62 61 64 66 64 61 64 67 67];
y=[63 64 58 52 57 62 62 58 51 58 63 63
  53 52 56 61 61 61 56 51 55 61 55 52
x=x'; x=x(:); y=y'; y=y(:);
ncrnr=[17 22 26 27 28 29 30 31 36 42 47 52];
clf; [xs,ys]=curv2d(x,y,10,ncrnr);
plot(xs,ys,'k-',x,y,'k*'), axis off;
title('A Spline Curve Drawing the Word MATLAB');
figure(gcf);
% print -deps matlbdat

%=======================================================================

function [X,Y]=spcrv2d(xd,yd,nseg,icrnr)
%
% [X,Y]=spcrv2d(xd,yd,nseg,icrnr)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function computes points (X,Y) on a
% spline curve through (xd,yd) allowing slope
% discontinuities at points with corner
% indices in icrnr. nseg plot segments are
% used between each successive pair of points.

if nargin<4, icrnr=[]; end
if nargin<3, nseg=10; end
zd=xd(:)+i*yd(:); n=length(zd);
N=[1;sort(icrnr(:));n]; Z=zd(1);
if N(1)==N(2); N(1)=[]; end
if N(end)==N(end-1); N(end)=[]; end
for k=1:length(N)-1
    zk=zd(N(k):N(k+1)); sk=length(zk)-1;
4.3 Numerical Differentiation Using Finite Differences

Differential equation problems are sometimes solved using difference formulas to approximate the derivatives in terms of function values at adjacent points. Deriving difference formulas by hand can be tedious, particularly when unequal point spacing is used. For this reason, we develop a numerical procedure to construct formulas of arbitrary order and arbitrary truncation error. Of course, as the desired order of derivative and the order of truncation error increases, more points are needed to interpolate the derivative. We will show below that approximating a derivative of order \( k \) with a truncation error of order \( h^m \) generally requires \((k + m)\) points unless symmetric central differences are used. Consider the Taylor series expansion

\[
F(x + \alpha h) = \sum_{k=0}^{\infty} \frac{F^{(k)}(x)}{k!} (\alpha h)^k
\]

where \( F^{(k)}(x) \) means the \( k \)'th derivative of \( F(x) \). This relation expresses values of \( F \) as linear combinations of the function derivatives at \( x \). Conversely, the derivative values can be cast in terms of function values by solving a system of simultaneous equations. Let us take a series of points defined by

\[
x_i = x + h\alpha_i, \quad 1 \leq i \leq n
\]

where \( h \) is a fixed step-size and \( \alpha_i \) are arbitrary parameters. Separating some leading terms in the series expansion gives

\[
F(x_i) = \sum_{k=0}^{n-1} \frac{\alpha_i^k}{k!} \left[ h^k F^{(k)}(x) \right] + \frac{\alpha_i^n}{n!} \left[ h^n F^{(n)}(x) \right] + \frac{\alpha_i^{n+1}}{(n+1)!} \left[ h^{(n+1)} F^{(n+1)}(x) \right] + O(h^{n+2}), \quad 1 \leq i \leq n.
\]

It is helpful to use the following notation:

- \( \alpha^k \) – a column vector with component \( i \) being equal to \( \alpha_i^k \)
- \( f \) – a column vector with component \( i \) being \( F(x_i) \)
- \( f^p \) – a column vector with component \( i \) being \( h^i F^{(i)}(x) \)
- \( A \) – \([\alpha^0, \alpha^1, \ldots, \alpha^{n-1}]\), a square matrix with columns which are powers of \( \alpha \).
Then the Taylor series expressed in matrix form is
\[ f = A * f^p + \frac{h^n F^{(n)}(x)}{n!} \alpha^n + \frac{h^{n+1} F^{(n+1)}(x)}{(n+1)!} \alpha^{n+1} + O(h^{n+2}). \]

Solving this system for the derivative matrix yields
\[ f^p = A^{-1} f - \frac{h^n F^{(n)}(x)}{n!} A^{-1} \alpha^n - \frac{h^{n+1} F^{(n+1)}(x)}{(n+1)!} A^{-1} \alpha^{n+1} + O(h^{n+2}). \]

In the last equation we have retained the first two remainder terms in explicit form to allow the magnitudes of these terms to be examined. Row \( k + 1 \) of the previous equation implies
\[ F^{(k)}(x) = h^{-k} (A^{-1} f)_{k+1} - \frac{h^{n-k}}{n!} F^{(n)}(x) (A^{-1} \alpha^n)_{k+1} - \frac{h^{n-k+1}}{(n+1)!} F^{(n+1)}(A^{-1} \alpha^{n+1})_{k+1} + O(h^{n-k+1}). \]

Consequently, the rows of \( A^{-1} \) provide coefficients in formulas to interpolate derivatives. For a particular number of interpolation points, say \( N \), the highest derivative approximated will be \( F^{(N-1)}(x) \) and the truncation error will normally be of order \( h^1 \). Conversely, if we need to compute a derivative formula of order \( k \) with the truncation error being \( m \), then it is necessary to use a number of points such that \( n - k = m \); therefore \( n = m + k \). For the case where interpolation points are symmetrically placed around the point where derivatives are desired, one higher power of accuracy order is achieved than might be expected. We can show, for example, that
\[ \frac{d^4 F(x)}{dx^4} = \frac{1}{h^4} (F(x - 2h) - 4F(x - h) + 6F(x) - 4F(x + h) + F(x + 2h)) + O(h^2) \]

because the truncation error term associated with \( h^1 \) is found to be zero. At the same time, we can show that a forward difference formula for \( f''''(x) \) employing equidistant point spacing is
\[ \frac{d^4 F(x)}{dx^4} = \frac{1}{h^4} (-2.5F(x) - 9F(x + h) + 12F(x + 2h) + 7F(x + 3h) - 1.5F(x + 4h)) + O(h^2). \]

Although the last two formulas contain arithmetically simple interpolation coefficients, due to equal point spacing, the method is certainly not restricted to equal spacing. The following program contains the function \texttt{derivtrp} which implements the ideas just developed. Since the program contains documentation that is output when it is executed, no additional example problem is included.
4.3.1 Example: Program to Derive Difference Formulas

Output from Example

finitdif;

COMPUTING \( F(x,k) \), THE \( k' \)TH DERIVATIVE OF \( f(x) \), BY FINITE DIFFERENCE APPROXIMATION

Input the derivative order (give 0 to stop, or ? for an explanation) > ?

Let \( f(x) \) have its \( k' \)th derivative denoted by \( F(k,x) \). The finite difference formula for a stepsize \( h \) is given by:

\[
F(x,k) = \text{Sum}(c(j)*f(x+a(j)*h), j=1:n)/h^k + \ldots
\]

TruncationError

with \( m=n-k \) being the order of truncation error which decreases like \( h^m \) according to:

\[
\text{TruncationError} = -(h^m)(e(1)*F(x,n) + \ldots + e(2)*F(x,n+1)*h + e(3)*F(x,n+2)*h^2 + O(h^3))
\]

Input the derivative order (give 0 to stop, or ? for an explanation) > 4

Give the required truncation order > 1

To define interpolation points \( X(j)=x+h*a(j) \), input at least 5 components for vector \( a \).

Components of \( a \) > -2, -1, 0, 1, 2

The formula for a derivative of order 4 is:

\[
F(x,k) = \text{sum}(c(j)*F(X(j),j=1:n)/h^4 + \text{order}(h^1))
\]

where \( c \) is given by:

\[
egin{array}{cccccc}
1.0000 & -4.0000 & 6.0000 & -4.0000 & 1.0000 \\
\end{array}
\]

and the truncation error coefficients are:

\[
egin{array}{cccc}
-0.0000 & 0.1667 & -0.0000 & 0.0125 \\
\end{array}
\]

Input the derivative order (give 0 to stop,
or ? for an explanation) > 3

Give the required truncation order > 2

To define interpolation points \( X(j) = x + h \cdot a(j) \),
input at least 5 components for vector \( a \).

Components of \( a \) > 0,1,2,3,4

The formula for a derivative of order 3 is:
\[
F(x,k) = \sum (c(j) \cdot F(X(j), j=1:n) / h^3) + \text{order}(h^2)
\]
where \( c \) is given by:

\[
-2.5000 \quad 9.0000 \quad -12.0000 \quad 7.0000 \quad -1.5000
\]

and the truncation error coefficients are:

\[
-1.7500 \quad -2.5000 \quad -2.1417 \quad -1.3750
\]

Input the derivative order (give 0 to stop,
or ? for an explanation) > 0

Program \textit{finitdif}

```matlab
function [c,e,m,crat,k,a]=finitdif(k,a)
% [c,e,m,crat,k,a]=finitdif(k,a)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This program computes finite difference formulas of
% general order. For explanation of the input and
% output parameters, see the following function
% findifco. When the program is executed without input
% arguments, then input is read interactively.

if nargin==0, disp(' ') % Use interactive input
    disp('COMPUTING F(x,k), THE K''TH DERIVATIVE OF')
    disp('f(x), BY FINITE DIFFERENCE APPROXIMATION')
    disp('')
    while 1
        disp('Input the derivative order (give 0 to stop,')
        K=input('or ? for an explanation) > ','s');
        k=str2num(K);
        if strcmp(K,'') | strcmp(K,'0'); disp(' '); return
        end
        c=-2.5000 9.0000 -12.0000 7.0000 -1.5000;
        e=-1.7500 -2.5000 -2.1417 -1.3750;
        m;
        crat=order(h^2);
        disp('F(x,k)=');
        disp('sum(')
        disp(c); disp('*F(X(j), j=1:n)/h^3) + order(h^2)')
    end
end
```

© 2003 by CRC Press LLC
elseif strcmp(K,'?')
    disp(' '), disp...
    'Let f(x) have its k''th derivative denoted by'
    disp(...
    'F(k,x). The finite difference formula for a'
    disp('stepsie h is given by:'), disp(' ')
    disp(...
    'F(x,k)=Sum(c(j)*f(x+a(j)*h), j=1:n)/h^k +...')
    disp('TruncationError'), disp(' ')
    disp('with m=n-k being the order of truncation')
    disp(...
    'error which decreases like h^m according to:')
    disp(' ')
    disp('TruncationError=-(h^m)*(e(1)*F(x,n)+...')
    disp(...
    'e(2)*F(x,n+1)*h+e(3)*F(x,n+2)*h^2+O(h^3))')
    disp(' ')
else
    disp(' ')
    m=input('Give the required truncation order > '); 
    n=m+k; N=num2str(n); disp(' '), disp(...
    'To define interpolation points X(j)=x+h*a(j),')
    disp(['input at least ',N,...
    ' components for vector a.'])
    disp(' '), aa=input('Components of a > ','s');
    a=eval(['[',aa,']']); n=length(a); m=n-k;
    [c,e,m,crat]=findifco(k,a); disp(' '), disp(...
    ['The formula for a derivative of order ','...
    K,' is:'])
    disp(['F(x,k)=sum(c(j)*F(X(j),j=1:n)/h^',K,...
    '+order(h^',num2str(m),')')]})
    disp('where c is given by:')
    disp(' '), disp(c), disp(' ')
    disp(...
    'and the truncation error coefficients are:')
    disp(' '), disp(e)
end
end
else
    [c,e,m,crat]=findifco(k,a);
end

%==================================================================

function [c,e,m,crat]=findifco(k,a)
This function approximates the \( k \)'th derivative of a function using function values at \( n \) interpolation points. Let \( f(x) \) be a general function having its \( k \)'th derivative denoted by \( F(x,k) \). The finite difference approximation for the \( k \)'th derivative employing a stepsize \( h \) is given by:

\[
F(x,k) = \frac{\text{Sum}(c(j) \cdot f(x+a(j) \cdot h), j=1:n)}{h^k} + \text{TruncationError}
\]

with \( m=n-k \) being the order of truncation error which decreases like \( h^m \)

\[
\text{TruncationError} = (h^m) \cdot (e(1) \cdot F(x,n)+...+e(2) \cdot F(x,n+1) \cdot h+e(3) \cdot F(x,n+2) \cdot h^2+O(h^3))
\]

\[
\text{a} - a \text{ vector of length } n \text{ defining the interpolation points } x+a(j) \cdot h \text{ where } x \text{ is an arbitrary parameter point}
\]

\[
\text{k} - \text{order of derivative evaluated at } x
\]

\[
\text{c} - \text{the weighting coefficients in the difference formula above. } c(j) \text{ is the multiplier for value } f(x+a(j) \cdot h)
\]

\[
\text{e} - \text{error component vector in the above difference formula}
\]

\[
\text{m} - \text{order of truncation order in the formula. The relation } m=n-k \text{ applies.}
\]

\[
\text{crat} - a \text{ matrix of integers such that } c \text{ is approximated by } \frac{\text{crat}(1,:)}{\text{crat}(2,:)}
\]

\[
a=a(:,); \ n=\text{length}(a); \ m=n-k; \ \text{mat}=\text{ones}(n,n+4); \ 
\text{for } j=2:n+4; \ \text{mat}(:,j)=a(j-1) \cdot \text{mat}(:,j-1); \ \text{end} \\
\text{A} = \text{pinv(mat(:,1:n))}; \ \text{ec} = -\text{A} \cdot \text{mat(:,n+1:n+4)}; \\
\text{c} = \text{A}(k+1,:); \ \text{e} = -\text{ec(k+1,:)}; \\
[\text{ctop}, \text{cbot}] = \text{rat(c,1e-8)}; \ \text{crat} = [\text{ctop}(:,')'; \text{cbot}(:,')] ;
\]
Chapter 5

Gauss Integration with Geometric Property
Applications

5.1 Fundamental Concepts and Intrinsic Integration Tools
in MATLAB

Numerical integration methods approximate a definite integral by evaluating the
integrand at several points and taking a weighted combination of those integrand
values. The weight factors can be obtained by interpolating the integrand at selected
points and integrating the interpolating function exactly. For example, the Newton-
Cotes formulas result from polynomial interpolation through equidistant base points.
This chapter discusses concepts of numerical integration needed in applications.

Let us assume that an integral over limits \( a \) to \( b \) is to be evaluated. We can write

\[
\int_{a}^{b} f(x) \, dx = \sum_{i=1}^{n} W_i f(x_i) + E
\]

where \( E \) represents the error due to replacement of the integral by a finite sum. This
is called an \( n \)-point quadrature formula. The points \( x_i \) where the integrand is evalu-
ated are the base points and the constants \( W_i \) are the weight factors. Most integration
formulas depend on approximating the integrand by a polynomial. Consequently,
they give exact results when the integrand is a polynomial of sufficiently low order.
Different choices of \( x_i \) and \( W_i \) will be discussed below.

It is helpful to express an integral over general limits in terms of some
fixed limits, say \(-1\) to \(1\). This is accomplished by introducing a linear change of variables

\[
x = \alpha + \beta t.
\]

Requiring that \( x = a \) corresponds to \( t = -1 \) and that \( x = b \) corresponds to \( t = 1 \)
gives \( \alpha = (a + b)/2 \) and \( \beta = (b - a)/2 \), so that one obtains

\[
\int_{a}^{b} f(x) \, dx = \frac{1}{2} (b - a) \int_{-1}^{1} f \left( \frac{a + b}{2} + \frac{b - a}{2} t \right) dt = \int_{-1}^{1} F(t) dt
\]

where \( F(t) = f[(a + b)/2 + (b - a)t/(b - a)]]/2 \). Thus, the dependence of the
integral on the integration limits can be represented parametrically by modifying the

© 2003 by CRC Press LLC
consequently, if an integration formula is known for limits $-1$ to $1$, we can write
\[ \int_a^b f(x)dx = \beta \sum_{i=1}^n W_i f(\alpha + \beta x_i) + E. \]

The idea of shifting integration limits can be exploited further by dividing the interval $a$ to $b$ into several parts and using the same numerical integration formula to evaluate the contribution from each interval. Employing $m$ intervals of length $\ell = (b - a)/m$, we get
\[ \int_a^b f(x)dx = \sum_{j=1}^m \int_{a+(j-1)\ell}^{a+j\ell} f(x)dx. \]

Each of the integrals in the summation can be transformed to have limits $-1$ to $1$ by taking
\[ x = \alpha_j + \beta t \]
with
\[ \alpha_j = a + (j - 1/2)\ell \text{ and } \beta = \ell/2. \]

Therefore we obtain the identity
\[ \int_a^b f(x)dx = \sum_{j=1}^m \int_{-1}^{1} f(\alpha_j + \beta t)dt. \]

Applying the same $n$-point quadrature formula in each of $m$ equal intervals gives what is termed a composite formula
\[ \int_a^b f(x)dx = \frac{\ell}{2} \sum_{j=1}^m \sum_{i=1}^n W_i f(\alpha_j + \beta x_i) + E. \]

By interchanging the summation order in the previous equation we get
\[ \int_a^b f(x)dx = \frac{\ell}{2} \sum_{i=1}^n W_i \sum_{j=1}^m f(\alpha_j + \beta x_i) + E. \]

Let us now turn to certain choices of weight factors and base points. Two of the most widely used methods approximate the integrand as either piecewise linear or piecewise cubic. Approximating the integrand by a straight line through the integrand end points gives the following formula
\[ \int_{-1}^{1} f(x)dx = f(-1) + f(1) + E. \]

A much more accurate formula results by using a cubic approximation matching the integrand at $x = -1, 0, 1$. Let us write
\[ f(x) = c_1 + c_2 x + c_3 x^2 + c_4 x^3. \]

© 2003 by CRC Press LLC
Then
\[ \int_{-1}^{1} f(x) dx = 2c_1 + \frac{2}{3}c_3. \]

Evidently the linear and cubic terms do not influence the integral value. Also, \( c_1 = f(0) \) and \( f(-1) + f(1) = 2c_1 + 2c_3 \) so that
\[ \int_{-1}^{1} f(x) dx = \frac{1}{3} [f(-1) + 4f(0) + f(1)] + E. \]

The error \( E \) in this formula is zero when the integrand is any polynomial of order 3 or lower. Expressed in terms of more general limits, this result is
\[ \int_{a}^{b} f(x) dx = \frac{(b-a)}{6} \left[ f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] + E \]
which is known as Simpson’s rule.

Analyzing the integration error for a particular choice of integrand and quadrature formula can be complex. In practice, the usual procedure taken is to apply a composite formula with \( m \) chosen large enough so the integration error is expected to be negligibly small. The value for \( m \) is then increased until no further significant change in the integral approximation results. Although this procedure involves some risk of error, adequate results can be obtained in most practical situations.

In the subsequent discussions the integration error that results by replacing an integral by a weighted sum of integrand values will be neglected. It must nevertheless be kept in mind that this error depends on the base points, weight factors, and the particular integrand. Most importantly, the error typically decreases as the number of function values is increased.

It is convenient to summarize the composite formulas obtained by employing a piecewise linear or piecewise cubic integrand approximation. Using \( m \) intervals and letting \( \ell = (b-a)/m \), it is easy to obtain the composite trapezoidal formula which is
\[ \int_{a}^{b} f(x) dx = \ell \left[ \frac{f(a) + f(b)}{2} + \sum_{j=1}^{m-1} f(a + j\ell) \right]. \]

This formula assumes that the integrand is satisfactorily approximated by piecewise linear functions. The MATLAB function \texttt{trapz} implements the trapezoidal rule. A similar but much more accurate result is obtained for the composite integration formula based on cubic approximation. For this case, taking \( m \) intervals implies \( 2m + 1 \) function evaluations. If we let \( g = (b-a)/(2m) \) and \( h = 2g \), then
\[ f_j = f(x_j) \quad \text{where} \quad x_j = a + gj, \quad j = 0, 1, 2, \ldots, 2m, \]
with \( f(x_0) = f(a) \) and \( f(x_{2m}) = f(b) \). Combining results for all intervals gives
\[ \int_{a}^{b} f(x) dx = \frac{h}{6} \left[ f(a) + 4f_1 + f(b) + \sum_{i=1}^{m-1} (4f_{2i+1} + 2f_{2i}) \right]. \]
This formula, known as the composite Simpson rule, is one of the most commonly used numerical integration methods. The following function `simpson` works for an analytically defined function or a function defined by spline interpolating through discrete data.

**Function for Composite Simpson Rule**

```matlab
function area=simpson(funcname,a,b,n,varargin)

% area=simpson(funcname,a,b,n,varargin)
% -------------------------------------
% Simpson’s rule integration for a general function
% defined analytically or by a data array
%
% funcname - either the name of a function valid
% for a vector argument x, or an array
% having two columns with x data in the
% first column and y data in the second
% column. If array data is given, then
% the function is determined by piecewise
% cubic spline interpolation.

% a,b - limits of integration
% n - odd number of function evaluations. If
% n is given as even, then the next
% higher odd integer is used.
% varargin - variable number of arguments passed
% for use in funcname
% area - value of the integral when the integrand
% is approximated as a piecewise cubic
% function

% User functions called: function funcname in the
% argument list
%----------------------------------------------------

if 2*fix(n/2)==n; n=n+1; end; n=max(n,3);
x=linspace(a,b,n);
if isstr(funcname)
y=feval(funcname,x,varargin{:});
else
y=spline(funcname(:,1),funcname(:,2),x);
end
area=(b-a)/(n-1)/3*( y(1)-y(n)+
```

© 2003 by CRC Press LLC
An important goal in numerical integration is to achieve accurate results with only a few function evaluations. It was shown for Simpson’s rule that three function evaluations are enough to exactly integrate a cubic polynomial. By choosing the base point locations properly, a much higher accuracy can be achieved for a given number of function evaluations than would be obtained by using evenly spaced base points. Results from orthogonal function theory lead to the following conclusions. If the base points are located at the zeros of the Legendre polynomials (all these zeros are between $-1$ and $1$) and the weight factors are computed as certain functions of the base points, then the formula

$$\int_{-1}^{1} f(x) \, dx = \sum_{i=1}^{n} W_i f(x_i)$$

is exact for a polynomial integrand of degree $2n - 1$. Although the theory proving this property is not elementary, the final results are quite simple. The base points and weight factors for a particular order can be computed once and used repeatedly. Formulas that use the Legendre polynomial roots as base points are called Gauss quadrature formulas. In a typical application, Gauss integration gives much more accurate results than Simpson’s rule for an equivalent number of function evaluations. Since it is equally easy to use, the Gauss formula is preferable to Simpson’s rule.

MATLAB also has three functions `quad` and `quad8` and `quadl` to numerically integrate by adaptive methods. These functions repeatedly modify approximations for an integral until the estimated error becomes smaller than a specified tolerance. In the current text, the function `quadl` is preferable over the other two functions, and `quadl` is always used when an adaptive quadrature function is needed. Readers should study carefully the system documentation for `quadl` to understand the various combinations of call list parameters allowed.

5.2 Concepts of Gauss Integration

This section summarizes properties of Gauss integration which, for the same number of function evaluations, are typically much more accurate than comparable Newton-Cotes formulas. It can be shown for Gauss integration [20] that

$$\int_{-1}^{1} f(x) \, dx = \sum_{j=1}^{n} w_j f(x_j) + E(f)$$
where the integration error term is

$$E = \frac{2^{2n-1} (n!)^4}{(2n+1)((2n)!)^3} x^{(2n)}(\xi), \quad -1 < \xi < 1.$$  

The base points in the Gauss formula of order $n$ are the roots of the Legendre polynomial of order $n$ and the weight factors are expressible concisely in terms of the base points. The quadrature error term for an $n$-point formula involves the integrand derivative of order $2n$, which implies a zero error for any polynomial of order $2n-1$ or lower. The coefficient of the derivative term in $E$ decreases very rapidly with increasing $n$, as can be seen in Figure 5.1.

For example, $n = 10$ gives a coefficient of $2.03 \times 10^{-21}$. Thus, a function having well behaved high order derivatives can be integrated accurately with a formula of fairly low order. The base points $x_j$ are all distinct, lie between $-1$ and $1$, and are the eigenvalues of a symmetric tridiagonal matrix [26] which can be analyzed very rapidly with the function `eigen`. Furthermore, the weight factors are simply twice the squares of the first components of the orthonormalized eigenvectors. Because `eigen` returns orthonormalized eigenvectors for symmetric matrices, only lines 58-60 in function `gequad` given below are needed to compute the base points and weight factors.
Function for Composite Gauss Integration

function [val,bp,wf]=gcquad(func,xlow,...
         xhigh,nquad,mparts,varargin)

% [val,bp,wf]=gcquad(func,xlow,...
       xhigh,nquad,mparts,varargin)

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function integrates a general function using
% a composite Gauss formula of arbitrary order. The
% integral value is returned along with base points
% and weight factors obtained by an eigenvalue based
% method. The integration interval is divided into
% mparts subintervals of equal length and integration
% over each part is performed with a Gauss formula
% making nquad function evaluations. Results are
% exact for polynomials of degree up to 2*nquad-1.
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% func - name of a function to be integrated
% having an argument list of the form
% func(x,p1,p2,...) where any auxiliary
% parameters p1,p2,... are passed through
% variable varargin. Use [ ] for the
% function name if only the base points
% and weight factors are needed.
% xlow,xhigh - integration limits
% nquad - order of Gauss formula chosen
% mparts - number of subintervals selected in
% the composite integration
% varargin - variable length parameter used to
% pass additional arguments needed in
% the integrand func
% val - numerical value of the integral
% bp,wf - vectors containing base points and
% weight factors in the composite
% integral formula

% A typical calculation such as:
% Fun=inline('(sin(w*t).^2).*exp(c*t)','t','w','c');
% A=0; B=12; nquad=21; mparts=10; w=10; c=8;
% [value,pcterr]=integrate(Fun,A,B,nquad,mparts,w,c);
42 \% gives value = 1.935685556078172e+040 which is
43 \% accurate within an error of 1.9e-13 percent.
44 \%
45 \% User m functions called: the function name passed
46 \% in the argument list
47 \%
48 \%----------------------------------------------
49:
50: if isempty(nquad), nquad=10; end
51: if isempty(mparts), mparts=1; end
52:
53: \% Compute base points and weight factors
54: \% for the single interval [-1,1]. (Ref:
55: \% 'Methods of Numerical Integration' by
56: \% P. Davis and P. Rabinowitz, page 93)
57:
58: u=(1:nquad-1)./sqrt((2*(1:nquad-1)).^2-1);
59: [vc,bp]=eig(diag(u,-1)+diag(u,1));
60: [bp,k]=sort(diag(bp)); wf=2*vc(1,k)'^2;
61:
62: \% Modify the base points and weight factors
63: \% to apply for a composite interval
64: \d=(xhigh-xlow)/mparts; \d1=d/2;
65: \dbp=d1*bp(:,); \dwf=d1*wf(:,); \dr=d*(1:mparts);
66: \cbp=dbp(:,ones(1,mparts))+ ... 
67: \dr(ones(nquad,1),:)+(xlow-d1);
68: \cwf=dwf(:,ones(1,mparts)); \wf=cwf(:,); \bp=cbp(:,);
69:
70: \% Compute the integral
71: if isempty(func)
72: \val=[];
73: else
74: \f=feval(func,bp,varargin{:}); \val=wf'*f(:,
75: end

5.3 Comparing Results from Gauss Integration and
Function QUADL

A program was written to compare the performance of the Gauss quadrature function \texttt{gcquad} and the numerical integrator \texttt{quadl} provided in MATLAB. \texttt{Quadl} is a robust adaptive integration routine which efficiently handles most integrands. It can even deal with special integrals having singularities, like $\log(x)$ or $1/\sqrt{x}$ at the origin. Integrating these functions from zero to one yields correct answers although messages occur warning about integrand singularities at the origin. No capabilities
are provided in quadl to directly handle vector integrands (except one component at a time), and no options are provided to suppress unwanted warning or error messages. In the timing program given below, warning messages from quadl were temporarily turned off for the tests.

Often there are examples involving vector-valued integrands that are to be integrated many times over fixed integration limits. A typical case is evaluation of coefficients in Fourier-Bessel series expansions. Then, computing a set of base points and weight factors once and using these coefficients repeatedly is helpful. To illustrate this kind of situation, let us numerically integrate the vector valued function

\[ f(x) = [\sqrt{x}; \log(x); \text{humps}(x); \exp(10x) \cos(10\pi x); \cos(20\pi x - 20 \sin(\pi x))] \]

from \( x = 0 \) to \( x = 1 \). Several components of this function are hard to integrate numerically because \( \sqrt{x} \) has infinite slope at \( x = 0 \), \( \log(x) \) is singular at \( x = 0 \), the fourth component is highly oscillatory with large magnitude variations, and the last component is highly oscillatory (integrating the last component gives the value of the integer order Bessel function \( J_{20}(20) \)).

The following function quadtest uses functions quadl and gcquad to integrate \( f(x) \) from \( x = 0 \) to \( x = 1 \). The Gauss integration employs a formula of order 100 with one subinterval, so integrands are effectively approximated by polynomials of order 199. To achieve accurate timing, it was necessary to evaluate the integrals repeatedly until a chosen number of seconds elapsed. Then average times were computed. The program output shows that gcquad was more accurate than quadl for all cases except for \( \log(x) \) involving a singular integrand. Computations times shown for each component of \( f(x) \) are the same when gcquad was used because the integration was done for all components at once, and then results were divided by five. The total time used by quadl was about 3.5 times as large as the time for gcquad.

We are not arguing that these results show gcquad is superior to quadl. However, it does imply that Gauss integration can be attractive in some instances. The geometry problems in the remainder of this chapter include boundary curves defined by cubic splines. Then, using Gauss integration of sufficiently high order produces exact results for the desired geometrical properties.

Output from Program quadtest

\[
\begin{align*}
\text{>> quadtest(10);} \\
\text{PRESS RETURN TO BEGIN COMPUTATION} & > ? \\
\end{align*}
\]

INTEGRATION TEST COMPARING FUNCTIONS QUADL AND GCQUAD
The functions being integrated are:
- \( \sqrt{x} \)
- \( \log(x) \)
- \( \text{humps}(x) \)
- \( \exp(10\times) \cdot \cos(10\times \pi \times) \)
- \( \cos(20 \times \pi \times - 20 \times \sin(\pi \times)) \)
Results Using Function quadl

<table>
<thead>
<tr>
<th>Integral values</th>
<th>Function evaluations</th>
<th>Percent error</th>
<th>Computation seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6667e-001</td>
<td>7.8000e+001</td>
<td>-1.9813e-004</td>
<td>6.9720e-003</td>
</tr>
<tr>
<td>-1.0000e+000</td>
<td>2.2900e+002</td>
<td>2.6064e-005</td>
<td>2.1071e-002</td>
</tr>
<tr>
<td>2.9858e+001</td>
<td>1.9800e+002</td>
<td>6.2164e-010</td>
<td>2.1162e-002</td>
</tr>
<tr>
<td>2.0263e+002</td>
<td>7.0800e+002</td>
<td>2.4425e-013</td>
<td>6.8660e-002</td>
</tr>
<tr>
<td>1.6475e-001</td>
<td>5.2800e+002</td>
<td>-1.0627e-008</td>
<td>5.1370e-002</td>
</tr>
</tbody>
</table>

Results Using Function gcquad

<table>
<thead>
<tr>
<th>Integral values</th>
<th>Function evaluations</th>
<th>Percent error</th>
<th>Computation seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6667e-001</td>
<td>1.0000e+002</td>
<td>1.5215e-005</td>
<td>9.5628e-003</td>
</tr>
<tr>
<td>-9.9994e-001</td>
<td>1.0000e+002</td>
<td>-6.2513e-003</td>
<td>9.5628e-003</td>
</tr>
<tr>
<td>2.9858e+001</td>
<td>1.0000e+002</td>
<td>8.8818e-014</td>
<td>9.5628e-003</td>
</tr>
<tr>
<td>2.0263e+002</td>
<td>1.0000e+002</td>
<td>-4.1078e-013</td>
<td>9.5628e-003</td>
</tr>
<tr>
<td>1.6475e-001</td>
<td>1.0000e+002</td>
<td>-1.5543e-013</td>
<td>9.5628e-003</td>
</tr>
</tbody>
</table>

(Total time using quadl)/(Total time using gcquad) equals 3.5395

Program Comparing Numerical Integration Methods

```matlab
function [L,G,names]=quadtest(secs)
%
% [L,G,names]=quadtest(secs)
% --------------------------
% This program compares the accuracy and
% computation times for several integrals
% evaluated using quadl and gcquad
%
% secs - the number of seconds each integration
% is repeated to get accurate timing. The
% default value is 60 seconds.
% L,G - matrices with columns containing
% results from quadl and from gcquad.
% The matrices are structured as:
% [IntegralValue,PercentError,...
% FunctionEvaluations,ComputationSeconds]
% names - character matrix with rows
% describing the functions
```

© 2003 by CRC Press LLC
which were integrated

User functions called: ftest, gcquad

---

global nvals

if nargin==0, secs=60; end

fprintf('
PRESS RETURN TO BEGIN COMPUTATION > ')
pause

% Summary of the five integrands used
names=strvcat('sqrt(x)','log(x)','humps(x)',
      'exp(10*x).*cos(10*pi*x)',...
      'cos(20*pi*x-20*sin(pi*x))');

fprintf(['

INTEGRATION TEST COMPARING',...
' FUNCTIONS QUADL AND GCQUAD
'])
fprintf('
The functions being integrated are:
')
disp(names)

% Compute exact values of integrals
exact=[2/3; -1; quadl(@humps,0,1,1e-12);
      real((exp(10+10*pi*i)-1)/(10+10*pi*i));
      besselj(20,20)];

% Find time to make a loop and call the clock
nmax=5000; nclock=0; t0=clock;
while nclock<nmax
    nclock=nclock+1; tclock=etime(clock,t0);
end
tclock=tclock/nclock;

% Evaluate each integral individually. Repeat
% the integrations for secs seconds to get
% accurate timing. Save results in array L.
for k=1:5
    nquad=0; tim=0; t0=clock;
    while tim<secs
        [v,nfuns]=quadl(@ftest,0,1,tol,[],k);
        nquad=nquad+1; tim=etime(clock,t0);
        end
    tim=tim/nquad-tclock; pe=100*(v/e(k)-1);
    L(k,:)=[v,nfuns,pe,tim];
end

warning on;

% Obtain time to compute base points and weight
% factors for a Gauss formula of order 100
nloop=100; t0=clock;
for j=1:nloop
    [dumy,bp,wf]=gcquad([],0,1,100,1);
end
tbpwf=etime(clock,t0)/nloop;

% Perform the Gauss integration using a
% vector integrand. Save results in array G
ngquad=0; tim=0; t0=clock;
while tim<secs
    v=ftest(bp,6)*wf;
    ngquad=ngquad+1; tim=etime(clock,t0);
end
tim=tim/ngquad+tbpwf-tclock; pe=100*(v./e-1);
G=[v,100*ones(5,1),pe,tim/5*ones(5,1)];

format short e
disp(' ')
disp(' Results Using Function quadl')
disp(...
    ' Integral Function Percent Computation')
disp(...
    ' values evaluations error seconds')
disp(L)
disp(' Results Using Function gcquad')
disp(...
    ' Integral Function Percent Computation')
disp(...
    ' values evaluations error seconds')
disp(G)

format short
disp(['(Total time using quadl)',...
    '(Total time using gcquad')]}
disp(['equals ',...
    num2str(sum(L(:,end))/sum(G(:,end)))])
disp(' ')

%====================================================================
function y=ftest(x,n)
% Integrands used by function quadl
global nvals
switch n
  case 1, y=sqrt(x); case 2, y=log(x);
  case 3, y=humps(x);
  case 4, y=exp(10*x).*cos(10*pi*x);
  case 5, y=cos(20*pi*x-20*sin(pi*x));
  otherwise
    x=x(:)'; y=[sqrt(x);log(x);humps(x);
      exp(10*x).*cos(10*pi*x);
      cos(20*pi*x-20*sin(pi*x))];
end
if n<6, nvals=nvals+length(x);
else, nvals=nvals+5*length(x); end

%=============================================
% function [val,bp,wf]=gcquad(func,xlow,...
% xhigh,nquad,mparts,varargin)
% See Appendix B

5.4 Geometrical Properties of Areas and Volumes

Geometrical properties of areas and volumes are often needed in physical applications such as linear stress analysis and rigid body dynamics. For example, consider a prismatic structural member having a general cross-section area denoted by $A$ with a boundary curve $L$. Analyzing the stresses occurring when the member undergoes axial compression and bi-axial bending leads to integrals of the form

$$C_{nm} = \int\int_{A} x^n y^m \, dx \, dy$$

for integers $n$ and $m$. The six most important cases and the related property names are:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>Symbol</th>
<th>Geometrical Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a</td>
<td>Area</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ax</td>
<td>First moment of area about the y-axis</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>ay</td>
<td>First moment of area about the x-axis</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>axx</td>
<td>Moment of inertia about the y-axis</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>ayy</td>
<td>Product of inertia with respect to the xy axes</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>ayy</td>
<td>Moment of inertia about the x-axis</td>
</tr>
</tbody>
</table>

The integral $C_{nm}$ can be evaluated for very general shapes by converting the area integral to a line integral over the boundary. Then, approximating the boundary curve
parametrically (by spline interpolation, for example) and using numerical integration
yield the desired values. Green’s theorem [119] relates area integrals and line inte-
grals according to
\[
\int_{A} \left[ \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} \right] dxdy = \oint_{L} Udy - Vdx
\]
where \( U(x, y) \) and \( V(x, y) \) are single-valued and differentiable functions inside and
on \( L \). This implies that
\[
\int_{A} x^n y^m dxdy = \frac{1}{n + m + 2} \int_{A} \left[ \frac{\partial}{\partial x} (x^{n+1}y^m) + \frac{\partial}{\partial y} (x^n y^{m+1}) \right] dxdy
\]
\[
= \frac{1}{n + m + 2} \oint_{L} x^n y^m (xdy - ydx),
\]
provided \( n + m + 2 \neq 0 \). We can even have negative \( n \) provided \( x = 0 \) is outside \( L \),
and negative \( m \) provided \( y = 0 \) is outside \( L \). The case \((n, m) = (0, -1)\) occurring
in curved beam theory can also be treated by line integration, but we will confine
attention to the six cases listed above.

If the boundary curve, \( L \), is parameterized as \( x(t), y(t), a \leq t \leq b \), then
\[
\oint_{L} x^n y^m (xdy - ydx) = \int_{a}^{b} x(t)^n y(t)^m [x(t)y'(t) - y(t)x'(t)] dt
\]
which is a one-dimensional integral amenable to numerical integration. When cubic
spline interpolation is used to represent the boundary, then \( x(t)y'(t) - y(t)x'(t) \) is a
piecewise polynomial function of degree four (not degree five as seems apparent at
first glance). Since a Gaussian quadrature formula of order \( N \) integrates exactly any
polynomial of degree \( 2N - 1 \) or less, the integral of interest can be integrated exactly
by taking \( 2N - 1 \geq 3n + 3m + 4 \). For our case, using a composite Gauss formula
of order six is appropriate. A program is given below to compute properties for a
general geometry that can have several separate parts, and these parts may contain
holes. The details of that program are discussed later.

The ideas for plane regions can be extended to three dimensions where volume,
gravity center location, and inertia tensor are the quantities being computed. Let
\[
\mathbf{R} = [x; y; z]
\]
be the Cartesian radius vector for points in a three-dimensional region \( W \) covered by
a surface \( S \). The Gauss divergence theorem [119] implies

\[
V = \iiint_W dx
dy
dz = \frac{1}{3}\iiint_S R \cdot \hat{n} dS,
\]

\[
V_r = \iiint_W R dx
dy
dz = \frac{1}{4}\iiint_S R(R \cdot \hat{n}) dS
\]

\[
V_{rr} = \iiint_W R' R dx
dy
dz = \frac{1}{5}\iiint_S R' R(R \cdot \hat{n}) dS.
\]

In the last equation, \( RR' \) is the matrix product \([x; y; z] \ast [x, y, z]\) and \( \hat{n} \) is the outward directed unit surface normal. We refer to \( V, V_r, \) and \( V_{rr} \) as the volume, the first moment of volume, and the second moment of volume. These quantities can be evaluated exactly for some special cases, such as polyhedra, and volumes of revolution.

The quantity \( V_{rr} \) is useful in rigid body dynamics where the inertia tensor, \( I_{rr} \), is needed to compute the rotational kinetic energy. The inertia tensor for a body having unit mass density can be computed from \( V_{rr} \) as

\[
I_{rr} = \text{eye}(3, 3) \text{ sum(diag}(V_{rr})) - V_{rr}.
\]

The corresponding inverse is

\[
V_{rr} = \frac{\text{eye}(3, 3)}{2} \text{ sum(diag}(I_{rr})) - I_{rr}.
\]

To illustrate the computation of volume properties, consider the instance where the surface has a parametric equation of the form

\[
R(u, v), \quad u_1 \leq u \leq u_2, \quad v_1 \leq v \leq v_2.
\]

For example, the ellipsoid defined by

\[
\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 + \left(\frac{z}{c}\right)^2 \leq 1
\]

has a surface equation

\[
R = [a \sin(u) \cos(v); b \sin(u) \sin(v); c \cos(v)], \quad 0 \leq u \leq \pi, \quad 0 \leq v \leq 2\pi.
\]

The unit surface normal and the differential of surface area can be computed as

\[
\hat{n} dS = \frac{\partial R}{\partial u} \times \frac{\partial R}{\partial v} dudv
\]

where the order of the cross product is chosen so that the outward directed normal is produced. Then,

\[
(R \cdot \hat{n}) dS = \det([R, R_u, R_v]) = D(u, v) du dv
\]

© 2003 by CRC Press LLC
and the integrals of interest become

\[
V = \frac{1}{3} \int_{u_1}^{u_2} \int_{v_1}^{v_2} D(u, v) dv du,
\]

\[
V_r = \frac{1}{4} \int_{u_1}^{u_2} \int_{v_1}^{v_2} R \ D(u, v) dv du,
\]

\[
V_{rr} = \frac{1}{5} \int_{u_1}^{u_2} \int_{v_1}^{v_2} R R' \ D(u, v) dv du.
\]

Note that the function \( D(u, v) \) vanishes at points where the radius vector \( R \) is perpendicular to the surface normal. A useful instance of the parametric form occurs when a closed curve is rotated to form a volume of revolution as illustrated in Figure 5.2.

Consider a curve defined parametrically in the \((x, z)\) plane as \( x(t), z(t), a \leq t \leq b \). If the curve is rotated about the \( z \) axis through angular limits \( \theta_1 \leq \theta \leq \theta_2 \), the lateral surface of the body has a surface equation

\[
R(t, \theta) = [\cos(\theta); \sin(\theta); 1]. * [x; x; z]
\]
and the volume property integrals reduce to

\[ V = \frac{1}{3} \oint_L x[\mathbf{dz} - zdx], \]

\[ \mathbf{V}_r = \frac{1}{4} \int_{\theta_1}^{\theta_1} \cos(\theta); \sin(\theta); 1 \cdot \oint_L [x^2; x^2; \mathbf{xz}] dxdz, \]

\[ \mathbf{V}_{rr} = \frac{1}{5} \int_{\theta_1}^{\theta_1} \cos(\theta); \sin(\theta); 1 \cdot \oint_L [x; x; z] \cdot [x; x; z] x dxdz. \]

The line integrals in the last three formulas have similar structure to those for plane area properties, but the highest polynomial power in the integrands of the form \( x^n z^m \) is one higher than that encountered for plane area properties. Taking a composite Gauss formula of order seven produces exact results for the volume properties when a spline interpolated boundary curve is used. A program employing these formulas is developed below.

### 5.4.1 Area Property Program

A program was written to compute the area, the centroidal coordinates, the inertial moments, and the product of inertia for general plane areas bounded by a series of spline curve segments. Shapes such as polygons, having one or more straight boundary segments, are also handled by allowing slope discontinuities at the ends of the straight segments. The program requires data points \( x_d(j), y_d(j), 1 \leq j \leq nd \) with the boundary traversed in a counterclockwise sense. The first and last points should be identical to make the curve closed. A set of point indices designating any slope discontinuities (such as those at the corners of a square) are also needed.

A typical geometry for the program appears in Figure 5.3. Program data created by the function `makrcrsq` employs 27 data points. A multiply connected geometry is treated as if it were a simply connected region by introducing fictitious cuts connecting outer boundaries and inner holes. Disconnected parts are also joined with zero width strips. Including the cuts and strips has no effect on the area properties because related boundary segments are traversed twice, but in opposite directions. Consequently, the corresponding line integral contributions from the fictitious parts cancel. The complete boundary is parameterized as a spline curve in complex form

\[ z(t) = x(t) + iy(t), \quad 1 \leq t \leq nd \]

with

\[ z(j) = x_d(j) + iy_d(j), \quad j = 1, 2, \ldots, nd. \]
The boundary curve and its derivatives are piecewise polynomial functions. Exact results for the geometrical properties are obtained by using the function \texttt{gcquad} to generate Gauss base points and weight factors for integration limits from 1 to \( nd \), and a number of integration segments equal to \( nd - 1 \). The various area properties are accumulated in vector mode for computational efficiency.

The function \texttt{runaprop} is the main driver for the program. It accepts boundary data, calls the function \texttt{aprop} to compute area properties, prints results, and plots the geometry. If no input data are given, the function \texttt{makrcsq} is called to create data for the illustrative example. The plot produced by the program resembles the one shown, without inclusion of point indices.
The numerical output for our example has the following simple form:

```>> runaprop;

GEOMETRICAL PROPERTY ANALYSIS USING FUNCTION APROP

  A   XCG   YCG   AXX   AXY   AYY
16.7147  3.0000  4.1254  176.3369  206.8620  359.6076

>>
```

Program for Properties of Spline Bounded Areas

```function [p,x,y,xd,yd]=areaprog(xd,yd,icrnr)
    
    % This function calls function aprop which computes geometrical properties for an area bounded by a spline curve through data points in (xd,yd).
    
    % User functions called: aprop

    if nargin==2,icrnr=[1,length(xd)]; end
    titl='AREA IN THE XY PLANE';

    if nargin==0
        [xd,yd,icrnr]=makrcsq;
        titl=...
        'HALF ANNULUS ABOVE A SQUARE WITH A HOLE';
    end

    disp(' ')
    disp([' GEOMETRICAL PROPERTY ANALYSIS',... ' USING FUNCTION APROP'])
    [p,z]=aprop(xd,yd,icrnr); x=real(z); y=imag(z);
    disp(' ');
    disp([', A   XCG   YCG   ',...
    '   AXX   AXY   AYY']);
    disp(p), close, plot(xd,yd,'ko',x,y,'k-')

    xlabel('x axis'), ylabel('y axis')
    title(titl),axis(cubrange([x(:),y(:)],1.2));
```
function [p,zplot]=aprop(xd,yd,kn)
%
% This function determines geometrical properties
% of a general plane area bounded by a spline
% curve
% xd,yd - data points for spline interpolation
% with the boundary traversed in counter-
% clockwise direction. The first and last
% points must match for boundary closure.
% kn - vector of indices of points where the
% slope is discontinuous to handle corners
% like those needed for shapes such as a
% rectangle.
% p - the vector [a,xcg,ycg,axx,axy,ayy]
% containing the area, centroid coordinates,
% moment of inertia about the y-axis,
% product of inertia, and moment of inertia
% about the x-axis
% zplot - complex vector of boundary points for
% plotting the spline interpolated geometry.
% The points include the numerical quadrature
% points interspersed with data values.
% User functions called: gcquad, curve2d

if nargin==0
td=linspace(0,2*pi,13); kn=[1,13];
xd=cos(td)+1; yd=sin(td)+1;
end
nd=length(xd); nseg=nd-1;
[dum,bp,wf]=gcquad([],1,nd,6,nseg);
[z,zplot,zp]=curve2d(xd,yd,kn,bp);
w=[ones(size(z)), z, z.*conj(z), z.^2].*...
repmat(imag(conj(z).*zp),1,4);
v=(wf' * w)./ [2,3,8,8]; vr=real(v); vi=imag(v);
p=[vr(1:2),vi(2),vr(3)+vr(4),vi(4),vr(3)-vr(4)];
p(2)=p(2)/p(1); p(3)=p(3)/p(1);
function [z,zplot,zp]=curve2d(xd,yd,kn,t)
%
% [z,zplot,zp]=curve2d(xd,yd,kn,t)
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function generates a spline curve through
% given data points with corners(slope dis-
% continuities) allowed as selected points.
% xd,yd - real data vectors of length nd
% defining the curve traversed in
% counterclockwise order.
% kn - vectors of point indices, between one
% and nd, where slope discontinuities
% occur
% t - a vector of parameter values at which
% points on the spline curve are
% computed. The components of t normally
% range from one to nd, except when t is
% a negative integer,-m. Then t is
% replaced by a vector of equally spaced
% values using m steps between each
% successive pair of points.
% z - vector of points on the spline curve
% corresponding to the vector t
% zplot - a complex vector of points suitable
% for plotting the geometry
% zp - first derivative of z with respect to
% t for the same values of t as is used
% to compute z
%
% User m functions called: splined
%----------------------------------------------

nd=length(xd); zd=xd(:)+i*yd(:); td=(1:nd)';
if isempty(kn), kn=[1;nd]; end
kn=sort(kn(:)); if kn(1)==1, kn=[1;kn]; end
if kn(end)==nd, kn=[kn;nd]; end
N=length(kn)-1; m=round(abs(t(1))); if -t(1)==m, t=linspace(1,nd,1+N*m)'; end
z=[]; zp=[]; zplot=[];
for j=1:N
    k1=kn(j); k2=kn(j+1); K=k1:k2;
k=find(k1<=t & t<k2);
if j==N, k=find(k1<=t & t<=k2); end
if isempty(k)
zk = spline(K, zd(K), t(k)); z = [z; zk];
zplot = [zplot; zd(k1); zk];
if nargout == 3
    zp = [zp; splined(K, zd(K), t(k))];
end
end
zplot = [zplot; zd(end)];

%==================================================================
function [x, y, icrnr] = makrcsq
%
% [x, y, icrnr] = makrcsq
% ~~~~~~~~~~~~~~~~~~~~
% This function creates data for a geometry
% involving half of an annulus placed above a
% square containing a square hole.
% x, y = data points characterizing the data
% icrnr = index vector defining corner points
% User m functions called: none
%==================================================================
xshift = 3.0; yshift = 3.0;
a = 2; b = 1; narc = 7; x0 = 0; y0 = 2*a - b;
xy = [a, -a, -b, b, -b, -b, -a, a, a,
     a, b, -b, -b, b, a, -a, a]';
theta = linspace(0, pi, narc)';
c = cos(theta); s = sin(theta);
xy = [xy; [x0 + a*c, y0 + a*s]];
c = flipud(c); s = flipud(s);
xy = [xy; [x0 + b*c, y0 + b*s]; [a, y0]; [a, a]];
x = xy(:, 1) + xshift; y = xy(:, 2) + yshift;
icrnr = [(1:12)'; 11+narc; 12+narc; ... 11+2*narc; 12+2*narc; 13+2*narc];

%==================================================================
% function [val, bp, wf] = gcquad(func, xlow, ...
%     xhigh, nquad, mparts, varargin)
% See Appendix B
%==================================================================
5.4.2 Program Analyzing Volumes of Revolution

Since the geometrical property computation for a volume of revolution is quite similar to that for area properties, the same functions \texttt{gcquad} and \texttt{curve2d} used in the area program are employed below to compute the volume, centroidal coordinates, and inertia tensor of a solid generated by rotating a spline curve through arbitrary angular limits about the \( z \) axis. In the following program, the function \texttt{volrevol} calls a general purpose function \texttt{volrev} which computes the geometrical properties and plots the related volume of revolution. The function \texttt{volrev} depends on \texttt{gcquad}, \texttt{curve2d}, function \texttt{rotasurf} to plot the body surface, and function \texttt{anglefun} which deals with rotation angle dependence. The function returns volume properties and surface coordinates on the solid. Area properties of the cross section are also obtained.

The geometry in Figure 5.4 was analyzed. The area, rotated through 270 degrees, consists of the bottom half of a semicircle capped on the outer radius by a square which is also capped by a smaller semicircle. Results from \texttt{volrev} were confirmed to agree closely with another function \texttt{srfv}, which is discussed in the next section. Because the function \texttt{srfv} employs triangular surface elements, the two computational models are not identical. This accounts for the slight difference in numerical results.

In conclusion, the volume property program to handle volumes of revolution for spline interpolated cross sections was found to be a useful extension of the methods developed earlier for properties of areas.

\begin{verbatim}
>> volrevol

PROPERTIES OF A VOLUME OF REVOLUTION
Results Using Function VOLREV
Volume = 59.1476
Rg = [0.91028 0.91028 0.13755]

Inertia Tensor =
1.0e+003 *
0.5773 0.1168 -0.0090
0.1168 0.5773 -0.0090
-0.0090 -0.0090 1.1010

Area Properties
\end{verbatim}
area xcentr zcentr.
2.9638 4.2350 0.1045
axx axz azz
53.8408 1.7264 1.3101
Results Using Function SRFV
Volume = 59.1056
Rg = [0.91016 0.91016 0.13749]
Inertia Tensor =
1.0e+003 *
0.5768 0.1167 -0.0090
0.1167 0.5768 -0.0090
-0.0090 -0.0090 1.0999
>>

Figure 5.4: Partial Volume of Revolution

Program for Properties of a Volume of Revolution

1: function volrevol
2: %
3: % volrevol
4: % -----------
% This program determines geometrical properties
% for a solid generated by rotating a closed spline
% curve through an arbitrary angle about the z-axis.
% A detailed description of the geometry is given in
% function volrev.
%
% User m functions called: volrev srfv
%----------------------------------------------
% Data for a cross section consisting of the lower
% half of a circle plus a square capped by the
% upper half of a smaller semicircle. The geometry
% is rotated through 270 degrees about the z-axis.

n1=9; t1=-pi:pi/n1:0; n2=6; t2=0:pi/n2:pi;
Zd=[0,exp(i*t1),1/2+i+exp(i*t2)/2,0];
xd=real(Zd)+4; zd=imag(Zd);

th=[-pi/2,pi]; nth=31;
kn=[1,2,n1+2,n1+3,n1+n2+3,n1+n2+4];

% Compute the geometrical properties
[v,rg,Irr,x,y,z,aprop]=volrev(...
    xd,zd,kn,th,nth);
disp(' ')
disp('PROPERTIES OF A VOLUME OF REVOLUTION')
disp(' ')
disp('Results Using Function VOLREV')
disp(['Volume = ',num2str(v)], disp(' ')
disp(['Rg = [',num2str(rg(:)),']'], disp(' ')
disp('Inertia Tensor ='), disp(Irr), disp(' ')
disp('Area Properties'), disp(' ')
disp(aprop(1:3))
disp(' axx axz azz')
disp(aprop(4:6))

% Run a second case to generate a dense set of
% surface coordinates to check results using
% function srfv.

N1=61; T1=-pi:pi/N1:0; N2=41; T2=0:pi/N2:pi;
Zd=[0,exp(i*T1),1/2+i+exp(i*T2)/2,0];
xxd=real(Zd)+4; zzd=imag(Zd);

th=[-pi/2,pi]; Nth=121;
Kn=[1,2,N1+2,N1+3,N1+N2+3,N1+N2+4];

[V,Rg,IRR,X,Y,Z]=volrev(...
    xxd,zzd,Kn,th,Nth,1);
[vt,rct,vrrt]=srfv(X,Y,Z);
disp('Results Using Function SRFV')
disp(['Volume = ',num2str(vt)])
disp(['Rg = [',num2str(rct(:))','])
disp('Inertia Tensor ='), disp(vrrt)

%===============================================

function [v,rg,IRR,X,Y,Z,aprop,xd,zd,kn]=...
    volrev(xd,zd,kn,th,nth,noplot)

% [v,rg,IRR,X,Y,Z,aprop,xd,zd,kn]=...
% volrev(xd,zd,kn,th,nth,noplot)

% This function computes geometrical properties
% for a volume of revolution resulting when a
% closed curve in the (x,z) plane is rotated,
% through given angular limits, about the z axis.
% The cross section of the volume is defined by
% a spline curve passed through data points
% (xd,zd) in the same manner as was done in
% function areaprop for plane areas.

% xd,zd - data vectors defining the spline
% interpolated boundary, which is
% traversed in a counterclockwise
% direction
% kn - indices of any points where slope
% discontinuity is allowed to turn
% sharp corners
% p - vector of volume properties containing
% [v, xcg, ycg, zcg, vxx, vyy, vzz,...
% vxy, vyz, vzx] where v is the volume,
% (xcg,ycg,zcg) are coordinates of the
% centroid, and the remaining properties
% are volume integrals of the following
% integrand:
% [x.^, y.^2, z.^2, xy, yz, zx]*dxdydz
% X,Y,Z - data arrays containing points on the
% surface of revolution. Plotting these
points shows the solid volume with
the ends left open. Function fill3
is used to plot the surface with ends
closed
aprop - a vector containing properties of the
area in the (x,z) plane which was used
to generate the volume. aprop=[area,...
xcentroidal, ycentroidal, axx, axz,azzi].

User m functions called: rotasurf, gcquad,
curve2d, anglefun, splined

if nargin==0
    t1=-pi:pi/6:0; t2=0:pi/6:pi;
    Zd=[0,exp(i*t1),1/2+i+exp(i*t2)/2,0,-1];
    xd=real(Zd)+4; zd=imag(Zd);
    kn=[1,2,8,9,15,16];
    th=[-pi/2,pi]; nth=31;
end

Plot a surface of revolution based on the
input data points
if nargin==6
    [X,Y,Z]=rotasurf(xd,zd,th,nth,1);
else
    [X,Y,Z]=rotasurf(xd,zd,th,nth); pause
end

Obtain base points and weight factors for the
composite Gauss formula of order seven used in
the numerical integration
nd=length(xd); nseg=nd-1;
[dum,bp,wf]=gcquad([],1,nd,7,nseg);

Evaluate complex points and derivative values
on the spline curve which is rotated to form
the volume of revolution
[u,uplot,up]=curve2d(xd,zd,Kn,bp);
plot(real(uplot),imag(uplot)), axis equal,shg
u=u(:); up=up(:); n=length(bp);
x=real(u); dx=real(up); z=imag(u);
dz=imag(up); da=x.*dz-z.*dx;

Evaluate line integrals for area properties
p=[ones(n,1), x, z, x.^2, x.*z, z.^2, x.^3,...
(x.^2).*z, x.*(z.^2)].*repmat(da,1,9);

p=(wf(:)'*p)./[2 3 3 4 4 5 5 5];

% Scale area properties by multipliers involving
% the rotation angle for the volume
f=anglefun(th(2))-anglefun(th(1));
v=f(1)*p(2); rg=f([2 3 1]).*p([4 4 5])/v;

% Generate multipliers from rotation angle
vrr=[f([4 5 2]); f([5 6 3]); f([2 3 1])].*;
    [p([7 7 8]); p([7 7 8]); p([8 8 9])];

Irr=eye(3)*sum(diag(vrr))-vrr;
aprop=[p(1),p(2:3)/p(1),p(4:6)];

%=================================

function f=anglefun(t)
% f=anglefun computes multipliers involving
% t, the rotation angle of the volume.
c=cos(t); s=sin(t);
f=[t,s,-c,(t+c*s)/2,s*s/2,(t-c*s)/2];

%=================================

function [x,y,z,xd,zd]=rotasurf(xd,zd,th,nth,noplot)
% This function generates points on a surface of
% revolution generated by rotating an area in
% the (x,z) plane about the z-axis
% xd,yz - coordinate data for the curve in the
% (x,y) which forms the cross section
% th - [ThetaMin,ThetaMax] defining limits of
% rotation angle about the z-axis
% nth - number of theta values used to generate
% surface values
% noplot - option given any value if no plot is
% desired. Otherwise omit this value.
% x,y,z - arrays of points on the surface
% User m functions called: none

if nargin==0
    n1=9; t1=-pi:pi/n1:0; n2=6; t2=0:pi/n2:pi;
    Zd=[0,exp(i*t1),1/2+i+exp(i*t2)/2,0];
    xd=real(Zd)+4; zd=imag(Zd);
th=[-pi/2,pi]; nth=31;
end
xd=xd(:); zd=zd(:); nd=length(xd);
t=linspace(th(1),th(2),nth);
x=xd*cos(t); y=xd*sin(t); z=repmat(zd,1,nth);
if nargin==5, return; end
close; surf(x,y,z), title('VOLUME OF REVOLUTION')
xlabel('x axis'), ylabel('y axis')
zlabel('z axis'), colormap([1 1 1]); hold on
fill3(x(:,1),y(:,1),z(:,1),'w')
fill3(x(:,end),y(:,end),z(:,end),'w')
axis equal, grid on, hold off, shg

%=========================================%
% function [z,zplot,zp]=curve2d(xd,yd,kn,t)
% See Appendix B
%=========================================%
% function [val,bp,wf]=gcquad(func,xlow,...
% xhigh,nquad,mparts,varargin)
% See Appendix B
%=========================================%
% function range=cubrange(xyz,ovrsiz)
% See Appendix B
%=========================================%
% function val=splined(xd,yd,x,if2)
% See Appendix B
%=========================================%
% function [v,rc,vrr]=srfv(x,y,z)
% See Appendix B

5.5 Computing Solid Properties Using Triangular Surface Elements and Using Symbolic Math

In this section a numerical method is developed to compute properties of a solid covered by triangular surface elements. An example problem is analyzed by a nu-
Numerical method and also by use of the symbolic math toolbox. Results of the two analyses are compared.

Many familiar solid bodies such as an ellipsoid, a conical frustum, or a torus have surfaces readily parameterized by equations of the form

\[ R = R(u, v), \quad U_1 \leq u \leq U_2, \quad V_1 \leq v \leq V_2. \]

This is the type of equation implied when MATLAB function \texttt{surf} uses rectangular \( X, Y, Z \) coordinate arrays to depict a curvilinear coordinate net covering a surface. The surface is approximated by a series of quadrilateral surface patches. Geometrical properties of the related solid can be computed approximately by dividing each quadrilateral into two triangular patches, and accumulating the surface integral contributions of the triangles. This approach is attractive because the surface integral properties of triangles can be computed exactly, and all triangles can be processed in parallel. Although the geometrical properties for a solid covered by triangular patches can be computed exactly, the reader should realize that many surface elements may be required to achieve several digit accuracy for highly curved surfaces.

To pursue our ideas, consider the solid in Figure 5.5 which resembles a twisted rope. This body has its outer surface (as distinguished from its ends) described by the following set of equations:

\[
\begin{align*}
  x &= x_0 + \rho \cos(p), \quad y = y_0 + \rho \sin(p), \quad z = z_0 - \xi \sin(mp) + \eta \cos(mp), \\
  \rho &= a + \xi \cos(mp) + \eta \sin(mp), \quad 0 \leq t \leq 2\pi, \quad 0 \leq p \leq 3\pi, \\
  \xi &= b \cos(t) [\cos(t)], \quad \eta = b \sin(t) [\cos(t)].
\end{align*}
\]

The cross section of the solid is two circular disks touching tangentially. The solid is swept out as the centroid of the area (where the circles touch) moves along a helical path and twists simultaneously. The parameter choices used in our example are

\[ a = 3, \quad b = 1, \quad m = 6, \quad x_0 = y_0 = 0, \quad z_0 = 3\pi/2 \]

which places the centroid of the solid on the \( y \)-axis and makes the ends of the rope lie in the \( xz \) plane. Then the geometrical property contributions from both end surfaces are zero because \( \hat{\eta} \cdot \vec{R} \) vanishes on the ends.

Let us next think about a solid with its surface composed of triangular patches. For a generic patch with corners at \( \vec{R}_i, \vec{R}_j, \vec{R}_k \), denote the surface area as \( S_T \) and the unit surface normal as \( \hat{\eta} \). Then

\[
\hat{\eta} S_T = \frac{1}{2}(\vec{R}_j - \vec{R}_i) \times (\vec{R}_k - \vec{R}_i),
\]

and the triangle centroid is at

\[
\vec{R}_C = \frac{1}{3}(\vec{R}_i + \vec{R}_j + \vec{R}_k).
\]

If \( h \) is the normal distance from the origin to the plane containing the triangle, then

\[
h = \hat{\eta} \cdot \vec{R}_i \quad \text{and} \quad S_T = |\hat{\eta} S_T|.
\]

The first two volume properties are just

\[
V_T = \frac{1}{3} \int \int_{S_T} \hat{\eta} \cdot \vec{R} \, dS = \frac{h}{3}S_T.
\]
Figure 5.5: Solid Resembling a Twisted Rope
and
\[
(V_R)_T = \frac{1}{4} \int_S (\hat{\eta} \cdot R) R dS = \frac{h}{4} S_T R_C.
\]

The remaining inertial property integral is
\[
(V_{RR})_T = \frac{1}{5} \int_S (\hat{\eta}R) R R' dS = \frac{h}{60} S_T (R_i R'_i + R_j R'_j + R_k R'_k + 9 R_C R'_C).
\]

These formulas were used to develop the function `srfv` which computes geometrical properties for a surface described by the same type of data arrays as those used by the function `surf`. Each quadrilateral patch is divided into two triangles, and the contributions of all triangles are accumulated in vectorized mode for computational efficiency.

The function `ropesymu` in the following program calls function `twistprop` to perform numerical computation, function `twistprop` to perform symbolic computation, and function `ropedraw` to plot the geometry of the twisted rope. `Twistprop` calls the function `srfv` which is a general routine to compute properties of solid bodies modeled with triangular surface elements. The numerical example employs point arrays of dimension 804 by 100 to obtain the numerical solution. Results for the numerical and symbolic computations are shown next along with the computer code. Note that the numerical and exact solutions agreed within 0.2 percent. The numerical solution took about 1.3 secs compared with 314 seconds for the symbolic solution. Even though the symbolic solution took 238 times as long to compute as the numerical solution, the symbolic coding was simple and might be appealing in specific situations where the related integrals can be evaluated exactly.

### 5.6 Numerical and Symbolic Results for the Example

**COMPARISON OF NUMERICAL AND SYMBOLIC GEOMETRICAL PROPERTIES FOR A TWISTED ROPE**

**FOR THE TRIANGULAR SURFACE PATCH MODEL**

Volume = 44.3239

\[
Rg = [1.6932e-015 \quad 0.64979 \quad 3.0068e-015]
\]

\[
Irr = \\
548.6015 \quad -0.0000 \quad 29.0040 \\
-0.0000 \quad 548.6015 \quad -0.0000 \\
29.0040 \quad -0.0000 \quad 423.7983
\]

Computation Time = 1.3194 Secs.
FOR THE SYMBOLIC MODEL
Volume = 44.4132
Rg = [0 0.64982 0]
Irr =
    549.7423 0 29.0639
    0 549.7423 0
    29.0639 0 424.7014

Computation Time = 314.28 Secs.

NUMERICAL APPROXIMATION ERROR USING TRIANGULAR SURFACE PATCHES. THE ERROR VALUES ARE DEFINED AS
NORM(APPROX.-EXACT)/NORM(EXACT)
Volume Error = 0.0020102
Centroidal Radius Error = 4.7287e-005
Inertia Tensor Error = 0.0020768

COMPARISON OF SOLUTION TIMES
(Symbolic Time)/(Numerical Time) = 238.1992

Program ropesymu

```matlab
function [vn,rcn,irrn,vs,rcs,irrs,times,nt,np]=...  
    ropesymu(A,B,M,X0,Y0,Z0,nt,np)

% [vn,rcn,irrn,vs,rcs,irrs,times,nt,np]=ropesymu(...  
% A,B,M,X0,Y0,Z0,nt,np)
%---------------------------------------------------------
% This program computes geometrical properties of a
% twisted rope having a cross section which is two
% circles of diameter B touching tangentially. The
% tangency point is at distance A from the rotation
% axis z. As the area is rotated, it is also twisted
% in a helical fashion. For a complete revolution
% about the z axis, the area is twisted through m
% turns. The resulting surface resembles a rope
% composed of two strands. Two results are obtained
% 1) by a numerical method where the surface is
% modeled with triangular surface patches and
% 2) by symbolic math. See functions twistrope and
% twistprop for descriptions of the problem parameters.
```

© 2003 by CRC Press LLC
% Numerical results and computation times for the two methods are compared, and the related surface geometry is plotted

% User functions called: twistrope twistprop ropedraw

%-------------------------------------

if nargin==0 % Default data case
A=3; B=1; m=6; np=201; nt=25;
X0=0; Y0=0; Z0=-3*pi/2; M=6;
end

disp('  
COMPARISON OF NUMERICAL AND SYMBOLIC
GEOMETRICAL PROPERTIES FOR A TWISTED ROPE

% Run the first time to get a crude grid for plotting
[vn,rcn,irrn,x,y,z,c]=twistrope(A,B,M,X0,Y0,Z0,nt,np);

% Numerical solution using a dense point grid to get close comparison with exact results. Calculations are run repeatedly for accurate timing.
Nt=4*nt; Np=4*np; n=50; tic;
for i=1:n
[vn,rcn,irrn]=twistrope(A,B,M,X0,Y0,Z0,Nt,Np);
end
timn=toc/n;

% Perform the symbolic analysis. This takes a long time.
tic;
[v,rc,vrr,vs,rcs,irrs]=twistprop(A,B,M,X0,Y0,Z0);
tims=toc; times=[timn,tims];

disp('  
FOR THE TRIANGULAR SURFACE PATCH MODEL

Volume = ',num2str(vn))
Rg = [' ',num2str(rcn(:))']
Irr = 'disp(irrn)
Computation Time = ',num2str(timn),' Secs.'

% Print numerical comparisons of results

FOR THE SYMBOLIC MODEL'

© 2003 by CRC Press LLC
GAUSS INTEGRATION WITH GEOMETRIC PROPERTY APPLICATIONS 175

65: disp(['Volume = ',num2str(vs)])
66: disp(['Rg = [' ,num2str(rcc(:)) ,']'])
67: disp('Irr = '), disp(irms)
68: disp(['Computation Time = ',num2str(tims),' Secs.'])
69: disp('
')
70: disp('NUMERICAL APPROXIMATION ERROR USING TRIANGULAR')
71: disp('SURFACE PATCHES. THE ERROR VALUES ARE DEFINED AS')
72: disp(' NORM(APPROX.-EXACT)/NORM(EXACT)')
73: evol=abs(vn-vs)/vs; erad=norm(rcc(:)-rccs)/norm(rcc);
74: einert=norm(irms-irms)/norm(irms);
75: disp(['Volume Error = ',num2str(evol)])
76: disp(['Centroidal Radius Error = ',num2str(erad)])
77: disp(['Inertia Tensor Error = ',num2str(einert)])
78: disp('
')
79: disp('COMPARISON OF SOLUTION TIMES')
80: disp(['(Symbolic Time)/(Numerical Time) = ',...
81: num2str(tims/timn)])
82: disp('
')
83: disp(' % Draw the surface using a crude grid to avoid')
84: disp(' % crowded grid lines')
85: ropedraw(A,B,np,nt,M,X0,Y0,Z0);
86: disp('%================================================================')
87: function [x,y,z,t]=ropedraw(a,b,np,nt,m,x0,y0,z0)
88: % [x,y,z,t]=ropedraw(a,b,np,nt,m,x0,y0,z0)
89: % This function draws the twisted rope.
90: if nargin==0
91: a=3; b=1; np=200; nt=25; m=6;
92: x0=0; y0=0; z0=-3*pi/2;
93: end
94: % Draw the surface
95: t=linspace(0,2*pi,nt); p=linspace(0,3*pi,np)';
96: xi=b*cos(t).*abs(cos(t)); eta=b*sin(t).*abs(cos(t));
97: rho=a+xi.*cos(m*p)+eta.*sin(m*p);
98: x=rho.*cos(p)+x0; y=rho.*sin(p)+y0;
99: z=-xi.*sin(m*p)+eta.*cos(m*p)+p+z0;
100: close; surf(x,y,z,t), title('TWISTED ROPE')
xlabel('x axis'), ylabel('y axis'), zlabel('z axis')
colormap('prism(4)'), axis equal, hold on

% Fill the ends
fill3(x(1,:),y(1,:),z(1,:),'w')
fill3(x(end,:),y(end,:),z(end,:),'w')
view([-40,10]), hold off, shg

%===========================================

function [v,rc,vrr,V,Rc,Irr]=twistprop(A,B,M,X0,Y0,Z0)
% 
% [v,rc,vrr,V,Rc,Irr]=twistprop(A,B,M,X0,Y0,Z0)
%  % This function computes geometrical properties of
%  % a twisted rope. Exact results are obtained using
%  % symbolic math to evaluate three surface integrals
%  % for the volume, centroidal radius, and inertia
%  % tensor. The symbolic calculations take about five
%  % minutes to run.
%  %
%  % A,B,N - parameters defining the twisted rope
%  % X0,Y0,Z0 - center coordinates for the centroid of
%  % the twisted rope
%  % v,rc - symbolic formulas for the volume and
%  % centroid radius
%  % vrr - symbolic formula for integral of
%  % r*r'*d(vol)
%  % V,Rc - numerical values for volume and
%  % centroid radius
%  % Irr - numerical value for the inertia tensor
%
if nargin==0
    A=6; B=1; M=6; X0=1; Y0=2; Z0=3;
end

syms a b m t p xi eta rho x y z r rt rp real
syms x0 y0 z0 real
syms n dv dv1 v vr1 vr rg vrr1 vrr real
a=sym(A); b=sym(B); Pi=sym('pi');
x0=sym(X0); y0=sym(Y0); z0=sym(Z0);

% Surface equation for the twisted rope
xi=b*cos(t)*abs(cos(t));
eta=b*sin(t)*abs(cos(t));
GAUSS INTEGRATION WITH GEOMETRIC PROPERTY APPLICATIONS

rho = a + x1*cos(m*p) + eta*sin(m*p);
x = rho*cos(p) + x0; y = rho*sin(p) + y0;
z = -x1*sin(m*p) + eta*cos(m*p) + p + z0;
Pi = sym('pi');

% Tangent vectors
r = [x; y; z]; rt = diff(r, t); rp = diff(r, p);

% Integrate to get the volume
dv = det([r, rp, rt]); dv1 = int(dv, t, 0, 2*Pi);
v = simple(int(dv1, p, 0, 3*Pi)/3);

% First moment of volume
vrl = int(r*dv, t, 0, 2*Pi);
vr = simple(int(vrl, p, 0, 3*Pi)/4);

% Radius to the centroid
rc = simple(vr/v);

% Integral of r*r'd(vol)
vrr1 = int(r*r'*dv, t, 0, 2*Pi);
vrr = simple(int(vrr1, p, 0, 3*Pi)/5);

% Obtain numerical values
V = double(subs(v, {a, b, m, x0, y0, z0}, ...
{A, B, M, X0, Y0, Z0}));
Rc = double(subs(rc, {a, b, m, x0, y0, z0}, ...
{A, B, M, X0, Y0, Z0}));
Irr = double(subs(vrr, {a, b, m, x0, y0, z0}, ...
{A, B, M, X0, Y0, Z0}));

% Rigid body inertia tensor for a
% body of unit mass density
Irr = eye(3, 3)*sum(diag(Irr)) - Irr;

%=================================================================

function [v, rc, vrr, x, y, z, t] = twistrope(...
a, b, m, x0, y0, z0, nt, np)

% [v, rc, vrr, x, y, z, t] = twistrope(...
a, b, m, x0, y0, z0, nt, nm)
%-----------------------------------------------
% Geometrical properties of a twisted rope.
% This example takes 1.3 seconds to run
if nargin<8, np=321; end; if nargin<7, nt=161; end
if nargin==0
    a=6; b=1; m=6; x0=1; y0=2; z0=3;
end
t=linspace(0,2*pi,nt); p=linspace(0,3*pi,np)';
t=repmat(t,np,1); p=repmat(p,1,nt);

% Surface equation for the twisted rope

xi=b*cos(t).*abs(cos(t));
eta=b*sin(t).*abs(cos(t));
rho=a+xi.*cos(m*p)+eta.*sin(m*p);
x=rho.*cos(p)+x0; y=rho.*sin(p)+y0;
z=-xi.*sin(m*p)+eta.*cos(m*p)+p+z0;

[v,rc,vrr]=srfv(x,y,z);

%===========================================

function [v,rc,vrr]=srfv(x,y,z)
% 
% [v,rc,vrr]=srfv(x,y,z)
% ~~~~~~~~~~~~~~~~~~~~~~
% This function computes the volume, centroidal
% coordinates, and inertial tensor for a volume
% covered by surface coordinates contained in
% arrays x,y,z
% x,y,z - matrices containing the coordinates
% of a grid of points covering the
% surface of the solid
% v - volume of the solid
% rc - centroidal coordinate vector of the
% solid
% vrr - inertial tensor for the solid with the
% mass density taken as unity
% User functions called: scatripl proptet
%-----------------------------------------------

% p=inline(...
% 'v*(eye(3)*(r(:))''*r(:))-r(:)*r(:)''','v','r');
%d=mean([x(:),y(:),z(:)]);
GAUSS INTEGRATION WITH GEOMETRIC PROPERTY APPLICATIONS

\%x=x-d(1); y=y-d(2); z=z-d(3);

\[ n,m=\text{size}(x); i=1:n-1; I=i+1; j=1:m-1; J=j+1; \]
\[ xij=x(i,j); yij=y(i,j); zij=z(i,j); \]
\[ xIj=x(I,j); yIj=y(I,j); zIj=z(I,j); \]
\[ xIJ=x(I,J); yIJ=y(I,J); zIJ=z(I,J); \]
\[ xij=x(i,J); yij=y(i,J); zij=z(i,J); \]
\[ xIj=x(I,J); yIj=y(I,J); zIj=z(I,J); \]
\[ \% \text{Tetrahedron volumes} \]
\[ v1=\text{scatripl}(xij,yij,zij,xIj,yIj,zIj,xIJ,yIJ,zIJ); \]
\[ v2=\text{scatripl}(xij,yij,zij,xIJ,yIJ,zIJ,xiJ,yiJ,ziJ); \]
\[ v=\text{sum}(\text{sum}(v1+v2)); \]
\[ \% \text{First moments of volume} \]
\[ X1=xij+xIj+xIJ; X2=xij+xIJ+xiJ; \]
\[ Y1=yij+yIj+yIJ; Y2=yij+yIJ+yiJ; \]
\[ Z1=zij+zIj+zIJ; Z2=zij+zIJ+ziJ; \]
\[ vx=\text{sum}(\text{sum}(v1.*X1+v2.*X2)); \]
\[ vy=\text{sum}(\text{sum}(v1.*Y1+v2.*Y2)); \]
\[ vz=\text{sum}(\text{sum}(v1.*Z1+v2.*Z2)); \]
\[ \% \text{Second moments of volume} \]
\[ vrr=\text{proptet}(v1,xij,yij,zij,xIj,yIj,zIj,... \]
\[ xIj,yIJ,zIJ,X1,Y1,Z1)+... \]
\[ \text{proptet}(v2,xij,yij,zij,xIJ,yIJ,zIJ,... \]
\[ xIJ,yIJ,zIJ,X2,Y2,Z2); \]
\[ rc=[vx,vy,vz]/v/4; vs=\text{sign}(v); \]
\[ v=\text{abs}(v)/6; vrr=vs*vrr/120; \]
\[ vrr=[vrr([1 4 5]), vrr([4 2 6]), vrr([5 6 3])']; \]
\[ vrr=\text{eye}(3,3)*\text{sum}(\text{diag}(vrr))-vrr; \]
\[ \%vrr=vrr-p(v,rc)+p(v,rc+d); rc=rc+d; \]
\%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\textbf{function} v=\text{scatripl}(ax,ay,az,bx,by,bz,cx,cy,cz)
\%\textbf{function} v=\text{scatripl}(ax,ay,az,bx,by,bz,cx,cy,cz)
\% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\% Scalar triple product dot(cross(a,b),c) where
\% the cartesian components of vectors a,b,and c
\% are given in arrays of the same size.
\% v=ax.*(by.*cz-bz.*cy)+ay.*(bz.*cx-bx.*cz)...
\% +az.*(bx.*cy-by.*cx);
5.7 Geometrical Properties of a Polyhedron

A polyhedron is a solid covered by polygonal faces. Since polyhedra with sufficiently many faces can approximate volumes of complex shape, computing the volume, centroidal position, and inertia tensor of a polyhedron has useful applications. A polyhedron can be treated as the combination of a number of pyramids with bases which are the polyhedron faces and apexes located at the coordinate origin. Once the geometrical properties of a pyramid are known, results for a polyhedron are found by combining results for all faces [111].

Consider a general volume \( V \) covered by surface \( S \). It follows from the divergence theorem of Gauss [59] that

\[
\iiint_V X^n Y^m Z^\ell \, dX \, dY \, dZ = \frac{1}{n + m + \ell + 3} \iiint_S X^n Y^m Z^\ell (\hat{\eta} \cdot \mathbf{R}) \, dS
\]

where \( \hat{\eta} \) is the outward directed surface normal and \( \mathbf{R} \) is the column vector \([X; Y; Z]\).
This formula implies

\[ V = \int \int \int_V dX \, dY \, dZ = \frac{1}{3} \int_S \hat{n} \cdot R \, dS, \]

\[ V_R = \int \int \int_V R \, dX \, dY \, dZ = \frac{1}{4} \int_S R (\hat{n} \cdot R) \, dS, \]

and

\[ V_{RR} = \int \int \int_V RR' \, dX \, dY \, dZ = \frac{1}{5} \int_S RR' (\hat{n} \cdot R) \, dS \]

where \( R' \) means the transpose of \( R \). Let us apply these formulas to a pyramid with the apex at \( R = 0 \) and the base being a planar region \( S_b \) of area \( A \). For points on the side of the pyramid of height \( h \) we find that \( \hat{n} \cdot R = 0 \), and for points on the base \( \hat{n} \cdot R = h \). Consequently

\[ V = \frac{1}{3} \int_{S_b} h \, dS = \frac{h}{3} A, \]

\[ V_R = \frac{1}{4} \int_{S_b} Rh \, dS = \frac{h}{4} \int_{S_b} R \, dS, \]

\[ V_{RR} = \frac{1}{5} \int_{S_b} RR' h \, dS = \frac{h}{5} \int_{S_b} RR' \, dS. \]

The volume is equal to one third of the height times the base area, regardless of the base shape. If \( R_b \) and \( R_p \) signify the centroidal radii of the base and the pyramid volume, respectively, we get

\[ R_p = \frac{V_R}{V} = \frac{\frac{h}{4} R_b A}{\frac{h}{3} A} = \frac{3}{4} R_b. \]

Therefore, the centroid of the volume lies \( \frac{3}{4} \) of the way along a line from the apex to the centroid of the base. For any planar area it is not hard to show that the area \( A \) and unit surface normal \( \hat{n} \) can be computed using the line integral

\[ \hat{n} A = \frac{1}{2} \int_L R \times dR. \]

The last formula simplifies for a polygon having corners at \( R_1, R_2, \ldots, R_n \) to yield

\[ \hat{n} A = \frac{1}{2} \sum_{j=1}^n R_j \times R_{j+1} \text{ where } R_{n+1} = R_1. \]
To compute the first and second area moments for a general planar area, it is helpful to introduce coordinates centered anywhere in the plane containing the base. We let 

\[ R = R_0 + \hat{i}x + \hat{j}y \]

where \( R_0 \) is a vector to a point in the plane of the base, and \( \hat{i} \) and \( \hat{j} \) are orthonormal unit vectors which are tangent to the plane and are chosen such that \( \hat{i}, \hat{j}, \hat{\eta} \) form a right-handed system. The local coordinates \((x, y)\) can be computed using

\[ x = (R - R_0)' \hat{i} \quad \text{and} \quad y = (R - R_0)' \hat{j}. \]

Then we get

\[ V_R = \frac{h}{4} \iint_{S_b} (R_0 + \hat{i}x + \hat{j}y) \, dx \, dy \]

\[ = \frac{h}{4} (R_0 + \hat{i} \bar{x} + \hat{j} \bar{y}) A \]

\[ = \frac{h}{4} R_b A \]

where \((\bar{x}, \bar{y})\) are the centroidal coordinates of the area measured relative to the local axes. Similarly we have

\[ V_{RR} = \frac{h}{5} \iint_{S_b} \left[ R_0 R'_0 + (R_0 \hat{i}' \hat{j}' + \hat{i} R_0')x + (R_0 \hat{j}' + \hat{j} R_0')y + \right. \]

\[ (\hat{i} \hat{j}' + \hat{j} \hat{i}') xy + \hat{i} \hat{i}' x^2 + \hat{j} \hat{j}' y^2 ] \, dx \, dy \]

\[ = \frac{h}{5} \left[ R_0 R'_0 + (R_0 \hat{i}' + \hat{i} R'_0) \bar{x} + (R_0 \hat{j}' + \hat{j} R'_0) \bar{y} \right] A + \]

\[ \frac{h}{5} \left[ \hat{i} \hat{i}' A_{xx} + \hat{j} \hat{j}' A_{yy} + (\hat{i} \hat{j}' + \hat{j} \hat{i}') A_{xy} \right] \]

where

\[ A_{xx} = \iint_{S_b} x^2 \, dx \, dy, \quad A_{xy} = \iint_{S_b} xy \, dx \, dy, \quad A_{yy} = \iint_{S_b} y^2 \, dx \, dy. \]

The formula for \( V_{RR} \) simplifies when \( R_0 \) is chosen as the centroidal radius \( R_b \). Then \( \bar{x} = \bar{y} = 0 \) so that

\[ V_{RR} = \frac{h}{5} \left[ R_0 R'_0 A + \hat{i} \hat{i}' A_{xx}^{b} + \hat{j} \hat{j}' A_{yy}^{b} + (\hat{i} \hat{j}' + \hat{j} \hat{i}') A_{xy}^{b} \right] \]

with the quantities \( A_{xx}^{b}, A_{yy}^{b}, A_{xy}^{b} \) denoting reference to the centroidal axes.

The analysis to compute polyhedron properties can now be completed using vector algebra along with area property calculations of the type introduced earlier. To
define data for a particular polyhedron we provide vectors $x, y, z$ containing global coordinates of all corners. We also employ a matrix named $\text{idface}$ having a row dimension equal to the number of faces on the polyhedron and a column dimension equal to the largest number of corners on any face. Row $i$ of $\text{idface}$ consists of corner indices of the $i$’th face with the row being padded with zeros on the right if necessary. Each face is traversed in the counterclockwise sense relative to the outward normal. Consider a figure showing a triangular block with a hole, having twelve corners and eight faces as shown in Figure 5.6. The required geometry descriptions are defined in example polhdrun. The results produced for this example are

$$\begin{align*}
\text{>> polhdrun;}
\text{v = 15} \\
\text{rc =}
& \begin{bmatrix}
0.0000 \\
2.6667 \\
1.3333
\end{bmatrix} \\
\text{vrr =}
& \begin{bmatrix}
5.0000 & 0.0000 & 0.0000 \\
0.0000 & 120.8333 & 60.4167 \\
0.0000 & 60.4167 & 40.8333
\end{bmatrix} \\
\text{irr =}
& \begin{bmatrix}
161.6667 & -0.0000 & -0.0000 \\
-0.0000 & 45.8333 & -60.4167 \\
-0.0000 & -60.4167 & 125.8333
\end{bmatrix}
\end{align*}$$

These values can be easily verified by manual calculations.
Figure 5.6: Surface Plot of a General Polyhedron
Program polhdrun

function polhdrun
% Example: polhdrun
% ~~~~~~~~~~~~~~~~~
% This program illustrates the use of routine
% polhedrn to calculate the geometrical
% properties of a polyhedron.
% User m functions called:
% crosmat, polyxy, cubrange, pyramid,
% polhdplt, polhedrn

x=[2 2 2 2 2 2 0 0 0 0 0 0]-1;
y=[0 4 4 2 3 3 0 4 4 2 3 3];
z=[0 0 4 1 1 2 0 0 4 1 1 2];
idface=[1 2 3 6 5 4 6 3; ... 
1 3 9 7 0 0 0 0; ... 
1 7 8 2 0 0 0 0; ... 
2 8 9 3 0 0 0 0; ... 
7 9 12 10 11 12 9 8; ... 
4 10 12 6 0 0 0 0; ... 
4 5 11 10 0 0 0 0; ... 
5 6 12 11 0 0 0 0];
polhdplt(x,y,z,idface,[1,1,1]);
[v,rc,vrr,irr]=polhedrn(x,y,z,idface)

%=============================================
% This function determines the volume,
% centroidal coordinates and inertial moments
% for an arbitrary polyhedron.
% x,y,z - vectors containing the corner
% indices of the polyhedron
% idface - a matrix in which row j defines the
% corner indices of the j’th face.
% Each face is traversed in a
counterclockwise sense relative to
the outward normal. The column
dimension equals the largest number
of indices needed to define a face.
Rows requiring fewer than the
maximum number of corner indices are
padded with zeros on the right.

v - the volume of the polyhedron
rc - the centroidal radius
vrr - the integral of R*R'*d(vol)
irr - the inertia tensor for a rigid body
of unit mass obtained from vrr as
eye(3,3)*sum(diag(vrr))-vrr

User m functions called: pyramid

function [area,xbar,ybar,axx,axy,ayy]=polyxy(x,y)
% This function computes the area, centroidal
coordinates, and inertial moments of an
arbitrary polygon.
x,y - vectors containing the corner
coordinates. The boundary is
traversed in a counterclockwise
direction
area - the polygon area
GAUSS INTEGRATION WITH GEOMETRIC PROPERTY APPLICATIONS

% xbar, ybar - the centroidal coordinates
% axx - integral of x^2 dxdy
% axy - integral of xy dxdy
% ayy - integral of y^2 dxdy

% User m functions called: none

%----------------------------------------------

n=1:length(x); n1=n+1;
x=[x(:);x(1)]; y=[y(:);y(1)];
a=(x(n).*y(n1)-y(n).*x(n1))';
area=sum(a)/2; a6=6*area;
xbar=a*(x(n)+x(n1))/a6; ybar=a*(y(n)+y(n1))/a6;
ayy=a*(y(n).^2+y(n).*y(n1)+y(n1).^2)/12;
axy=a*(x(n).*(2*y(n)+y(n1))+x(n1).* ...% This function determines geometrical
% properties of a pyramid with the apex at the
% origin and corner coordinates of the base
% stored in the rows of r.

% r - matrix containing the corner
% coordinates of a polygonal base stored
% in the rows of matrix r.

% v - the volume of the pyramid
% vr - the first moment of volume relative to
% the origin
% vrr - the second moment of volume relative
% to the origin
% h - the pyramid height
% area - the base area
% n - the outward directed unit normal to
% the base

% User m functions called: crosmat, polyxy

function [v,vr,vrr,h,area,n]=pyramid(r)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function determines geometrical
% properties of a pyramid with the apex at the
% origin and corner coordinates of the base
% stored in the rows of r.
% r - matrix containing the corner
% coordinates of a polygonal base stored
% in the rows of matrix r.
% v - the volume of the pyramid
% vr - the first moment of volume relative to
% the origin
% vrr - the second moment of volume relative
% to the origin
% h - the pyramid height
% area - the base area
% n - the outward directed unit normal to
% the base
% User m functions called: crosmat, polyxy
%----------------------------------------------
ns=size(r,1);
na=sum(crosmat(r,r([2:ns,1,:]),))/2;
area=norm(na); n=na/area; p=null(n');
i=p(:,1); j=p(:,2);
if det([p,n])<0, j=-j; end;
r1=r(1,:); rr=r-r1(ones(ns,1),:);
x=rr*i; y=rr*j;
[areat,xc,yc,aaxx,aaxy,ayy]=polyxy(x,y);
rc=r1'+xc*i+yc*j; h=r1*n;
v=h*area/3; vr=v*3/4*rc;
aaxx=aaxx-area*xc^2; ayy=ayy-area*yc^2;
axy=axy-area*xc*yc;
vrh=h/5*(area*rc*rc'+aaxx*i*i'+aaxy*j*j'+ ... 
axy*(i*j'+j*i'));

%=============================================

© 2003 by CRC Press LLC

function polhdplt(x,y,z,idface,colr)
%
% polhdplt(x,y,z,idface,colr)
% ----------------------------------
% This function makes a surface plot of an
% arbitrary polyhedron.
% x,y,z - vectors containing the corner
% indices of the polyhedron
% idface - a matrix in which row j defines the
% corner indices of the j'th face.
% Each face is traversed in a
% counterclockwise sense relative to
% the outward normal. The column
% dimension equals the largest number
% of indices needed to define a face.
% Rows requiring fewer than the
% maximum number of corner indices are
% padded with zeros on the right.
% colr - character string or a vector
% defining the surface color
%----------------------------------------------
if nargin<5, colr=[1 0 1]; end
hold off, close; nf=size(idface,1);
for k=1:nf
i=idface(k,:); i=i(find(i>0));
xi=x(i); yi=y(i); zi=z(i);
fill3(xi,yi,zi,colr); hold on;
end
axis(v); grid on;
xlabel('x axis'); ylabel('y axis'); zlabel('z axis');
title('Surface Plot of a General Polyhedron');
figure(gcf); hold off;

%=============================================% end of function
%=============================================% function range=cubrange(xyz,ovrsiz)
% See Appendix B
%=============================================% function c=crosmat(a,b)
% % $c = \textrm{crosmat}(a,b)$
% % This function computes the vector cross product for vectors stored in the rows
% % of matrices a and b, and returns the result in the rows of c.
% % User m functions called: none
% %---------------------------------------------
c=[a(:,2).*b(:,3)-a(:,3).*b(:,2),...
a(:,3).*b(:,1)-a(:,1).*b(:,3),...
a(:,1).*b(:,2)-a(:,2).*b(:,1)];
%=============================================% function range=cubrange(xyz,ovrsiz)
% See Appendix B
5.8 Evaluating Integrals Having Square Root Type Singularities

Consider the problem of evaluating the following three integrals having square root type singularities at one or both ends of the integration interval:

\[ I_1 = \int_a^b \frac{f(x)}{\sqrt{x-a}} \, dx, \quad I_2 = \int_a^b \frac{f(x)}{\sqrt{b-x}} \, dx, \quad I_3 = \int_a^b \frac{f(x)}{\sqrt{(x-a)(b-x)}} \, dx. \]

The singularities in these integrals can be removed using substitutions \( x - a = t^2, b - x = t^2 \), and \((x-a)(b-x) = (b+a)/2 + (b-a)/2 \cos(t)\) which lead to

\[ I_1 = 2 \int_0^{\sqrt{b-a}} f(a + t^2) \, dt, \quad I_2 = 2 \int_0^{\sqrt{b-a}} f(b - t^2) \, dt, \]
\[ I_3 = \pi \int_0^\infty f\left( \frac{b+a}{2} + \frac{b-a}{2} \cos(t) \right) dt. \]

These modified integrals can be evaluated using \texttt{gequad} or \texttt{quadv} by creating integrands with appropriate argument shifts. Two integration functions \texttt{quadgsqrt} and \texttt{quadlsqrt} were written to handle each of the three integral types. Shown below is a program called \texttt{sqrtquadtest} which computes results for the case where \( f(x) = e^{ux} \cos(vx) \) with constants \( u \) and \( v \) being parameters passed to the integrators using the \texttt{varargin} construct in MATLAB. Function \texttt{quadgsqrt} uses Gauss quadrature to evaluate \( I_1 \) and \( I_2 \), and uses Chebyshev quadrature [1] to evaluate \( I_3 \). When \( f(x) \) is a polynomial, then taking parameter \texttt{norder} in function \texttt{quadgsqrt} equal to the polynomial order gives exact results. With \texttt{norder} taken sufficiently high, more complicated functions can also be integrated accurately. Function \texttt{quadlsqrt} evaluates the three integral types using the adaptive integrator \texttt{quadl}, which accommodates \( f(x) \) of quite general form. The program shown below integrates the test function for parameter choices corresponding to \([a, b, u, v] = [1, 4, 3, 10]\) with \texttt{norder}=10 in \texttt{quadgsqrt} and \texttt{tol}=1e-12 in \texttt{quadlsqrt}. Output from the program for this data case appears as comments at lines 14 thru 35 of \texttt{sqrtquadtest}. The integrators apparently work well and give results agreeing to fifteen digits. However, \texttt{quadlsqrt} took more than four hundred times as long to run as \texttt{quadgsqrt}. Furthermore, the structure of \texttt{quadgsqrt} is such that it could easily be modified to accommodate a form of \( f(x) \) which returns a vector.

5.8.1 Program Listing

Singular Integral Program

```matlab
function [vg,tg,vL,tL,pctdiff]=sqrtquadtest
```

© 2003 by CRC Press LLC
```matlab
% [vg,tg,vL,tL,pctdiff]=sqrtquadtest
% This function compares the accuracy and
% computation time for functions quadgsqrt
% and quadlsqrt to evaluate:
% integral(exp(u*x)*cos(v*x)/radical(x), a<x<b)
% where radical(x) is sqrt(x-a), sqrt(b-x), or
% sqrt((x-a)*(b-x))
%
%--------------------------------------------------------------------
% Program Output
% >> sqrtquadtest;
% EVALUATING INTEGRALS WITH SQUARE ROOT TYPE
% SINGULARITIES AT THE END POINTS
% Function integrated:
% ftest(x,u,v)=exp(u*x).*cos(v*x)
%
% a = 1   b = 4
% u = 3   v = 10
%
% Results from function gquadsqrt
% 4.836504484e+003 -8.060993912e+003 -4.264510048e+003
% Computation time = 0.0159 sec.
%
% Results from function quadlsqrt
% 4.836504484e+003 -8.060993912e+003 -4.264510048e+003
% Computation time = 7.03 sec.
%
% Percent difference for the two methods
% -3.6669e-012 -1.5344e-012 1.4929e-012
% >>
%--------------------------------------------------------------------
% The test function
% ftest=inline('exp(u*x).*cos(v*x)','x','u','v');
% Limits and function parameters
a=1; b=4; u=3; v=10;
nloop=100; tic;
```
for j=1:nloop
v1g=quadgsqrt(f,test,1,a,b,40,1,u,v);
v2g=quadgsqrt(f,test,2,a,b,40,1,u,v);
v3g=quadgsqrt(f,test,3,a,b,40,1,u,v);
end
vg=[v1g,v2g,v3g]; tg=toc/nloop;
disp('EVALUATING INTEGRALS WITH SQUARE ROOT TYPE')
disp(' SINGULARITIES AT THE END POINTS')
disp(' Function integrated:')
disp('f(test(x,u,v)=exp(u*x).*cos(v*x)')
disp(' Function integrated:')
disp(['a = ',num2str(a),' b = ',num2str(b)])
disp(['u = ',num2str(u),' v = ',num2str(v)])
disp('Results from function gquadsqrt')
fprintf('%17.9e %17.9e %17.9e
',vg)
disp([',',num2str(tg),' sec.'])
tol=1e-12; tic;
v1L=quadlsqrt(f,test,1,a,b,tol,[],u,v);
v2L=quadlsqrt(f,test,2,a,b,tol,[],u,v);
v3L=quadlsqrt(f,test,3,a,b,tol,[],u,v);
vl=[v1L,v2L,v3L]; tL=toc;
disp(' Results from function quadlsqrt')
fprintf('%17.9e %17.9e %17.9e
',vl)
disp([',',num2str(tL),' sec.'])
pctdiff=100*(vg-vL)./vL; disp('Percent difference for the two methods')
fprintf('%13.4e %12.4e %12.4e
',pctdiff)

function v=quadgsqrt(...
    func,type,a,b,norder,nsegs,varargin)
% v=quadgsqrt(func,type,a,b,norder,nsegs,varargin)
% This function evaluates an integral having a
GAUSS INTEGRATION WITH GEOMETRIC PROPERTY APPLICATIONS

% square root type singularity at one or both ends
% of the integration interval a<x<b. Composite
% Gauss integration is used with func(x) treated
% as a polynomial of degree norder.
% The integrand has the form:
% func(x)/sqrt(x-a) if type==1.
% func(x)/sqrt(b-x) if type==2.
% func(x)/sqrt((x-a)*(b-x)) if type==3.
% The integration interval is subdivided into
% nsegs subintervals of equal length.
% func - a character string or function handle
% naming a function continuous in the
% interval from x=a to x=b
% type - 1 if the integrand is singular at x=a
% 2 if the integrand is singular at x=b
% 3 if the integrand is singular at both
% x=a and x=b.
% a,b - integration limits with b>a
% norder - polynomial interpolation order within
% each interval. Lowest norder is 20.
% nsegs - number of integration subintervals
% User m functions called: gcquad

if nargin<6, nsegs=1; end;
if nargin<5, norder=50; end
switch type
  case 1 % Singularity at the left end.
    % Use Gauss quadrature
    [dumy,bp,wf]=gcquad(...
        '0',sqrt(b-a),norder+1,nsegs);
    t=a+bp.^2; y=feval(func,t,varargin{:});
    v=wf(:)'*y(:)*2;
  case 2 % Singularity at the right end.
    % Use Gauss quadrature
    [dumy,bp,wf]=gcquad(...
        '0',sqrt(b-a),norder+1,nsegs);
    t=b-bp.^2; y=feval(func,t,varargin{:});
    v=wf(:)'*y(:)*2;
  case 3 % Singularity at both ends.
end

© 2003 by CRC Press LLC
% Use Chebyshev integration
n=norder; bp=cos(pi/(2*n+2)*(1:2:2*n+1));
c1=(b+a)/2; c2=(b-a)/2; t=c1+c2*bp;
y=feval(func,t,varargin{:});
v=pi/(n+1)*sum(y);
end

%========================================

function v=quadlsqrt(fname,type,a,b,tol,trace,varargin)

% v=quadlsqrt(fname,type,a,b,tol,trace,varargin)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function uses the MATLAB integrator quadl
% to evaluate integrals having square root type
% singularities at one or both ends of the
% integration interval a < x < b.
% The integrand has the form:
% func(x)/sqrt(x-a) if type==1.
% func(x)/sqrt(b-x) if type==2.
% func(x)/sqrt((x-a)*(b-x)) if type==3.
% func - the handle for a function continuous
% from x=a to x=b
% type - 1 if the integrand is singular at x=a
% 2 if the integrand is singular at x=b
% 3 if the integrand is singular at both
% x=a and x=b.
% a,b - integration limits with b > a
if nargin<6 | isempty(trace), trace=0; end
if nargin<5 | isempty(tol), tol=1e-8; end
if nargin<7
    varargin{1}=type; varargin{2}=[a,b];
    varargin{3}=fname;
else
    n=length(varargin); c=[a,b]; varargin{n+1}=type;
    varargin{n+2}=c; varargin{n+3}=fname;
end
if type==1 | type==2
    v=2*quadl(@fshift,0,sqrt(b-a),
              tol,trace,varargin{:});
else
5.9 Gauss Integration of a Multiple Integral

Gauss integration can be used to evaluate multiple integrals having variable limits. Consider the instance typified by the following triple integral

\[ I = \int_{c_1}^{c_2} \int_{b_1(z)}^{b_2(z)} \int_{a_1(y,z)}^{a_2(y,z)} F(x,y,z) \, dx \, dy \, dz. \]

This integral can be changed into one with constant limits by the substitutions

\[
\begin{align*}
  z &= c_p + c_m u, \quad -1 \leq u \leq 1, \\
  y &= b_p + b_m t, \quad -1 \leq t \leq 1, \\
  x &= a_p + a_m s, \quad -1 \leq s \leq 1
\end{align*}
\]
where
\[ c_p = \frac{c_2 + c_1}{2}, \quad c_m = \frac{c_2 - c_1}{2}, \]
\[ b_p = \frac{b_2 + b_1}{2}, \quad b_m = \frac{b_2 - b_1}{2}, \]
\[ a_p = \frac{a_2 + a_1}{2}, \quad a_m = \frac{a_2 - a_1}{2}. \]

The above integral becomes
\[ I = \int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} c_m b_m a_m f(s, t, u) \, ds \, dt \, du \]
where
\[ f(s, t, u) = F(a_p + a_m s, b_p + b_m t, c_p + c_m u), \]
\[ a_m = a_m(y, z) = a_m(b_p + b_m t, c_p + c_m u), \]
\[ b_m = b_m(z) = b_m(c_p + c_m u). \]

Thus, the integral has the form
\[ I = \int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} G(s, t, u) \, ds \, dt \, du \]
where
\[ G = c_m b_m a_m f. \]

Performing the integration over each limit using an \( n \)-point quadrature formula with weight factors \( w_i \) and base points \( x_i \), yields
\[ I = \sum_{k=1}^{n} \sum_{j=1}^{n} \sum_{i=1}^{n} w_k w_j w_i G(x_i, x_j, x_k). \]

A function allowing an integrand and integration limits of general form was developed. An example is considered where the inertial moment of a sphere having unit radius, unit mass density, and centered at \((0, 0, 0)\) is to be obtained about an axis through \(x = 2, y = 0, \) parallel to the \( z \)-axis. The related integral
\[ I = \int_{-1}^{1} \int_{-1}^{\sqrt{1-z^2}} \int_{-\sqrt{1-z^2}}^{\sqrt{1-z^2}} [(x - 2)^2 + y^2] \, dx \, dy \, dz \]
has a value of \(88\pi/15\). Shown below is a function \texttt{quadit3d} and related limit and integrand functions. The function \texttt{triplint(n)} computes the ratio of the numerically integrated function to the exact result. The function specification \texttt{triplint(20)} yields a value of 1.000067. Even though the triple integration procedure is not computationally very fast, it is nevertheless robust enough to produce accurate results when a sufficiently high integration order is chosen.
5.9.1 Example: Evaluating a Multiple Integral

Triple Integration Program

```matlab
function val=triplint(n)

% Triple integration example on inertial moment of a sphere.

% User m functions called: fsphere, bs1, bs2, as1, as2

if nargin==0, n=20; end
val=quadit3d('fsphere',[-1,1],'bs1','bs2','as1','as2',n)/(88*pi/15);

function s = quadit3d(f,c,b1,b2,a1,a2,w)

% This function computes the iterated integral

% s = integral(...
% f(x,y,z), x=a1..a2, y=b1..b2, z=c1..c2)

% where a1 and a2 are functions of y and z, b1 and b2 are functions of z, and c is a vector
% containing constant limits on the z variable.
% Hence, as many as five external functions may be involved in the call list. For example,
% when the integrand and limits are:

% f = x^2+y^2+z^2
% a2 = sqrt(4-y^2-z^2)
% a1 = -a2
% b2 = sqrt(4-z^2)
% b1 = -b2
% c = [-2,2]

% Then the exact value is 128*pi/5.
```

© 2003 by CRC Press LLC
% The approximation produced from a 20 point
% Gauss formula is accurate within .007 percent.

% f - a function f(x,y,z) which must return
% a vector value when x is a vector,
% and y and z are scalar.
% a1,a2 - integration limits on the x variable
% which may specify names of functions
% or have constant values. If a1 is a
% function it should have a call list
% of the form a1(y,z). A similar form
% applies to a2.
% b1,b2 - integration limits on the y variable
% which may specify functions of z or
% have constant values.
% c - a vector defined by c=[c1,c2] where
% c1 and c2 are fixed integration
% limits for the z direction.
% w - this argument defines the quadrature
% formula used. It has the following
% three possible forms. If w is omitted,
% a Gauss formula of order 12 is used.
% If w is a positive integer n, a Gauss
% formula of order n is used. If w is an
% n by 2 matrix, w(:,1) contains the base
% points and w(:,2) contains the weight
% factors for a quadrature formula over
% limits -1 to 1.
% s - the numerically evaluated integral

% User m functions called: gcquad

if nargin<7
% function gcquad generates base points
% and weight factors
    n=12; [dummy,x,W]=gcquad(’,-1,1,n,1);
elseif size(w,1)==1 & size(w,2)==1
    n=w; [dummy,x,W]=gcquad(’,-1,1,n,1);
else
    n=size(w,1); x=w(:,1); W=w(:,2);
end
s=0; cp=(c(1)+c(2))/2; cm=(c(2)-c(1))/2;
for k=1:n
    zk=cp+cm*x(k);
    if ischar(b1), B1=feval(b1,zk);
    else, B1=b1; end
    if ischar(b2), B2=feval(b2,zk);
    else, B2=b2; end
    Bp=(B2+B1)/2; Bm=(B2-B1)/2; sj=0;
    for j=1:n
        yj=Bp+Bm*x(j);
        if ischar(a1), A1=feval(a1,yj,zk);
        else, A1=a1; end
        if ischar(a2), A2=feval(a2,yj,zk);
        else, A2=a2; end
        Ap=(A2+A1)/2; Am=(A2-A1)/2;
        fval=feval(f, Ap+Am*x, yj, zk);
        si=fval(:).'*W(:); sj=sj+W(j)*Am*si;
    end
    s=s+W(k)*Bm*sj;
end
s=cm*s;

%=======================================================================
function v=fsphere(x,y,z)
    v=(x-2).^2+y.^2;
%=======================================================================
function x=as1(y,z)
    x=lower x integration limit.
\begin{verbatim}
x=-sqrt(1-y.^2-z.^2);

function x=as2(y,z)

% x=as2(y,z)

% Upper x integration limit.

x=sqrt(1-y.^2-z.^2);

function y=bs1(z)

% y=bs1(z)

% Lower y integration limit.

y=-sqrt(1-z.^2);

function y=bs2(z)

% y=bs2(z)

% Upper y integration limit.

y=sqrt(1-z.^2);

function [val,bp,wf]=gcquad(func,xlow,...
    xhigh,nquad,mparts,varargin)

% See Appendix B
\end{verbatim}
Chapter 6

Fourier Series and the Fast Fourier Transform

6.1 Definitions and Computation of Fourier Coefficients

Trigonometric series are useful to represent periodic functions. A function defined for \(-\infty < x < \infty\) has a period of \(2\pi\) if \(f(x+2\pi) = f(x)\) for all \(x\). In most practical situations, such a function can be expressed as a complex Fourier series

\[
f(x) = \sum_{j=-\infty}^{\infty} c_j e^{ijx}
\]

where \(i = \sqrt{-1}\).

The numbers \(c_j\), called complex Fourier coefficients, are computed by integration as

\[
c_j = \frac{1}{2\pi} \int_{0}^{2\pi} f(x) e^{-ix} \, dx.
\]

The Fourier series can also be rewritten using sines and cosines as

\[
f(x) = c_0 + \sum_{j=1}^{\infty} (c_j + c_{-j}) \cos(jx) + i(c_j - c_{-j}) \sin(jx).
\]

Denoting

\[a_j = c_j + c_{-j}\]
\[b_j = i(c_j - c_{-j})\]

yields

\[
f(x) = \frac{1}{2} a_0 + \sum_{j=1}^{\infty} a_j \cos(jx) + b_j \sin(jx)
\]

which is called a Fourier sine-cosine expansion. This series is especially appealing when \(f(x)\) is real valued. For that case \(c_{-j} = \overline{c_j}\) for all \(j\), which implies that \(c_0\) must be real and

\[a_j = 2 \, \text{real}(c_j), \quad b_j = -2 \, \text{imag}(c_j) \quad \text{for } j > 0.
\]

Suppose we want a Fourier series expansion for a more general function \(f(x)\) having period \(p\) instead of \(2\pi\). If we introduce a new function \(g(x)\) defined by

\[g(x) = f \left( \frac{px}{2\pi} \right)
\]

where \(\frac{px}{2\pi}\) is an integer.
then \( g(x) \) has a period of \( 2\pi \). Consequently, \( g(x) \) can be represented as

\[
g(x) = \sum_{j=-\infty}^{\infty} c_j e^{jx}.
\]

From the fact that \( f(x) = g(2\pi x/p) \) we deduce that

\[
f(x) = \sum_{j=-\infty}^{\infty} c_j e^{2\pi jx/p}.
\]

A need sometimes occurs to expand a function as a series of sine terms only, or as a series of cosine terms only. If the function is originally defined for \( 0 < x < \frac{p}{2} \), then making \( f(x) = -f(p-x) \) for \( \frac{p}{2} < x < p \) gives a series involving only sine terms. Similarly, if \( f(x) = +f(p-x) \) for \( \frac{p}{2} < x < p \), only cosine terms arise. Thus we get

\[
f(x) = c_0 + \sum_{j=1}^{\infty} (c_j + c_{-j}) \cos(2\pi jx/p) \text{ if } f(x) = f(p-x),
\]

or

\[
f(x) = \sum_{j=1}^{\infty} i(c_j - c_{-j}) \sin(2\pi jx/p) \text{ if } f(x) = -f(p-x).
\]

When the Fourier series of a function is approximated using a finite number of terms, the resulting approximating function may oscillate in regions where the actual function is discontinuous or changes rapidly. This undesirable behavior can be reduced by using a smoothing procedure described by Lanczos [60]. Use is made of Fourier series of a closely related function \( \tilde{f}(x) \) defined by a local averaging process according to

\[
\tilde{f}(x) = \frac{1}{\Delta} \int_{x-\Delta/2}^{x+\Delta/2} f(\zeta) d\zeta
\]

where the averaging interval \( \Delta \) should be a small fraction of the period \( p \). Hence we write \( \Delta = \alpha p \) with \( \alpha < 1 \). The functions \( \tilde{f}(x) \) and \( f(x) \) are identical as \( \alpha \to 0 \). Even for \( \alpha > 0 \), these functions also match exactly at any point \( x \) where \( f(x) \) varies linearly between \( x - \frac{\Delta}{2} \) and \( x + \frac{\Delta}{2} \). An important property of \( \tilde{f}(x) \) is that it agrees closely with \( f(x) \) for small \( \alpha \) but has a Fourier series which converges more rapidly than the series for \( f(x) \). Furthermore, from its definition,

\[
\tilde{f}(x) = \sum_{j=-\infty}^{\infty} c_j \frac{1}{\alpha^2} \int_{x-\alpha p}^{x+\alpha p} e^{2\pi jx/p} dx = \sum_{j=-\infty}^{\infty} \tilde{c}_j e^{2\pi jx/p}
\]

where \( \tilde{c}_0 = c_0 \) and \( \tilde{c}_j = c_j \sin(\pi j\alpha)/(\pi j\alpha) \) for \( j \neq 0 \). Evidently the Fourier coefficients of \( \tilde{f}(x) \) are easily obtainable from those of \( f(x) \). When the series for \( f(x) \) converges slowly, using the same number of terms in the series for \( \tilde{f}(x) \) often gives an approximation preferable to that provided by the series for \( f(x) \). This process is called smoothing.
6.1.1 Trigonometric Interpolation and the Fast Fourier Transform

Computing Fourier coefficients by numerical integration is very time consuming. Consequently, we are led to investigate alternative methods employing trigonometric polynomial interpolation through evenly spaced data. The resulting formulas are the basis of an important algorithm called the Fast Fourier Transform (FFT). Although the Fourier coefficients obtained by interpolation are approximate, these coefficients can be computed very rapidly when the number of sample points is an integer power of 2 or a product of small primes. We will discuss next the ideas behind trigonometric polynomial interpolation among evenly spaced data values.

Suppose we truncate the Fourier series and only use harmonics up to some order $N$. We assume $f(x)$ has period $2\pi$ so that

$$f(x) = \sum_{j=-N}^{N} c_j e^{ijx}. $$

This trigonometric polynomial satisfies $f(0) = f(2\pi)$ even though the original function might actually have a finite discontinuity at 0 and $2\pi$. Consequently, we may choose to use, in place of $f(0)$, the limit as $\epsilon \to 0$ of $[f(\epsilon) + f(2\pi - \epsilon)]/2$.

It is well known that the functions $e^{ijx}$ satisfy an orthogonality condition for integration over the interval 0 to $2\pi$. They also satisfy an orthogonality condition regarding summation over equally spaced data. The latter condition is useful for deriving a discretized approximation of the integral formula for the exact Fourier coefficients. Let us choose data points

$$x_j = \left(\frac{\pi}{N}\right) j, \quad 0 \leq j \leq (2N-1), $$

and write the simultaneous equations to make the trigonometric polynomial match the original function at the equally spaced data points. To shorten the notation we let

$$t = e^{\pi/N},$$

and write

$$f_k = \sum_{j=-N}^{N} c_j t^{kj}. $$

Suppose we pick an arbitrary integer $n$ in the range $-N < n < N$. Multiplying the last equation by $t^{-kn}$ and summing from $k = 0$ to $2N-1$ gives

$$\sum_{k=0}^{2N-1} f_k t^{-kn} = \sum_{k=0}^{2N-1} t^{-kn} \sum_{j=-N}^{N} c_j t^{kj}. $$

Interchanging the summation order in the last equation yields

$$\sum_{k=0}^{2N-1} f_k t^{-kn} = \sum_{j=-N}^{N} c_j \sum_{k=0}^{2N-1} t^k.$$
where $\zeta = e^{(j-n)\pi/N}$. Summing the inner geometric series gives
\[
\sum_{k=0}^{2N-1} \zeta^k = \begin{cases} 
1 - \zeta^{2N} & \text{for } \zeta \neq 1, \\
2N & \text{for } \zeta = 1.
\end{cases}
\]
We find, for all $k$ and $n$ in the stated range, that
\[
\zeta^{2N} = e^{i2\pi(k-n)} = 1.
\]
Therefore we get
\[
\sum_{k=0}^{2N-1} f_k t^{-kn} = 2N c_n, \quad -N < n < N.
\]
In the cases where $n = \pm N$, the procedure just outlined only gives a relationship governing $c_N + c_{-N}$. Since the first and last terms cannot be computed uniquely, we customarily take $N$ large enough to discard these last two terms and write simply
\[
c_n = \frac{1}{2N} \sum_{k=0}^{2N-1} f_k t^{-kn}, \quad -N < n < N.
\]
This formula is the basis for fast algorithms (called FFT for Fast Fourier Transform) to compute approximate Fourier coefficients. The periodicity of the terms depending on various powers of $e^{i\pi/N}$ can be utilized to greatly reduce the number of trigonometric function evaluations. The case where $N$ equals a power of 2 is especially attractive. The mathematical development is not provided here. However, the related theory was presented by Cooley and Tukey in 1965 [21] and has been expounded in many textbooks [53, 96]. The result is a remarkably concise algorithm which can be comprehended without studying the details of the mathematical derivation. For our present interests it is important to understand how to use MATLAB’s intrinsic function for the FFT ($\text{fft}$).

Suppose a periodic function is evaluated at a number of equidistant points ranging over one period. It is preferable for computational speed that the number of sample points should equal an integer power of two ($n = 2^m$). Let the function values for argument vector
\[
x = p/n * (0 : n - 1)
\]
be an array $f$ denoted by
\[
f \iff [f_1, f_2, \ldots, f_n].
\]
The function evaluation $\text{fft}(f)$ produces an array of complex Fourier coefficients multiplied by $n$ and arranged in a peculiar fashion. Let us illustrate this result for $n = 8$. If
\[
f = [f_1, f_2, \ldots, f_8]
\]
then $\text{fft}(f)/8$ produces
\[
c = [c_0, c_1, c_2, c_3, c_4, c_{-3}, c_{-2}, c_{-1}].
\]
The term denoted by $c_*$ actually turns out to equal $c_4 + c_{-4}$, so it would not be used in subsequent calculations. We generalize this procedure for arbitrary $n$ as follows. Let $N = n/2 - 1$. In the transformed array, elements with indices of $1, \cdots, N + 1$ correspond to $c_0, \cdots, c_N$ and elements with indices of $n, n - 1, n - 2, \cdots, N + 3$ correspond to $c_{-1}, c_{-2}, c_{-3}, \cdots, c_{-N}$. It is also useful to remember that a real valued function has $c_{-n} = \text{conj}(c_n)$. To fix our ideas about how to evaluate a Fourier series, suppose we want to sum an approximation involving harmonics from order zero to order $(n\text{sum} - 1)$. We are dealing with a real valued function defined by $\text{func}$ with a real argument vector $x$. The following code expands $\text{func}$ and sums the series for argument $x$ using $n\text{sum}$ terms.

```matlab
function fouroval=fftaprox(func,period,nfft,nsum,x)
fc=feval(func,period/nfft*(0:nfft-1));
fcc=fft(fc)/nfft; fc(1)=fc(1)/2;
w=2*pi/period*(0:nsum-1);
fouroval=2*real(exp(i*x(:)*w)*fc(:));
```

### 6.2 Some Applications

Applications of Fourier series arise in numerous practical situations such as structural dynamics, signal analysis, solution of boundary value problems, and image processing. Three examples are given below that illustrate use of the FFT. The first example calculates Bessel functions and the second problem studies forced dynamic response of a lumped mass system. The final example presents a program for constructing Fourier expansions and displaying graphical results for linearly interpolated or analytically defined functions.

#### 6.2.1 Using the FFT to Compute Integer Order Bessel Functions

The FFT provides an efficient way to compute integer order Bessel functions $J_n(x)$ which are important in various physical applications [119]. Function $J_n(x)$ can be obtained as the complex Fourier coefficient of $e^{im\theta}$ in the generating function described by

$$e^{ix \sin(\theta)} = \sum_{n=-\infty}^{\infty} J_n(x)e^{in\theta}.$$  

Orthogonality conditions imply

$$J_n(x) = \frac{1}{2\pi} \int_{0}^{2\pi} e^{i(x \sin(\theta) - n\theta)} \, d\theta.$$
The Fourier coefficients represented by $J_n(x)$ can be computed approximately with the FFT. The infinite series converges very rapidly because the function it represents has continuous derivatives of all finite orders. Of course, $e^{ix}\sin(\theta)$ is highly oscillatory for large $|x|$, thereby requiring a large number of sample points in the FFT to obtain accurate results. For $n < 30$ and $|x| < 30$, a 128-point transform is adequate to give about ten digit accuracy for values of $J_n(x)$. The following code implements the above ideas and plots a surface showing how $J_n$ changes in terms of $n$ and $x$.

![Surface Plot for $J_n(x)$](image)

**Figure 6.1:** Surface Plot for $J_n(x)$
MATLAB Example

Bessel Function Program plotjrun

function plotjrun
% Example: plotjrun
% ~~~~~~~~~~~~~~~~~
% This program computes integer order Bessel
% functions of the first kind by using the FFT.
% %
% % User m functions required: jnft
% %
% x=0:.5:20; n=0:20; J=jnft(n,x); surf(x,n,J);
title('Surface Plot For J_{n}(x)');
ylabel('order n'), xlabel('argument x')
zlabel('function value'), figure(gcf);
print -deps plotjrun

function J=jnft(n,z,nft)
% J=jnft(n,z,nft)
% ~~~~~~~~~~~~~~~~~~~~~
% Integer order Bessel functions of the
% first kind computed by use of the Fast
% Fourier Transform (FFT).
% %
% % n - integer vector defining the function
% orders
% % z - a vector of values defining the
% arguments
% % nft - number of function evaluations used
% in the FFT calculation. This value
% should be an integer power of 2 and
% should exceed twice the largest
% component of n. When nft is omitted
% from the argument list, then a value
% equal to 512 is used. More accurate
% values of J are computed as nft is
% increased. For max(n) < 30 and
% max(z) < 30, nft=256 gives about
% ten digit accuracy.
% % J - a matrix of values for the integer
% order Bessel function of the first kind. Row position matches orders defined by n, and column position corresponds to arguments defined by components of z.

if nargin<3, nft=512; end;
J=exp(sin((0:nft-1)'*(2*pi/nft))*(i*z(:).'))/nft;
J=fft(J); J=J(1+n,:).';
if sum(abs(imag(z)))<max(abs(z))/1e10
J=real(J);
end

6.2.2 Dynamic Response of a Mass on an Oscillating Foundation

Fourier series are often used to describe time dependent phenomena such as earthquake ground motion. Understanding the effects of foundation motions on an elastic structure is important in design. The model in Figure 6.2 embodies rudimentary aspects of this type of system and consists of a concentrated mass connected by a spring and viscous damper to a base which oscillates with known displacement $Y(t)$. The system is assumed to have arbitrary initial conditions $y(0) = y_0$ and $\dot{y}(0) = v_0$ when the base starts moving. The resulting displacement and acceleration of the mass are to be computed.

We assume that $Y(t)$ can be represented well over some time interval $p$ by a Fourier series of the form

$$ Y(t) = \sum_{n=-\infty}^{\infty} c_n e^{i\omega_n t}, \quad \omega_n = \frac{2n\pi}{p} $$

where $c_{-n} = \text{conj}(c_n)$ because $Y$ is real valued. The differential equation governing this problem is

$$ m\ddot{y} + c\dot{y} + ky = kY(t) + c\dot{Y}(t) = F(t) $$

where the forcing function can be expressed as

$$ F(t) = \sum_{n=-\infty}^{\infty} c_n[k + ic\omega_n]e^{i\omega_n t} = kc_0 + 2\text{real}\left( \sum_{n=1}^{\infty} f_n e^{i\omega_n t} \right) $$

and

$$ f_n = c_n(k + ic\omega_n). $$

The corresponding steady-state solution of the differential equation is representable
as

\[ y_s(t) = \sum_{n=-\infty}^{\infty} y_n e^{i\omega_n t} \]

where \( y_{-n} = \text{conj}(y_n) \) since \( y_s(t) \) is real valued. Substituting the series solution into the differential equation and comparing coefficients of \( e^{i\omega_n t} \) on both sides leads to

\[ y_n = \frac{c_n (k + ic\omega_n)}{k - m\omega_n^2 + ic\omega_n}. \]

The displacement, velocity, and acceleration corresponding to the steady-state (also called particular) solution are

\[
\begin{align*}
y_s(t) &= c_0 + 2 \text{ real} \left( \sum_{n=1}^{\infty} y_n e^{i\omega_n t} \right), \\
\dot{y}_s(t) &= 2 \text{ real} \left( \sum_{n=1}^{\infty} i\omega_n y_n e^{i\omega_n t} \right), \\
\ddot{y}_s(t) &= -2 \text{ real} \left( \sum_{n=1}^{\infty} \omega_n^2 y_n e^{i\omega_n t} \right).
\end{align*}
\]

The initial conditions satisfied by \( y_s \) are

\[
\begin{align*}
y_s(0) &= c_0 + 2 \text{ real} \left( \sum_{n=1}^{\infty} y_n \right), \\
\dot{y}_s(0) &= 2 \text{ real} \left( \sum_{n=1}^{\infty} i\omega_n y_n \right).
\end{align*}
\]

Because these values usually will not match the desired initial conditions, the total solution consists of \( y_s(t) \) and \( y_h(t) \) which satisfies the homogeneous differential equation

\[ m\ddot{y}_h + c\dot{y}_h + ky_h = 0. \]
The solution is
\[ y_h = g_1 e^{s_1 t} + g_2 e^{s_2 t} \]

where \( s_1 \) and \( s_2 \) are roots satisfying
\[ ms^2 + cs + k = 0. \]

The roots are
\[ s_1 = \frac{-c + \sqrt{c^2 - 4mk}}{2m}, \quad s_2 = \frac{-c - \sqrt{c^2 - 4mk}}{2m}. \]

Since the total solution is
\[ y(t) = y_s(t) + y_h(t) \]

the constants \( g_1 \) and \( g_2 \) are obtained by solving the two simultaneous equations
\[ g_1 + g_2 = y(0) - y_s(0), \quad s_1 g_1 + s_2 g_2 = \dot{y}(0) - \dot{y}_s(0). \]

The roots \( s_1 \) and \( s_2 \) are equal when \( c = 2\sqrt{mk} \). Then the homogeneous solution assumes an alternate form given by \((g_1 + g_2 t)e^{st}\) with \( s = -c/(2m) \). In this special case we find that
\[ g_1 = y(0) - y_s(0), \quad g_2 = \dot{y}(0) - \dot{y}_s(0) - sg_1. \]

It should be noted that even though roots \( s_1 \) and \( s_2 \) will often be complex numbers, this causes no difficulty since MATLAB handles the complex arithmetic automatically (just as it does when the FFT transforms real function values into complex Fourier coefficients).

The harmonic response solution works satisfactorily for a general forcing function as long as the damping coefficient \( c \) is nonzero. A special situation can occur when \( c = 0 \), because the forcing function may resonate with the natural frequency of the undamped system. If \( c \) is zero, and for some \( n \) we have \( \sqrt{k/m} = 2\pi n/p \), a condition of harmonic resonance is produced and a value of zero in the denominator occurs when the corresponding \( y_n \) is computed. In the undamped resonant case the particular solution grows like \([te^{\omega \pi n t}]\), quickly becoming large. Even when \( c \) is small and \( \sqrt{k/m} \approx 2\pi n/p \), undesirably large values of \( y_n \) can result. Readers interested in the important phenomenon of resonance can find more detail in Meirovitch [68].

This example concludes by using a base motion resembling an actual earthquake excitation. Seismograph output employing about 2700 points recorded during the Imperial Valley, California, earthquake of 1940 provided the displacement history for Figure 6.3. The period used to describe the motion is 53.8 seconds. A program was written to analyze system response due to a simulated earthquake base excitation. The following program modules are used:
<table>
<thead>
<tr>
<th>runimpv</th>
<th>sets data values and generates graphical results</th>
</tr>
</thead>
<tbody>
<tr>
<td>fouaprox</td>
<td>generates Fourier series approximations for a general function</td>
</tr>
<tr>
<td>imptp</td>
<td>piecewise linear function approximating the Imperial Valley earthquake data</td>
</tr>
<tr>
<td>shkbftss</td>
<td>computes steady-state displacement and acceleration for a spring-mass-dashpot system subjected to base motion expandable in a Fourier series</td>
</tr>
<tr>
<td>hsmck</td>
<td>computes the homogeneous solution for the spring-mass-dashpot system subjected to general initial conditions</td>
</tr>
</tbody>
</table>

Numerical results were obtained for a system having a natural period close to one second \( (2\pi/6 \approx 1.047) \) and a damping factor of 5 percent. The function \texttt{imptp} was employed as an alternative to the actual seismograph data to provide a concisely expressible function which still embodies characteristics of a realistic base motion. Figure 6.4 shows a plot of function \texttt{imptp} along with its approximation by a twenty-term Fourier series. The series representation is surprisingly good considering the fact that such a small number of terms is used. The use of two-hundred terms gives an approximation which graphically does not deviate perceptibly from the actual function. Results showing how rapidly the Fourier coefficients diminish in magnitude with increasing order appear in Figure 6.5. The dynamical analysis produced displacement and acceleration values for the mass. Figure 6.6 shows both the total displacement as well as the displacement contributed from the homogeneous solution alone. Evidently, the steady-state harmonic response function captures well most of the motion, and the homogeneous part could probably be neglected without serious error. Figure 6.7 also shows the total acceleration of the mass which is, of course, proportional to the resultant force on the mass due to the base motion.

Before proceeding to the next example, the reader should be sure to appreciate the following important fact. Once a truncated Fourier series expansion of the forcing function using some appropriate number of terms is chosen, the truncated series defines an input function for which the response is computed exactly. If the user takes enough terms in the truncated series so that he/she is well satisfied with the function it approximates, then the computed response value for \( y(t) \) will also be acceptable. This situation is distinctly different from the more complicated type of approximations occurring when finite difference or finite element methods produce discrete approximations for continuous field problems. Understanding the effects of grid size discretization error is more complex than understanding the effects of series truncation in the example given here.
Figure 6.3: Normalized Base Displacement

Figure 6.4: Result from a 20-Term Fourier Series
Figure 6.5: Coefficient Magnitude in Base Motion Expansion

Figure 6.6: Total and Homogeneous Response
MATLAB Example

Program runimpy

function runimpy
% Example: runimpy
% ~~~~~~~~~~~~~~~~~
% This is a driver program for the
% earthquake example.
% 
% User m functions required:
%  % fouaprox, imptp, hsmck,
%  % shkbftss, lintrp
%
% Make the undamped period about one
% second long
m=1; k=36;
%
% Use damping equal to 5 percent of critical
c=.05*(2*sqrt(m*k));

© 2003 by CRC Press LLC
% Choose a period equal to length of
% Imperial Valley earthquake data
prd=53.8;

nft=1024; tmin=0; tmax=prd;
ntimes=200; nsum=80; % ntimes=501; nsum=200;
tplt=linspace(0,prd,ntimes);
y20trm=fouaprox('imptp',prd,tplt,20);
plot(tplt,y20trm,'-',tplt,imptp(tplt),'--');
xlabel('time, seconds');
ylabel('unitized displacement');
title('Result from a 20-Term Fourier Series')
figure(gcf);
disp('Press [Enter] to continue');
dumy=input('','s');

% print -deps 20trmplt

% Show how magnitudes of Fourier coefficients
% decrease with increasing harmonic order
fcof=fft(imptp((0:1023)/1024,1))/1024;
clf; plot(abs(fcof(1:100)));
xlabel('harmonic order');
ylabel('coefficient magnitude');
title(['Coefficient Magnitude in Base ' ...
    'Motion Expansion']); figure(gcf);
disp('Press [Enter] to continue');
dumy=input('','s');
% print -deps coefsize

% Compute forced response
[t,ys,ys0,vs0,as]= ...
shkbftss(m,c,k,'imptp',prd,nft,nsum, ...
    tmin,tmax,ntimes);

% Compute homogeneous solution
[t,yh,ah]= ...
hsmck(m,c,k,-ys0,-vs0,tmin,tmax,ntimes);

% Obtain the combined solution
y=ys(:)+yh(:); a=as(:)+ah(:);
clf; plot(t,y,'-',t,yh,'--');
xlabel('time'); ylable('displacement');
title('Total and Homogeneous Response');
legend('Total response','Homogeneous response');
figure(gcf);
disp('Press [Enter] to continue');
dumy=input('','s');
print -deps displac;
clf; plot(t,a,'-');
xlabel('time'); ylabel('acceleration')
title('Acceleration Due to Base Oscillation')
figure(gcf); print -deps accel
%
%=====================================================================
% function y=fouaprox(func,per,t,nsum,nft)
%
% y=fouaprox(func,per,t,nsum,nft)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Approximation of a function by a Fourier series.
% %
% func - function being expanded
% per - period of the function
% t - vector of times at which the series is to be evaluated
% nsum - number of terms summed in the series
% nft - number of function values used to compute Fourier coefficients. This should be an integer power of 2.
% The default is 1024
% %
% User m functions called: none.
%=====================================================================

if nargin<5, nft=1024; end;
nsum=min(nsum,fix(nft/2));
c=fft(feval(func,per/nft*(0:nft-1)))/nft;
c(1)=c(1)/2; c=c(:); c=c(1:nsum);
w=2*pi/per*(0:nsum-1);
y=2*real(exp(i*t(:)*w)*c);
%
%=====================================================================
% function ybase=imptp(t,period)
%
% ybase=imptp(t,period)
%
This function defines a piecewise linear function resembling the ground motion of the earthquake which occurred in 1940 in the Imperial Valley of California. The maximum amplitude of base motion is normalized to equal unity.

% period - period of the motion (optional argument)
% t - vector of times between tmin and tmax
% ybase - piecewise linearly interpolated base motion

% User m functions called: lintrp

%----------------------------------------------

% User m functions called: shkbftss

© 2003 by CRC Press LLC
This function determines the steady-state solution of the scalar differential equation

\[ m \ddot{y}(t) + c \dot{y}(t) + k y(t) = k y_{\text{base}}(t) + c y_{\text{base}}'(t) \]

where \( y_{\text{base}} \) is a function of period \( \text{prd} \) which is expandable in a Fourier series

\[ m, c, k \quad \text{- Mass, damping coefficient, and spring stiffness} \]
\[ y_{\text{base}} \quad \text{- Function or vector of displacements equally spaced in time which describes the base motion over a period} \]
\[ \text{prd} \quad \text{- Period used to expand } x_{\text{base}} \text{ in a Fourier series} \]
\[ \text{nft} \quad \text{- The number of components used in the FFT (should be a power of two). If nft is input as zero, then } y_{\text{base}} \text{ must be a vector and nft is set to length}(y_{\text{base}}) \]
\[ \text{nsum} \quad \text{- The number of terms to be used to sum the Fourier series expansion of } y_{\text{base}} \text{. This should not exceed nft/2.} \]
\[ \text{tmin}, \text{tmax} \quad \text{- The minimum and maximum times for which the solution is to be computed} \]
\[ t \quad \text{- A vector of times at which the solution is computed} \]
\[ ys \quad \text{- Vector of steady-state solution values} \]
\[ ys0, vs0 \quad \text{- Position and velocity at } t=0 \]
\[ as \quad \text{- Acceleration } y_{\text{s''}}(t), \text{ if this quantity is required} \]

User m functions called: none.
198: nft=length(ybase); ybft=ybase(:)
199: else
200: tbft=prd/nft*(0:nft-1);
201: ybft=fft(feval(ybase,tbft))/nft;
202: ybft=ybft(:);
203: end
204: nsum=min(nsum,fix(nft/2)); ybft=ybft(1:nsum);
205: w=2*pi/prd*(0:nsum-1);
206: t=tmin+(tmax-tmin)/(ntimes-1)*(0:ntimes-1)';
207: etw=exp(i*t*w); w=w(:);
208: ysft=ybft.*(k+i*c*w)./(k+w.*(i*c-m*w));
209: ysft(1)=ysft(1)/2;
210: ys=2*real(etw*ysft); ys0=2*real(sum(ysft));
211: vs0=2*real(sum(i*w.*ysft));
212: if nargout > 4
213: ysft=-ysft.*w.^2; as=2*real(etw*ysft);
214: end
215: %============================================= 
216: %function [t,yh,ah]= ...
217: % hsmck(m,c,k,y0,v0,tmin,tmax,ntimes)
218: % [t,yh,ah]=hsmck(m,c,k,y0,v0,tmin,tmax,ntimes)
219: % % Solution of
220: % m*yh''(t) + c*yh'(t) + k*yh(t) = 0
221: % subject to initial conditions of
222: % yh(0) = y0 and yh'(0) = v0
223: % m,c,k - mass, damping and spring
224: % constants
225: % y0,v0 - initial position and velocity
226: % tmin,tmax - minimum and maximum times
227: % ntimes - number of times to evaluate
228: % t - vector of times
229: % yh - displacements for the
230: % homogeneous solution
231: % ah - accelerations for the
232: % homogeneous solution
233: % User m functions called: none.
234: %----------------------------------------------
t=tmin+(tmax-tmin)/(ntimes-1)*(0:ntimes-1);
r=sqrt(c*c-4*m*k);
if r~=0
  s1=(-c+r)/(2*m); s2=(-c-r)/(2*m);
g=[1,1;s1,s2][y0,v0];
yh=real(g(1)*exp(s1*t)+g(2)*exp(s2*t));
if nargout > 2
  ah=real(s1*s1*g(1)*exp(s1*t)+ ...
          s2*s2*g(2)*exp(s2*t));
end
else
  s=-c/(2*m);
g1=y0; g2=v0-s*g1; yh=(g1+g2*t).*exp(s*t);
if nargout > 2
  ah=real(s*(2*g2+s*g1+s*g2*t).*exp(s*t));
end
end
%=============================================%
% function y=lintrp(xd,yd,x)
% See Appendix B

6.2.3 General Program to Plot Fourier Expansions

The final example in this chapter is a program to compute Fourier coefficients of
general real valued functions and to display series with varying numbers of terms
so that a user can see how rapidly such series converge. Since a truncated Fourier
series is a continuous differentiable function, it cannot perfectly represent a discon-
tinuous function such as a square wave. Near points where jump discontinuities
occur, Fourier series approximations oscillate [18]. The same kind of behavior oc-
curs less seriously near points of slope discontinuity. Adding more terms does not
cure the problem at jump discontinuities. The behavior, known as Gibbs phenom-
enon, produces approximations which overshoot the function on either side of the
discontinuity. Illustrations of this behavior appear below.

A program was written to expand real functions of arbitrary period using Fourier
series approximations computed with the FFT. A piecewise linear function can be
specified interactively by giving data points over a period. Alternatively, a function
which is user defined can be employed. For instance, a function varying like a sine
curve with the bottom half cut off would be

```matlab
function y=chopsine(x,period)
y=sin(pi*x/period).*(x<period)
```

The program consists of the following functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fouseris</td>
<td>main driver</td>
</tr>
<tr>
<td>sine</td>
<td>example for exact function input</td>
</tr>
<tr>
<td>lintrp</td>
<td>function for piecewise linear interpolation</td>
</tr>
<tr>
<td>fousum</td>
<td>sum a real valued Fourier series</td>
</tr>
<tr>
<td>read</td>
<td>reads several data items on one line</td>
</tr>
</tbody>
</table>

Comments within the program illustrate how to input data interactively. Details of different input options can be found by executing the program.

Let us see how well the FFT approximates a function of period 3 defined by piecewise linear interpolation through \((x, y)\) values of \((0,1)\), \((1,1)\), \((1,-1)\), \((2,-1)\), \((3,1)\), and \((4,0)\). The function has jump discontinuities at \(x = 0\), \(x = 1\), and \(x = 4\). A slope discontinuity also occurs at \(x = 3\). Program results using a twenty-term approximation appear in Figure 6.8. Results produced by 100- and 250-term series plotted near \(x = 1\) are shown in Figures 6.9 and 6.10. Clearly, adding more terms does not eliminate the oscillation. However, the oscillation at a jump discontinuity can be reduced with the Lanczos smoothing procedure. Results for a series of 250 terms smoothed over an interval equal to the period times 0.01 appear in Figure 6.11. The oscillation is reduced at the cost of replacing the infinite slope at a discontinuity point by a steep slope of fifty-to-one for this case. Figure 6.12 shows a plot produced using an exact function definition as indicated in the second program execution. The reader may find it instructive to investigate how well Fourier series converge by running the program for other function choices.
Figure 6.8: Fourier Series for Harmonics up to Order 20

Figure 6.9: Fourier Series for Harmonics up to Order 100
Figure 6.10: Fourier Series for Harmonics up to Order 250

Figure 6.11: Smoothed Fourier Series for Harmonics up to Order 250
Figure 6.12: Exact Function Example for Harmonics up to Order 20
Examples of Fourier Series Expansions

Output for Piecewise Linear Example

>> fouseris

FOURIER SERIES EXPANSION FOR A PIECEWISE LINEAR OR ANALYTICALLY DEFINED FUNCTION

Input the period of the function
? > 4

Input the number of data points to define the function by piecewise linear interpolation (input a zero if the function is defined analytically by the user).
? > 6

Input the x,y values one pair per line
? > 0,1
? > 1,1
? > 1,-1
? > 2,-1
? > 3,1
? > 4,0

To plot the series input xmin, xmax, and the highest harmonic not exceeding 255 (input 0,0,0 to stop) (Use a negative harmonic number to save your graph)
? > 0,4,20

To plot the series smoothed over a fraction of the period, input the smoothing fraction (give 0.0 for no smoothing).
? > 0

Press RETURN to continue

To plot the series input xmin, xmax, and the highest harmonic not exceeding 255 (input 0,0,0 to stop) (Use a negative harmonic number to save your graph)
? > 0,0,0

Output for Analytically Defined Example

>> fouseris
FOURIER SERIES EXPANSION FOR A PIECEWISE LINEAR OR ANALYTICALLY DEFINED FUNCTION

Input the period of the function
? > pi/2

Input the number of data points to define the function by piecewise linear interpolation (input a zero if the function is defined analytically by the user).
? > 0

Select the method used for exact function definition:

1 <=> Use an existing function with syntax defined by the following example:

function y=sine(x,period)
%
% y=sine(x,period)
% ~~~~~~~~~~~~~~~~~
% This function specifies all or part of a sine wave.
% % x - vector of argument values
% % period - period of the function
% % y - vector of function values
% % User m functions called: none
%-----------------------------------------------
y=sin(rem(x,period));

or

2 <=> Use a one-line character string definition involving argument x and period p. For example a sine wave with the bottom cut off would be defined by:
sin(x*2*pi/p).*((x<p/2)

1 or 2 ? > 1

Enter the name of your function
? > sine

To plot the series input xmin, xmax, and the highest harmonic not exceeding 255 (input 0,0,0 to stop)
(Use a negative harmonic number to save your graph)
? > 0, pi, -20

To plot the series smoothed over a fraction of the period, input the smoothing fraction
(give 0.0 for no smoothing).
? > 0

Give a file name to save the current graph > exactplt

Press RETURN to continue

To plot the series input xmin, xmax, and the highest harmonic not exceeding 255 (input 0, 0, 0 to stop)
(Use a negative harmonic number to save your graph)
? > 0, 0, 0

Fourier Series Program fouseris

function fouseris
  \% Example: fouseris
  \% -------------------
  \% This program illustrates the convergence rate
  \% of Fourier series approximations derived by
  \% applying the FFT to a general function which
  \% may be specified either by piecewise linear
  \% interpolation in a data table or by
  \% analytical definition in a function given by
  \% the user. The linear interpolation model
  \% permits inclusion of jump discontinuities.
  \% Series having varying numbers of terms can
  \% be graphed to demonstrate Gibbs phenomenon
  \% and to show how well the truncated Fourier
  \% series represents the original function.
  \% Provision is made to plot the Fourier series
  \% of the original function or a smoothed
  \% function derived by averaging the original
  \% function over an arbitrary fraction of the
  \% total period.
  \%
  \% User m functions required:
  \%    fousum, lintrp, inputv, sine

© 2003 by CRC Press LLC
% The following parameters control the number of fft points used and the number of points used for graphing.
% nft=1024; ngph=1001; nmax=int2str(nft/2-1);

fprintf('
FOURIER SERIES EXPANSION FOR A PIECEWISE LINEAR OR ANALYTICALLY DEFINED FUNCTION
');

Input the period of the function
period=input('>?');

fprintf('
How many points define the function by piecewise linear interpolation?
(Give a zero for analytical definition)
');

nd=input('>?');
if nd > 0, xd=zeros(nd,1); yd=xd;
    fprintf('
Input the x,y values one pair per line
');
    for j=1:nd
        [xd(j),yd(j)]=inputv('>?');
    end
else
    fprintf('
Select the method for analytical function definition:
');
    fprintf('
1 <=> Use an existing function with syntax of the form:
function y=funct(x,period), or
2 <=> Give a character string in argument x and period p. (Such as: sign(sin(2*pi*x/p))
to make a square wave)
');
    nopt=input('Enter 1 or 2 ? > ');
    if nopt == 1
        fprintf('
Enter the name of your function
');
        fnam=input('>?','s');
        yfc=feval(fnam,xfc,period); c=fft(yfc);
    else
        fprintf('
Select the method for
');
        fprintf('
1 => Use an existing function
2 => Give a character string in argument x and period p. (Such as: sign(sin(2*pi*x/p))
to make a square wave)
');
        nopt=input('Enter 1 or 2 ? > ');
        if nopt == 1
            fprintf('
Enter the name of your function
');
            fnam=input('>?','s');
            yfc=feval(fnam,xfc,period); c=fft(yfc);
        else
            ...
        end
    end

% Use nft interpolated data points to compute the fft
yfc=lintrp(xd,yd,xfc); c=fft(yfc);
69: fprintf(‘
Input the one-line definition’);
70: fprintf(‘ in terms of x and p
’);
71: strng=input(‘> ? ’,’s’);
72: x=xfc; p=period;
73: yfc=eval(strng); c=fft(yfc);
74: end
75: end
76: while 1
77: fprintf(‘
To plot the series input xmin,’);
78: fprintf(‘ xmax, and the highest’);
79: fprintf(‘
harmonic not exceeding ’,…
80: nmax,’ (press [Enter] to stop])’);
81: fprintf(‘
(Use a negative harmonic number’);
82: fprintf(‘ to save your graph)
’);
83: [xl,xu,nh]=inputv(‘> ? ’);
84: if isnan(xl), break; end
85: pltsav=(nh < 0); nh=abs(nh);
86: xtmp=xl+((xu-xl)/ngph)*(0:ngph);
87: fprintf(‘
To plot the series smoothed ’);
88: fprintf(‘over a fraction of the’);
89: fprintf(‘
period, input the smoothing ‘);
90: fprintf(‘(give 0.0 for no smoothing).
’);
91: alpha=input(‘> ? ’);
92: yfou=fousum(c,xtmp,period,nh,alpha);
93: xtmp=xtmp; idneg=find(xtmp<0);
94: xng=abs(xtmp(idneg));
95: xtmp(idneg)=xtmp(idneg)+ …
96: period*ceil(xng/period);
97: if nd>0
98: yexac=lintrp(xd,yd,rem(xxtmp,period));
99: else
100: if nopt == 1
101: yexac=feval(fnam,xtmp,period);
102: else
103: x=xtmp; yexac=eval(strng);
104: end
105: end
106: in=int2str(nh);
107: if alpha == 0
108: titl=[’Fourier Series for Harmonics ’ …
109: ’up to Order ’,in];
110: else
111: titl=[’Smoothed Fourier Series for ’ …
% Harmonics up to Order ',in];

clf; plot(xtmp,yfou,'-',xtmp,yexact,'--');
ylabel('y axis'); xlabel('x axis'); zoom on
title(titl); grid on; figure(gcf); disp('');
input('You can zoom in with the mouse button.');
if pltsav disp('');
  filnam=input(['Give a file name to ',...
    'save the current graph > ? ',']','s');
  if length(filnam) > 0 eval(['print -deps ',filnam]);
end
end

%=============================================

function y=sine(x,period)
% y=sine(x,period)
% ~~~~~~~~~~~~~~~~
% Function for all or part of a sine wave.
% x,period - vector argument and period
% y - function value
% y=sin(rem(x,period));

%=============================================

function yreal=fousum(c,x,period,k, alpha)
% % yreal = fousum(c,x,period,k, alpha)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Sum the Fourier series of a real
% valued function.
% x - The vector of real values at which the series is evaluated.
% c - A vector of length n containing Fourier coefficients output by the fft function
% period - The period of the function
% k - The highest harmonic used in the Fourier sum. This must

© 2003 by CRC Press LLC
x = x(:); n = length(c);
if nargin < 4, k = n/2-1; alpha = 0; end
if nargin < 5, alpha = 0; end
if nargin < 3, period = 2*pi; end
L = period/2; k = min(k, n/2-1); th = (pi/L)*x;
i = sqrt(-1); z = exp(i*th);
y = c(k+1)*ones(size(th)); pa = pi*alpha;
if alpha > 0
   jj = (1:k);
c(jj+1) = c(jj+1).*sin(jj*pa)./(jj*pa);
end
for j = k:-1:2, y = c(j)+y.*z; end
yreal = real(c(1)+2*y.*z)/n;
function varargout=inputv(prompt)

% See Appendix B
Chapter 7

Dynamic Response of Linear Second Order Systems

7.1 Solving the Structural Dynamics Equations for Periodic Applied Forces

The dynamics of a linear structure subjected to periodic forces obeys the matrix differential equation

\[ M \ddot{X} + C \dot{X} + KX = F(t), \]

with initial conditions

\[ X(0) = D_0, \quad \dot{X}(0) = V_0. \]

The solution vector \( X(t) \) has dimension \( n \) and \( M, C, \) and \( K \) are real square matrices of order \( n \). The mass matrix, \( M \), the damping matrix, \( C \), and the stiffness matrix, \( K \), are all real. The forcing function \( F(t) \), assumed to be real and having period \( L \), can be approximated by a finite trigonometric series as

\[ F(t) = \sum_{k=-N}^{N} c_k e^{i\omega_k t} \] where \( \omega_k = \frac{2\pi k}{L} \)

and \( i = \sqrt{-1} \). The Fourier coefficients \( c_k \) are vectors that can be computed using the FFT. The fact that \( F(t) \) is real also implies that \( c_{-k} = \text{conj}(c_k) \) and, therefore,

\[ F(t) = c_0 + 2 \text{ real} \left( \sum_{k=1}^{n} c_k e^{i\omega_k t} \right). \]

The solution of the differential equation is naturally resolvable into two distinct parts. The first is the so called particular or forced response which is periodic and has the same general mathematical form as the forcing function. Hence, we write

\[ X_p = \sum_{k=-n}^{n} X_k e^{i\omega_k t} = X_0 + 2 \text{ real} \left( \sum_{k=1}^{n} X_k e^{i\omega_k t} \right). \]

Substituting this series into the differential equation and matching coefficients of \( e^{i\omega_k t} \) on both sides yields

\[ X_k = (K - \omega_k^2 M + i\omega_k C)^{-1} c_k. \]
The particular solution satisfies initial conditions given by

\[ X_p(0) = X_0 + 2 \text{ real } \left( \sum_{k=1}^{n} c_k \right) \quad \text{and} \quad \dot{X}_p(0) = 2 \text{ real } \left( \sum_{k=1}^{n} \omega_k c_k \right) . \]

Since these conditions usually will not equal the desired values, the particular solution must be combined with what is called the homogeneous or transient solution \( X_h \), where

\[ M \ddot{X}_h + c \dot{X}_h + KX_h = 0, \]

with

\[ X_h(0) = D_0 - X_p(0), \quad \dot{X}_h(0) = V_0 - \dot{X}_p(0). \]

The homogeneous solution can be constructed by reducing the original differential equation to first order form. Let \( Z \) be the vector of dimension \( 2n \) which is the concatenation of \( X \) and \( \dot{X} = V \). Hence, \( Z = [X; V] \) and the original equation of motion is

\[ \frac{dZ}{dt} = AZ + P(t) \]

where

\[ A = \begin{bmatrix} 0 & I \\ -M^{-1}K & -M^{-1}C \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} 0 \\ m^{-1}F \end{bmatrix} . \]

The homogeneous differential equation resulting when \( P = 0 \) can be solved in terms of the eigenvalues and eigenvectors of matrix \( A \). If we know the eigenvalues \( \lambda_j \) and eigenvectors \( U_j \) satisfying

\[ AU_j = \lambda_j U_j, \quad 1 \leq j \leq 2n, \]

then the homogeneous solution can be written as

\[ Z = \sum_{j=1}^{2n} z_j U_j e^{\lambda_j t}. \]

The weighting coefficients \( z_j \) are computed to satisfy the desired initial conditions which require

\[ \begin{bmatrix} U_1, U_2, \cdots, U_{2n} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_{2n} \end{bmatrix} = \begin{bmatrix} X_0 - X_p(0) \\ V_0 - \dot{X}_p(0) \end{bmatrix} . \]

We solve this system of equations for \( z_1, \cdots, z_{2n} \) and replace each \( U_j \) by \( z_j U_j \). Then the homogeneous solution is

\[ X_h = \sum_{j=1}^{n} U_j(1:n)e^{\lambda_j t} \]
where \( U_j(1:n) \) means we take only the first \( n \) elements of column \( j \).

In most practical situations, the matrix \( C \) is nonzero and the eigenvalues \( \lambda_1, \ldots, \lambda_{2n} \) have negative real parts. Then the exponential terms \( e^{\lambda_j t} \) all decay with increasing time, which is why \( X_h \) is often known as the transient solution. In other cases, where the damping matrix \( C \) is zero, the eigenvalues \( \lambda_j \) are typically purely imaginary, and the homogeneous solution does not die out. In either instance, it is often customary in practical situations to ignore the homogeneous solution because it is usually small when compared to the contribution of the particular solution.

### 7.1.1 Application to Oscillations of a Vertically Suspended Cable

Let us solve the problem of small transverse vibrations of a vertically suspended cable. This system illustrates how the natural frequencies and mode shapes of a linear system can be combined to satisfy general initial conditions on position and velocity.

The cable in Figure 7.1 is idealized as a series of \( n \) rigid links connected at frictionless joints. Two vectors, consisting of link lengths \( [\ell_1, \ell_2, \ldots, \ell_n] \) and masses \( [m_1, m_2, \ldots, m_n] \) lumped at the joints, characterize the system properties. The accelerations in the vertical direction will be negligibly small compared to transverse accelerations, because the transverse displacements are small. Consequently, the tension in the chain will remain close to the static equilibrium value. This means the tension in link \( i \) is

\[
T_i = g b_i \text{ where } b_i = \sum_{j=i}^{n} m_j.
\]

We assume that the transverse displacement \( y_i \) for mass \( m_i \) is small compared to the total length of the cable. A free body diagram for mass \( i \) is shown in Figure 7.2. The small deflection angles are related to the transverse deflections by

\[
\theta_{i+1} = (y_{i+1} - y_i) \ell_{i+1} \text{ and } \theta_i = (y_i - y_{i-1}) / \ell_i.
\]

Summation of forces shows that the horizontal acceleration is governed by

\[
m_i \ddot{y}_i = g (b_{i+1}/\ell_{i+1}) (y_{i+1} - y_i) - g (b_i/\ell_i) (y_i - y_{i-1})
\]

\[
= g (b_i/\ell_i) y_{i-1} - g (b_i/\ell_i + b_{i+1}/\ell_{i+1}) y_i + g (b_{i+1}/\ell_{i+1}) y_{i+1}.
\]

In matrix form this equation is

\[
M \ddot{Y} + KY = 0
\]

where \( M \) is a diagonal matrix of mass coefficients and \( K \) is a symmetric tridiagonal matrix. The natural modes of free vibration are dynamical states where each element of the system simultaneously moves with harmonic motion of the same frequency. This means we seek motions of the form \( Y = U \cos(\omega t) \), or equivalently \( Y = U \sin(\omega t) \), which implies

\[
KU_j = \lambda_j M U_j \text{ where } \lambda_j = \omega_j^2 \text{ for } 1 \leq j \leq n.
\]
Solving the eigenvalue problem \((M^{-1}K)U = \lambda U\) gives the natural frequencies \(\omega_1, \ldots, \omega_n\) and the modal vectors \(U_1, \ldots, U_n\). The response to general initial conditions is then obtained by superposition of the component modes. We write

\[
Y = \sum_{j=1}^{n} \cos(\omega_j t)U_j c_j + \sin(\omega_j t)U_j d_j / \omega_j
\]

where the coefficients \(c_1, \ldots, c_n\) and \(d_1, \ldots, d_n\) (not to be confused with Fourier coefficients) are determined from the initial conditions as

\[
\begin{bmatrix}
U_1, \cdots, U_n
\end{bmatrix}
\begin{bmatrix}
c_1 \\
\vdots \\
c_n
\end{bmatrix} = Y(0), \; c = U^{-1}Y(0),
\]

\[
\begin{bmatrix}
U_1, \cdots, U_n
\end{bmatrix}
\begin{bmatrix}
d_1 \\
\vdots \\
d_n
\end{bmatrix} = \dot{Y}(0), \; d = U^{-1}\dot{Y}(0).
\]

The following program determines the cable response for general initial conditions. The natural frequencies and mode shapes are computed along with an animation of the motion.

The cable motion produced when an initially vertical system is given the same initial transverse velocity for all masses was studied. Graphical results of the analysis appear in Figures 7.3 through 7.6. The surface plot in Figure 7.3 shows the cable deflection pattern in terms of longitudinal position and time. Figure 7.4 shows the deflection pattern at two times. Figure 7.5 traces the motion of the middle and the
free end. At $t = 1$, the wave propagating downward from the support point is about halfway down the cable. By $t = 2$, the wave has reached the free end and the cable is about to swing back. Finally, traces of cable positions during successive stages of motion appear in Figure 7.6.
The cable was initially vertical and was given a uniform transverse velocity. A 30 link model was used.

Figure 7.3: Surface Showing Cable Deflection

Figure 7.4: Cable Transverse Deflection at $t = 1$ and $t = 2$
Figure 7.5: Position Versus Time for the Cable Middle and End

Figure 7.6: Trace of Cable Motion
MATLAB Example

Program cablinea

function cablinea
% Example: cablinea
% ~~~~~~~~~~~~~~~~~
% This program uses modal superposition to
% compute the dynamic response of a cable
% suspended at one end and free at the other.
% The cable is given a uniform initial
% velocity. Time history plots and animation
% of the motion are provided.
% User m functions required:
% cablemk, udfrevib, canimate

% Initialize graphics
hold off; axis('normal'); close;

% Set physical parameters
n=30; gravity=1.; masses=ones(n,1)/n;
lengths=ones(n,1)/n;

% Obtain mass and stiffness matrices
[m,k]=cablemk(masses,lengths,gravity);

% Assign initial conditions & time limit
% for solution
dsp=zeros(n,1); vel=ones(n,1);
tmin=0; tmax=10; ntim=30;

% Compute the solution by modal superposition
[t,u,modvc,natfrq]=...
udfrevib(m,k,dsp,vel,tmin,tmax,ntim);

% Interpret results graphically
nt1=sum(t<=tmin); nt2=sum(t<=tmax);
u=[zeros(ntim,1),u];
y=cumsum(lengths); y=[0;y(:)];

% Plot deflection surface
disp(' '), disp('TRANSVERSE MOTION OF A CABLE')
surf(y,t,u); xlabel('y axis'); ylabel('time');
41:  zlabel('transverse deflection');
42:  title('Surface Showing Cable Deflection');
43:  colormap('default'), view([30,30]); figure(gcf);
44:  disp(['Press [Enter] to see the cable ',...
45:         'position at two times'])
46:  pause, %print -deps surface
47:
48:  % Show deflection configuration at two times
49:  % Use closer time increment than was used
50:  % for the surface plots.
51:  mtim=4*ntim;
52:  [tt,uu,modvc,natfrq]=...  
53:     udfrevib(m,k,dsp,vel,tmin,tmax,mtim);
54:  uu=[zeros(mtim,1),uu];
55:  tp1=.1*tmax; tp2=.2*tmax;
56:  s1=num2str(tp1); s2=num2str(tp2);
57:  np1=sum(tt<=tp1); np2=sum(tt<=tp2);
58:  u1=uu(np1,:); u2=uu(np2,:);
59:  yp=flipud(y(:)); ym=max(yp);
60:  plot(u1,yp,'-',u2,yp,'--');
61:  ylabel('distance from bottom');
62:  xlabel('transverse displacement');
63:  title(['Cable Transverse Deflection '...
64:         'at t = ',s1,' and t = ',s2]);
65:  legend('t = 1', 't = 2');
66:
67:  xm=.2*max([u1(:);u2(:)]);
68:  ntxt=int2str(n); n2=1+fix(n/2);
69:  str=strvcat(...
70:     ['The cable was initially vertical and was',...
71:        'given a uniform transverse velocity.',...
72:        ['A ',ntxt,' link model was used.']);
73:  text(xm,.9*ym,str), figure(gcf);
74:  disp([['Press [Enter] to show the time ',...
75:          'response at the middle and free end']])
76:  pause, %print -deps twoposn
77:  % Plot time history for the middle and the end
78:  clf; plot(tt,uu(:,n2),'--',tt,uu(:,n+1),'-');
79:  xlabel('dimensionless time');
80:  ylabel('transverse displacement');
81:  title(['Position versus Time for the '...
82:        'Cable Middle and End'])
83:  legend('Midpoint', 'Lower end');
84:  figure(gcf);
disp('Press [Enter] for a motion trace')
pause, %print -deps 2timhist

% Plot animation of motion history
clf; canimate(y,u,t,0,.5*max(t),1);
%print -deps motntrac
disp('Press [Enter] to finish'), pause, close;

%=============================================%  
function [m,k]=cablemk(masses,lngths,gravty)
%
% [m,k]=cablemk(masses,lngths,gravty)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Form the mass and stiffness matrices for
% the cable.
%
% masses - vector of masses
% lngths - vector of link lengths
% gravty - gravity constant
% m,k - mass and stiffness matrices
%
% User m functions called: none.
%=============================================%  

m=diag(masses);
b=flipud(cumsum(flipud(masses(:))))* ...  
gravty./lngths;
n=length(masses); k=zeros(n,n); k(n,n)=b(n);
for i=1:n-1
k(i,i)=b(i)+b(i+1); k(i,i+1)=-b(i+1);
k(i+1,i)=k(i,i+1);
end

%=============================================%  
function [t,u,mdvc,natfrq]=...  
udfrevib(m,k,u0,v0,tmin,tmax,nt)
%
% [t,u,mdvc,natfrq]= ...  
% udfrevib(m,k,u0,v0,tmin,tmax,nt)
%
% This function computes undamped natural
% frequencies, modal vectors, and time response
% by modal superposition. The matrix
% differential equation and initial conditions
% are
% m u'' + k u = 0,  u(0) = u0,  u'(0) = v0
% m,k - mass and stiffness matrices
% u0,v0 - initial position and velocity vectors
% tmin,tmax - time limits for solution evaluation
% nt - number of times for solution
% t - vector of solution times
% u - matrix with row j giving the system response at time t(j)
% mdvc - matrix with columns which are modal vectors
% natfrq - vector of natural frequencies
% User m functions called: none.
%---------------------------------------------------------------

% Call function eig to compute modal vectors and frequencies
% [mdvc,w]=eig(m\k);
% [w,id]=sort(diag(w)); w=sqrt(w);
% Arrange frequencies in ascending order
% mdvc=mdvc(:,id); z=mdvc\[u0(:,),v0(:,)];
% Generate vector of equidistant times
% t=linspace(tmin,tmax,nt);
% Evaluate the displacement as a function of time
% u=(mdvc*diag(z(:,1)))\cos(w*t)+...
% (mdvc*diag(z(:,2)./w))*sin(w*t);
% t=t(:); u=u'; natfrq=w;
%=============================================
% data values stored in array u. The
% different columns of u correspond to position
% values in vector y. The successive rows of u
% correspond to different times. Parameter
% tpause controls the speed of the animation.
% 
% u - matrix of values for which
% animated plots of u versus y
% are required
% y - spatial positions for different
% columns of u
% t - time vector at which positions
% are known
% tmin,tmax - time limits for graphing of the
% solution
% norub - parameter which makes all
% position images remain on the
% screen. Only one image at a
% time shows if norub is left out.
% A new cable position appears each
% time the user presses any key
% 
% User m functions called: none.
%---------------------------------------------------------------

% If norub is input,
% all images are left on the screen
if nargin < 6
    rubout = 1;
else
    rubout = 0;
end

% Determine window limits
umin=min(u(:)); umax=max(u(:)); udif=umax-umin;
 uavg=.5*(umin+umax);
ymin=min(y); ymax=max(y); ydif=ymax-ymin;
yavg=.5*(ymin+ymax);
ywmin=yavg-.55*ydif; ywmax=yavg+.55*ydif;
uwmin=uavg-.55*udif; uwmax=uavg+.55*udif;
n1=sum(t<=tmin); n2=sum(t<=tmax);
t=t(n1:n2); u=u(n1:n2,:);
u=fliplr (u); [ntime,nxpts]=size(u);

hold off; cla; ey=0; eu=0; axis('square');
% Plot successive positions
for j=1:ntime
  ut=u(j,:); plot(ut,y,’-’);
  figure(gcf); pause(.5);
  % Erase image before next one appears
  if rubout & j < ntime, cla, end
end

7.2 Direct Integration Methods

Using stepwise integration methods to solve the structural dynamics equation provides an alternative to frequency analysis methods. If we invert the mass matrix and save the result for later use, the \( n \) degree-of-freedom system can be expressed concisely as a first order system in \( 2n \) unknowns for a vector \( z = [x; v] \), where \( v \) is the time derivative of \( x \). The system can be solved by applying the variable step-size differential equation integrator \texttt{ode45} as indicated in the following function:

```matlab
function [t,x]=strdynrk(t,x0,v0,m,c,k,functim)
% [t,x]=strdynrk(t,x0,v0,m,c,k,functim)
global Mi C K F n n1 n2
Mi=inv(m); C=c; K=k; F=functim;
n=size(m,1); n1=1:n; n2=n+1:2*n;
[t,z]=ode45(@sde,t,[x0(:,1);v0(:,1)]); x=z(:,n1);
%================================
function zp=sde(t,z)
global Mi C K F n n1 n2
zp=[z(n2); Mi*(feval(F,t)-C*z(n2)-K*z(n1))];
%================================
function f=func(t)
% m=eye(3,3); k=[2,-1,0;-1,2,-1;0,-1,2];
% c=.05*k;
f=[-1;0;1]*sin(1.413*t);
```

In this function, the inverted mass matrix is stored in a global variable \( Mi \), the damping and stiffness matrices are in \( C \) and \( K \), and the forcing function name is stored in a character string called \( \texttt{functim} \). Although this approach is easy to im-

© 2003 by CRC Press LLC
plement, the resulting analysis can be very time consuming for systems involving several hundred degrees of freedom. Variable step integrators make adjustments to control stability and accuracy which can require very small integration steps. Consequently, less sophisticated formulations employing fixed step-size are often employed in finite element programs. We will investigate two such algorithms derived from trapezoidal integration rules \([7, 113]\). The two fundamental integration formulas \([26]\) needed are:

\[
\int_a^b f(t)\,dt = \frac{h}{2} [f(a) + f(b)] - \frac{h^3}{12} f''(\epsilon_1)
\]

and

\[
\int_a^b f(t)\,dt = \frac{h}{2} [f(a) + f(b)] + \frac{h^2}{12} [f'(a) - f'(b)] + \frac{h^5}{720} f^{(4)}(\epsilon_2)
\]

where \(a < \epsilon_i < b\) and \(h = b - a\). The first formula, called the trapezoidal rule, gives a zero truncation error term when applied to a linear function. Similarly, the second formula, called the trapezoidal rule with end correction, has a zero final term for a cubic integrand.

The idea is to multiply the differential equation by \(dt\), integrate from \(t\) to \((t+h)\), and employ numerical integration formulas while observing that \(M, C,\) and \(K\) are constant matrices, or

\[
M \int_t^{t+h} \dot{V} \,dt + C \int_t^{t+h} \dot{X} \,dt + K \int_t^{t+h} X \,dt = \int_t^{t+h} P(t) \,dt
\]

and

\[
\int_t^{t+h} \dot{X} \,dt = \int_t^{t+h} V \,dt.
\]

For brevity we utilize a notation characterized by \(X(t) = X_0, X(t+h) = X_1, \dot{X} = X_1 - X_0\). The trapezoidal rule immediately leads to

\[
\left[M + \frac{h}{2} C + \frac{h^2}{4} K\right] \ddot{V} = \int_t^{t+h} P(t)\,dt - h \left[CV_0 + K(X_0 + \frac{h}{2} V_0)\right] + O(h^3).
\]

The last equation is a balance of impulse and momentum change involving the effective mass matrix

\[
M_e = \left[M + \frac{h}{2} C + \frac{h^2}{4} K\right]
\]

which can be inverted once and used repeatedly if the step-size is not changed.

To integrate the forcing function we can use the midpoint rule \([26]\) which states that

\[
\int_a^b P(t) \,dt = hP \left(\frac{a + b}{2}\right) + O(h^3).
\]
Solving for $\ddot{V}$ yields

$$\ddot{V} = \left[ M + \frac{h}{2} C + \frac{h^2}{4} K \right]^{-1} \left[ P \left( t + \frac{h}{2} \right) - CV_0 - K \left( X_0 + \frac{h}{2} V_0 \right) h \right] + O(h^3).$$

The velocity and position at $(t + h)$ are then computed as

$$V_1 = V_0 + \ddot{V}, \quad X_1 = X_0 + \frac{h}{2} [V_0 + V_1] + O(h^3).$$

A more accurate formula with truncation error of order $h^5$ can be developed from the extended trapezoidal rule. This leads to

$$M \ddot{V} + C \dot{X} + K \left[ \frac{h}{2} \dot{X} + 2X_0 \right] = \int_t^{t+h} P(t) dt + O(h^5)$$

and

$$\dot{X} = \frac{h}{2} \dot{V} + 2V_0 + \frac{h^2}{12} [V_0 - V_1] + O(h^5).$$

Multiplying the last equation by $M$ and employing the differential equation to reduce the $V_0 - V_1$ terms gives

$$M \ddot{X} = \frac{h}{2} M \dot{V} + \frac{h^2}{12} [P_0 - P_1] + O(h^5).$$

These results can be arranged into a single matrix equation to be solved for $\dot{X}$ and $\ddot{V}$:

$$\begin{bmatrix} (M - \frac{h^2}{12} K) & (M - \frac{h^2}{12} K) \\ (C + \frac{h^2}{6} K) & (C + \frac{h^2}{6} K) \end{bmatrix} \begin{bmatrix} \ddot{V} \\ \dot{X} \end{bmatrix} = \begin{bmatrix} hMV_0 + \frac{h^2}{12} (P_0 - P_1) \\ \int P dt - hKX_0 \end{bmatrix} + O(h^5).$$

A Gauss two-point formula [26] evaluates the force integral consistent with the desired error order so that

$$\int_t^{t+h} P(t) dt = \frac{h}{2} \left[ P(t + \alpha h) + P(t + \beta h) \right] + O(h^5)$$

where $\alpha = \frac{3 - \sqrt{3}}{6}$ and $\beta = \frac{3 + \sqrt{3}}{6}$.

### 7.2.1 Example on Cable Response by Direct Integration

Functions implementing the last two algorithms appear in the following program which solves the previously considered cable dynamics example by direct integration. Questions of computational efficiency and numerical accuracy are examined for two different step-sizes. Figures 7.7 and 7.8 present solution times as multiples of the times needed for a modal response solution. The accuracy measures employed...
are described next. Note that the displacement response matrix has rows describing system positions at successive times. Consequently, a measure of the difference between approximate and exact solutions is given by the vector

\[
\text{error\_vector} = \sqrt{\text{sum}((x\_aprox - x\_exact)^2)}
\]

Typically this vector has small initial components (near \( t = 0 \)) and larger components (near the final time). The error measure is compared for different integrators and time steps in the figures. Note that the fourth order integrator is more efficient than the second order integrator because a larger integration step can be taken without excessive loss in accuracy. Using \( h = 0.4 \) for \texttt{mkde4i} achieved nearly the same accuracy as that given by \texttt{mkde2i} with \( h = 0.067 \). However, the computation time for \texttt{mkde2i} was several times as large as that for \texttt{mkde4i}.

In the past it has been traditional to use only second order methods for solving the structural dynamics equation. This may have been dictated by considerations on computer memory. Since workstations widely available today have relatively large memories and can invert a matrix of order two hundred in about half a second, it appears that use of high order integrators may gain in popularity.

The following computer program concludes our chapter on the solution of linear,

\[
\begin{align*}
\text{error\_vector} &= \sqrt{\text{sum}((x\_aprox - x\_exact)^2)}
\end{align*}
\]

© 2003 by CRC Press LLC
constant-coefficient matrix differential equations. Then we will study, in the next chapter, the Runge-Kutta method for integrating nonlinear problems.
MATLAB Example

Program deislner

sfunction deislner
%
% Example: deislner
% -------------------
% Solution error for simulation of cable
% motion using a second or a fourth order
% implicit integrator.
%
% This program uses implicit second or fourth
% order integrators to compute the dynamical
% response of a cable which is suspended at
% one end and is free at the other end. The
% cable is given a uniform initial velocity.
% A plot of the solution error is given for
% two cases where approximate solutions are
% generated using numerical integration rather
% than modal response which is exact.
%
% User m functions required:
% mckde2i, mckde4i, cablemk, udfrevib,
% plterror

% Choose a model having twenty links of
% equal length

fprintf(...
'\nPlease wait: solution takes a while\n')
clear all
n=20; gravty=1.; n2=1+fix(n/2);
masses=ones(n,1)/n; lengths=ones(n,1)/n;

% First generate the exact solution by
% modal superposition
[m,k]=cablemk(masses,lengths,gravty);
c=zeros(size(m));
dsp=zeros(n,1); vel=ones(n,1);
t0=0; tfin=50; ntim=126; h=(tfin-t0)/(ntim-1);

% Numbers of repetitions each solution is
% performed to get accurate cpu times for
% the chosen step sizes are shown below.
% Parameter jmr may need to be increased to
% give reliable cpu times on fast computers

jmr=500;
j2=fix(jmr/50); J2=fix(jmr/25);
j4=fix(jmr/20); J4=fix(jmr/10);

% Loop through all solutions repeatedly to
% obtain more reliable timing values on fast
% computers
tic;
for j=1:jmr;
    [tmr,xmr]=udfrevib(m,k,dsp,vel,t0,tfin,ntim);
end
tcprmr=toc/jmr;

% Second order implicit results
i2=10; h2=h/i2; tic;
for j=1:j2
    [t2,x2]=mckde2i(m,c,k,t0,dsp,vel,tfin,h2,i2);
end
tcp2=toc/j2; tr2=tcp2/tcprmr;

I2=5; H2=h/I2; tic;
for j=1:J2
    [T2,X2]=mckde2i(m,c,k,t0,dsp,vel,tfin,H2,I2);
end
Tcp2=toc/J2; Tr2=Tcp2/tcprmr;

% Fourth order implicit results
i4=2; h4=h/i4; tic;
for j=1:j4
    [t4,x4]=mckde4i(m,c,k,t0,dsp,vel,tfin,h4,i4);
end
tcp4=toc/j4; tr4=tcp4/tcprmr;

I4=1; H4=h/I4; tic;
for j=1:J4
    [T4,X4]=mckde4i(m,c,k,t0,dsp,vel,tfin,H4,I4);
end
Tcp4=toc/J4; Tr4=Tcp4/tcprmr;

% Plot error measures for each solution
plterror(xmr,t2,h2,x2,T2,H2,X2,...
function [t,x,tcp] = ...
  mckde2i(m,c,k,t0,x0,v0,tmax,h,incout,forc)
%
% This function uses a second order implicit \% to solve the matrix differential
% equation
% \[ m \ddot{x} + c \dot{x} + k x = \text{forc}(t) \]
% where \( m, c, \text{and} k \) are constant matrices and
% \( \text{forc} \) is an externally defined function.
%\%
% Input:
%-------
% \( m, c, k \) mass, damping and stiffness matrices
% \( t0 \) starting time
% \( x0, v0 \) initial displacement and velocity
% \( tmax \) maximum time for solution evaluation
% \( h \) integration stepsize
% \( \text{incout} \) number of integration steps between
% successive values of output
% \( \text{forc} \) externally defined time dependent
% forcing function. This parameter
% should be omitted if no forcing
%\%
% Output:
%-------
% \( t \) time vector going from \( t0 \) to \( tmax \)
% in steps of
% \( \text{h*incout} \) to yield a matrix of
% solution values such that row \( j \)
% is the solution vector at time \( t(j) \)
% \( \text{tcp} \) computer time for the computation
%\%
% User m functions called: none.

if (nargin > 9); force=1; else, force=0; end
if nargout ==3, tcp=clock; end
hbig = h * incout;
t = (t0 : hbig : tmax)'; n = length(t);
ns = (n - 1) * incout; ts = t0 + h * (0 : ns)';
xnow = x0(:); vnow = v0(:);
nvar = length(x0);
jrow = 1; jstep = 0; h2 = h / 2;

% Form the inverse of the effective
% stiffness matrix
mnv = h * inv(m + h2 * (c + h2 * k));

% Initialize the output matrix for x
x = zeros(n, nvar); x(1, :) = xnow';
zroforc = zeros(length(x0), 1);

% Main integration loop
for j = 1 : ns
tj = ts(j); tjh = tj + h2;
if force
   dv = feval(forc, tjh);
else
   dv = zroforc;
end

dv = mnv * (dv - c * vnow - k * (xnow + h2 * vnow));
vnext = vnow + dv; xnext = xnow + h2 * (vnow + vnext);
jstep = jstep + 1;
if jstep == incout
   jstep = 0; jrow = jrow + 1; x(jrow, :) = xnext';
end
xnow = xnext; vnow = vnext;
end
if nargout == 3
   tcp = etime(clock, tcp);
else
   tcp = [];
end

function [t, x, tcp] = ...
mckde4i(m, c, k, t0, x0, v0, tmax, h, incout, forc)
% [t, x, tcp] = ...
% mckde4i(m, c, k, t0, x0, v0, tmax, h, incout, forc)
This function uses a fourth order implicit integrator with fixed stepsize to solve the matrix differential equation
\[
m x'' + c x' + k x = \text{forc}(t)
\]
where \(m, c,\) and \(k\) are constant matrices and \(\text{forc}\) is an externally defined function.

\% Input:
\% 
\% \(m, c, k\) mass, damping and stiffness matrices
\% \(t0\) starting time
\% \(x0, v0\) initial displacement and velocity
\% \(tmax\) maximum time for solution evaluation
\% \(h\) integration stepsize
\% \(incout\) number of integration steps between successive values of output
\% \(\text{forc}\) externally defined time dependent forcing function. This parameter should be omitted if no forcing function is used.

\% Output:
\% 
\% \(t\) time vector going from \(t0\) to \(tmax\) in steps of \(h*incout\)
\% \(x\) matrix of solution values such that row \(j\) is the solution vector at time \(t(j)\)
\% \(tcp\) computer time for the computation

\% User m functions called: none.

\%----------------------------------------------

if nargin > 9, force=1; else, force=0; end
if nargout ==3, tcp=clock; end
hbig=h*incout; t=(t0:hbig:tmax)';
n=length(t); ns=(n-1)*incout; nvar=length(x0);
jrow=1; jstep=0; h2=h/2; h12=h*h/12;

\% Form the inverse of the effective stiffness matrix for later use.

m12=m-h12*k;
mnv=inv([[-h2*m-h12*c],m12]; [m12,(c+h2*k)]);
% The forcing function is integrated using a 2 point Gauss rule
r3=sqrt(3); b1=h*(3-r3)/6; b2=h*(3+r3)/6;

% Initialize output matrix for x and other variables
xnow=x0(:); vnow=v0(:);
tnow=t0; zroforc=zeros(length(x0),1);

if force
    fnow=feval(forc,tnow);
else
    fnow=zroforc;
end

x=zeros(n,nvar); x(1,:)=xnow'; fnext=fnow;

% Main integration loop
for j=1:ns
    tnow=t0+(j-1)*h; tnext=tnow+h;
    if force
        fnext=feval(forc,tnext);
        di1=h12*(fnow-fnext);
        di2=h2*(feval(forc,tnow+b1)+ feval(forc,tnow+b2));
        z=mnv*[di1+m*(h*vnow)); (di2-k*(h*xnow))];
        fnow=fnext;
    else
        z=mnv*[m*(h*vnow); -k*(h*xnow)];
    end
    vnext=vnow + z(1:nvar);
    xnext=xnow + z((nvar+1):2*nvar);
    jstep=jstep+1;

    if jstep == incout
        jstep=0; jrow=jrow+1; x(jrow,:)=xnext';
    end

    % Save results every incout steps
    if nargout==3
        tcp=etime(clock,tcp);
    else
        tcp=etime(clock,tcp);
    end

% Update quantities for next step
xnow=xnext; vnow=vnext; fnow=fnext;
if nargout==3
    tcp=etime(clock,tcp);
else
function [m,k]=cablemk(masses,lngths,gravty)
% [m,k]=cablemk(masses,lngths,gravty)
% Form the mass and stiffness matrices for the cable.
% masses -- vector of masses
% lngths -- vector of link lengths
% gravty -- gravity constant
% m,k -- mass and stiffness matrices
% User m functions called: none.

m=diag(masses);
b=flipud(cumsum(flipud(masses(:))))* ...
  gravty./lngths;
n=length(masses); k=zeros(n,n); k(n,n)=b(n);
for i=1:n-1
  k(i,i)=b(i)+b(i+1); k(i,i+1)=-b(i+1);
  k(i+1,i)=k(i,i+1);
end

function plterror(xmr,t2,h2,x2,T2,H2,X2,...
  t4,h4,x4,T4,X4,Tr2,Tr2,Tr4,Tr4)
% plterror(xmr,t2,h2,x2,T2,H2,X2,...
% Plots error measures showing how different integrators and time steps compare with the exact solution using modal response.
% User m functions called: none

% Compare the maximum error in any component
% at each time with the largest deflection
% occurring during the complete time history
maxd=max(abs(xmr(:))); er2=max(abs(x2-xmr)')/maxd;
Er2=max(abs(X2-xmr)')/maxd;
er4=max(abs(x4-xmr)')/maxd;
Er4=max(abs(X4-xmr)')/maxd;

plot(t2,er2,'-',T2,Er2,'--');
title(['Solution Error For Implicit ',...
    '2nd Order Integrator']);
xlabel('time');
ylabel('solution error measure');
lg1=['h= ', num2str(h2), ...
    ', relative cputime= ', num2str(tr2)];
lg2=['h= ', num2str(H2), ...
    ', relative cputime= ', num2str(Tr2)];
legend(lg1,lg2,2); figure(gcf);
% print -deps deislne2

plot(t4,er4,'-',T4,Er4,'--');
title(['Solution Error For Implicit ',...
    '4th Order Integrator']);
xlabel('time');
ylabel('solution error measure');
lg1=['h= ', num2str(h4), ...
    ', relative cputime= ', num2str(tr4)];
lg2=['h= ', num2str(H4), ...
    ', relative cputime= ', num2str(Tr4)];
legend(lg1,lg2,2); figure(gcf);
% print -deps deislne4
disp(' '), disp('All Done')

%=============================================

% function [t,u,mdvc,natfrq]=...
% udfrevib(m,k,u0,v0,tmin,tmax,nt)
% See Appendix B

© 2003 by CRC Press LLC
Chapter 8

Integration of Nonlinear Initial Value Problems

8.1 General Concepts on Numerical Integration of Nonlinear Matrix Differential Equations

Methods for solving differential equations numerically are one of the most valuable analysis tools now available. Inexpensive computer power and user friendly software are stimulating wider use of digital simulation methods. At the same time, intelligent use of numerically integrated solutions requires appreciation of inherent limitations of the techniques employed. The present chapter discusses the widely used Runge-Kutta method and applies it to some specific examples.

When physical systems are described by mathematical models, it is common that various system parameters are only known approximately. For example, to predict the response of a building undergoing earthquake excitation, simplified formulations may be necessary to handle the elastic and frictional characteristics of the soil and the building. Our observation that simple models are used often to investigate behavior of complex systems does not necessarily amount to a rejection of such procedures. In fact, good engineering analysis depends critically on development of reliable models which can capture salient features of a process without employing unnecessary complexity. At the same time, analysts need to maintain proper caution regarding trustworthiness of answers produced with computer models. Nonlinear system response sometimes changes greatly when only small changes are made in the physical parameters. Scientists today realize that, in dealing with highly nonlinear phenomena such as weather prediction, it is simply impossible to make reliable long term forecasts [45] because of various unalterable factors. Among these are a) uncertainty about initial conditions, b) uncertainty about the adequacy of mathematical models describing relevant physical processes, c) uncertainty about error contributions arising from use of spatial and time discretizations in construction of approximate numerical solutions, and d) uncertainty about effects of arithmetic roundoff error. In light of the criticism and cautions being stated about the dangers of using numerical solutions, the thrust of the discussion is that idealized models must not be regarded as infallible, and no numerical solution should be accepted as credible without adequately investigating effects of parameter perturbation within uncertainty limits of the parameters. To illustrate how sensitive a system can be to initial conditions,
might consider a very simple model concerning motion of a pendulum of length \( \ell \) given an initial velocity \( v_0 \) starting from a vertically downward position. If \( v_0 \) exceeds \( 2\sqrt{g\ell} \), the pendulum will reach a vertically upward position and will go over the top. If \( v_0 \) is less than \( 2\sqrt{g\ell} \), the vertically upward position is never reached. Instead, the pendulum oscillates about the bottom position. Consequently, initial velocities of \( 1.999\sqrt{g\ell} \) and \( 2.001\sqrt{g\ell} \) produce quite different system behavior with only a tiny change in initial velocity. Other examples illustrating the difficulties of computing the response of nonlinear systems are cited below. These examples are not chosen to discourage use of the powerful tools now available for numerical integration of differential equations. Instead, the intent is to encourage users of these methods to exercise proper caution so that confidence in the reliability of results is fully justified.

Many important physical processes are governed by differential equations. Typical cases include dynamics of rigid and flexible bodies, heat conduction, and electrical current flow. Solving a system of differential equations subject to known initial conditions allows us to predict the future behavior of the related physical system. Since very few important differential equations can be solved in closed form, approximations which are directly or indirectly founded on series expansion methods have been developed. The basic problem addressed is that of accurately computing \( Y(t + h) \) when \( Y(t) \) is known, along with a differential equation governing system behavior from time \( t \) to \( (t + h) \). Recursive application of a satisfactory numerical approximation procedure, with possible adjustment of step-size to maintain accuracy and stability, allows approximate prediction of system response subsequent to the starting time.

Numerical methods for solving differential equations are important tools for analyzing engineering systems. Although valuable algorithms have been developed which facilitate construction of approximate solutions, all available methods are vulnerable to limitations inherent in the underlying approximation processes. The essence of the difficulty lies in the fact that, as long as a finite integration step-size is used, integration error occurs at each time step. These errors sometimes have an accumulative effect which grows exponentially and eventually destroys solution validity. To some extent, accuracy problems can be limited by regulating step-size to keep local error within a desired tolerance. Typically, decreasing an integration tolerance increases the time span over which a numerical solution is valid. However, high costs for supercomputer time to analyze large and complex systems sometimes preclude generation of long time histories which may be more expensive than is practically justifiable.
8.2 Runge-Kutta Methods and the ODE45 Integrator Provided in MATLAB

Formulation of one method to solve differential equations is discussed in this section. Suppose a function \( y(x) \) satisfies a differential equation of the form \( y'(x) = f(x, y) \), subject to \( y(x_0) = y_0 \), where \( f \) is a known differentiable function. We would like to compute an approximation of \( y(x_0 + h) \) which agrees with a Taylor’s series expansion up to a certain order of error. Hence,

\[
y(x_0 + h) = \tilde{y}(x_0, h) + O(h^{n+1})
\]

where \( O(h^{n+1}) \) denotes a quantity which decreases at least as fast as \( h^{n+1} \) for small \( h \). Taylor’s theorem allows us to write

\[
y(x_0 + h) = y(x_0) + y'(x_0)h + \frac{1}{2}y''(x_0)h^2 + O(h^3)
\]

\[
= y_0 + f(x_0, y_0)h + \frac{1}{2}[f_x(x_0, y_0) + f_y(x_0, y_0)f_0]h^2 + O(h^3)
\]

where \( f_0 = f(x_0, y_0) \). The last formula can be used to compute a second order approximation \( \tilde{y}(x_0 + h) \), provided the partial derivatives \( f_x \) and \( f_y \) can be evaluated. However, this may be quite difficult since the function \( f(x, y) \) may not even be known explicitly.

The idea leading to Runge-Kutta integration is to compute \( y(x_0 + h) \) by making several evaluations of function \( f \) instead of having to differentiate that function. Let us seek an approximation in the form

\[
\tilde{y}(x_0 + h) = y_0 + h[k_0 f_0 + k_1 f(x_0 + \alpha h, y_0 + \beta h f_0)].
\]

We choose \( k_0, k_1, \alpha, \) and \( \beta \) to make \( \tilde{y}(x_0 + h) \) match the series expansion of \( y(x) \) as well as possible. Since

\[
f(x_0 + \alpha h, y_0 + \beta h f_0) = f_0 + [f_x(x_0, y_0)\alpha + f_y(x_0, y_0)f_0\beta]h + O(h^2),
\]

we must have

\[
\tilde{y}(x_0 + h) = y_0 + h[(k_0 + k_1)f_0 + k_1(f_x(x_0, y_0)\alpha + f_y(x_0, y_0)\beta f_0)]h + O(h^2)
\]

\[
= y_0 + (k_0 + k_1)f_0 h + [f_x(x_0, y_0)\alpha k_1 + f_y(x_0, y_0)f_0\beta k_1]h^2 + O(h^3).
\]

The last relation shows that

\[
y(x_0 + h) = \tilde{y}(x_0 + h) + O(h^3)
\]

provided

\[
k_0 + k_1 = 1, \quad \alpha k_1 = \frac{1}{2}, \quad \beta k_1 = \frac{1}{2}.
\]
This system of three equations in four unknowns has an infinite number of solutions; one of these is \( k_0 = k_1 = \frac{1}{2}, \alpha = \beta = 1 \). This implies that

\[
y(x_0 + h) = y(x_0) + \frac{1}{2} [f_0 + f(x_0 + h, y_0 + hf_0)]h + O(h^3).
\]

Neglecting the truncation error \( O(h^3) \) gives a difference approximation known as Heun’s method [61], which is classified as a second order Runge-Kutta method. Reducing the step-size by \( h \) reduces the truncation error by about a factor of \( \left(\frac{1}{2}\right)^3 = \frac{1}{8} \). Of course, the formula can be used recursively to compute approximations to \( y(x_0 + h), y(x_0 + 2h), y(x_0 + 3h), \ldots \). In most instances, the solution accuracy decreases as the number of integration steps is increased and results eventually become unreliable. Decreasing \( h \) and taking more steps within a fixed time span helps, but this also has practical limits governed by computational time and arithmetic roundoff error.

The idea leading to Heun’s method can be extended further to develop higher order formulas. One of the best known is the fourth order Runge-Kutta method described as follows

\[
y(x_0 + h) = y(x_0) + h [k_1 + 2k_2 + 2k_3 + k_4]/6
\]

where

\[
k_1 = f(x_0, y_0), \quad k_2 = f(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}k_1),
\]

\[
k_3 = f(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}k_2), \quad k_4 = f(x_0 + h, y_0 + k_3h).
\]

The truncation error for this formula is order \( h^5 \); so, the error is reduced by about a factor of \( \frac{1}{16} \) when the step-size is halved. The development of the fourth order Runge-Kutta method is algebraically quite complicated [43]. We note that accuracy of order four is achieved with four evaluations of \( f \) for each integration step. This situation does not extend to higher orders. For instance, an eighth order formula may require twelve evaluations per step. This price of more function evaluations may be worthwhile provided the resulting truncation error is small enough to permit much larger integration steps than could be achieved with formulas of lower order. MATLAB provides the function \texttt{ode45} which uses variable step-size and employs formulas of order four and five. (Note: In MATLAB 6.x the integrators can output results for an arbitrary time vector using, for instance, even time increments.)

### 8.3 Step-size Limits Necessary to Maintain Numerical Stability

It can be shown that, for many numerical integration methods, taking too large a step-size produces absurdly large results that increase exponentially with successive
time steps. This phenomenon, known as numerical instability, can be illustrated with the simple differential equation

\[ y'(t) = f(t, y) = \lambda y \]

which has the solution \( y = ce^{\lambda t} \). If the real part of \( \lambda \) is positive, the solution becomes unbounded with increasing time. However, a pure imaginary \( \lambda \) produces a bounded oscillatory solution, whereas the solution decays exponentially for \( \text{real}(\lambda) < 0 \). Applying Heun’s method [43] gives

\[ y(t + h) = y(t) \left[ 1 + (\lambda h) + \frac{(\lambda h)^2}{2} \right] \]

This shows that at each integration step the next value of \( y \) is obtained by multiplying the previous value by a factor

\[ p = 1 + (\lambda h) + \frac{(\lambda h)^2}{2} \]

which agrees with the first three Taylor series terms of \( e^{\lambda h} \). Clearly, the difference relation leads to

\[ y_n = y_0 p^n \]

As \( n \) increases, \( y_n \) will approach infinity unless \( |p| \leq 1 \). This stability condition can be interpreted geometrically by regarding \( \lambda h \) as a complex variable \( z \) and solving for all values of \( z \) such that

\[ 1 + z + \frac{z^2}{2} = \zeta e^{i\theta}, \ |\zeta| \leq 1, \ 0 \leq \theta \leq 2\pi. \]

Taking \( \zeta = 1 \) identifies the boundary of the stability region, which is normally a closed curve lying in the left half of the complex plane. Of course, \( h \) is assumed to be positive and the real part of \( \lambda \) is nonpositive. Otherwise, even the exact solution would grow exponentially. For a given \( \lambda \), the step-size \( h \) must be taken small enough to make \( |\lambda h| \) lie within the stability zone. The larger \( |\lambda| \) is, the smaller \( h \) must be to prevent numerical instability.

The idea illustrated by Heun’s method can be easily extended to a Runge-Kutta method of arbitrary order. A Runge-Kutta method of order \( n \) reproduces the exact solution through terms of order \( n \) in the Taylor series expansion. The differential equation \( y' = \lambda y \) implies

\[ y(t + h) = y(t) e^{\lambda h} \]

and

\[ e^{\lambda h} = \sum_{k=0}^{n} \frac{(\lambda h)^k}{k!} + O(h^{n+1}). \]

Consequently, points on the boundary of the stability region for a Runge-Kutta method of order \( n \) are found by solving the polynomial

\[ 1 - e^{i\theta} + \sum_{k=1}^{n} \frac{z^k}{k!} = 0 \]

© 2003 by CRC Press LLC
for a dense set of \( \theta \)-values ranging from 0 to \( 2\pi \). Using MATLAB’s intrinsic function \texttt{roots} allows easy calculation of the polynomial roots which may be plotted to show the stability boundary. The following short program accomplishes the task. Program output for integrators of order four and six is shown in Figures 8.1 and 8.2. Note that the region for order 4 resembles a semicircle with radius close to 2.8. Using \(|\lambda h| > 2.8\), with Runge-Kutta of order 4, would give results which rapidly become unstable. The figures also show that the stability region for Runge-Kutta of order 6 extends farther out on the negative real axis than Runge-Kutta of order 4 does. The root finding process also introduces some meaningless stability zones in the right half plane which should be ignored.

Figure 8.1: Stability Zone for Explicit Integrator of Order 4
Figure 8.2: Stability Zone for Explicit Integrator of Order 6
MATLAB Example

Program rkdestab

```matlab
% Example: rkdestab
% ~~~~~~~~~~~~~~~~~~
% This program plots the boundary of the region
% of the complex plane governing the maximum
% step size which may be used for stability of
% a Runge-Kutta integrator of arbitrary order.

% npts - a value determining the number of
% points computed on the stability
% boundary of an explicit Runge-Kutta
% integrator.
% xrang - controls the square window within
% which the diagram is drawn.
% [ -3, 3, -3, 3] is appropriate for
% the fourth order integrator.

% User m functions required: none

hold off; clf; close;
fprintf('
STABILITY REGION FOR AN ');
fprintf('EXPLICIT RUNGE-KUTTA');
fprintf('
 INTEGRATOR OF ARBITRARY ');
fprintf('
 ORDER

');
while 1
disp(' ');
nordr=input('Give the integrator order ? > ');
if isempty(nordr) | nordr==0, break; end
% fprintf('
Input the number of points ');% npts=input('the boundary (100 is typical) ? > ');% npts=100;
r=zeros(npts,nordr); v=1./gamma(nordr+1:-1:2);
d=2*pi/(npts-1); i=sqrt(-1);
% Generate polynomial roots to define the
% stability boundary
% for j=1:npts
% polynomial coefficients
v(nordr+1)=1-exp(i*(j-1)*d);
```

© 2003 by CRC Press LLC
8.4 Discussion of Procedures to Maintain Accuracy by Varying Integration Step-size

When we solve a differential equation numerically, our first inclination is to seek output at even increments of the independent variable. However, this is not the most natural form of output appropriate to maintain integration accuracy. Whenever solution components are changing rapidly, a small time step may be needed, whereas using a small time step might be quite inefficient at times where the solution remains smooth. Most modern ODE programs employ variable step-size algorithms which decrease the integration step-size whenever some local error tolerance is violated and conversely increase the step-size when the increase can be performed without loss of accuracy. If results at even time increments are needed, these can be determined by interpolation of the non-equidistant values. The differential equation integrators provide the capability to output results at an arbitrary vector of times over the integration interval.

Although the derivation of algorithms to regulate step-size is an important topic, development of these methods is not presented here. Several references [43, 46, 51, 61] discuss this topic with adequate detail. The primary objective in regulating step-size is to gain computational efficiency by taking as large a step-size as possible while maintaining accuracy and minimizing the number of function evaluations.

Practical problems involving a single first order differential equation are rarely encountered. More commonly, a system of second order equations occurs which is then transformed into a system involving twice as many first order equations. Several hundred, or even several thousand dependent variables may be involved. Evaluating the necessary time derivatives at a single time step may require computationally in-

139x385 to 474x389
tensive tasks such as matrix inversion. Furthermore, performing this fundamental calculation several thousand times may be necessary in order to construct time responses over time intervals of practical interest. Integrating large systems of nonlinear differential equations is one of the most important and most resource intensive aspects of scientific computing.

Instead of deriving the algorithms used for step-size control in ode45, we will outline briefly the ideas employed to integrate \( y'(t) = f(t, y) \) from \( t \) to \( (t + h) \). It is helpful to think of \( y \) as a vector. For a given time step and \( y \) value, the program makes six evaluations of \( f \). These values allow evaluation of two Runge-Kutta formulas, each having different truncation errors. These formulas permit estimation of the actual truncation error and proper step-size adjustment to control accuracy. If the estimated error is too large, the step-size is decreased until the error tolerance is satisfied or an error condition occurs because the necessary step-size has fallen below a set limit. If the estimated error is found to be smaller than necessary, the integration result is accepted and the step-size is increased for the next pass. Even though this type of process may not be extremely interesting to discuss, it is nevertheless an essential part of any well designed program for integrating differential equations numerically. Readers should become familiar with the error control features employed by ODE solvers. Printing and studying the code for ode45 is worthwhile. Studying the convergence tolerance used in connection with function odeset is also instructive. It should be remembered that solutions generated with tools such as ode45 are vulnerable to accumulated errors from roundoff and arithmetic truncation. Such errors usually render unreliable the results obtained sufficiently far from the starting time.

This chapter concludes with the analysis of several realistic nonlinear problems having certain properties of their exact solutions known. These known properties are compared with numerical results to assess error growth. The first problem involves an inverted pendulum for which the loading function produces a simple exact displacement function. Examples concerning top dynamics, a projectile trajectory, and a falling chain are presented.

### 8.5 Example on Forced Oscillations of an Inverted Pendulum

The inverted pendulum in Figure 8.3 involves a weightless rigid rod of length \( l \) which has a mass \( m \) attached to the end. Attached to the mass is a spring with stiffness constant \( k \) and an unstretched length of \( \gamma l \). The spring has length \( l \) when the pendulum is in the vertical position. Externally applied loads consist of a driving moment \( M(t) \), the particle weight, and a viscous damping moment \( cl^2 \dot{\theta} \). The differential equation governing the motion of this system is

\[
\ddot{\theta} = -(c/m)\dot{\theta} + (g/l) \sin(\theta) + M(t)/(ml^2) - (2k/m) \sin(\theta)(1 - \alpha/\lambda)
\]
Figure 8.3: Forced Vibration of an Inverted Pendulum
where
\[ \lambda = \sqrt{5 - 4 \cos(\theta)}. \]

This system can be changed to a more convenient form by introducing dimensionless variables. We let \( t = (\sqrt{l/g})\tau \) where \( \tau \) is dimensionless time. Then
\[
\ddot{\theta} = -\alpha \dot{\theta} + \sin(\theta) + P(\tau) - \beta \sin(\theta)(1 - \gamma/\lambda)
\]

where
\[
\alpha = (c/m)\sqrt{l/g} = \text{viscous damping factor},
\beta = 2(k/m)/(g/l),
\lambda = \sqrt{5 - 4 \cos(\theta)},
\gamma = (\text{unstretched spring length})/l,
P(\tau) = M/(mgl) = \text{dimensionless driving moment}.
\]

It is interesting to test how well a numerical method can reconstruct a known exact solution for a nonlinear function. Let us assume that the driving moment \( M(\tau) \) produces a motion having the equation
\[ \theta_e(\tau) = \theta_0 \sin(\omega \tau) \]
for arbitrary \( \theta_0 \) and \( \omega \). Then
\[ \dot{\theta}_e(\tau) = \omega \theta_0 \cos(\omega \tau) \]
and
\[ \ddot{\theta}_e(\tau) = -\omega^2 \theta_e. \]

Consequently, the necessary driving moment is
\[ P(\tau) = -\omega^2 \theta_e - \sin(\theta_e) + \gamma \omega \theta_0 \cos(\omega \tau) + \beta \sin(\theta_e) \left[ 1 - \gamma/\sqrt{5 - 4 \cos(\theta_e)} \right]. \]

Applying this forcing function, along with the initial conditions \( \theta(0) = 0, \dot{\theta}(0) = \theta_0 \omega \)
should return the solution \( \theta = \theta_e(\tau) \). For a specific numerical example we choose \( \theta_0 = \pi/8, \omega = 0.5 \), and four different combinations of \( \beta, \gamma \), and \text{tol} \. The second order differential equation has the form \( \ddot{\theta} = f(\tau, \theta, \dot{\theta}) \). This is expressed as a first order matrix system by letting \( y_1 = \theta, y_2 = \dot{\theta} \), which gives
\[ \dot{y}_1 = y_2, \quad \dot{y}_2 = f(\tau, y_1, y_2). \]

A function describing the system for solution by \texttt{ode45} is provided at the end of this section. Parameters \( \theta_0, \omega_0, \alpha, \zeta \), and \( \beta \) are passed as global variables.
We can examine how well the numerically integrated $\theta$ match $\theta_e$ by using the error measure
\[ |\theta(\tau) - \theta_e(\tau)|. \]
Furthermore, the exact solution satisfies
\[ \theta_e^2 + (\dot{\theta}_e/\omega)^2 = \theta_0^2. \]
Plotting $\dot{\theta}/(\theta_0\omega)$ on a horizontal axis and $\theta/\theta_0$ on a vertical axis should produce a unit circle. Violation of that condition signals loss of solution accuracy.

How certain physical parameters and numerical tolerances affect terms in this problem can be demonstrated by the following four data cases:

1. The spring is soft and initially unstretched. A liberal integration tolerance is used.
2. The spring is soft and initially unstretched. A stringent integration tolerance is used.
3. The spring is stiff and initially stretched. A liberal integration tolerance is used.
4. The spring is stiff and initially stretched. A stringent integration tolerance is used.

The curves in Figure 8.4 show the following facts:

1. When the spring is unstretched initially, the numerical solution goes unstable quickly.
2. Stretching the spring initially and increasing the spring constant improves numerical stability of the solution.
3. Decreasing the integration tolerance increases the time period over which the solution is valid.

An additional curve illustrating the numerical inaccuracy of results for Case 1 appears in Figure 8.5. A plot of $\theta(\tau)$ versus $\dot{\theta}(\tau)/\omega$ should produce a circle. However, solution points quickly depart from the desired locus.
Figure 8.4: Error Growth in Numerical Solution
Figure 8.5: $\theta$ versus ($\theta'(\tau)/\omega$) for Case One
MATLAB Example

Program prun

function prun
% Example: prun
% ~~~~~~~~~~~~~
% Dynamics of an inverted pendulum integrated
% by use of ode45.
%
% User m functions required: pinvert, mom

global ncal
th0=pi/8; w=.5; tmax=30; ncal=0;

fprintf('
FORCED OSCILLATION OF AN ');
fprintf('INVERTED PENDULUM
');
fprintf('
Note: Generating four sets of
');
fprintf('numerical results takes a while.
');

% loose spring with liberal tolerance
alp=0.1; bet=1.0; gam=1.0; tol=1.e-4;
a1=num2str(alp); b1=num2str(bet);
g1=num2str(gam); e1=num2str(tol);
options=odeset('RelTol',tol);
[t1,z1]= ... ode45(@pinvert,[0,tmax],[0;w*th0],...
options,alp,bet,gam,th0,w);

% loose spring with stringent tolerance
alp=0.1; bet=1.0; gam=1.0; tol=1.e-10;
a2=num2str(alp); b2=num2str(bet);
g2=num2str(gam); e2=num2str(tol);
options=odeset('RelTol',tol);
[t2,z2]= ... ode45(@pinvert,[0,tmax],[0;w*th0],...
options,alp,bet,gam,th0,w);

% tight spring with liberal tolerance
alp=0.1; bet=4.0; gam=0.5; tol=1.e-4;
a3=num2str(alp); b3=num2str(bet);
g3=num2str(gam); e3=num2str(tol);
41: options=odeset('RelTol',tol);
42: [t3,z3]= ode45(@(pinvert,[0,tmax],[0;w*th0],...  
43:     options,alp,bet,gam,th0,w);
44: n3=ncal; ncal=0;
45:  
46: % tight spring with stringent tolerance
47: alp=0.1; bet=4.0; gam=0.5; tol=1.e-10;
48: a4=num2str(alp); b4=num2str(bet);
49: g4=num2str(gam); e4=num2str(tol);
50: options=odeset('RelTol',tol);
51: [t4,z4]= ode45(@(pinvert,[0,tmax],[0;w*th0],...  
52:     options,alp,bet,gam,th0,w);
53: n4=ncal; ncal=0; save pinvert.mat;
54:  
55: % Plot results
56: clf; semilogy(  
57:     t1,abs(z1(:,1)/th0-sin(w*t1)),'-r',...  
58:     t2,abs(z2(:,1)/th0-sin(w*t2)),'--g',...  
59:     t3,abs(z3(:,1)/th0-sin(w*t3)),'-.b',...  
60:     t4,abs(z4(:,1)/th0-sin(w*t4)),':m');
61: title('Error Growth in Numerical Solution')
62: xlabel('dimensionless time');
63: ylabel('error measure');
64: c1=['Case 1: alp=',a1,', bet=',b1,', gam=',g1,', tol=',e1];
65: c2=['Case 2: alp=',a2,', bet=',b2,', gam=',g2,', tol=',e2];
66: c3=['Case 3: alp=',a3,', bet=',b3,', gam=',g3,', tol=',e3];
67: c4=['Case 4: alp=',a4,', bet=',b4,', gam=',g4,', tol=',e4];
68: legend(c1,c2,c3,c4,4); shg
69: dum=input('Press [Enter] to continue','s');
70:  
71: % plot a phase diagram for case 1
72: clf; plot(z1(:,:2)/w,z1(:,:1));
73: axis('square'); axis([-1,1,-1,1]);
74: xlabel('	heta''(\tau)/\omega'); ylabel('	heta');
75: title(['\theta versus ( \theta''(\tau) / \omega ) for Case One']); figure(gcf);
76: %print -deps ccrclplt
77: disp(''); disp('All Done');
%function zdot=pinvert(t,z,alp,bet,gam,th0,w)
%  
% zdot=pinvert(t,z,alp,bet,gam,th0,w)
%  
% Equation of motion for the pendulum
%  
% t - time value
% z - vector [theta ; thetadot]
% alp,bet,gam,th0,w
%     - physical parameters in the
differential equation
% zdot - time derivative of z
%  
% User m functions called: mom
%  
%global ncal
ncal=ncal+1; th=z(1); thd=z(2);
c=cos(th); s=sin(th); lam=sqrt(5-4*c);
zdot=[thd; mom(t,alp,bet,gam,th0,w)+
  s-alp*thd-bet*s*(1-gam/lam)];

%function me=mom(t,alp,bet,gam,th0,w)
%  
% me=mom(t,alp,bet,gam,th0,w)
%  
% t - time
% alp,bet,gam,th0,w
%     - physical parameters in the
differential equation
% me - driving moment needed to produce
%     exact solution
%  
% User m functions called: none.
%  
th=th0*sin(w*t);
tha=w*th0*cos(w*t); thdd=-th*w^2;
s=sin(th); c=cos(th); lam=sqrt(5-4*c);
me=thdd-s+alp*tha+bet*s*(1-gam/lam);
8.6 Dynamics of a Spinning Top

The dynamics of a symmetrical spinning top can be analyzed simply by computing the path followed by the gravity center in Cartesian coordinates. Consider a top spinning with its apex (or tip) constrained to remain at the origin. The gravity center lies at position \( r \) along the axis of symmetry and the only applied forces are the weight \(-mgk\) through the gravity center and the support reaction at the tip of the top. The inertial properties involve a moment of inertia \( J_a \) about the symmetry axis and a transverse inertial moment \( J_t \) relative to an axis normal to the symmetry axis and passing through the apex of the top. The velocity of the gravity center and the angular velocity \( \Omega \) are related by

\[
\mathbf{v} = \dot{\mathbf{r}} = \mathbf{\Omega} \times \mathbf{r}.
\]

This implies that \( \mathbf{\Omega} \) can be expressed in terms of radial and transverse components as

\[
\mathbf{\Omega} = \ell^{-2} \mathbf{r} \times \mathbf{v} + \ell^{-1} \omega_a \mathbf{r}
\]

where \( \ell = |\mathbf{r}| \) and \( \omega_a \) is the magnitude of the angular velocity component in the radial direction. The angular momentum with respect to the origin is therefore

\[
\mathbf{H} = J_t \ell^{-2} \mathbf{r} \times \mathbf{v} + J_a \ell^{-1} \omega_a \mathbf{r}
\]

and the potential plus kinetic energy is given by

\[
K = mgz + \frac{1}{2} \left( J_t \ell^{-2} \mathbf{v} \cdot \mathbf{v} + J_a \omega_a^2 \right)
\]

where \( z \) is the height of the gravity center above the origin.

The equations of motion can be found using the principle that the moment of all applied forces about the origin must equal the time rate of change of the corresponding angular momentum. Hence

\[
\mathbf{M} = J_t \ell^{-2} \mathbf{r} \times \mathbf{a} + J_a \ell^{-1} \left[ \omega_a \mathbf{v} + \dot{\omega}_a \mathbf{r} \right]
\]

where \( \mathbf{a} = \dot{\mathbf{v}} = \ddot{\mathbf{r}} \) is the total acceleration of the gravity center. The radial component of the last equation is obtainable by a dot product with \( \mathbf{r} \) to give

\[
\mathbf{r} \cdot \mathbf{M} = J_a \ell \dot{\omega}_a
\]

where simplifications result because \( \mathbf{r} \cdot (\mathbf{r} \times \mathbf{a}) = 0 \) and \( \mathbf{r} \cdot \mathbf{v} = 0 \). The remaining components of \( \mathbf{M} \) for the transverse direction result by taking \( \mathbf{r} \times \mathbf{M} \) and noting that

\[
\mathbf{r} \times (\mathbf{r} \times \mathbf{a}) = (\mathbf{r} \cdot \mathbf{a}) \mathbf{r} - \ell^2 \mathbf{a} = -\ell^2 \mathbf{a}_t
\]

\(^{1}\)In this section the quantities \( \mathbf{v}, \mathbf{r}, \mathbf{\Omega}, \mathbf{H}, \mathbf{a}, \) and \( a \) all represent vector quantities.

© 2003 by CRC Press LLC
where \( a_t \) is the vector component of total acceleration normal to the direction of \( r \). This leads to
\[
\mathbf{r} \times \mathbf{M} = -J_t \mathbf{a}_t + J_a \ell^{-1} \mathbf{\omega}_a \mathbf{r} \times \mathbf{v}.
\]
Since the gravity center moves on a spherical surface of radius \( \ell \) centered at the origin, the radial acceleration is given by
\[
\mathbf{a}_r = -\mathbf{v} \cdot \mathbf{v} \ell^{-2} \mathbf{r}
\]
and the total acceleration equation becomes
\[
\mathbf{a} = \frac{\mathbf{r} \times \mathbf{M}}{J_t} + \frac{J_a \ell^{-1} \mathbf{\omega}_a}{J_t} \mathbf{r} \times \mathbf{v} - \mathbf{v} \cdot \mathbf{v} \ell^{-2} \mathbf{r}.
\]
In the case studied here, only the body weight \(-mg \hat{k}\) causes a moment about the origin so
\[
\mathbf{M} = -mg \mathbf{r} \times \hat{k}, \quad \mathbf{r} \cdot \mathbf{M} = 0
\]
and
\[
\mathbf{r} \times \mathbf{M} = -mg \left[ z \mathbf{r} - \ell^2 \hat{k} \right].
\]
The radial component of the moment equation simply gives \( \dot{\mathbf{\omega}}_a = 0 \), so the axial component of angular velocity retains its initial value throughout the motion.

Integrating the differential equations
\[
\dot{\mathbf{v}} = \mathbf{a}, \quad \dot{\mathbf{r}} = \mathbf{v}
\]
numerically subject to appropriate initial conditions produces a trajectory of the gravity center motion. The simple formulation presented here treats \( x, y, \) and \( z \) as if they were independent variables even though
\[
x^2 + y^2 + z^2 = \ell^2, \quad xv_x + yv_y + zv_z = 0
\]
are implied. The type of analysis traditionally used in advanced dynamics books [48] would employ Euler angles, thereby assuring exact satisfaction of \( |\mathbf{r}| = \ell \). The accuracy of the solution method proposed here can be checked by finding a) whether the total energy of the system remains constant and b) whether the component of angular momentum in the \( z \)-direction remains constant. However, even when constraint conditions are satisfied exactly, reliability of numerical simulations of nonlinear systems over long time periods becomes questionable due to accumulated inaccuracies caused by arithmetic roundoff and the approximate nature of integration formulas.

The program \texttt{toprun} integrates the equations of motion and interprets the results. This program reads data to specify properties of a conical top along with the initial position and the angular velocity. Intrinsic function \texttt{ode45} is employed to integrate the motion equation defined in function \texttt{topde}. The path followed by the gravity center is plotted and error measures regarding conservation of energy and angular momentum are computed. Figures 8.6 and 8.7 show results for a top having properties given by the test case suggested in the interactive data input. A top which
has its symmetry axis initially horizontal along the $y$-axis is given an angular velocity of $[0, 10, 2]$. Integrating the equation of motion with an error tolerance of $10^{-8}$ leads to the response shown in the Figure 8.6. Error measures computed regarding the fluctuation in predicted values of total energy and angular momentum about the $z$-axis (Figure 8.7) fluctuate about one part in 100,000. It appears that the analysis employing Cartesian coordinates does produce good results.
Figure 8.7: Variation in Total Energy and $z$-axis Angular Momentum
Program Output and Code

Program Toprun

```matlab
function toprun
% Example: toprun
% -----------------
% Example that analyzes the response of a
% spinning conical top.
% % User m functions required:
% topde, cubrange, inputv

disp(' '); disp(['*** Dynamics of a Homogeneous ', ... 'Conical Top ***']); disp(' '); disp(['Input the gravity constant and the ', ... 'body weight (try 32.2,5)']); [grav,wt]=inputv('? '); mass=wt/grav; tmp=zeros(3,1); disp(' '); disp(['Input the height and base radius ', ... '(try 1,.5)']); [ht,rb]=inputv('? '); len=.75*ht; jtrans=3*mass/20*(rb*rb+4*ht*ht); jaxial=3*mass*rb*rb/10; disp(' '); disp(['Input a vector along the initial ', ... 'axis direction (try 0,1,0)']); [tmp(1),tmp(2),tmp(3)]=inputv('? '); e3=tmp(:)/norm(tmp); r0=len*e3; disp(' '); disp(['Input the initial angular velocity ', ... '(try 0,10,2)']); [tmp(1),tmp(2),tmp(3)]=inputv('? '); omega0=tmp; omegax=e3'*omega0(:); rdot0=cross(omega0,r0); z0=[r0(:);rdot0(:)]; uz=[0;0;1]; c1=wt*len^2/jtrans; c2=omegax*jaxial/jtrans; disp(' '); disp(['Input tfinal,and the integration ', ... 'tolerance (try 4.2, 1e-8)']); [tfinl,tol]=inputv('? '); disp(' '); fprintf( ...
```

© 2003 by CRC Press LLC
'Please wait for solution of equations.\n');

% Integrate the equations of motion
odeoptn=odeset('RelTol',tol);
[tout,zout]=ode45(@topde,[0,tfinl],z0,...
    odeoptn,uz,c1,c2);
t=tout; x=zout(:,1); y=zout(:,2); z=zout(:,3);
vx=zout(:,4); vy=zout(:,5); vz=zout(:,6);

% Compute total energy and angular momentum
c3=jtrans/(len*len); taxial=jaxial/2*omegax^2;
r=zout(:,1:3)'; v=zout(:,4:6)';
etotal=wt*r(3,:)'+taxial+c3/2*sum(v.*v)';
h=(jaxial*omegax/len*r+c3*cross(r,v))';

% Plot the path of the gravity center
clf; axis('equal');
axis(cubrange([x(:),y(:),z(:)])); plot3(x,y,z);
title('Path of the Top Gravity Center');
xlabel('x axis'); ylabel('y axis');
zlabel('z axis'); grid on; figure(gcf);

disp(' '); disp(...
    'Press [Enter] to plot error measures'), pause

% print -deps toppath

n=2:length(t);

% Compute energy and angular momentum error
et=etotal(1); enrger=abs(100*(etotal(n)-et)/et);
hzs=abs(h(1,3));
angmzer=abs(100*(h(n,3)-hzs)/hzs);
vec=[enrger(:);angmzer(:)];
minv=min(vec); maxv=max(vec);

clf;
semilogy(t(n),enrger,'-r',t(n),angmzer,':m');
axis('normal'); xlabel('time');
ylabel('percent variation');
title(['Percent Variation in Total Energy ', ...
    'and z-axis Angular Momentum']);
legend(' Energy (Upper Curve)', ...
    ' Ang. Mom. (Bottom Curve)',4);
figure(gcf), pause

% print -deps topvar
disp(' '), disp('All Done')

%=============================================  
% function zdot=topde(t,z,uz,c1,c2)  
% % zdot=topde(t,z,uz,c1,c2)  
% % This function defines the equation of motion  
% % for a symmetrical top. The vector z equals  
% % [r(:);v(:)] which contains the Cartesian  
% % components of the gravity center radius and  
% % its velocity.  
% % t - the time variable  
% z - the vector [x; y; z; vx; vy; vz]  
% uz - the vector [0;0;1]  
% c1 - wt*len^2/jtrans  
% c2 - omegax*jaxial/jtrans  
% zdot - the time derivative of z  
% % User m functions called: none  
% %---------------------------------------------  
% z=z(:); r=z(1:3); len=norm(r); ur=r/len;  
% % Make certain the input velocity is  
% % perpendicular to r  
% v=z(4:6); v=v-(ur'*v)*ur;  
% vdot=-c1*(uz-ur*ur(3))+c2*cross(ur,v)- ...  
% ((v'*v)/len)*ur;  
% zdot=[v;vdot];  
% %=============================================  
% % function varargout=inputv(prompt)  
% % See Appendix B  
% %=============================================  
% % function range=cubrange(xyz,ovrsiz)  
% % See Appendix B

© 2003 by CRC Press LLC
8.7 Motion of a Projectile

The problem of aiming a projectile to strike a distant target involves integrating a system of differential equations governing the motion and adjusting the initial inclination angle to achieve the desired hit [101]. A reasonable model for the projectile motion assumes atmospheric drag proportional to the square of the velocity. Consequently, the equations of motion are

\[ \dot{v}_x = -cv_x, \quad \dot{v}_y = -g - cv_y, \quad \dot{x} = v_x, \quad \dot{y} = v_y \]

where \( g \) is the gravity constant and \( c \) is a ballistic coefficient depending on such physical properties as the projectile shape and air density.

The natural independent variable in the equations of motion is time. However, horizontal position \( x \) is a more desirable independent variable, since the target will be located at some distant point \((x_f, y_f)\) relative to the initial position \((0, 0)\) where the projectile is launched. We can formulate the differential equations in terms of \( x \) by using the relationship

\[ \frac{dx}{v_x} = dt \quad \text{or} \quad \frac{dt}{dx} = \frac{1}{v_x}. \]

Then

\[ \frac{dy}{dx} = \frac{v_y}{v_x}, \quad \frac{dv_y}{dx} = v_x \frac{dv_y}{dt} = v_x \frac{dv_y}{dx}, \quad \frac{dv_x}{dx} = -cv, \quad \frac{dv_y}{dx} = -\left( g + cvv_y \right). \]

and the equations of motion become

\[ \frac{dy}{dx} = \frac{v_y}{v_x}, \quad \frac{dt}{dx} = \frac{1}{v_x}, \quad \frac{dv_x}{dx} = -cv, \quad \frac{dv_y}{dx} = -\left( g + cvv_y \right). \]

Taking a vector \( z \) defined by

\[ z = [v_x; v_y; y; t] \]

leads to a first order matrix differential equation

\[ \frac{dz}{dx} = \left[ \begin{array}{c} -cvv_x; \\ -(g + cvv_y); \\ v_y; \\ 1 \end{array} \right] \]

where

\[ v = \sqrt{v_x^2 + v_y^2}. \]

The reader should note that an ill-posed problem can occur if the initial velocity of the projectile is not large enough so that the maximum desired value of \( x \) is reached before \( v_x \) is reduced to zero from atmospheric drag. Consequently, error checking is needed to handle such a circumstance. The functions \texttt{tract} and \texttt{projcteq} employ intrinsic function \texttt{ode45} to compute the projectile trajectory. Graphical results
produced by the default data case appear in Figure 8.8. The function `traject` will be employed again in Chapter 12 for an optimization problem where a search procedure is used to compute the initial inclination angle needed to hit a target at some specified distant position. In this section we simply provide the functions to integrate the equations of motion.

**Program Output and Code**

**Function traject**

```plaintext
function [y,x,t]=traject ...

% [y,x,t]=traject ...
% (angle,vinit,gravty,cdrag,xfinl,noplot)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function integrates the dynamical equations for a projectile subjected to
% gravity loading and atmospheric drag proportional to the square of the velocity.
```
% angle - initial inclination of the projectile in degrees
% vinit - initial velocity of the projectile (muzzle velocity)
% gravity - the gravitational constant
% cdrag - drag coefficient specifying the drag force per unit mass which equals cdrag*velocity^2.
% xfinl - the projectile is fired toward the right from x=0. xfinl is the largest x value for which the solution is computed. The initial velocity must be large enough that atmospheric damping does not reduce the horizontal velocity to zero before xfinl is reached. Otherwise an error termination will occur.
% noplot - plotting of the trajectory is omitted when this parameter is given an input value
% y,x,t - the y, x and time vectors produced by integrating the equations of motion

% Global variables:
% grav, - two constants replicating gravity and cdrag, for use in function projcteq
% vtol - equal to vinit/1e6, used in projcteq to check whether the horizontal velocity has been reduced to zero

% User m functions called: projcteq
global grav dragc vtol

% Default data case generated when input is null
if nargin ==0
    angle=45; vinit=600; gravity=32.2;
    cdrag=0.002; xfinl=1000;
end;

% Assign global variables and evaluate
% initial velocity
grav=gravity; dragc=cdrag; ang=pi/180*angle;
vtol=vinit/1e6;
z0=[vinit*cos(ang); vinit*sin(ang); 0; 0];

% Integrate the equations of motion defined
% in function projcteq
deoptn=odeset('RelTol',1e-6);
[x,z]=ode45(@projcteq,[0,xfinl],z0,deoptn);

% Plot the trajectory curve
if nargin < 6
    plot(x,y,'-',xf,yf,'o');
    xlabel('x axis'); ylabel('y axis');
    title(['Projectile Trajectory for ', ...
          'Velocity Squared Drag']);
    axis('equal'); grid on; figure(gcf);
    % print -deps trajplot
end

%=============================================
% function zp=projcteq(x,z)
% %
% % This function defines the equation of motion
% % for a projectile loaded by gravity and
% % atmospheric drag proportional to the square
% % of the velocity.
% % x - the horizontal spatial variable
% % z - a vector containing [vx; vy; y; t];
% % zp - the derivative dz/dx which equals
% % [vx'(x); vy'(x); y'(x); t'(x)];
% % Global variables:
% % grav - the gravity constant
% % dragc - the drag coefficient divided by

% % Integrate the equations of motion defined
% % in function projcteq
% deoptn=odeset('RelTol',1e-6);
% [x,z]=ode45(@projcteq,[0,xfinl],z0,deoptn);
% % Plot the trajectory curve
% if nargin < 6
%     plot(x,y,'-',xf,yf,'o');
%     xlabel('x axis'); ylabel('y axis');
%     title(['Projectile Trajectory for ', ...
%           'Velocity Squared Drag']);
%     axis('equal'); grid on; figure(gcf);
%     % print -deps trajplot
% end

%=============================================
% function zp=projcteq(x,z)
% %
% % This function defines the equation of motion
% % for a projectile loaded by gravity and
% % atmospheric drag proportional to the square
% % of the velocity.
% % x - the horizontal spatial variable
% % z - a vector containing [vx; vy; y; t];
% % zp - the derivative dz/dx which equals
% % [vx'(x); vy'(x); y'(x); t'(x)];
% % Global variables:
% % grav - the gravity constant
% % dragc - the drag coefficient divided by
% gravity
% vtol - a global variable used to check whether vx is zero
% 
% User m functions called: none

%----------------------------------------------

global grav dragc vtol

vx=z(1); vy=z(2); v=sqrt(vx^2+vy^2);

% Check to see whether drag reduced the horizontal velocity to zero before the xfinl was reached.
if abs(vx) < vtol
    disp(' ');
    disp('*************************************');
    disp('ERROR in function projcteq. The initial velocity of the projectile was not large enough for xfinal to be reached.');
    disp('EXECUTION IS TERMINATED. ');
    disp('*************************************');
    disp(''),error(' ');
end

zp=[-dragc*v; -(grav+dragc*v*vy)/vx; vy/vx; 1/vx];

8.8 Example on Dynamics of a Chain with Specified End Motion

The dynamics of flexible cables is often modeled using a chain of rigid links connected by frictionless joints. A chain having specified end motions illustrates the behavior of a system governed by nonlinear equations of motion and auxiliary algebraic constraints. In particular, we will study a gravity loaded cable fixed at both ends. The total cable length exceeds the distance between supports, so that the static deflection configuration resembles a catenary.

A simple derivation of the equations of motion employing principles of rigid body dynamics is given next. Readers not versed in principles of rigid body dynamics [48] may nevertheless understand the subsequent programs by analyzing the equations of motion which have a concise mathematical form. The numerical solutions vividly illustrate some numerical difficulties typically encountered in multibody dynamical studies. Such problems are both computationally intensive, as well as highly sensitive to accumulated effects of numerical error.

The mathematical model of interest is the two-dimensional motion of a cable (or chain) having \( n \) rigid links connected by frictionless joints. A typical link \( i \) has its
mass $m_i$ concentrated at one end. The geometry is depicted in Figure 8.9. The chain ends undergo specified motions $R_0(t) = [X_0(t) : Y_0(t)]$ for the first link and $R_n(t) = [X_n(t) : Y_n(t)]$ for the last link. The direction vector along link $i$ is described by $r_i = [x_i ; y_i] = \ell_i[\cos(\theta_i) ; \sin(\theta_i)]$. We assume that each joint $i$ is subjected to a force $F_i = [f_xi ; f_yi]$ where $0 \leq i \leq n$. Index values $i = 0$ and $i = n$ denote unknown constraint forces which must act at the outer ends of the first and last links to achieve the required end displacements. The forces applied at the interior joints are arbitrary. It is convenient to characterize the dynamics of each link in terms of its direction angle. Thus

$$
\dot{r}_i = r_i' \dot{\theta}_i, \quad \ddot{r}_i = r_i'' \dot{\theta}_i^2 + r_i' \ddot{\theta}_i - r_i \dot{\theta}_i^2
$$

where primes and dots denote differentiation with respect to $\theta_i$ and $t$, respectively. Therefore

$$
\dot{r}_i = [-y_i ; x_i] \dot{\theta}_i, \quad \ddot{r}_i = [-\ddot{y}_i ; x_i] \dot{\theta}_i - [x_i ; y_i] \dot{\theta}_i^2.
$$

The global position vector of joint $i$ is

$$
R_i = R_0 + \sum_{j=1}^{i} r_j = R_0 + \sum_{j=1}^{n} <i-j>r_j
$$

where the symbol $< k > = 1$ for $k \geq 0$, and 0 for $k < 0$. Consequently, the velocity and acceleration of joint $i$ are

$$
\dot{R}_i = \dot{R}_0 + \sum_{j=1}^{n} <i-j>r_j' \dot{\theta}_j,
$$

$$
\ddot{R}_i = \ddot{R}_0 + \sum_{j=1}^{n} <i-j>r_j'' \dot{\theta}_j - \sum_{j=1}^{n} <i-j>r_j \dot{\theta}_j^2.
$$
The ends of the chain each have specified motions; so not all of the inclination angles are independent. Consequently,

$$\sum_{j=1}^{n} r_j = R_n - R_0,$$

$$\sum_{j=1}^{n} r'_j \dot{\theta}_j = \dot{R}_n - \dot{R}_0,$$

$$\sum_{j=1}^{n} r''_j - \sum_{j=1}^{n} r'_j \ddot{\theta}_j = \ddot{R}_n - \ddot{R}_0.$$ 

Combining the last constraint equation with equations of motion written for masses $m_1, \cdots, m_n$ yields a complete system of $(n + 2)$ equations determining $\dot{\theta}_1, \cdots, \dot{\theta}_n$ and the components of $F_n$. The fact that all masses are concentrated at frictionless joints shows that link $i$ is a two-force member carrying an internal load directed along $r_i$. Consequently, the D’Alembert principle [48] implies that the sum of all external and inertial loads from joints $i, i + 1, \cdots, n$ must give a resultant passing through joint $i$ in the direction of $r_i$. Since $r'_i$ and $r_i$ are perpendicular, requiring a vector to be in the direction of $r_i$ is equivalent to making it normal to $r'_i$. Therefore

$$r'_i \cdot \left( \sum_{j=1}^{n} \left< j - i > \left\{ F_j - m_j \dot{R}_j \right\} \right) = 0, \ 1 \leq i \leq n.$$ 

The last $n$ equations involve $\dot{\theta}_i$ and two end force components $f_{xi}$ and $f_{yi}$. Some algebraic rearrangement results in a matrix differential equation of concise form containing several auxiliary coefficients defined as follows:

$$b_i = \sum_{k=1}^{n} m_k, \quad m_{ij} = m_{ji} = b_i, \ 1 \leq i \leq n, \ 1 \leq j \leq i,$$

$$a_{ij} = m_{ij} (x_i x_j + y_i y_j), \ 1 \leq i \leq n, \ 1 \leq j \leq n$$

$$b_{ij} = m_{ij} (x_i y_j - x_j y_i), \ 1 \leq i \leq n, \ 1 \leq j \leq n$$

$$p_{xi} = \sum_{j=i}^{n-1} f_{xi}, \quad p_{yi} = \sum_{j=i}^{n-1} f_{yi}, \ 1 \leq i \leq n.$$ 

For $i = n$, the last two sums mean $p_{xn} = p_{yn} = 0$. Furthermore, we denote the acceleration components of the chain ends as $\dot{R}_0 = [a_{x0} \ a_{y0}]$ and $\ddot{R}_n = [a_{xn} \ a_{yn}]$.

Using the various quantities just defined, the equations of motion become

$$\sum_{j=1}^{n} a_{ij} \dot{\theta}_j + y_i f_{xn} - x_i f_{yn} = \sum_{j=1}^{n} b_{ij} \ddot{\theta}_j + x_i (p_{yi} - b_i a_{y0}) - y_i (p_{xi} - b_i a_{x0})$$

$$= e_i, \ 1 \leq i \leq n.$$
The remaining two components of the constraint equations completing the system are
\[
\sum_{j=1}^{n} y_j \ddot{y}_j = -\sum_{j=1}^{n} x_j \ddot{x}_j - a_{xn} + a_{x0} = e_{n+1},
\]
\[
\sum_{j=1}^{n} x_j \ddot{x}_j = \sum_{j=1}^{n} y_j \ddot{y}_j + a_{yn} - a_{y0} = e_{n+2}.
\]
Consequently, we get the following symmetric matrix equation to solve for \(\ddot{\theta}_1, \ldots, \ddot{\theta}_n, f_{xn}\) and \(f_{yn}\)
\[
\begin{bmatrix}
A & X & Y \\
X^T & 0 & 0 \\
Y^T & 0 & 0
\end{bmatrix}
\begin{bmatrix}
\ddot{\theta} \\
f_{xn} \\
-f_{yn}
\end{bmatrix}
= \begin{bmatrix} E \end{bmatrix}
\]
where \(X, Y, E\) and \(\theta\) are column matrices, and the matrix \(A = [a_{ij}]\) is symmetric. Because most numerical integrators for differential equations solve first order systems, it is convenient to employ the vector \(Z = [\theta ; \dot{\theta}]\) having \(2n\) components. Then the differential equation \(Z' = H(t, Z)\) is completely defined when \(\dot{\theta}\) has been computed for known \(Z\). The system is integrated numerically to give \(\theta\) and \(\dot{\theta}\) as functions of time. These quantities can then be used to compute the global Cartesian coordinates of the link configurations, thereby completely describing the time history of the chain.

The general equations of motion simplify somewhat when the chain ends are fixed and the external forces only involve gravity loads. Then \(p_{x1} = 0\) and \(p_{yn} = -g(b_i - b_n)\) which gives
\[
\sum_{j=1}^{n} m_{ij} (x_i x_j + y_i y_j) \ddot{y}_j - x_i f_{yn} + y_i f_{xn} =
\]
\[
g(b_i - b_n) + \sum_{j=1}^{n} m_{ij} (x_i y_j - x_j y_i) \ddot{y}_j^2, \quad 1 \leq i \leq n.
\]
The last two equations to complete the set are:
\[
\sum_{j=1}^{n} x_j \ddot{x}_j = \sum_{j=1}^{n} y_j \ddot{y}_j, \quad \sum_{j=1}^{n} y_j \ddot{y}_j = -\sum_{j=1}^{n} x_j \ddot{x}_j.
\]
A program was written to simulate motion of a cable fixed at both ends and released from rest. The cable falls under the influence of gravity from an initially elevated position. Function \texttt{ode45} is used to perform the numerical integration. The program consists of three functions \texttt{cablenl}, \texttt{plotmotn}, and \texttt{equamo}. Function \texttt{cablenl} creates the data, calls \texttt{ode45} to perform the integration, and displays the output from the simulation. Function \texttt{plotmotn} plots the motion for specified time limits. Results can be shown using animation or plots superimposing successive positions of the cable. Most of the analysis in the program is performed in function \texttt{equamo} which forms the equations of motion which are passed to \texttt{ode45} for integration.
A configuration with eight identical links was specified. For simplicity, the total mass, total cable length, and gravity constant were all normalized to equal unity. The numerical integration error was controlled using a relative tolerance of 1e-6 and an absolute error tolerance of 1e-8. Results of the simulation appear below. Figure 8.10 shows cable positions during the early stages of motion when results of the numerical integration are reliable. However, the numerical solution eventually becomes worthless due to accumulated numerical inaccuracies yielding the motion predictions indicated in Figure 8.11. The nature of the error growth can be seen clearly in Figure 8.12 which plots the x-coordinate of the chain midpoint as a function of time. Since the chosen mass distribution and initial deflection is symmetrical about the middle, the subsequent motion will remain symmetrical unless the numerical solution becomes invalid. The x coordinate of the midpoint should remain at a constant value of $\sqrt{2}/4$, but it appears to abruptly go unstable near $t = 18$. More careful examination indicates that this numerical instability does not actually occur suddenly. Instead, it grows exponentially from the outset of the simulation. The error is caused by the accumulation of truncation errors intrinsic to the numerical integration process allowing errors at each step which are regulated within a small but finite tolerance. A global measure of symmetry loss of the y deflection pattern is plotted on a semilog scale in Figure 8.13. Note that the error curve has a nearly linear slope until the solution degenerates completely near $t = 18$. The reader can verify that choosing less stringent error tolerances produces solutions which become inaccurate sooner than $t = 18$. It should also be observed that this dynamical model exhibits another important characteristic of highly nonlinear systems, namely, extreme sensitivity to physical properties. Note that shortening the last link by only one part in ten thousand makes the system deflection quickly lose all appearance of symmetry by $t = 6$. Hence, two systems having nearly identical physical parameters and initial conditions may behave very differently a short time after motion is initiated. The conclusion implied is that analysts should thoroughly explore how parameter variations affect response predictions in nonlinear models.
INTEGRATION OF NONLINEAR INITIAL VALUE PROBLEMS

Figure 8.10: Motion During Initial Phase

Figure 8.11: Motion After Solution Degenerates

© 2003 by CRC Press LLC
Figure 8.12: Horizontal Position of the Cable Midpoint

Figure 8.13: Growing Loss of Symmetry in Vertical Deflection
Example on Nonlinear Cable Motion

Program cablenl

function [t,x,y,theta,cptim]=cablenl
% [t,x,y,theta,cptim]=cablenl
% Example: cablenl
% ~~~~~~~~~~~~~~~~
% Numerical integration of the matrix
% differential equations for the nonlinear
% dynamics of a cable of rigid links with
% the outer ends of the cable fixed.
% % t - time vector for the solution
% x,y - matrices with nodal coordinates
% stored in the columns. The time
% history of point j is in x(:,j)
% and y(:,j)
% theta - matrix with inclination angles
% stored in the columns
% cptim - number of seconds to integrate
% the equations of motion
% % User m functions required:
% % plotmotn, equamo

clear all; close;

% Make variables global for use by
% function equamo
global first_ n_ m_ len_ grav_ b_ mas_ py_

fprintf('
NONLINEAR DYNAMICS OF A FALLING CABLE
')
fprintf('
Note: The calculations take awhile
')

% Set up data for a cable of n_ links,
% initially arranged in a triangular
% deflection configuration.

% parameter controlling initialization of
% auxiliary parameters used in function
% equamo

© 2003 by CRC Press LLC
first_ = 1;
\% number of links in the cable
n_ = 8; n = n_; nh = n_ / 2;
\% vector of lengths and gravity constant
len_ = 1/n*ones(n,1); grav_ = 1;
\% vector of mass constants
m_ = ones(1,n_)/n_; 
\% initial position angles
th0 = pi/4*[ones(nh,1); -ones(nh,1)];
\% time limits, integration tolerances, 
\% and the number of solution points
tmin = 0; tmax = 25; nt = 201; 
t = linspace(0,tmax,nt)';
tolrel = 1e-6; tolabs = 1e-8; len = len_;
\% Perform the numerical integration using a 
\% variable stepsize Runge-Kutta integrator 
tic; 
\{t,w\} = ode45(@equamo,t,z0,odetol); 
\theta = w(:,1:n_); ctim = toc; 
\% Compute node point coordinates 
Z = [zeros(nt,1),repmat(len',nt,1).*exp(i*\theta)]; 
Z = cumsum(Z.'); ' ; x = real(Z); y = imag(Z); 
\% Plot the horizontal position of the midpoint 
cf; plot(t,x(:,1+n_/2)); 
ylabel(‘x coordinate’); xlabel(‘time’)
title(['Horizontal Position of the ' ... 
\‘Cable Midpoint’]) 
grid on; figure(gcf); 
\% print -deps xmidl 
disp(’ ‘), disp(... 
'Press [Enter] to see the error growth curve’); 
\pause, close 
\% Show error growth indicated by symmetry 
\% loss of the vertical deflection symmetry. 
\% An approximately linear trend on the semilog 
\% plot indicates exponential growth of the error.
unsymer=sqrt(sum((y-y(:,end:-1:1)).^2,2));
hold off; axis('normal'); clf;
semilogy(t,unsymer);
xlabel('time'); ylabel('asymmetry error');
title(['Growing Loss of Symmetry in ' ... 
' Vertical Deflection']);
grid on; figure(gcf);

% print -deps unsymerr

disp(' '), disp(...
'Press [Enter] to see the response animation');

% Show animation of the cable response

disp(' ') disp('The motion can be animated or a trace')
disp('can be shown for successive positions')
disp(['between t = ',num2str(tmin),...
' and t = ',num2str(tmax)])

% Plot the position for different times limits
titl='CABLE MOTION FOR T = ';
while 1

disp(' '), disp(...
['Choose a plot option (1 <=> animate, ',...
' 2 <=> trace,)']

opt=input('3 <=> stop) > ? '); if opt==3, break, end

disp(' '), disp(...
'Give a time vector such as 0:.1:15')

Tp=input('Time vector > ? ','s');
if isempty(Tp), break, end

tp=eval(Tp); tp=tp(:); T=[titl,Tp];

xp=interplq(t,x,tp); yp=interplq(t,y,tp);
if opt ==1, plotmotn(xp,yp,T)
else, plotmotn(xp,yp,T,1), end
end
fprintf('
All Done
')

%=============================================

function plotmotn(x,y,titl,isave)

% This function plots the cable time
% history described by coordinate values
% stored in the rows of matrices x and y.
%  
% x, y - matrices having successive rows
% which describe position
% configurations for the cable
% titl - a title shown on the plots
% isave - parameter controlling the form
% of output. When isave is not input,
% only one position at a time is shown
% in rapid succession to animate the
% motion. If isave is given a value,
% then successive are all shown at
% once to illustrate a kinematic
% trace of the motion history.
%
% User m functions called: none
%------------------------------------------------------------------
% Set a square window to contain all
% possible positions
[nt,n]=size(x);
if nargin==4, save =1; else, save=0; end
xmin=min(x(:)); xmax=max(x(:));
ymin=min(y(:)); ymax=max(y(:));
w=max(xmax-xmin,ymax-ymin)/2;
xmd=(xmin+xmax)/2; ymd=(ymin+ymax)/2;
hold off; clf; axis('normal'); axis('equal');
range=[xmd-w,xmd+w,ymd-w,ymd+w];
title(titl)
xlabel('x axis'); ylabel('y axis')
if save==0
  for j=1:nt
    xj=x(j,:); yj=y(j,:);
    plot(xj,yj,'-k',xj,yj,'ok');
    axis(range), axis off
    title(titl)
    figure(gcf), drawnow, pause(.1)
  end
  pause(2)
else
  hold off; close
  for j=1:nt
    xj=x(j,:); yj=y(j,:);
    plot(xj,yj,'-k',xj,yj,'ok');
axis(range), axis off, hold on
end
title(titl)
figure(gcf), drawnow, hold off, pause(2)
end

% Save plot history for subsequent printing
% print -deps plotmotn
%=============================================

function zdot=equamo(t,z)

% zdot=equamo(t,z)
% ~~~~~~~~~~~~~~~~
% Equation of motion for a cable fixed at
% both ends and loaded by gravity forces only
%
% t    current time value
% z    column vector defined by
%       [thet(t);theta'(t)]
% zdot  column vector defined by
%       the concatenation
%       z'(t) = [theta'(t);theta''(t)]
%
% User m functions called: none.
%---------------------------------------------
% Values accessed as global variables
global first_ n_ m_ len_ grav_ b_ mas_ py_
% Initialize parameters first time
% function is called
if first_==1, first_=0;
% mass parameters
m_=m_(:); b_=flipud(cumsum(flipud(m_))); mas_=b_(:,ones(n_,1));
mas_=tril(mas_)+tril(mas_,-1)';
% load effects from gravity forces
py_=-grav_*betas=-b_ betas(n_));
end

% Solve for zdot = [theta'(t); theta''(t)];
% n=n_; len=len_; th=z(1:n); td=z(n+1:2*n); td2=td.*td;

© 2003 by CRC Press LLC
x = len.*cos(th); y = len.*sin(th);

% Matrix of mass coefficients and
% constraint conditions
amat = [[mas_.*((x'*y+y'*y), x, y];
        [x, y; zeros(2, 2)]'];

% Right side vector involves applied forces
% and inertial terms
bmat = x*y'; bmat = mas_.*(bmat-bmat');

% Solve for angular acceleration.
% Most computation occurs here.
soln = amat\[x.*py_+bmat*td2; y'*td2; -x'*td2];

% Final result needed for use by the
% numerical integrator
zdot = [td; soln(1:n)];

8.9 Dynamics of an Elastic Chain

The preceding article analyzed a chain of rigid links requiring only one rotation
angle per link. Next we study a similar model of an elastic chain involving sev-
eral point masses connected by elastic springs which can only support tension. The
equations of motion are easy to formulate in terms of the horizontal and vertical coor-
dinates of each mass. The dimensionality needed to handle the elastic chain is twice
that needed for a similar rigid link model. It is natural to utilize a three-dimensional
model that easily simplifies for two dimensional motion.

Consider a chain having \( n \) mass particles

\[ m_j, \ 1 \leq j \leq n \]

connected by \( n + 1 \) springs having unstretched lengths

\[ l_j, \ 1 \leq j \leq n + 1 \]

The position of particle \( m_j \) is denoted by vector \( r_j \) with \( r_0(t) \) and \( r_{n+1}(t) \) signi-
fying the outer end positions of the first and last springs, which are assumed to be
known functions of time. Furthermore, concentrated forces \( P_j(t) \) are applied to the
particles. The tensile force in spring number \( j \) is

\[ T_j = k_j (1 - l_j/L_j) (L_j > l_j) R_j \]

where

\[ R_j = r_{j+1} - r_j, \ L_j = |R_j|, \]
and \( k_j \) denotes a spring constant. Then the equations of motion are given by

\[
\dot{r}_j(t) = v_j, \quad \dot{v}_j(t) = \left( P_j + T_j - T_{j-1} - c_j v_j \right) / m_j, \quad 1 \leq j \leq n
\]

where viscous drag forces defined by the particle velocities times damping coefficients \( c_j \) are included. These equations are easy to form using array operations. Furthermore, the two-dimensional case can be simplified further by using complex numbers to represent the particle positions.

A program was written to compute the response of a chain released from at rest in a horizontal position with the springs unstretched. The chain is subjected to gravity loading and the ends of the chain are rotated at constant speed around circular paths. The left and right ends rotate counterclockwise and clockwise respectively. A special case where the right end of the chain is free is provided by setting the last spring constant to zero. Another case where the chain ends do not move occurs when the radii of the end path motions are set to zero.

The following program called \texttt{sprnchan} computes the response of a chain with an arbitrary number of identical masses connected by identical springs. The radii and the rotation rate of the end motions, as well as the amount of viscous damping can be changed easily. Function \texttt{sprnchan} reads data from function \texttt{chaindata} and calls \texttt{ode45} to integrate the equations of motion which are formed with functions \texttt{spreqmof} and \texttt{endmo}. Using the output from \texttt{ode45}, function \texttt{plotmotn} provides visual descriptions of the response. The motion can be presented using either animation or by superimposing plots of successive positions of the chain in chosen time intervals. To run a different problem, the sample data function \texttt{chaindata} can be saved using a different name; and the variables \( n, t_{\text{max}}, n_t, \text{xorfree}, \text{rend}, \omega, \text{and cdamp} \) can be changed appropriately. Furthermore, modifying the program to handle different variations of stiffness and mass, as well as different end conditions would be straightforward. Figures 8.14 and 8.15 show program results where 1) the left end of the chain was rotated and the right end was detached and 2) both ends of the chain were rotated simultaneously in opposite directions. The time response was computed for a maximum time value of 20, but the chosen time traces only show small subintervals chosen so that successive positions do not overlap excessively. Readers may find it interesting to observe the animation responses resulting from different data choices.
ELASTIC CHAIN MOTION FOR $[\text{cdamp}, \omega] = [1, 6]$ and $T = 0.08:1.5$

Figure 8.14: Chain with Left End Rotating and Right End Free

$[\text{cdamp}, \omega] = [1, 6], 0 < t < 0.6$

Figure 8.15: Chain with Both Ends Rotating
function [t,z,cptim]=sprnchan
% [t,z,cptim]=sprnchan
% -----------------------
% DYNAMIC SIMULATION OF AN ELASTIC CHAIN
% This program simulates the dynamics of an elastic chain modeled by a series of mass particles joined by elastic springs. The outer springs at each end are connected to foundations moving on circular paths at constant speed. The system is released from rest in a horizontal position. Forces on the system include gravity, linear viscous drag, and foundation motion. If the last spring in the chain is assigned zero stiffness, then the last particle is freed from the foundation and a swinging chain with the upper end shaken is analyzed. The principal variables for the problem are listed below. Different data choices can be made by changing function chaindata.
% tlim - vector of time values at which the solution is computed
% m - vector of mass values for the particles
% k - vector of stiffness values for springs connecting the particles. If the last spring constant is set to zero, then the right end constraint is removed
% L - vector of unstretched spring lengths
% zend - complex position coordinate of the outer end of the last spring (assuming the outer end of the first spring is held at z=0)
% zinit - vector of complex initial displacement values for each mass particle. Initial velocity values are zero.
% fext - vector of constant complex force components applied to the individual masses
% c - vector of damping coefficients specifying drag on each particle linearly proportional to the particle velocity
% tolrel - relative error tolerance for function ode45
% tolatex - absolute error tolerance for function ode45
% t - vector of times returned by ode45
% z - matrix of complex position and velocity
% values returned by ode45. A typical row
% z(j,:) gives the system position and
% velocity for time t(j). The first half of
% the row contains complex position values
% and the last half contains velocity values
% omega - frequency at which the ends of the chain
% are shaken
% yend - amplitude of the vertical motion of the
% chain ends. If this is set to zero then
% the chain ends do not move
% endmo - the function defining the end motion of
% the chain
% spreqmof - the function defining the equation of
% motion to be integrated using ode45
% User m functions called: chaindata, spreqmof,
% endmo, plotmotn
%----------------------------------------------

global zend omega Rend

fprintf('
DYNAMICS OF A FALLING ELASTIC CHAIN

')
disp('Give a file name to define the data. Try')
datname=input('chaindata as an example > ? ','s');
eval(['[n,tmax,nt,fixorfree,rend,omega,cdamp]=',datname,']]);

% The following data values are scaled in terms of
% the parameters returned by the data input function

% Time vector for solution output
tmin=0; tlim=linspace(tmin,tmax,nt);

% Number of masses, gravity constant, mass vector
g=32.2; len0=1; mas=1/g; m=mas*ones(n,1);

% Spring lengths and spring constants
L=len0*ones(n+1,1); ksp=5*mas*g*(n+1)/(2*len0);
k=ksp*ones(n+1,1);

% If the far end of the chain is free, then the
% last spring constant is set equal to zero
k(n+1)=fixorfree*k(n+1);
% Viscous damping coefficients
c = cdamp*sqrt(mas*ksp)/40*ones(n,1);

% Chain end position and initial position of each mass. Parameters concerning the end positions are passed as global variables.
% global zend omega Rend
zend = len0*(n+1); zinit = cumsum(L(1:n));
Rend = rend*zend;

% Function name giving end position of the chain re=@endmo;

% Gravity forces and integration tolerance
fext = -i*g*m; tolrel = 1e-6; tolabs = 1e-8;

% Initial conditions for the ode45 integrator
n = length(m); r0 = [zinit;zeros(n,1)];

% Integrate equations of motion
options = odeset('reltol',tolrel,'abstol',tolabs);
fprintf('
Please Wait While the Equations
')
fprintf('of Motion Are Being Integrated
')
pause(1), tic;

[t,r]=ode45(@spreqmof,tlim,r0,options,...
m,k,L,re,fext,c);

cptim = toc; cpt = num2str(fix(10*cptim)/10);
fprintf(...
['
Computation time was ',cpt,' seconds
'])

% Extract displacement history and add end positions
R = endmo(t); z = [R(:,1),r(:,1:n)];
if k(n+1)~=0, z = [z,R(:,2)]; end
X = real(z); Y = imag(z);

% Show animation or motion trace of the response.
% disp('Press [Enter] to continue'), pause
disp('...'

disp('The motion can be animated or a trace')
disp('can be shown for successive positions')
disp(['between t = ',num2str(tmin),...
function [n,tmax,nt,fixorfree,rend,omega,...
cdamp]=chaindata
% This example function creates data defining
% the chain. The function can be renamed and
% modified to handle different problems.

n=8; % Number or point masses
tmax=20; % Maximum time for the solution
nt=401; % Number of time values from 0 to tmax
fixorfree=0; % Determines whether the right end
% position is controlled or free. Use
% zero for free or one for controlled.
rend=0.05; % Amplitude factor for end motion. This can be zero if the ends are fixed.
omega=6; % Frequency at which the ends are rotated.
cdamp=1; % Coefficient regulating the amount of viscous damping. Reduce cdamp to give less damping.

function rdot=spreqmof(t,r,m,k,L,re,fext,c)
% This function forms the two-dimensional equation of motion for a chain of spring-connected particles. The positions of the ends of the chain may be time dependent and are computed from a function named in the input parameter re. The applied external loading consists of constant loads on the particles and linear viscous damping proportional to the particle velocities. Data parameters for the problem are defined in a function file specified by the user. Function chaindata gives a typical example.

N=length(r); n=N/2; z=r(1:n); v=r(n+1:N);
R=feval(re,t);
zdif=diff([R(1);z;R(2)]); len=abs(zdif);
fsp=zdif./len.*((len-L).*(len-L>0)).*k; fdamp=-c.*v;
accel=(fext+fdamp+fsp(2:n+1)-fsp(1:n))./m;
rdot = [v; accel];

%=======================================

function rends = endmo(t)

% rends = endmo(t)
% ~~~~~~~~~~~~~
% This function specifies the varying end positions.
% In this example the ends rotate at frequency omega
% around circles of radius Rend.
% %
% User m functions called: none
% %----------------------------------------------

global zend Rend omega

s = Rend * exp(i * omega * t); rends = [s, zend - conj(s)];

%=============================================
Chapter 9

Boundary Value Problems for Partial Differential Equations

9.1 Several Important Partial Differential Equations

Many physical phenomena are characterized by linear partial differential equations. Such equations are attractive to study because (a) principles of superposition apply in the sense that linear combinations of component solutions can often be used to build more general solutions and (b) finite difference or finite element approximations lead to systems of linear equations amenable to solution by matrix methods. The accompanying table lists several frequently encountered equations and some applications. We only show one- or two-dimensional forms, although some of these equations have relevant applications in three dimensions.

In most practical applications the differential equations must be solved within a finite region of space while simultaneously prescribing boundary conditions on the function and its derivatives. Furthermore, initial conditions may exist. In dealing with the initial value problem, we are trying to predict future system behavior when initial conditions, boundary conditions, and a governing physical process are known. Solutions to such problems are seldom obtainable in a closed finite form. Even when series solutions are developed, an infinite number of terms may be needed to provide generality. For example, the problem of transient heat conduction in a circular cylinder leads to an infinite series of Bessel functions employing characteristic values which can only be computed approximately. Hence, the notion of an “exact” solution expressed as an infinite series of transcendental functions is deceiving. At best, we can hope to produce results containing insignificantly small computation errors.

The present chapter applies eigenfunction series to solve nine problems. Examples involving the Laplace, wave, beam, and heat equations are given. Nonhomogeneous boundary conditions are dealt with in several instances. Animation is also provided whenever it is helpful to illustrate the nature of the solutions.
### 9.2 Solving the Laplace Equation inside a Rectangular Region

Functions which satisfy Laplace’s equation are encountered often in practice. Such functions are called harmonic; and the problem of determining a harmonic function subject to given boundary values is known as the Dirichlet problem [119]. In a few cases with simple geometries, the Dirichlet problem can be solved explicitly. One instance is a rectangular region with the boundary values of the function being expandable in a Fourier sine series. The following program employs the FFT to construct a solution for boundary values represented by piecewise linear interpolation. Surface and contour plots of the resulting field values are also presented.

The problem of interest satisfies the differential equation

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad , \quad 0 < x < a \quad , \quad 0 < y < b
\]

with the boundary conditions of the form

\[
\begin{align*}
  u(x, 0) &= F(x) \quad , \quad 0 < x < a \\
  u(x, b) &= G(x) \quad , \quad 0 < x < a \\
  u(0, y) &= P(y) \quad , \quad 0 < y < b \\
  u(a, y) &= Q(y) \quad , \quad 0 < y < b
\end{align*}
\]

The series solution can be represented as

\[
u(x, y) = \sum_{n=1}^{\infty} f_n a_n(x, y) + g_n a_n(x, b - y) + p_n b_n(x, y) + q_n b_n(a - x, y)\]
where

\[ a_n(x, y) = \sin \left( \frac{n\pi x}{a} \right) \sinh \left( \frac{n\pi (b - y)}{a} \right) / \sinh \left( \frac{n\pi b}{a} \right), \]

\[ b_n(x, y) = \sinh \left( \frac{n\pi (a - x)}{b} \right) \sin \left( \frac{n\pi y}{b} \right) / \sinh \left( \frac{n\pi a}{b} \right), \]

and the constants \( f_m, g_m, p_n, \) and \( q_n \) are coefficients in the Fourier sine expansions of the boundary value functions. This implies that

\[ F(x) = \sum_{n=1}^{\infty} f_n \sin \left( \frac{n\pi x}{a} \right), \quad G(x) = \sum_{n=1}^{\infty} g_n \sin \left( \frac{n\pi x}{a} \right), \]

\[ P(y) = \sum_{n=1}^{\infty} p_n \sin \left( \frac{n\pi y}{b} \right), \quad Q(y) = \sum_{n=1}^{\infty} q_n \sin \left( \frac{n\pi y}{b} \right). \]

The coefficients in the series can be computed by integration as

\[ f_n = \frac{2}{a} \int_0^a F(x) \sin \left( \frac{n\pi x}{a} \right) dx, \quad g_n = \frac{2}{a} \int_0^a G(x) \sin \left( \frac{n\pi x}{a} \right) dx, \]

\[ p_n = \frac{2}{a} \int_0^b P(y) \sin \left( \frac{n\pi y}{b} \right) dy, \quad q_n = \frac{2}{a} \int_0^b Q(y) \sin \left( \frac{n\pi y}{b} \right) dy, \]

or approximate coefficients can be obtained using the FFT. The latter approach is chosen here and the solution is evaluated for an arbitrary number of terms in the series.

The chosen problem solution has the disadvantage of employing eigenfunctions that vanish at the ends of the expansion intervals. Consequently, it is desirable to combine the series with an additional term allowing exact satisfaction of the corner conditions for cases where the boundary value functions for adjacent sides agree. This implies requirements such as \( F(a) = Q(0) \) and three other similar conditions. It is evident that the function

\[ u_p(x, y) = c_1 + c_2 x + c_3 y + c_4 xy \]

is harmonic and varies linearly along each side of the rectangle. Constants \( c_1, \ldots, c_4 \) can be computed to satisfy the corner values and the total solution is represented as \( u_p \) plus a series solution involving modified boundary conditions.

The following program \texttt{laplace} solves the Dirichlet problem for the rectangle. Function values and gradient components are computed and plotted. Functions used in this program are described below. The example data set defined in the driver program was chosen to produce interesting surface and contour plots. Different boundary conditions can be handled by slight modifications of the input data. In this example 100 term series are used. Figure 9.1 through Figure 9.4 show function and gradient components, as well as a contour plot of function values. Readers may find it instructive to run the program and view these figures from different angles.
laplacec inputs data, calls computation modules, and plots results
datafunc defines an example data case
tulinbc particular solution for linearly varying boundary conditions
recseris sums the series for function and gradient values
sincos generates coefficients in a Fourier sine series
lintrp piecewise linear interpolation function allowing jump discontinuities

using the interactive figure rotating capability provided in MATLAB. Note that the figure showing the function gradient in the $x$ direction used view([225,20]) to show clearly the jump discontinuity in this quantity.
Figure 9.1: Surface Plot of Function Values

Figure 9.2: Contour Plot of Function Values
DERIVATIVE OF $U(X,Y)$ IN THE X DIRECTION

**Figure 9.3:** Function Derivative in the $x$ Direction

DERIVATIVE OF $U(X,Y)$ IN THE Y DIRECTION

**Figure 9.4:** Function Derivative in the $y$ Direction

© 2003 by CRC Press LLC
MATLAB Example

Program laplarec

```matlab
function [u,ux,uy,X,Y]=laplarec(...
    ubot,utop,ulft,urht,a,b,nx,ny,N)
%
% [u,ux,uy,X,Y]=laplarec(...
%                      ubot,utop,ulft,urht,a,b,nx,ny,N)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This program evaluates a harmonic function and its
% first partial derivatives in a rectangular region.
% The method employs a Fourier series expansion.
% ubot - defines the boundary values on the bottom
% side. This can be an array in which
% ubot(:,1) is x coordinates and ubot(:,2)
% is function values. Values at intermediate
% points are obtained by piecewise linear
% interpolation. A character string giving
% the name of a function can also be used.
% Then the function is evaluated using 200
% points along a side to convert ubot to an
% array. Similar comments apply for utop,
% ulft, and urht introduced below.
% utop - boundary value definition on the top side
% ulft - boundary value definition on the left side
% urht - boundary value definition on the right side
% a,b - rectangle dimensions in x and y directions
% nx,ny - number of x and y values for which the
% solution is evaluated
% N - number of terms used in the Fourier series
% u - function value for the solution
% ux,uy - first partial derivatives of the solution
% X,Y - coordinate point arrays where the solution
% is evaluated
%
% User m functions used: datafunc ulinbc
% recseris ftsincof
%
disp('')
disp('SOLVING THE LAPLACE EQUATION IN A RECTANGLE')
disp('')
if nargin==0
```

© 2003 by CRC Press LLC
disp('Give the name of a function defining the data')
datafun=input(' (try datafunc as an example): > ?','s');
[ubot,utop,ulft,urht,a,b,nx,ny,N]=feval(datfun);
end

% Create a grid to evaluate the solution
x=linspace(0,a,nx); y=linspace(0,b,ny);
[X,Y]=meshgrid(x,y); d=(a+b)/1e6;
xd=linspace(0,a,201)'; yd=linspace(0,b,201)';

% Check whether boundary values are given using external functions. Convert these to arrays
if isstr(ubot)
    ud=feval(ubot,xd); ubot=[xd,ud(:)];
end
if isstr(utop)
    ud=feval(utop,xd); utop=[xd,ud(:)];
end
if isstr(ulft)
    ud=feval(ulft,yd); ulft=[yd,ud(:)];
end
if isstr(urht)
    ud=feval(urht,yd); urht=[yd,ud(:)];
end

% Determine function values at the corners
ub=interp1(ubot(:,1),ubot(:,2),[d,a-d]);
ut=interp1(utop(:,1),utop(:,2),[d,a-d]);
ul=interp1(ulft(:,1),ulft(:,2),[d,b-d]);
ur=interp1(urht(:,1),urht(:,2),[d,b-d]);
U=[ul(1)+ub(1),ub(2)+ur(1),ur(2)+ut(2),...
    ut(1)+ul(2)]/2;

% Obtain a solution satisfying the corner values and varying linearly along the sides
[v,vx,vy]=ulinbc(U,a,b,X,Y);

% Reduce the corner values to zero to improve behavior of the Fourier series solution
% near the corners
f=inline('u0+(u1-u0)/L*x','x','u0','u1','L');
ubot(:,2)=ubot(:,2)-f(ubot(:,1),U(1),U(2),a);
utop(:,2)=utop(:,2)-f(utop(:,1),U(4),U(3),a);
ulft(:,2)=ulft(:,2)-f(ulft(:,1),U(1),U(4),b);
urht(:,2)=urht(:,2)-f(urht(:,1),U(2),U(3),b);

% Evaluate the series and combine results for the various component solutions
[ub,ubx,uby]=recseris(ubot,a,b,1,x,y,N);
[ut,utx,uty]=recseris(utop,a,b,2,x,y,N);
[ul,ulx,uly]=recseris(ulft,a,b,3,x,y,N);
[ur,urx,ury]=recseris(urht,a,b,4,x,y,N);
u=v+ub+ut+ul+ur; ux=vx+ubx+utx+ulx+urx;
uy=vy+uby+uty+uly+ury; close

% Show results graphically
surf(X,Y,u), xlabel('x axis'), ylabel('y axis')
zlabel('U(X,Y)')
title('HARMONIC FUNCTION IN A RECTANGLE')
shg, pause
% print -deps laprecsr
contour(X,Y,u,30); title('Contour Plot');
xlabel('x direction'); ylabel('y direction');
colorbar, shg, pause
% print -deps laprecont
surf(X,Y,ux), xlabel('x axis'), ylabel('y axis')
zlabel('DU(X,Y)/DX')
title('DERIVATIVE OF U(X,Y) IN THE X DIRECTION')
shg, pause
% print -deps laprecdx
surf(X,Y,uy), xlabel('x axis'), ylabel('y axis')
zlabel('DU(X,Y)/DY')
title('DERIVATIVE OF U(X,Y) IN THE Y DIRECTION')
% print -deps laprecdy
shg

%============================================

function [ubot,utop,ulft,urht,a,b,...
    nx,ny,N]=datafunc
This is a sample data case which can be
modified to apply to other examples

ubot, utop - vectors of function values on the
bottom and top sides
ulft, urht - vectors of function values on the
right and left sides
a, b - rectangle dimensions along the
x and y axis
nx, ny - number of grid values for the x
and y directions
N - number of terms used in the
Fourier series solution

a=3; b=2; e=1e-5; N=100;
x=linspace(0,1,201)'; s=sin(pi*x);
c=cos(pi*x); ubot=[a*x,2-4*s];
up=interp1([0,1/3,2/3,1],
[-2,2,2,-2],x);
ulft=[b*x,2*c]; urht=ulft; nx=51; ny=31;

function [u,ux,uy]=ulinbc(U,a,b,X,Y)

U - corner values of the harmonic function
[U(1),...U(4)] => corner coordinates
(0,0), (0,a), (a,b), (0,b)
a,b - rectangle dimensions in the x and y
directions
X,Y - array coordinates where the solution
is evaluated
u - function values evaluated for X,Y
ux,uy - first derivative components evaluated
for the X,Y arrays
c=[1,0,0,0;1,a,0,0;1,a,b,a*b;1,0,b,0]\U(:,);

u=c(1)+c(2)*X+c(3)*Y+c(4)*X.*Y;
ux=c(2)+c(4)*Y; uy=c(3)+c(4)*X;

function [u,ux,uy,X,Y]=recseris(udat,a,b,iside,x,y,N)
%
% [u,ux,uy,X,Y]=recseris(udat,a,b,iside,x,y,N)
% %============================================
% This function computes a function harmonic in
% a rectangle with general function values given
% on one side and zero function values on the
% other three sides.
% udat - a data array to determine the function
% values by piecewise linear interpolation
% along the side having nonzero values.
% udat(:,1) contains either x or y values
% along a side, and udat(:,2) contains
% corresponding function values
% a,b - side lengths for the x and y directions
% iside - an index indicating the side for which
% function values are given.
% [1,2,3,4]<=>[bottom,top,left,right]
% x,y data vectors defining a grid
% [X,Y]=meshgrid(x,y) on which the function
% and its first partial derivatives are
% computed
% N - number of series terms used (up to 500)
% u,ux,uy - arrays of values of the harmonic function
% and its first partial derivatives
% X,Y arrays of coordinate values for which
% function values were computed.

x=x(:)'; y=y(:); ny=length(y); N=min(N,500);
if iside<3, period=2*a; else, period=2*b; end
c=ftsincof(udat,period); n=1:N; c=c(n);
if iside<3 % top or bottom sides
    npa=pi/a*n; c=c./(1-exp(-2*b*npa));
    sx=sin(npa(:)*x); cx=cos(npa(:)*x);
    if iside==1 % bottom side
        dy=exp(-y*npa); ey=exp(-(2*b-y)*npa);
        u=repmat(c,ny,1).*(dy-ey)*sx;
        c=repmat(c.*npa,ny,1);
ux=c.*(dy-ey)*cx; uy=-c.*((dy+ey)*sx;
else % top side
dy=exp((y-b)*npa); ey=exp(-(y+b)*npa);
u=repmat(c,ny,1).*(dy-ey)*sx;
c=repmat(c.*npa,ny,1);
ux=c.*(dy-ey)*cx; uy=c.*(dy+ey)*sx;
end
else % left or right sides
npb=pi/b*n; c=c./(1-exp(-2*a*npb));
sy=sin(y*npb); cy=cos(y*npb);
if iside==3 % left side
dx=exp(-npb(:)*x);
ex=exp(-npb(:)*(2*a-x));
u=repmat(c,ny,1).*sy*(dx-ex);
c=repmat(c.*npb,ny,1);
ux=c.*sy*(-dx-ex); uy=c.*cy*(dx-ex);
else % right side
dx=exp(-npb(:)*(a-x));
ex=exp(-npb(:)*(a+x));
u=repmat(c,ny,1).*sy*(dx-ex);
c=repmat(c.*npb,ny,1);
ux=c.*sy*(dx+ex); uy=c.*cy*(dx-ex);
end
end
[X,Y]=meshgrid(x,y);

function c=ftsincof(y,period)
% This function computes 500 Fourier sine coefficients for a piecewise linear function defined by a data array.
% y - an array defining the function over half a period as Y(x)=interp1(y(:,1),y(:,2),x)
% period - the period of the function
xft=linspace(0,period/2,513);
uxt=interp1(y(:,1),y(:,2)/512,xft);
c=fft([uxt,-uxt(512:-1:2)]);
c=-imag(c(2:501));
9.3 The Vibrating String

Transverse motion of a tightly stretched string illustrates one of the simplest occurrences of one-dimensional wave propagation. The transverse deflection satisfies the wave equation

\[ a^2 \frac{\partial^2 u}{\partial X^2} = \frac{\partial^2 u}{\partial T^2} \]

where \( u(X, T) \) satisfies initial conditions

\[ u(X, 0) = F(X), \quad \frac{\partial u(X, 0)}{\partial T} = G(X) \]

with boundary conditions

\[ u(0, T) = 0, \quad u(\ell, T) = 0 \]

where \( \ell \) is the string length. If we introduce the dimensionless variables \( x = X/\ell \) and \( t = T/(\ell/a) \) the differential equation becomes

\[ u_{xx} = u_{tt} \]

where subscripts denote partial differentiation. The boundary conditions become

\[ u(0, t) = u(1, t) = 0 \]

and the initial conditions become

\[ u(x, 0) = f(x), \quad u_t(x, 0) = g(x). \]

Let us consider the case where the string is released from rest initially so \( g(x) = 0 \). The solution can be found in series form as

\[ u(x, t) = \sum_{n=1}^{\infty} a_n p_n(x) \cos(\omega_n t) \]

where \( \omega_n \) are natural frequencies and satisfaction of the differential equation of motion requires

\[ p''_n(x) + \omega_n^2 p_n(x) = 0 \]

so

\[ p_n = A_n \sin(\omega_n x) + B_n \cos(\omega_n x). \]

The boundary condition \( p_n(0) = B_n = 0 \) and \( p_n(1) = A_n \sin(\omega_n) \) requires \( A_n \neq 0 \) and \( \omega_n = n\pi \), where \( n \) is an integer. This leads to a solution in the form

\[ u(x, t) = \sum_{n=1}^{\infty} a_n \sin(n\pi x) \cos(n\pi t). \]
The remaining condition on initial conditions requires
\[ \sum_{n=1}^{\infty} a_n \sin(n\pi x) = f(x), \quad 0 < x < 1. \]

Therefore, the coefficients \( a_n \) are obtainable from an odd-valued Fourier series expansion of \( f(x) \) vanishing at \( x = 0 \) and \( x = 2 \). We see that \( f(-x) = -f(x) \) and \( f(x + 2) = f(x) \), and the coefficients are obtainable by integration as
\[ a_n = 2 \int_{0}^{1} f(x) \sin(n\pi x) \, dx. \]

However, an easier way to compute the coefficients is to use the FFT. A solution will be given for an arbitrary piecewise linear initial condition.

Before implementing the Fourier series solution, let us digress briefly to examine the case of an infinite string governed by
\[ a^2 u_{XX} = u_{TT}, \quad -\infty < X < \infty \]
and initial conditions
\[ u(X, 0) = F(X), \quad u_T(X, 0) = G(X). \]

The reader can verify directly that the solution of this problem is given by
\[ u(X, T) = \frac{1}{2} [F(X - aT) + F(X + aT)] + \frac{1}{2a} \int_{X-aT}^{X+aT} G(x) \, dx. \]

When the string is released from rest, \( G(X) \) is zero and the solution reduces to
\[ \frac{F(X - aT) + F(X + aT)}{2} \]
which shows that the initial deflection splits into two parts with one half translating to the left at speed \( a \) and the other half moving to the right at speed \( a \). This solution can also be adapted to solve the problem for a string of length \( \ell \) fixed at each end. The condition \( u(0, T) = 0 \) implies
\[ F(-aT) = -F(aT) \]
which shows that \( F(X) \) must be odd valued. Similarly, \( u(\ell, T) = 0 \) requires
\[ F(\ell - aT) + F(\ell + aT) = 0. \]

Combining this condition with \( F(X) = -F(X) \) shows that
\[ F(X + 2\ell) = F(X) \]
so, \( F(X) \) must have a period of \( 2\ell \). In the string of length \( \ell \), \( F(X) \) is only known for \( 0 < X < \ell \), and we must take
\[
F(X) = -F(2\ell - X), \quad \ell < X < 2\ell.
\]
Furthermore the solution has the form
\[
u(X, T) = \frac{F(x_p) + F(x_m)}{2}
\]
where \( x_p = X + aT \) and \( x_m = X - aT \). The quantity \( x_p \) will always be positive and \( x_m \) can be both positive and negative. The necessary sign change and periodicity can be achieved by evaluating \( F(X) \) as
\[
\text{sign}(X) \cdot F(\text{rem}(|X|), 2 \ast \ell)
\]
where \( \text{rem} \) is the intrinsic remainder function used in the exact solution implemented in function \texttt{strngwav} presented earlier in section 2.7.

A computer program employing the Fourier series solution was written for an initial deflection that is piecewise linear. The series solution allows the user to select varying numbers of terms in the series to examine how well the initial deflection configuration is represented by a truncated sine series. A function animating the time response shows clearly how the initial deflection splits in two parts translating in opposite directions. In the Fourier solution, dimensionless variables are employed to make the string length and the wave speed both equal one. Consequently, the time required for the motion to exactly return to the starting position equals two, representing how long it takes for a disturbance to propagate from one end to the other and back. When the motion is observed for \( 0 < x < 1 \), it is evident that waves reflected from a wall are inverted. The program employs the following functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{stringft}</td>
<td>function to input initial deflection data</td>
</tr>
<tr>
<td>\texttt{sincos}</td>
<td>uses \texttt{fft} to generate coefficients in a sine series</td>
</tr>
<tr>
<td>\texttt{initdef}</td>
<td>defines the initial deflection by piecewise linear interpolation</td>
</tr>
<tr>
<td>\texttt{strvib}</td>
<td>evaluates the series solution for general ( x ) and ( t )</td>
</tr>
<tr>
<td>\texttt{smotion}</td>
<td>animates the string motion</td>
</tr>
<tr>
<td>\texttt{inputv}</td>
<td>facilitates interactive data input</td>
</tr>
<tr>
<td>\texttt{lintrp}</td>
<td>performs interpolation to evaluate a piecewise linear function</td>
</tr>
</tbody>
</table>

Results are shown below for a string which was deflected initially in a square wave. The example was chosen to illustrate the approximation produced when a small number of Fourier coefficients, in this case 30, is used. Ripples are clearly evident in the surface plot of \( u(x, t) \) in Figure 9.5. The deflection configuration of
the string at $t = 1$ when the initial deflection form has passed through half a period of motion appears in Figure 9.6. One other example given in Figure 9.7 shows the deflection surface produced using 100 series terms and a triangular initial deflection pattern. The surface describes $u(x, t)$ through one period of motion.
Wave Propagation in a String

String Deflection as a Function of Position and Time

Figure 9.6: Wave Propagation in a String

Figure 9.7: Surface for Triangular Initial Deflection
Program Output and Code

Output from Example stringft

>> stringft;

FOURIER SERIES SOLUTION FOR WAVES
IN A STRING WITH LINEARLY INTERPOLATED
INITIAL DEFLECTION AND FIXED ENDS

Enter the number of interior data points (the fixed end point coordinates are added automatically)
? 4

The string stretches between fixed endpoints at x=zero and x=one.

Enter 4 sets of x,y to specify interior initial deflections (one pair per line)

? .33,0
? .33,-1
? .67,-1
? .67,0

Give the number of series terms and the maximum value of t
(give 0,0 to stop)
? 30,1

Press [Enter] to see the animation

Give the number of series terms and the maximum value of t
(give 0,0 to stop)
? 0,0

>>

String Vibration Program

1  function [x,t,y]=stringft(Xdat,Ydat)
2   

© 2003 by CRC Press LLC
% Example: [x,t,y]=stringft(xdat,Ydat)
% ~~~~~~~~~~~~~~~
% This program analyzes wave motion in a string
% having arbitrary piecewise linear initial
% deflection. A Fourier series expansion is used
% to construct the solution
% Xdat,Ydat - data vectors defining the initial
deflections at interior points. The
deflections at x=0 and x=1 are set
to zero automatically. For example,
try Xdat=[.2,.3,.7,.8],
Ydat=[0,-1,-1,0]
% x,t,y - arrays containing the position, time
and deflection values
% User m functions required:
sincof, initdefl, strvib, smotion, inputv,
lintrp

global xdat ydat

disp(’ ’), disp(...
’FOURIER SERIES SOLUTION FOR WAVES’)
disp(...
’IN A STRING WITH LINEARLY INTERPOLATED’)
disp(...’
’INITIAL DEFLECTION AND FIXED ENDS’)
if nargin==0
 disp(’ ’)
disp([’Enter the number of interior ’,...
’data points (the fixed’)])
disp([’end point coordinates are ’,...
’added automatically’)])
n=input(’? ’); if isempty(n), break, end
xdat=zeros(n+2,1); ydat=xdat; xdat(n+2)=1;
disp(’ ’)
disp([’The string stretches between ’,...
’fixed endpoints at’])
disp(’x=zero and x=one.’),disp(’ ’)
disp([’Enter ’,num2str(n),... 
’sets of x,y to specify interior’])
disp([’initial deflections ’,...
’(one pair per line’)], disp(’ ’)
for j=2:n+1,[xdat(j),ydat(j)]=inputv; end;
else
    xdat=[0;Xdat(:);1]; ydat=[0;Ydat(:);0];
end

a=sincof(@initdefl,1,1024); % sine coefficients
nx=51; x=linspace(0,1,nx);
xx=linspace(0,1,151);
while 1
    disp('')
    disp('Give the number of series terms')
    disp('and the maximum value of t')
    disp('give 0,0 to stop')
    [ntrms,tmax]=inputv;
    if isnan(ntrms) | norm([ntrms,tmax])==0
        break, end
    nt=ceil(nx*tmax); t=linspace(0,tmax,nt);
    y=strvib(a,t,x,1,ntrms); % time history
    yy=strvib(a,t,xx,1,ntrms);
    [xo,to]=meshgrid(x,t);
    hold off; surf(xo,to,y);
    grid on; colormap([1 1 1]);
    xlabel('x axis'); ylabel('time axis');
    zlabel('transverse deflection');
    title(['String Deflection as a Function ', ...
       'of Position and Time']);
    disp(''), disp('Press [Enter] to')
    disp('see the animation'), shg, pause
    % print -deps strdefl
    smotion(xx,yy,'Wave Propagation in a String');
    disp(''); pause(1);
end
% print -deps strwave

function y=initdefl(x)
% y=initdefl(x)
% This function defines the linearly
% interpolated initial deflection
% configuration.
% x - a vector of points at which the initial
deflection is to be computed
% y - transverse initial deflection value for
  argument x
% xdat, ydat - global data vectors used for
  linear interpolation
% User m functions required: lintrp
%----------------------------------------------
global xdat ydat
y=lintrp(xdat,ydat,x);

%==================================================================
function y=strvib(a,t,x,hp,n)
% y=strvib(a,t,x,hp,n)
% Sum the Fourier series for the string motion.
% a - Fourier coefficients of initial
deflection
% t,x - vectors of time and position values
% hp - the half period for the series
% n - the number of series terms used
% y - matrix with y(i,j) equal to the
deflection at position x(i) and
time t(j)
% User m functions required: none
%----------------------------------------------
w=pi/hp*(1:n); a=a(1:n); a=a(:)';
x=x(:); t=t(:)';
y=((a(ones(length(x),1),:).* ...
function smotion(x,y,titl)
% smotion(x,y,titl)
% ~~~~~~~~~~~~~~~~~
% This function animates the string motion.
% x - a vector of position values along the string
% y - a matrix of transverse deflection values where successive rows give
deflections at successive times
% titl - a title shown on the plot (optional)
% User m functions required: none
%----------------------------------------------

if nargin < 3, titl=' '; end
xmin=min(x); xmax=max(x);
ymin=min(y(:)); ymax=max(y(:));
[nt,nx]=size(y); clf reset;
for j=1:nt
plot(x,y(j,:), 'k');
axis([xmin,xmax,2*ymin,2*ymax]);
axis('off'); title(titl);
drawnow; figure(gcf); pause(.1)
end

function a=sincof(func,hafper,nft)
% a=sincof(func,hafper,nft)
% ~~~~~~~~~~~~~~~~~~~~~~~
% This function calculates the sine coefficients.
% func - the name of a function defined over
% a half period
% hafper - the length of the half period of the
defunction
% nft - the number of function values used
% in the Fourier series
% a - the vector of Fourier sine series
% coefficients

© 2003 by CRC Press LLC
% User m functions required:  none
%-------------------------------------------------------------

n2=nft/2; x=haffper/n2*(0:n2);
y=feval(func,x); y=y(:);
a=fft([y;-y(n2:-1:2)]); a=-imag(a(2:n2))/n2;

%=============================================

% function y=lintrp(xd,yd,x)
% See Appendix B

%=============================================

% function varargout=inputv(prompt)
% See Appendix B

9.4 Force Moving on an Elastic String

The behavior of a semi-infinite string acted on by a moving transverse force illustrates an interesting aspect of wave propagation. Consider a taut string initially at rest and un-deflected when a force moving at constant speed is applied. This simple example shows how a wave front moves ahead of the force when the velocity of wave propagation in the string exceeds the speed of the force, but the force acts at the front of the disturbance when the force moves faster than the wave speed of the string. The governing differential equations, initial conditions, and boundary conditions are:

\[ a^2 u_{xx}(x, t) = u_{tt}(x, t) + \frac{F_0}{\rho} \delta(x - vt) , \quad t > 0 , \quad 0 < x < \infty, \]

\[ u(0, t) = 0 , \quad u(\infty, t) = 0, \]

\[ u(x, 0) = 0 , \quad u_t(x, 0) = 0 , \quad 0 < x < \infty. \]

In these equations \( a \) is the speed of wave propagation in the string and \( v \) is the speed at which a concentrated downward force \( F_0 \) moves toward the right along the string, \( \rho \) is the mass per unit length of the string, and \( \delta \) is the Dirac delta function. This problem can be solved using the Fourier sine transform pair defined by

\[ U(p, t) = \int_0^\infty u(x, t) \sin(px)dx , \quad u(x, t) = \frac{2}{\pi} \int_0^\infty U(p, t) \sin(px)dp. \]

© 2003 by CRC Press LLC
Transforming the differential equation and initial conditions, and making use of the boundary conditions gives

\[
-p^2 a^2 U(p, t) = U_{tt}(p, t) + \frac{F_0}{\rho} \sin(pvt), \quad U(p, 0) = 0, \quad U_t(p, 0) = 0.
\]

It follows that

\[
U(p, t) = \frac{F_0}{a \rho (a^2 - v^2)} \left[ \frac{v \sin(apt) - a \sin(vpt)}{p^2} \right],
\]

provided \( v \neq a \). Applying the inverse transformation then gives the desired displacement response as

\[
u(x, t) = \frac{F_0}{2a \rho (a^2 - v^2)} \left[ (v - a)x - v|x - at| + a|x - vt| \right].
\]

### 9.4.1 Computer Analysis

The following MATLAB program analyzes the response predicted by the last equation. A surface plot shows \( u(x, t) \). Positions of the force at successive times are also marked by a heavy dark line superimposed on the surface. Then an animation shows the string deflection and the point of action of the force throughout the chosen time interval. As the force moves along the string, no deflection occurs ahead of the force if the speed of the force exceeds the speed of wave propagation for the string. Otherwise, a disturbance propagates ahead of the force at the wave speed of the string. Graphical results from the program are shown first. Then the computer code is listed.

Let us first consider what happens when the force moves slower than the wave speed. Taking so \( v = 1.0, \ a = 1.2 \) gives the following results in Figure 9.8. Since the point of application of the load is denoted by an arrow, it is clear from the last figure that the disturbance moves ahead of the load when the load moves slower than the wave speed for the string. Next consider what happens when the force moves faster than the wave speed for the string. For example taking \( v = 1, \ a = 0.80 \) gives significantly different output. In this instance, no disturbance occurs at a point until the load passes the point. This case is illustrated in Figure 9.9. The reader may find it instructive to run the program for different combinations of force speed and wave speed. The program does not account for the case where \( v \) exactly equals \( a \), but these values can be taken close enough together to see what the limiting case will give. We simply increase \( a \) to 1.00001 times \( a \).
Figure 9.8: Force Moving Slower than the Wave Speed

Figure 9.9: Force Moving Faster than the Wave Speed
function [u,X,T,uf,t]=forcmove(a,v,tmax,nt)
%
% [u,X,T,uf,t]=forcmove(a,v,tmax,nt)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function computes the dynamic response
% of a taut string subjected to an upward
% directed concentrated force moving along the
% string at constant speed. The string is
% fixed at x=0 and x=+infinity. The system
% is initially at rest when the force starts
% moving toward the right from the left end. If
% the force speed exceeds the wave propagation
% speed, then no disturbance occurs ahead of
% the force. If the force speed is slower
% than the wave propagation speed, then the
% deflection propagates ahead of the force at
% the wave propagation speed.
%
% v - speed of the moving load
% a - speed of wave propagation in the
% string
% tmax - maximum time for which the
% solution is computed
% u - matrix of deflection values where
% time and position vary row-wise and
% column-wise, respectively
% T,X - matrices of time and position values
% corresponding to the deflection
% uf - deflection values where the force acts
% t - vector of times (same as columns of T)
%
% User m functions used: ustring
if nargin==0, a=.8; v=1; tmax=10; nt=15; end
if a>v
    titl='FORCE SPEED SLOWER THAN THE WAVE SPEED';
elseif a<v
    titl='FORCE SPEED FASTER THAN THE WAVE SPEED';
else

© 2003 by CRC Press LLC
titl='FORCE SPEED EQUAL TO THE WAVE SPEED';
a=v*1.00001;
end

% Obtain solution values and plot results
[u,X,T,uf,t]=ustring(a,v,tmax,nt);
if a>v, xf=X(:,2); uf=u(:,2); xw=X(:,3);
else, xf=X(:,3); uf=u(:,3); end
close, subplot(211)
waterfall(X,T,-u), xlabel('x axis')
ylabel('time'), zlabel('deflection')
title(titl), grid on, hold on
% plot3(xf,t,-uf,'.k',xf,t,-uf,'k')
plot3(xf,t,-uf,'k','linewidth',2);
colormap([0 0 0]), view([-10,30]), shg
umin=min(u(:)); umax=max(u(:)); xmax=X(1,4);
range=[0,xmax,2*umin,2*umax]; hold on
Titl=['A = ',num2str(a),', V = ',num2str(v),...
', ' T = %4.2f']; subplot(212) , axis off
% Use a dense set of points for animation
nt=80; [uu,XX,TT,uuf,tt]=ustring(a,v,tmax,nt);
ux=max(abs(uu(:))); uu=uu/ux; uuf=uuf/ux;
XX=XX/xmax; range=[0,1,-1,1]; h=.4;
arx=h*[0,.02,-.02,0,0]; ary=h*[0,.25,.25,0,1];
for j=1:nt
uj=uu(j,:); xj=XX(j,:);
xfj=v/xmax*tt(j); ufj=uuf(j);
plot(xj,-uj,'k',xfj+arx,-ufj-ary,'-k')
axis off, time=(sprintf(Titl,tt(j)));
text(.3,.5,time), axis(range), drawnow
pause(.05), figure(gcf), if j<nt, cla, end
end
% print -deps forcmove
hold off; subplot

%==============================================================
function [u,X,T,uf,t]=ustring(a,v,tmax,nt)
% [u,X,T,uf,t]=ustring(a,v,tmax,nt)
% This function computes the deflection u(x,t)
% of a semi-infinite string subjected to a
% moving force. The equation for the normalized
% deflection is
% \[ u(x,t) = \frac{1}{a} \left( \frac{1}{a^2 - v^2} \right) \left( (v-a-v*|x-a*t|) + a*v|v-a*t| \right); \]
% a - speed of wave propagation in the string
% v - speed of the force moving to the right
% tmax - maximum time for computing the solution
% nt - number of time values
% uu - array of displacement values normalized
% by dividing by a factor equal to the force
% magnitude over twice the density per unit
% length. Position varies column-wise and
% time varies row-wise in the array.
% X, T - position and time arrays for the solution
% uf - deflection vector under the force
% t - time vector for the solution (same as the
% columns of T)
% t=linspace(0,tmax,nt)'; xmax=1.05*tmax*max(a,v);
% u=zeros(nt,4); nx=4; X=zeros(nt,nx); X(:,nx)=xmax;
% c=1/a/(a^2-v^2); xw=a*t; xf=v*t; T=repmat(t,1,4);
% uw=c*xw*(v-a+abs(v-a)); uf=c*xf*(v-a-abs(v-a));
% if a>v, X(:,2)=xf; X(:,3)=xw; u(:,2)=uf;
% else, X(:,2)=xw; X(:,3)=xf; u(:,2)=uw; end

9.5 Waves in Rectangular or Circular Membranes

Wave propagation in two dimensions is illustrated well by the transverse vibration
of an elastic membrane. Membrane dynamics is discussed here for general boundary
shapes. Then specific solutions are given for rectangular and circular membranes
subjected to a harmonically varying surface force. In the next chapter, natural mode
vibrations of an elliptical membrane are also discussed. We consider a membrane
occupying an area \( S \) of the \( x, y \) plane bounded by a curve \( L \) where the deflection is
zero. The differential equation, boundary conditions, and initial conditions governing
the transverse deflection \( U(x, y, t) \) are

\[
\nabla^2 U = c^{-2} U_{tt} - P(x, y, t), (x, y) \in S,
\]

\[
U(x, y, 0) = U_0(x, y), U_t(x, y, 0) = V_0(x, y), (x, y) \in S,
\]

\[
U(x, y, t) = 0, (x, y) \in L.
\]

The parameter \( c \) is the speed of wave propagation in the membrane and \( P \) is the
applied normal load per unit area divided by the membrane tension per unit length.

© 2003 by CRC Press LLC
When $P = 0$, the motion is resolvable into a series of normal mode vibrations [22] of the form $u_n(x, y) \sin(\Omega_n t + \epsilon_n)$ satisfying
\[
\nabla^2 u_n(x, y) = -\Lambda^2_n u_n(x, y), \quad (x, y) \in S, \quad u_n(x, y) = 0, \quad (x, y) \in L
\]
where $\Lambda_n = \Omega_n / c$ is a positive real frequency parameter, and $u_n$ satisfies
\[
\iint u_n(x, y) u_m(x, y) \, dxdy = C_n \delta_{nm}, \quad C_n = \iint u_n(x, y)^2 \, dxdy
\]
where $\delta_{nm}$ is the Kronecker delta symbol. If the initial displacement and initial velocity are representable by a series of the modal functions, then the homogeneous solution satisfying general initial conditions is
\[
U(x, y, t) = \sum_{n=1}^{\infty} u_n(x, y) [A_n \cos(\Omega_n t) + B_n \sin(\Omega_n t) / \Omega_n]
\]
where
\[
A_n = \iint U_0(x, y) u_n(x, y) \, dxdy / C_n, \quad B_n = \iint V_0(x, y) u_n(x, y) \, dxdy / C_n.
\]
The nonhomogeneous case will be treated where the applied normal force on the membrane varies harmonically as
\[
P(x, y, t) = p(x, y) \cos(\Omega t)
\]
and $\Omega$ does not match any natural frequency of the membrane. We assume that the membrane is initially at rest with zero deflection and $p(x, y)$ is expandable as
\[
p(x, y) = \sum_{n=1}^{\infty} P_n u_n(x, y) \, dxdy, \quad P_n = \iint p(x, y) u_n(x, y) \, dxdy / C_n.
\]
Then the forced response solution satisfying zero initial conditions is found to be
\[
U(x, y, t) = \sum_{n=1}^{\infty} \frac{P_n}{\Lambda^2 - \Lambda^2_n} u_n(x, y) [\cos(\Omega t) - \cos(\Omega_n t)].
\]
This equation shows clearly that when the frequency of the forcing function is close to any one of the natural frequencies, then large deflection amplitudes can occur.

Next we turn to specific solutions for rectangular and circular membranes. Consider the normal mode functions for a rectangular region defined by $0 \leq x \leq a$, $0 \leq y \leq b$. It can be shown that the modal functions are
\[
u_{nm}(x, y) = \sin(n \pi x / a) \sin(m \pi y / b), \quad \Omega_{nm} = c \pi \sqrt{(n/a)^2 + (m/b)^2}
\]
and $C_n = ab / 4$. In the simple case where the applied surface force is a concentrated load applied at $(x_0, y_0)$, then
\[
p(x, y) = p_0 \delta(x - x_0) \delta(y - y_0)
\]
where \( \delta \) is the Dirac delta function. The series solution for a forced response solution is found to be
\[
U(x, y, t) = c^2 \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} \frac{P_{nm}}{\Omega^2 - \Omega_{nm}^2} \sin\left(\frac{n\pi x}{a}\right) \sin\left(\frac{m\pi y}{b}\right) \left[\cos(\Omega t) - \cos(\Omega_{nm} t)\right]
\]
with
\[
P_{nm} = \frac{4p_0}{ab} \sin(n\pi x_0/a) \sin(m\pi y_0/b).
\]
A similar kind of solution is obtainable as a series of Bessel functions when the membrane is circular. Transforming the wave equation to polar coordinates \((r, \theta)\) gives
\[
U_{rr} + \frac{1}{r} U_r + \frac{1}{r^2} U_{\theta\theta} = c^2 U_{tt} - P(r, \theta, t), \quad 0 \leq r \leq a, \quad -\pi \leq \theta \leq \pi, \quad t > 0.
\]
To reduce the algebraic complexity of the series solution developed below, it is helpful to introduce dimensionless variables \( \rho = r/a \) and \( \tau = c t / a \). Then the boundary value problem involving a harmonic forcing function becomes
\[
U_{\rho\rho} + \frac{1}{\rho} U_{\rho} + \frac{1}{\rho^2} U_{\theta\theta} = U_{\tau\tau} - p(\rho, \theta) \sin(\omega \tau), \quad 0 \leq \rho \leq 1, \quad -\pi \leq \theta \leq \pi, \quad \tau > 0,
\]
where \( \omega = \Omega a/c \). The modal functions for this problem are
\[
u_{nm}(\rho, \theta) = J_n(\lambda_{nm} \rho) \cos(n\theta + \epsilon_n)
\]
involving the integer order Bessel functions, with \( \lambda_{nm} \) being the \( m \)th positive root of \( J_n(\rho) \). These modal functions satisfy the orthogonality conditions discussed above and we employ the series expansion
\[
p(\rho, \theta) = \sum_{n=0}^{\infty} \sum_{m=1}^{\infty} J_n(\lambda_{nm} \rho) \text{ real}(A_{nm} e^{in\theta})
\]
where
\[
A_{nm} = \frac{2}{\pi(1 + \delta_{n0}) J_{n+1}^2(\lambda_{nm})} \int_{-\pi}^{\pi} \int_0^1 p(\rho, \theta) \rho J_n(\lambda_{nm} \rho) e^{-in\theta} d\rho d\theta.
\]
Then the forced response solution becomes
\[
U(\rho, \theta, \tau) = \sum_{n=0}^{\infty} \sum_{m=1}^{\infty} \frac{J_n(\lambda_{nm} \rho)}{\omega^2 - \lambda_{nm}^2} \text{ real}(A_{nm} e^{in\theta}) \left[\cos(\omega \tau) - \cos(\lambda_{nm} \tau)\right].
\]
In the special case where a concentrated force acts at \( \rho = \rho_0, \theta = 0 \), so that
\[
p(\rho, \theta) = p_0 \delta(\rho - \rho_0) \delta(\theta),
\]
then evaluating the double integral gives
\[
A_{nm} = p_0 \rho_0 J_n(\lambda_{nm} \rho_0)
\]
and \( \text{ real}(A_{nm} e^{-in\theta}) \) simplifies to \( A_{nm} \cos(n\theta) \).
9.5.1 Computer Formulation

Program membwave was written to depict wave propagation in a rectangular or circular membrane. Input data specifies information on membrane dimensions, forcing function frequency, force position coordinates, wave speed, and maximum time for solution generation. The primary computation tasks involve summing the double series defining the solutions. In the case of the circular membrane, the Bessel function roots determining the natural frequencies must also be computed. The various program modules are listed in the following table.

<table>
<thead>
<tr>
<th>membwave</th>
<th>reads data, calls other computational modules, and outputs time response</th>
</tr>
</thead>
<tbody>
<tr>
<td>memrecwv</td>
<td>sums the series for dynamic response of a rectangular membrane</td>
</tr>
<tr>
<td>memcirwv</td>
<td>calls besjroot to obtain the natural frequencies and sums the series for the circular membrane response</td>
</tr>
<tr>
<td>besjroot</td>
<td>computes a table of Bessel function roots</td>
</tr>
<tr>
<td>membanim</td>
<td>animates the dynamic response of the membrane</td>
</tr>
</tbody>
</table>

9.5.2 Input Data for Program membwave

Listed below are data cases showing animations of both rectangular and circular membranes. Waves propagate outward in a circular pattern from the point of application of the oscillating concentrated load. The membrane response becomes more complex as waves reflect from all parts of the boundary. In order to fully appreciate the propagating wave phenomenon, readers should run the program for several combinations of forcing function frequency and maximum time. The two surface plots below show deflected positions before waves have reached the entire boundary, so some parts of the membrane surface still remain undisturbed.

```plaintext
>> membwave;

WAVE MOTION IN A RECTANGULAR OR CIRCULAR MEMBRANE HAVING AN OSCILLATING LOAD

Select the geometry type:
Enter 1 for a rectangle, 2 for a circle > ? 1

Specify the rectangle dimensions:
Give values for a,b > ? 2,1

Give coordinates (x0,y0) where the
force act. Enter x0, y0 > ? 1.5, .5

Enter the wave speed > ? 1

The first forty-two natural frequencies are:

<table>
<thead>
<tr>
<th>3.5124</th>
<th>4.4429</th>
<th>5.6636</th>
<th>6.4766</th>
<th>7.0248</th>
<th>7.0248</th>
</tr>
</thead>
</table>

Input the frequency of the forcing function > 17.5

Input the maximum solution evaluation time.
> ? 5

Press return for animation
or enter 0 to stop > ?

Press return for animation
or enter 0 to stop > ? 0

All done

>> membwave;

WAVE MOTION IN A RECTANGULAR OR CIRCULAR
MEMBRANE HAVING AN OSCILLATING LOAD

Select the geometry type:
Enter 1 for a rectangle, 2 for a circle > ? 2

The circle radius equals one. Give the radial
distance r0 from the circle center to the
d Distance > ? .5

Enter the wave speed > ? 1

The first forty-two natural frequencies are:

<table>
<thead>
<tr>
<th>2.4048</th>
<th>3.8317</th>
<th>5.1356</th>
<th>5.5201</th>
<th>6.3801</th>
<th>7.0156</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1735</td>
<td>11.0647</td>
<td>11.0864</td>
<td>11.6199</td>
<td>11.7916</td>
<td>12.2251</td>
</tr>
</tbody>
</table>

© 2003 by CRC Press LLC
Input the frequency of the forcing function ? 17.5

Input the maximum solution evaluation time.  
> ? 5

Press return for animation  
or enter 0 to stop > ?

Press return for animation  
or enter 0 to stop > ? 0

All done
>

Figure 9.10:  Wave Propagation in a Rectangular Membrane
Figure 9.11: Wave Propagation in a Circular Membrane
Program membwave

function [u,x,y,t]= membwave(type,dims,alp,w,tmax)
%
% [u,x,y,t]=membwave(type,dims,alp,w,tmax)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This program illustrates waves propagating in
% a membrane of rectangular or circular shape
% with an oscillatory concentrated load acting at
% an arbitrary interior point. The membrane has
% fixed edges and is initially undeflected and
% at rest. The response u(x,y,t) is computed and
% animated plots depicting the motion are shown.
% %
% % type - 1 for rectangle, 2 for circle
% % dims - vector giving problem dimensions. For
type=1, dims=[a,b,x0,y0] where a and
% b are rectangle dimensions along the
% x and y axes. Also the oscillating
% force acts at (x0,y0). For type=2,
% a circular membrane of unit radius is
% analyzed with the concentrated force
% acting at (r0,0) where r0=dims(1);
% % alp - wave propagation velocity in the
% membrane
% % w - frequency of the applied force. This
% can be zero if the force is constant.
% % x0,y0 - coordinates of the point where
% the force acts
% % x,y,t - vectors of position and time values
% for evaluation of the solution
% % u - an array of size [length(x),...
% length(y),length(t)]
% in which u(i,j,k) contains the
% normalized displacement at
% y(i),x(j),t(k). The displacement is
% normalized by dividing by
% max(abs(u(:)))
% disp(' ')
disp('WAVE MOTION IN A RECTANGULAR OR CIRCULAR')
disp('MEMBRANE HAVING AN OSCILLATING LOAD')

© 2003 by CRC Press LLC
if nargin > 0 % Data passed through the call list
% must specify: type, dims, alp, w, tmax
% Typical values are: a=2; b=1; alp=1;
% w=18.4; x0=1; y0=0.5; tmax=5;
if type==1
    a=dims(1); b=dims(2); x0=dims(3); y0=dims(4);
    [u,x,y,t]=memrecwv(a,b,alp,w,x0,y0,tmax);
else
    r0=dims(1);
end
else % Interactive data input
    disp(' '), disp('Select the geometry type:')
    type=input(['Enter 1 for a rectangle, ',...'
    '2 for a circle > ']);
    if type ==1
        disp(' ') 
        disp('Specify the rectangle dimensions:')
        s=input('Give values for a,b > ? ','s');
        s=eval(['[',s,']']); a=s(1); b=s(2);
        disp(' ') 
        disp('Give coordinates (x0,y0) where the')
        s=input('force acts. Enter x0,y0 > ? ','s');
        s=eval(['[',s,']']); x0=s(1); y0=s(2);
        disp(' ') , alp=input('Enter the wave speed > ? ');

        N=40; M=40; pan=pi/a*(1:N)'; pbm=pi/b*(1:M);
        W=alp*sqrt(repmat(pan.^2,1,M)+repmat(pbm.^2,N,1));
        wsort=sort(W(:)); wsort=reshape(wsort(1:42),6,7)'
        disp('['
        disp('The first forty-two natural ','
        'frequencies are:'])
        disp(wsort)
        w=input(...
        'Input the frequency of the forcing function ? ');
    else
        disp(' '), disp('... 
        'The circle radius equals one. Give the radial')
        disp('... 
        'distance r0 from the circle center to the')
        r0=input('force > ? ');
        disp(' ') , alp=input('Enter the wave speed > ? ');

© 2003 by CRC Press LLC
% First 42 Bessel function roots
wsort=alp*[...
2.4048 3.8317 5.1356 5.5201 6.3801 7.0156
10.1735 11.0647 11.0864 11.6199 11.7916 12.2251
17.6159 17.8014 17.9599 18.0711 18.2876 18.4335];

disp(' '), disp(['The first forty-two ',...
'natural frequencies are:'])
disp(wsort)
w=input(...
'Input the frequency of the forcing function ? ');
end
disp(' ')
disp('Input the maximum solution evaluation time.')
tmax=input(' > ? ');
end

if type==1
[u,x,y,t]=memrecwv(a,b,alp,w,x0,y0,tmax);
else
th=linspace(0,2*pi,81); r=linspace(0,1,20);
[u,x,y,t]=memcirwv(r,th,r0,alp,w,tmax);
end

% Animate the solution
membanim(u,x,y,t);

%======================================================================

function [u,x,y,t]= memrecwv(a,b,alp,w,x0,y0,tmax)
%
% [u,x,y,t]=memrecwv(a,b,alp,w,x0,y0,tmax)
%======================================================================
% This function illustrates wave motion in a
% rectangular membrane subjected to a concentrated
% oscillatory force applied at an arbitrary
% interior point. The membrane has fixed edges
% and is initially at rest in an undeflected
% position. The resulting response u(x,y,t) is
% computed and a plot of the motion is shown.
% a,b    - side dimensions of the rectangle

© 2003 by CRC Press LLC
% alp - wave propagation velocity in the
% membrane
% w - frequency of the applied force. This
% can be zero if the force is constant.
% x0,y0 - coordinates of the point where
% the force acts
% x,y,t - vectors of position and time values
% for evaluation of the solution
% u - an array of size [length(y),
% length(x),length(t)] in which u(i,j,k)
% contains the normalized displacement
% corresponding to y(i), x(j), t(k). The
% displacement is normalized by dividing
% by max(abs(u(:))).
%
% The solution is a double Fourier series of form
%
% u(x,y,t)=Sum(A(n,m,x,y,t), n=1..N, m=1..M)
% where
% A(n,m,x,y,t)=sin(n*pi*x0/a)*sin(n*pi*x/a)*...
% sin(m*pi*y0/b)*sin(m*pi*y/b)*...
% (cos(w*t)-cos(W(n,m)*t))/...
% ( w^2-W(n,m)^2)
% and the membrane natural frequencies are
% W(n,m)=pi*alp*sqrt((n/a)^2+(m/b)^2)
%
if nargin==0
  a=2; b=1; alp=1; tmax=3; w=13; x0=1.5; y0=0.5;
end
if a<b
  nx=31; ny=round(b/a*21); ny=ny+rem(ny+1,2);
else
  ny=31; nx=round(a/b*21); nx=nx+rem(nx+1,2);
end
x=linspace(0,a,nx); y=linspace(0,b,ny);

N=40; M=40; pan=pi/a*(1:N)'; pbm=pi/b*(1:M);
W=alp*sqrt(repmat(pan.'^2,1,M)+repmat(pbm.^2,N,1));
wsort=sort(W(:)); wsort=reshape(wsort(1:30),5,6)';
Nt=ceil(40*tmax*alp/min(a,b));
t=tmax/(Nt-1)*(0:Nt-1);

% Evaluate fixed terms in the series solution
mat=sin(x0*pan)*sin(y0*pbm)./(w^2-W.^2);
sxn=sin(x(:)*pan'); smy=sin(pbm'*y(:)');

© 2003 by CRC Press LLC
u=zeros(ny,nx,Nt);
for j=1:Nt
    A=mat.*(cos(w*t(j))-cos(W*t(j)));
    uj=sxn*(A*smy); u(:,:,j)=uj';
end

%=================================================================

function [u,x,y,t,r,th]=memcirwv(r,th,r0,alp,w,tmax)
% [u,x,y,t,r,th]=memcirwv(r,th,r0,alp,w,tmax)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function computes the wave response in a
% circular membrane having an oscillating force
% applied at a point on the radius along the
% positive x axis.
% 
% r,th - vectors of radius and polar angle values
% r0  - radial position of the concentrated force
% w   - frequency of the applied force
% tmax - maximum time for computing the solution
% 
% % User m function used: besjroot
% if nargin==0
r0=.4; w=15.5; th=linspace(0,2*pi,81);
r=linspace(0,1,21); alp=1;
end
Nt=ceil(20*alp*tmax); t=tmax/(Nt-1)*(0:Nt-1);

% Compute the Bessel function roots needed in
% the series solution. This takes a while.
lam=besjroot(0:20,20,1e-3);

% Compute the series coefficients
[nj,nk]=size(lam); r=r(:)'; nr=length(r);

% th=th(:); nth=length(th); nt=length(t);
N=repmat((0:nj-1)',1,nk); Nvec=N(:)';
c=besselj(N,lam*r0)./(besselj(...
N+1,lam).^2.*(lam.^2-w^2));
c(1,:)=c(1,:)/2; c=c(':');

% Sum the series of Bessel functions
lamvec=lam(:)'; wlam=w./lamvec;
c=cos(th*Nvec).*repmat(c,nth,1);
rmat=besselj(repmat(Nvec',1,nr),lamvec'*r);
u=zeros(nth,nr,nt);
for k=1:nt
tvec=-cos(w*t(k))+cos(lamvec*t(k));
u(:,:,k)=c.*repmat(tvec,nth,1)*rmat;
end
u=2/pi*u; x=cos(th)*r; y=sin(th)*r;

%==================================================================

function rts=besjroot(norder,nrts,tol)
%
% rts=besjroot(norder,nrts,tol)
% -----------------------------
% This function computes an array of positive roots
% of the integer order Bessel functions besselj of
% the first kind for various orders. A chosen number
% of roots is computed for each order
% norder - a vector of function orders for which
% roots are to be computed. Taking 3:5
% for norder would use orders 3,4 and 5.
% nrts - the number of positive roots computed for
% each order. Roots at x=0 are ignored.
% rts - an array of roots having length(norder)
% rows and nrts columns. The element in
% column k and row i is the k'th root of
% the function besselj(norder(i),x).
% tol - error tolerance for root computation.

if nargin<3, tol=1e-5; end
jn=inline('besselj(n,x)','x','n');
N=length(norder); rts=ones(N,nrts)*nan;
opt=optimset('TolFun',tol,'TolX',tol);
for k=1:N
n=norder(k); xmax=1.25*pi*(nrts-1/4+n/2);
xsrch=.1:pi/4:xmax; fb=besselj(n,xsrch);
f= length(fb); K=find(fb(1:nf-1).*fb(2:nf)<0);
if length(K)<nrts
   disp('Search error in function besjroot')
   rts=nan; return
else
   K=K(1:nrts);
   for i=1:nrts

interval=xsrch(K(i):K(i)+1);
   rts(k,i)=fzero(jn,interval,opt,n);
end
end
end

%=================================================================

function membanim(u,x,y,t)
%=================================================================
% This function animates the motion of a
% vibrating membrane
% u array in which component u(i,j,k) is the
% displacement for y(i),x(j),t(k)
% x,y arrays of x and y coordinates
% t vector of time values
% Compute the plot range
if nargin==0;
   [u,x,y,t]=memrecwv(2,1,1,15.5,1.5,.5,5);
end
xmin=min(x(:)); xmax=max(x(:));
ymin=min(y(:)); ymax=max(y(:));
xmid=(xmin+xmax)/2; ymid=(ymin+ymax)/2;
d=max(xmax-xmin,ymax-ymin)/2; Nt=length(t);
range=[xmid-d,xmid+d,ymid-d,ymid+d,...
      3*min(u(:)),3*max(u(:))];
while 1 % Show the animation repeatedly
   disp(' '), disp('Press return for animation')
dumy=input('or enter 0 to stop > ? ','s');
   if ~isempty(dumy)
      disp(' '), disp('All done'), break
   end
% Plot positions for successive times
   for j=1:Nt
      surf(x,y,u(:, :,j)), axis(range)
      xlabel('x axis'), ylabel('y axis')
      zlabel('u axis'), titl=sprintf(...
         'MEMBRANE POSITION AT T=%5.2f',t(j));
      title(titl), colormap([1 1 1])
   end
Analyzing the dynamic response caused when a time dependent moment acts on the end of an Euler beam involves a boundary value problem for a fourth order linear partial differential equation. In the following example we consider a beam of uniform cross section which is pin-ended (hinged at the ends) and is initially at rest. Suddenly, a harmonically varying moment $M_0 \cos(\Omega_0 T)$ is applied to the right end as shown in Figure 9.12. Determination of the resulting displacement and bending moment in the beam is desired. Let $U$ be the transverse displacement, $X$ the longitudinal distance from the right end, and $T$ the time. The differential equation, boundary conditions, and initial conditions characterizing the problem are

$$EI \frac{\partial^4 U}{\partial X^4} = -A\rho \frac{\partial^2 U}{\partial T^2}, \quad 0 < X < L, \quad T > 0,$$

$$U(0, T) = 0, \quad \frac{\partial^2 U}{\partial X^2}(0, T) = 0, \quad U(L, T) = 0, \quad \frac{\partial^2 U}{\partial X^2}(L, T) = M_0 \cos(\Omega_0 T)/(EI),$$

$$U(0, T) = 0, \quad \frac{\partial U}{\partial T}(0, T) = 0,$$

where $L$ is the beam length, $EI$ is the product of the elastic modulus and the moment of inertia, and $A\rho$ is the product of the cross section area and the mass density.

This problem can be represented more conveniently by introducing dimensionless variables

$$x = \frac{X}{L}, \quad t = \sqrt{\frac{EI}{A\rho L^2}}, \quad u = \frac{EI}{M_0L^2} U, \quad \omega = \sqrt{\frac{A\rho}{EI}} L^2 \Omega_0, \quad m = \frac{\partial^2 u}{\partial x^2}.$$
The new boundary value problem is then
\[
\frac{\partial^4 u}{\partial x^4} = -\frac{\partial^2 u}{\partial t^2}, \quad 0 < x < 1, \quad t > 0,
\]
\[
u(0, t) = 0, \quad \frac{\partial^2 u}{\partial x^2}(0, t) = 0, \quad u(1, t) = 0, \quad \frac{\partial^2 u}{\partial x^2}(1, t) = \cos(\omega t),
\]
\[
u(x, 0) = 0, \quad \frac{\partial u}{\partial t}(x, 0) = 0, \quad 0 < x < 1.
\]

The problem can be solved by combining a particular solution \( w \) which satisfies the differential equation and nonhomogeneous boundary conditions with a homogeneous solution in series form which satisfies the differential equation and homogeneous boundary conditions. Thus we have \( u = w + v \). The particular solution can be found in the form
\[
w = f(x) \cos(\omega t)
\]
where \( f(x) \) satisfies
\[
f'''(x) = \omega^2 f(x)
\]
and
\[
f(0) = f''(0) = f(1) = 0, \quad f''(1) = 1.
\]
This ordinary differential equation is solvable as
\[
f(x) = \sum_{k=1}^{4} c_k e^{s_k x}
\]
where
\[
s_k = \sqrt{\omega} e^{\pi i (k-1)/2}
\]
and \( i = \sqrt{-1} \). The boundary conditions require
\[
\sum_{k=1}^{4} c_k = 0, \quad \sum_{k=1}^{4} s_k^2 c_k = 1, \quad \sum_{k=1}^{4} c_k e^{s_k} = 0, \quad \sum_{k=1}^{4} c_k s_k^2 e^{s_k} = 0.
\]
Solving these simultaneous equations determines the particular solution. The initial displacement for the particular solution can be expanded in a Fourier series as an odd valued function of period 2. Hence we can write
\[
w(0, t) = f(x) = \sum_{k=-\infty}^{\infty} c_k e^{i\pi k x} = \sum_{k=1}^{\infty} a_k \sin(k \pi x), \quad \frac{\partial w}{\partial t}(0, t) = 0
\]
involve complex Fourier coefficients, \( c_k \), and \( a_k = -2 \text{Im}(c_k) \). The homogeneous solution is representable as
\[
v(x, t) = -\sum_{k=1}^{\infty} a_k \cos(\pi^2 k^2 t) \sin(k \pi x)
\]
so that $w + v$ combine to satisfy the desired initial conditions of zero displacement and velocity.

Of course, perfect satisfaction of the initial conditions cannot be achieved without taking an infinite number of terms in the Fourier series. However, the series converges very rapidly because the coefficients are of order $n^{-3}$. When a hundred or more terms are used, an approximate solution produces results which satisfy the differential equation and boundary conditions, and which insignificantly violate the initial displacement condition. It is important to remember the nature of this error when examining the bending moment results presented below. Effects of high frequency components are very evident in the moment. Despite the oscillatory character of the moments, these results are exact for the initial displacement conditions produced by the truncated series. These displacements agree closely with the exact solution.

A program was written to evaluate the series solution to compute displacements and moments as functions of position and time. Plots and surfaces showing these quantities are presented along with timewise animations of the displacement and moment across the span. The computation involves the following steps:

1. Evaluate $f(x)$;
2. Expand $f(x)$ using the FFT to get coefficients for the homogeneous series solution;
3. Combine the particular and homogeneous solution by summing the series for any number of terms desired by the user;
4. Plot $u$ and $m$ for selected times;
5. Plot surfaces showing $u(x, t)$ and $m(x, t)$;
6. Show animated plots of $u$ and $m$.

The principal parts of the program are shown in the table below.

<table>
<thead>
<tr>
<th>bemimpac</th>
<th>reads data and creates graphical output</th>
</tr>
</thead>
<tbody>
<tr>
<td>beamresp</td>
<td>converts material property data to dimensionless form and calls ndbemrsp</td>
</tr>
<tr>
<td>ndbemrsp</td>
<td>construct the solution using Fourier series</td>
</tr>
<tr>
<td>sumser</td>
<td>sums the series for displacement and moment</td>
</tr>
<tr>
<td>animate</td>
<td>animates the time history of displacement and moment</td>
</tr>
</tbody>
</table>

The numerical results show the response for a beam subjected to a moment close to the first natural frequency of the beam. It can be shown that, in the dimensionless problem, the system of equations defining the particular solution becomes singular
when $\omega$ assumes values of the form $k^2 \pi^2$ for integer $k$. In that instance the series solution provided here will fail. However, values of $\omega$ near to resonance can be used to show how the displacements and moments quickly become large. In our example we let $EI$, $Ap$, $l$, and $M_0$ all equal unity, and $\omega \approx 0.95\pi^2$. Figures 9.13 and 9.14 show displacement and bending moment patterns shortly after motion is initiated. The surfaces in Figures 9.15 and 9.16 also show how the displacement and moment grow quickly with increasing time. The reader may find it interesting to run the program for various choices of $\omega$ and observe how dramatically the chosen forcing frequency affects results.
Figure 9.14: Bending Moment in the Beam

Figure 9.15: Displacement Growth Near Resonance
Bending Moment as a Function of Time and Position

Figure 9.16: Moment Growth Near Resonance

MATLAB Example

Program bemimpac

function bemimpac
% Example: bemimpac
% ~~~~~~~~~~~~~~~
% This program analyzes an impact dynamics problem for an elastic Euler beam of
% constant cross section which is simply supported at each end. The beam is initially
% at rest when a harmonically varying moment m0*cos(w0*t) is applied to the right end.
% The resulting transverse displacement and bending moment are computed. The
% displacement and moment are plotted as functions of x for the three time values.
% Animated plots of the entire displacement and moment history are also given.
% User m functions required:

© 2003 by CRC Press LLC
% beamresp, beamanim, sumser, ndbemrsp

fprintf('
DYNAMICS OF A BEAM WITH AN OSCILLATING END MOMENT\n');
i=1; arho=1; len=1; m0=1; w0=.90*pi^2;
tmin=0; tmax=5; nt=101;
xmin=0; xmax=len; nx=151; ntrms=200;
[t,x,displ,mom]=beamresp(ei,arho,len,m0,w0,...
tmin,tmax,nt,xmin,xmax,nx,ntrms);
disp('')
disp('Press [Enter] to see the deflection for three positions'), pause

np=[3 5 8]; clf; pltsave=0;
dip=displ(np,:); mop=mom(np,:);
plot(x,dip(1,:),'-k',x,dip(2,:),':b',...
x,dip(3,:),'-r');
xlabel('x axis'); ylabel('displacement');

hh=gca;
[r(1:2)=get(hh,'XLim'); r(3:4)=get(hh,'YLim')];

xp=r(1)+(r(2)-r(1))/10;

mp=r(3)+(r(4)-r(3))/10;
tstr=['Displacement for Nearly Resonant Moment Acting at Right End'
'
 Moment Acting at Right End'];
title(tstr);

legend('t=0.10','t=0.20','t=0.35',3)
disp('')
disp('Press [Enter] to the bending moment for three positions')

if pltsave, print -deps 3positns, end

clf;
plot(x,mop(1,:),'-k',x,mop(2,:),':b',...
x,mop(3,:),'-r');

h=gca;
[r(1:2)=get(h,'XLim'); r(3:4)=get(h,'YLim')];

mp=r(3)+(r(4)-r(3))/10;
xlabel('x axis'); ylabel('moment');
tstr=['Bending Moment for Nearly Resonant Moment Acting at Right End'
'
 Moment Acting at Right End'];
title(tstr);
text(xp,mp,['Number of series terms ' ...
'used = ',int2str(ntrms)]);

legend('t=0.10','t=0.20','t=0.35',3)
disp('')
disp('Press [Enter] to the bending moment for three positions')

if pltsave, print -deps 3positns, end

clf;
plot(x,mop(1,:),'-k',x,mop(2,:),':b',...
x,mop(3,:),'-r');

h=gca;
[r(1:2)=get(h,'XLim'); r(3:4)=get(h,'YLim')];

mp=r(3)+(r(4)-r(3))/10;
xlabel('x axis'); ylabel('moment');
tstr=['Bending Moment for Nearly Resonant Moment Acting at Right End'
'
 Moment Acting at Right End'];
title(tstr);
text(xp,mp,['Number of series terms ' ...


used = int2str(ntrms));

legend('t=0.10','t=0.20','t=0.35',2),
disp(' '), disp(...
'Press [Enter] to see the deflections surface')
shg, pause
if pltsave, print -deps 3moments, end

inct=2; incx=2;
ht=0.75; it=1:inct:.8*nt; ix=1:incx:nx;

 tt=t(it); xx=x(ix);
 dd=displ(it,ix); mm=mom(it,ix);
 a=surf(xx,tt,dd);
 tstr=['Transverse Deflection as a ' ... 
 'Function of Time and Position'];
 title(tstr);
xlabel('x axis'); ylabel('time');
 zlabel('transverse deflection');
disp(' '), disp(['Press [Enter] to ',...
 'see the bending moment surface'])
shg, pause
if pltsave, print -deps bdeflsrf, end

 a=surf(xx,tt,mm);
title(['Bending Moment as a Function ' ... 
 'of Time and Position'])
xlabel('x axis'); ylabel('time');
 zlabel('bending moment');
disp(' '), disp('Press [Enter] to see animation of');
disp('the beam deflection'), shg, pause
if pltsave, print -deps bmomsrf, end

 beamanim(x,displ,.1,'Transverse Deflection', ... 
 'x axis','deflection'), disp(' 
Press [Enter] to see animation');
disp('of the bending moment'); pause
 beamanim(x,mom,.1,'Bending Moment History', ... 
 'x axis','moment');
 fprintf('
All Done
'); close;

%======================================================================

function [t,x,displ,mom]= ...
beamresp(ei,arho,len,m0,w0,tmin,tmax, ... 
 nt,xmin,xmax,nx,ntrms)

% [t,x,displ,mom]=beamresp(ei,arho,len,m0, ...
% w0, tmin, tmax, nt, xmin, xmax, nx, ntrms)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function evaluates the time dependent
% displacement and moment in a constant
% cross section, simply supported beam which
% is initially at rest when a harmonically
% varying moment is suddenly applied at the
% right end. The resulting time histories of
% displacement and moment are computed.
%
% ei - modulus of elasticity times
% moment of inertia
% arho - mass per unit length of the
% beam
% len - beam length
% m0, w0 - amplitude and frequency of the
% harmonically varying right end
% moment
% tmin, tmax - minimum and maximum times for
% the solution
% nt - number of evenly spaced
% solution times
% xmin, xmax - minimum and maximum position
% coordinates for the solution.
% These values should lie between
% zero and len (x=0 and x=len at
% the left and right ends).
% nx - number of evenly spaced solution
% positions
% ntrms - number of terms used in the
% Fourier sine series
% t - vector of nt equally spaced time
% values varying from tmin to tmax
% x - vector of nx equally spaced
% position values varying from
% xmin to xmax
% displ - matrix of transverse
% displacements with time varying
% from row to row, and position
% varying from column to column
% mom - matrix of bending moments with
% time varying from row to row,
% and position varying from column
to column

© 2003 by CRC Press LLC
% User m functions called: ndbemrsp
%---------------------------------------------------------------

tcof=sqrt(arho/ei)*len^2; dcof=m0*len^2/ei;
tmin=tmin/tcof; tmax=tmax/tcof; w=w0*tcof;
xmin=xmin/len; xmax=xmax/len;
[t,x,displ,mom]=...
dbemrsp(w,tmin,tmax,nt,xmin,xmax,nx,ntrms);
t=t*tcof; x=x*len;
displ=displ*dcof; mom=mom*m0;

%============================================= 

function beamanim(x,u,tpause,titl,xlabl,ylabl)
%
% beamanim(x,u,tpause,titl,xlabl,ylabl,save)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function draws an animated plot of data
% values stored in array u. The different
% columns of u correspond to position values
% in vector x. The successive rows of u
% correspond to different times. Parameter
% tpause controls the speed of animation.
%
% u        - matrix of values to animate plots
% of u versus x
% x        - spatial positions for different
%           columns of u
% tpause   - clock seconds between output of
%           frames. The default is .1 secs
%           when tpause is left out. When
%           tpause=0, a new frame appears
%           when the user presses any key.
% titl     - graph title
% xlabl    - label for horizontal axis
% ylabl    - label for vertical axis
%
% User m functions called: none
%---------------------------------------------------------------

if nargin<6, ylabl=''; end;
if nargin<5, xlabl=''; end
if nargin<4, titl=''; end;
if nargin<3, tpause=.1; end;

© 2003 by CRC Press LLC
[ntime,nxpts]=size(u);
umin=min(u(:)); umax=max(u(:));
udif=umax-umin; uavg=.5*(umin+umax);
xmin=min(x); xmax=max(x);
xdif=xmax-xmin; xavg=.5*(xmin+xmax);
xwmin=xavg-.55*xdif; xwmax=xavg+.55*xdif;
uwmin=uavg-.55*udif; uwmax=uavg+.55*udif; clf;
axis([xwmin,xwmax,uwmin,uwmax]); title(titl);
xlabel(xlabl); ylabel(ylabl); hold on;

for j=1:ntime
ut=u(j,:);
plot(x,ut,'-'); axis('off'); figure(gcf);
if tpause==0
    pause;
else
    pause(tpause);
end
if j==ntime, break, else, cla; end
end
% print -deps cntltrac
hold off; clf;

%========================================================================

function [u,t,x] = sumser(a,b,c,funt,funx, ...
tmin,tmax,nt,xmin,xmax,nx)

% [u,t,x] = sumser(a,b,c,funt,funx,tmin, ... 
% tmax,nt,xmin,xmax(nx)
%========================================================================

% This function evaluates a function U(t,x)
% which is defined by a finite series. The
% series is evaluated for t and x values taken
% on a rectangular grid network. The matrix u
% has elements specified by the following
% series summation:
% % u(i,j) = sum( a(k)*funt(t(i)*b(k))*...
% % k=1:nsum
% % funx(c(k)*x(j))
% % where nsum is the length of each of the
% % vectors a, b, and c.

© 2003 by CRC Press LLC
% a,b,c  - vectors of coefficients in
% the series
% funt,funx - handles of functions accepting
% matrix argument. funt is
% evaluated for an argument of
% the form funt(t*b) where t is
% a column and b is a row. funx
% is evaluated for an argument
% of the form funx(c*x) where
% c is a column and x is a row.
% tmin,tmax,nt - produces vector t with nt
% evenly spaced values between
% tmin and tmax
% xmin,xmax,nx - produces vector x with nx
% evenly spaced values between
% xmin and xmax
% u - the nt by nx matrix
% containing values of the
% series evaluated at t(i),x(j),
% for i=1:nt and j=1:nx
% t,x - column vectors containing t
% and x values. These output
% values are optional.

% User m functions called: none.
%----------------------------------------------

tt=(tmin:(tmax-tmin)/(nt-1):tmax)';
xx=(xmin:(xmax-xmin)/(nx-1):xmax); a=a(:).';
u=a(ones(nt,1),:).*feval(funt,tt*b(:).')*feval(funx,c(:)*xx);
if nargout>1, t=tt; x=xx'; end

%=============================================

function [t,x,displ,mom]= ...
    ndbemrsp(w,tmin,tmax,nt,xmin,xmax,nx,ntrms)
% [t,x,displ,mom]=ndbemrsp(w,tmin,tmax,nt,...
% xmin,xmax,nx,ntrms)
% This function evaluates the nondimensional
% displacement and moment in a constant
% cross section, simply supported beam which
% is initially at rest when a harmonically
% varying moment of frequency \( w \) is suddenly
% applied at the right end. The resulting
% time history is computed.
%
% \( w \) - frequency of the harmonically
% varying end moment
% \( t_{	ext{min}}, t_{	ext{max}} \) - minimum and maximum
% dimensionless times
% \( n_{t} \) - number of evenly spaced
% solution times
% \( x_{	ext{min}}, x_{	ext{max}} \) - minimum and maximum
% dimensionless position
% coordinates. These values
% should lie between zero and
% one (\( x=0 \) and \( x=1 \) give the
% left and right ends).
% \( n_{x} \) - number of evenly spaced
% solution positions
% \( n_{t_{	ext{rms}}} \) - number of terms used in the
% Fourier sine series
% \( t \) - vector of \( n_{t} \) equally spaced
% time values varying from
% \( t_{	ext{min}} \) to \( t_{	ext{max}} \)
% \( x \) - vector of \( n_{x} \) equally spaced
% position values varying
% from \( x_{	ext{min}} \) to \( x_{	ext{max}} \)
% \( \text{displ} \) - matrix of dimensionless
% displacements with time
% varying from row to row,
% and position varying from
% column to column
% \( \text{mom} \) - matrix of dimensionless
% bending moments with time
% varying from row to row, and
% position varying from column
to column
%
% User m functions called: sumser
%----------------------------------------------

if nargin < 8, \( w=0 \); end; nft=512; nh=nft/2;
xft=1/nh*(0:nh)’;
x=xmin+(xmax-xmin)/(nx-1)*(0:nx-1)’;
t=tmin+(tmax-tmin)/(nt-1)*(0:nt-1)’;
cwt=cos(w*t);
% Get particular solution for nonhomogeneous
% end condition
if w == 0 % Case for a constant end moment
    cp = [1 0 0 0; 0 0 2 0; 1 1 1 1; 0 0 2 6] \ ...
        [0; 0; 0; 1];
    yp = [ones(size(x)), x, x.^2, x.^3] * cp; yp = yp';
    mp = [zeros(nx, 2), 2*ones(nx, 1), 6*x] * cp;
    mp = mp';
    ypft = [ones(size(xft)), xft, xft.^2, xft.^3] * cp;
else % Case where end moment oscillates
    s = sqrt(w) * [1, i, -1, -i]; es = exp(s);
    cp = [ones(1, 4); s.^2; es; es.*s.^2] \ ...
        [0; 0; 0; 1];
    yp = real(exp(x*s) * cp); yp = yp';
    mp = real(exp(x*s) * (cp.*s(:).^2)); mp = mp';
    ypft = real(exp(xft*s) * cp);
end

% Fourier coefficients for
% particular solution
yft = -fft([ypft; -ypft(nh:-1:2)]) / nft;

% Sine series coefficients for
% homogeneous solution
acof = -2*imag(yft(2:ntrms+1));
ccof = pi*(1:ntrms)'; bcof = ccof.^2;

% Sum series to evaluate Fourier
% series part of solution. Then combine
% with the particular solution.
displ = sumser(acof, bcof, ccof, @cos, @sin, ...
    tmin, tmax, nt, xmin, xmax, nx);
    displ = displ + cwt * yp; acof = acof.*bcof;
mom = sumser(acof, bcof, ccof, 'cos', 'sin', ...
    tmin, tmax, nt, xmin, xmax, nx);
mom = -mom + cwt * mp;
9.7 Forced Vibration of a Pile Embedded in an Elastic Medium

Structures are often supported by piles embedded in soil foundations. The response of these systems, when the foundation is shaken in the manner occurring in an earthquake, has considerable practical interest. Let us examine a simple model approximating a single pile connected to an overlying structure. The pile is treated as a beam of uniform cross section buried in an elastic medium. An attached mass at the top causes inertial resistance to translation and rotation. The beam, shown in Figure 9.19 in a deflected position, has length \( \ell \) with \( x = 0 \) denoting the lower end and \( x = \ell \) denoting the top. Rotating the member 90° from the vertical is done to agree with the coordinate referencing traditionally used in beam analysis. We are interested in the steady-state response when the foundation displacement is \( y_o \cos(\omega t) \).

For convenience we use a complex valued forcing function and get the final results by taking the real part of the complex valued solution. The transverse bending response is to be computed when the surrounding elastic medium has an oscillatory motion of the form \( y_f = y_o e^{i\omega t} \).

The differential equation governing transverse oscillations of the beam is

\[
EI \frac{\partial^4 y(x, t)}{\partial x^4} = -A\rho \frac{\partial^2 y(x, t)}{\partial t^2} + k \left( y_o e^{i\omega t} - y \right)
\]

where \( EI \) is the product of the elastic modulus and the inertial moment of the beam, \( A\rho \) is the product of the cross section area and the mass per unit volume, and \( k \) describes the foundation stiffness in terms of force per unit length per unit of transverse deflection. The shear \( V \) and moment \( M \) in the beam are related to the deflection \( y(x, t) \) by

\[
V = EI \frac{\partial^3 y(x, t)}{\partial x^3}, \quad M = EI \frac{\partial^2 y(x, t)}{\partial x^2}.
\]

In the current analysis we consider forced response of frequency \( \omega \) described in the form

\[
y(x, t) = f(x)e^{i\omega t}
\]

so that

\[
V = EI f'''(x)e^{i\omega t}, \quad M = EI f''(x)e^{i\omega t}.
\]

The boundary conditions at \( x = 0 \) require vanishing moment and shear:

\[
f''(0) = 0, \quad f'''(0) = 0.
\]

The boundary conditions at \( x = \ell \) are more involved because inertial resistance of the end mass must be handled. We assume that the gravity center of the end mass is located along the axis of the beam at a distance \( h \) above the top end. Furthermore,
the attached body has a mass $m_0$ and inertial moment $J_0$ about its gravity center. The angular acceleration $\ddot{\theta}$ and the transverse acceleration $a_m$ are expressible as

$$\ddot{\theta} = \frac{\partial^3 y(\ell, t)}{\partial x \partial t^2} = -\omega^2 f'(\ell) e^{i\omega t}$$

and

$$a_m = -\frac{\partial^2 y(\ell, t)}{\partial t^2} + h \ddot{\theta} = -\omega^2 e^{i\omega t} \left[ f(\ell) + h f'(\ell) \right].$$

Writing equations of motion for the end mass gives

$$m_0 a_m = V(\ell, t) \quad \text{and} \quad J_0 \ddot{\theta}_m = -hV(\ell, t) - M(\ell, t).$$

Representing these conditions in terms of $f(x)$ yields

$$-\omega^2 m_0 [f(\ell) + h f'(\ell)] = EI f''''(\ell) \quad \text{and} \quad \omega^2 J_0 f'(\ell) = EI [f''(\ell) + h f'''(\ell)].$$

Furthermore, the factor $e^{i\omega t}$ cancels out of the differential equation

$$EI f''''(x) = (A\rho \omega^2 - k) f(x) + y_o k.$$ 

The general solution of this fourth order linear differential equation is expressed as

$$f(x) = \frac{y_o k}{k - A\rho \omega^2} \left[ 1 + \sum_{j=1}^{4} c_j e^{s_j x} \right]$$

where $s_j$ are complex roots given by

$$s_j = \left( \frac{A\rho \omega^2 - k}{EI} \right)^{1/4} e^{i(j-1)\pi/2}, \quad j = 1, 2, 3, 4.$$
The conditions of zero moment and shear at \( x = 0 \) lead to

\[
\sum_{j=1}^{4} s_j^2 c_j = 0, \quad \sum_{j=0}^{4} s_j^3 c_j = 0.
\]

The shear and moment conditions at \( x = \ell \) require

\[
\sum_{j=1}^{4} s_j^3 e^{s_j \ell} c_j = -m_0 \omega^2 \left[ 1 + \sum_{j=1}^{4} (1 + h s_j) e^{s_j \ell} c_j \right]
\]

and

\[
\sum_{j=1}^{4} \left(s_j^2 + h s_j^3\right) e^{s_j \ell} c_j = J_0 \omega^2 \sum_{j=1}^{4} s_j e^{s_j \ell} c_j.
\]

The system of four simultaneous equations can be solved for \( c_1, \ldots, c_4 \). Then the forced response solution corresponding to a foundation motion

\[
\text{real} \left( y_o e^{i\omega t} \right) = y_o \cos(\omega t)
\]

is given by

\[
y(x, t) = \text{real} \left( f(x) e^{i\omega t} \right)
\]

where \( f(x) \) is complex valued.

The function pilevibs evaluates the displacement, moment, and shear for \( 0 \leq x \leq \ell, \ 0 \leq t \leq 2\pi/\omega \). Surface plots of these quantities are shown in Figures 9.18 through 9.20. Figure 9.21 is a single frame from an animation depicting how the pile and the attached mass move.
Figure 9.18: Deflection Surface for a Vibrating Pile

Figure 9.19: Bending Moment in a Vibrating Pile
Figure 9.20: Shear Force in a Vibrating Pile

Figure 9.21: Frame from Pile Animation
Program Output and Code

Program pilevibs

function pilevibs
% Example: pilevibs
% ~~~~~~~~~~~~~~~~~
% The routine is used to solve an example
% problem using function pvibs. The example
% involves a steel pile 144 inches long which
% has a square cross section of 4 inch depth.
% The pile is immersed in soil having an elastic
% modulus of 200 psi. The attached mass weighs
% 736 lb. The foundation is shaken at an
% amplitude of 0.5 inch with a frequency of
% 20 cycles per second.
%
% User m functions required: pvibs

clear;
L=144; d=4; a=d^2; I=d^4/12; e=30e6; ei=e*I;
g=32.2*12; Density_steel=0.284;
rho=Density_steel/g;
Cap_w=36; Cap_h=18; Cap_t=4;
m0=Cap_w*Cap_h*Cap_t*rho;
j0=m0/12*(Cap_h^2+Cap_w^2);
h=Cap_h/2; arho=a*rho;
e_soil=200; k=e_soil*d; y0=0.5; w=40*pi;
nx=42; nt=25;
[t,x,y,m,v]= pvibs(y0,ei,arho,L,k,w,h,m0,j0,nx,nt);

function [t,x,y,m,v]= pvibs(y0,ei,arho,L,k,w,h,m0,j0,nx,nt)
% This function computes the forced harmonic
% response of a pile buried in an oscillating
% elastic medium. The lower end of the pile is 
% free from shear and moment. The top of the 
% pile carries an attached body having general 
% mass and inertial properties. The elastic 
% foundation is given a horizontal oscillation 
% of the form 
% 
% $y_f = \text{real}(y_0 \exp(i\omega t))$
% 
% The resulting transverse forced response of 
% the pile is expressed as 
% 
% $y(x,t) = \text{real}(f(x) \exp(i\omega t))$
% 
% where $f(x)$ is a complex valued function. The 
% bending moment and shear force in the pile 
% are also computed.
% 
% $y_0$ - amplitude of the foundation oscillation 
% $e_i$ - product of moment of inertia and 
% elastic modulus for the pile 
% $a_{rho}$ - mass per unit length of the pile 
% $L$ - pile length 
% $k$ - the elastic resistance constant for the 
% foundation described as force per unit 
% length per unit of transverse 
% deflection 
% $\omega$ - the circular frequency of the 
% foundation oscillation which vibrates 
% like $\text{real}(y_0 \exp(i\omega t))$ 
% $h$ - the vertical distance above the pile 
% upper end to the gravity center of the 
% attached body 
% $m_0$ - the mass of the attached body 
% $j_0$ - the mass moment of inertia of the 
% attached body with respect to its 
% gravity center 
% $n_x$ - the number of equidistant values along 
% the pile at which the solution is 
% computed 
% $n_t$ - the number of values of $t$ values at 
% which the solution is computed such 
% that $0 \leq \omega t \leq 2\pi$ 
% $t$ - a vector of time values such that the
% pile moves through a full period of motion. This means 0 <= t <= 2*pi/w
% x - a vector of x values with 0 <= x <= L
% y - the transverse deflection y(x,t) for the pile with t varying from row to row, and x varying from column to column
% m,v - matrices giving values bending moment and shear force
%
% User m functions called: none
%----------------------------------------------

% Default data for a steel pile 144 inches long
if nargin==0
y0=0.5; ei=64e7; arho=0.0118; L=144; k=800;
w=125.6637; h=9; m0=1.9051; j0=257.1876;
nx=42; nt=25;
end

w2=w^2; x=linspace(0,L,nx)';
t=linspace(0,2*pi/w,nt);

% Evaluate characteristic roots and complex exponentials
s=((arho*w2-k)/ei)^(1/4)*[1,i,-1,-i];
s2=s.^2; s3=s2.*s;
c0=y0*k/(k-w2*arho); esl=exp(s*L);
esx=exp(x*s); eiwt=exp(i*w*t);

% Solve for coefficients to satisfy the boundary conditions
c=[s2; s3; esl.*(h*s3+s2-j0*w2/ei*s)];
esl.*(s3+m0*w2/ei*(1+h*s))]/ ... [0;0;0;-c0*m0*w2/ei];

% Compute the deflection, moment and shear
y=real((c0+esx*c)*eiwt)';
ype=real(s.*esl*c*eiwt)';
m=real(ei*s2(ones(nx,1),:).*esx*c*eiwt)';
v=real(ei*s3(ones(nx,1),:).*esx*c*eiwt)';
t=t'; x=x'; hold off; clf;

% Make surface plots showing the deflection, moment, and shear over a complete period of
% the motion
surf(x,t*w,y);
xlabel('x axis'); ylabel('t*w axis');
zlabel('transverse deflection');
title('Deflection Surface for a Vibrating Pile');
grid on; figure(gcf)
% print -deps pilesurf
disp('Press [Enter] to continue'), pause

surf(x,t*w,m);
xlabel('x axis'); ylabel('t*w axis');
zlabel('bending moment');
title('Bending Moment in the Pile')
grid on; figure(gcf)
% print -deps pilemom;
disp('Press [Enter] to continue'), pause

surf(x,t*w,v);
xlabel('x axis'); ylabel('t*w axis');
zlabel('shear force');
title('Shear Force in the Pile');
grid on; figure(gcf)
% print -deps pilesher
disp('Press [Enter] to see animation'), pause

% Draw an animation depicting the pile response
% to the oscillation of the foundation
fu=.10/max(y(:)); p=[-0.70, 0.70, -.1, 1.3];
u=fu*y; upe=fu*L*ype; d=.15;
xm=[0,0,1,1,0,0]*d;
ym=[0,-1,-1,1,1,0]*d; zm=xm+i*ym;
close;
for jj=1:4
  for j=1:nt
    z=exp(i*atan(upe(j)))*zm;
    xx=real(z); yy=imag(z);
    ut=[u(j,:),u(j,nx)+yy]; xt=[x/L,1+xx];
    plot(ut,xt,'-'); axis(p); axis('square');
    title('Forced Vibration of a Pile');
    axis('off'); drawnow; figure(gcf);
  end
end
% print -deps pileanim
fprintf('
All Done
');

© 2003 by CRC Press LLC
9.8 Transient Heat Conduction in a One-Dimensional Slab

Let us analyze the temperature history in a slab which has the left side insulated while the right side temperature varies sinusoidally according to \( U_0 \sin(\Omega T) \). The initial temperature in the slab is specified to be zero. The pertinent boundary value problem is

\[
\alpha \frac{\partial^2 U}{\partial X^2}(X,T) = \frac{\partial U}{\partial T}(X,T), \quad 0 < X < \ell, \quad T > 0,
\]

\[
\frac{\partial U}{\partial X}(0, T) = 0, \quad U(\ell, T) = U_0 \sin(\Omega T),
\]

\[ U(X, 0) = 0, \quad 0 < X < \ell \]

where \( U, X, T, \) and \( \alpha \) are, respectively, the temperature, position, time, and thermal diffusivity.

The problem can be converted to dimensionless form by letting

\[
u = \frac{U}{U_0}, \quad x = \frac{X}{\ell}, \quad t = \frac{\alpha T}{\ell^2}, \quad \omega = \frac{\Omega^2}{\alpha}.
\]

Then we get

\[
\frac{\partial^2 \nu}{\partial x^2} = \frac{\partial \nu}{\partial t}, \quad 0 < x < 1, \quad t > 0,
\]

\[
\frac{\partial \nu}{\partial x}(0, t) = 0, \quad \nu(1, t) = \text{imag} \left( e^{i\omega t} \right), \quad \nu(x, 0) = 0.
\]

The solution consists of two parts as \( \nu = w + v \), where \( w \) is a particular solution satisfying the differential equation and nonhomogeneous boundary conditions, and \( v \) is a solution satisfying homogeneous boundary conditions and specified to impose the desired zero initial temperature when combined with \( w \). The appropriate form for the particular solution is

\[
w = \text{imag} \left[ f(x) e^{i\omega t} \right].
\]

Making \( w \) satisfy the heat equation requires

\[
f''(x) = if(x).
\]

Consequently

\[
f(x) = c_1 \sin(\phi x) + c_2 \cos(\phi x)
\]

where \( \phi = \sqrt{-i\omega} \). The conditions of zero gradient at \( x = 0 \) and unit function value at \( x = 1 \) determine \( c_1 \) and \( c_2 \). We get the particular solution as

\[
w = \text{imag} \left[ \frac{\cos(\phi x)}{\cos(\phi)} e^{i\omega t} \right].
\]
This forced response solution evaluated at \( t = 0 \) yields

\[
 w(x, 0) = \text{imag} \left[ \frac{\cos(\phi x)}{\cos(\phi)} \right].
\]

The general solution of the heat equation satisfying zero gradient at \( x = 0 \) and zero function value at \( x = 1 \) is found to be

\[
 v(x, t) = \sum_{n=1}^{\infty} a_n \cos(\lambda_n x) e^{-\lambda_n^2 t}
\]

where \( \lambda_n = \pi (2n - 1)/2 \). To make the initial temperature equal zero in the combined solution, the coefficients \( a_n \) are chosen to satisfy

\[
 \sum_{n=1}^{\infty} a_n \cos(\lambda_n x) = -\text{imag} \left[ \frac{\cos(\phi x)}{\cos(\phi)} \right].
\]

The orthogonality of the functions \( \cos(\lambda_n x) \) implies

\[
 a_n = -2 \int_0^1 \text{imag} \left[ \frac{\cos(\phi x)}{\cos(\phi)} \right] \cos(\lambda_n x) dx
\]

which can be integrated to give

\[
 a_n = -\text{imag} \left[ \frac{(\sin(\lambda_n + \phi)/(\lambda_n + \phi) + \sin(\lambda_n - \phi)/(\lambda_n - \phi))}{\cos(\phi)} \right].
\]

This completely determines the solution. Taking any finite number of terms in the series produces an approximate solution exactly satisfying the differential equation and boundary conditions. Exact satisfaction of the zero initial condition would theoretically require an infinite number of series terms. However, the terms in the series decrease like \( O(1/n^3) \) and using a 250-term series produces initial temperature values not exceeding \( 10^{-6} \). Thus, the finite series is satisfactory for practical purposes.

The above equations were evaluated in a function called \texttt{heat}. Function \texttt{slab-heat} was also written to plot numerical results. The code and resulting Figures 9.23 and 9.24 appear below. This example illustrates nicely how well MATLAB handles complex arithmetic and complex valued functions.
Figure 9.22: Temperature Variation in a Slab

Figure 9.23: Temperature History at Ends and Middle
function slabheat
% Example: slabheat
% ~~~~~~~~~~~~~~~~~
% This program computes the temperature
% variation in a one-dimensional slab with
% the left end insulated and the right end
% given a temperature variation sin(w*t).
% User m functions required:  heat
[u1,t1,x1]=heat(12,0,2,50,0,1,51,250);
surf(x1,t1,u1); axis([0 1 0 2 -2 2]);
title('Temperature Variation in a Slab');
xlabel('x axis'); ylabel('time');
zlabel('temperature'); view([45,30])
colormap('default'), shg
disp(' '), disp('Press [Enter] to continue')
pause
% print -deps tempsurf
[u2,t2,x2]=heat(12,0,2,150,0,1,3,250);
plot(t2,u2(:,1),--,t2,u2(:,2),:',,...
t2,u2(:,3),-');
title(['Temperature History at Ends' ...'
and Middle']);
xlabel('dimensionless time');
ylabel('dimensionless temperature');
text1='Left End'; text2='Middle';
text3='Right End';
legend(text1,text2,text3,3); shg
% print -deps templot
disp(' '), disp('All Done');

function [u,t,x]= ...
    heat(w,tmin,tmax,nt,xmin,xmax,nx,nsum)
% [u,t,x]=heat(w,tmin,tmax,nt,xmin,xmax,nx,nsum)
% This function evaluates transient heat
% conduction in a slab which has the left end
% (x=0) insulated and has the right end (x=1)
% subjected to a temperature variation
% sin(w*t). The initial temperature of the slab
% is zero.

% w - frequency of the right side
% temperature variation
% tmin, tmax - time limits for solution
% nt - number of uniformly spaced
% time values used
% xmin, xmax - position limits for solution.
% Values should lie between zero and one.
% nx - number of equidistant x values
% nsum - number of terms used in the
% series solution
% u - matrix of temperature values.
% Time varies from row to row.
% x varies from column to column.
% t, x - vectors of time and x values

% User m functions called: none.

%----------------------------------------------

t=tmin+(tmax-tmin)/(nt-1)*(0:nt-1);
x=xmin+(xmax-xmin)/(nx-1)*(0:nx-1)’;
W=sqrt(-i*w); ln=pi*((1:nsum)-1/2);
v1=ln+W; v2=ln-W;
a=-imag((sin(v1)./v1+sin(v2)./v2)/cos(W));
u=imag(cos(W*x)*exp(i*w*t)/cos(W))+ ...
(a(ones(nx,1),:).*cos(x*ln))* ...
exp(-ln(:).^2*t);
u=u’; t=t(:);

9.9 Transient Heat Conduction in a Circular Cylinder with Spatially Varying Boundary Temperature

9.9.1 Problem Formulation

Transient heat conduction in a circular cylinder can be analyzed using an infinite
series of Bessel functions. Consider a cylinder having an initial temperature distri-
bution \( u_0(r, \theta) \) when the boundary is suddenly given a temperature variation \( f(\theta) \) depending on the polar angle but independent of time. The problem is conveniently formulated in polar coordinates using dimensionless radius and time variables. The differential equation, boundary conditions, and initial conditions are as follows:

\[
\begin{align*}
  u_{rr} + \frac{1}{r} u_r + \frac{1}{r^2} u_{\theta \theta} &= u_t, \quad 0 \leq r \leq 1, \ t > 0, \\
  u(1, \theta, t) &= f(\theta) = \sum_{n=-\infty}^{\infty} f_n e^{in\theta}, \quad 0 \leq \theta \leq 2\pi, \\
  u(r, \theta, 0) &= u_0(r, \theta), \quad 0 \leq r \leq 1, \ 0 \leq \theta \leq 2\pi.
\end{align*}
\]

With the boundary condition expressed as a complex Fourier series, the steady-state solution satisfying the differential equation and the boundary conditions is

\[
v(r, \theta) = -f_0 + 2 \text{ real} \left( \sum_{n=0}^{\infty} f_n z^n \right) \text{ where } z = re^{i\theta}.
\]

The total solution is the steady-state solution combined with a transient solution \( w(r, \theta, t) \) chosen to satisfy the initial condition and boundary conditions expressed as

\[
w(r, \theta, 0) = u_0(r, \theta) - v(r, \theta), \ w(1, \theta, t) = 0.
\]

The transient solution is a Fourier-Bessel series involving double subscripted coefficients depending on the functions \( v(r, \theta) \) and \( u_0(r, \theta) \). It is found that

\[
w(r, \theta, t) = \sum_{n=0}^{\infty} \sum_{k=1}^{\infty} J_n(\lambda_{nk}r) \left[ A_{nk} \cos(n\theta) + B_{nk} \sin(n\theta) \right] \exp\left(-\lambda_{nk}^2 t\right)
\]

where, for \( n > 0 \) and \( k \geq 1 \), we have

\[
A_{nk} + iB_{nk} = C_{nk} = \frac{2}{\pi J_{n+1}(\lambda_{nk})} \int_0^1 \int_0^{2\pi} w(r, \theta, 0) r J_n(\lambda_{nk}r) \exp(in\theta) dr d\theta
\]

with \( \lambda_{nk} \) denoting the \( k^{th} \) positive root of \( J_n(r) \). The last formula almost applies for \( n = 0 \) except that \( A_{0k} = C_{0k}/2 \) and \( B_{0k} = 0 \). The coefficients for \( n = 0 \) pertain to the radially symmetric case independent of the polar angle. Evaluating this series solution involves several steps which are: 1) Expanding the boundary condition in a complex Fourier series to obtain the steady-state solution; 2) Determining the zeros of the integer order Bessel functions \( J_n(r) \); 3) Computing the series coefficients by numerical integration; and 4) Summing the series solution for various \( (r, \theta) \) values with enough terms being used in the series to assure adequate satisfaction of the initial conditions and boundary conditions.
9.9.2 Computer Formulation

A computer program was written to analyze the time dependent temperature field. The program specifies general initial temperature and boundary temperature. The series solution is evaluated on a polar coordinate grid and an animation of the temperature variation from initial to steady state is shown. The program modules include: 1) heatcyl which calls the computational modules and plots results; 2) besjtabl returns Bessel function roots used in the series solution; 3) tempinit specifies the initial temperature field; 4) tempstgy computes the steady state solution; 5) tempdif computes the difference in the initial and the final temperature fields; 6) foubesco evaluates coefficients in the Fourier-Bessel series; and (7) tempsum sums the Fourier-Bessel series for a vector of time values. Figures 9.25 through 9.28 show the initial, final, and two intermediate temperature states. The program animates the temperature history so the transition from initial to steady-state can be visualized.
Figure 9.25: Temperature at t=0.02

Figure 9.26: Temperature at t=0.05
Figure 9.27: Steady State Temperature Distribution
Program heatcyln

function heatcyln

% heatcyln
% ~~~~~~~~~
% This program analyzes the time varying temperature
% history in a circular cylinder which initially has
% a radially symmetric temperature varying para-
% bolically. Then a spatially varying but constant
% boundary temperature distribution is imposed. The
% total solution is composed of a harmonic steady
% state solution plus a transient component given by
% a Fourier-Bessel series.
% User functions called:
%   besjtabl, tempinit, tempstdy, foubesco,
%   tempsum, tempdif, gcquad

global ubdry besjrt

% Obtain Bessel function roots needed in the
% transient solution
besjrt=besjtabl(0:20,20);

% Define the steady state temperature imposed
% on the outer boundary for t>0
th=linspace(0,pi,100)';
ud=cos(2*th).*(th<=pi/2)+...
  (-3+4/pi*th).*((th>pi/2&th<3*pi/4);
ud=[ud;ud(end-1:-1:1)];
ubdry=[linspace(0,360,199)',ud];
theta=linspace(0,2*pi,65);
r=linspace(0,1,15);

% Compute and plot the initial and final
% temperature fields
[uinit,z]=tempinit(theta,r);
[usteady,z]=tempstdy(theta,r);
umin=min([usteady(:);uinit(:)]);
ymax=max([usteady(:);uinit(:)]);
range=[-1,1,-1,1,umin,ymax];
x=real(z); y=imag(z);
surf(x,y,uinit), colormap('default')
title('INITIAL TEMPERATURE DISTRIBUTION')
xlabel('x axis'), ylabel('y axis')
zlabel('temperature'), axis(range), disp(' ') 
disp('Press [Enter] to see the steady')
disp('state temperature distribution')
shg, pause, disp(' ')
% print -deps tempinit

surf(x,y,usteady)
title('STEADY STATE TEMPERATURE DISTRIBUTION')
xlabel('x axis'), ylabel('y axis')
zlabel('temperature'), axis(range), shg
% print -deps tempstdy

% Compute coefficients used in the Fourier-
% Bessel series for the transient solution
[c,lam,cptim]=foubesco(@tempdif,20,20,40,128);

% Set a time interval sufficient to nearly
% reach steady state
tmax=.4; nt=81; t=linspace(0,tmax,nt);

% Evaluate the transient solution
[u,tsum]=tempsum(c,theta,r,t,lam);
u(:,:,1)=uinit-usteady;

% Plot time history for the total solution
while 1
    disp('Press [Enter] to see the animation')
disp('or enter 0 to stop'), v=input('> ? ');
    if isempty(v), v=1; end
    if v~=1, break, end
    for j=1:nt
        utotal=usteady+u(:,:,j);
surf(x,y,utotal)
        titl=sprintf(['Temperature at time =',... '
           ', '%6.3f'],t(j)); title(titl)
xlabel('x axis'), ylabel('y axis')
zlabel('temperature'), axis(range);
drawnow; shg, pause(.3)
    end
end
%=============================================
function [u,z]=tempstdy(theta,r)
% Steady state temperature distribution in a circular cylinder of unit radius with piecewise linear boundary values described in global array ubdry.

thft=2*pi/(1024)*(0:1023); n=100;
ufft=interp1(pi/180*ubdry(:,1),ubdry(:,2)/1024,thft);
c=fft(ufft); z=exp(i*theta(:))*r(:)';
u=-real(c(1))+2*real(polyval(c(n:-1:1),z));

%=============================================

function [u,z]=tempinit(theta,r)
% Initial temperature varying parabolically with the radius
theta=theta(:); r=r(:)'; z=exp(i*theta)*r;
u=ones(length(theta),1)*(1-r.^2);

%=============================================

function [u,z]=tempdif(theta,r)
% Difference between the steady state temperature and the initial temperature
u1=tempstdy(theta,r); [u2,z]=tempinit(theta,r);
u=u2-u1;

%=============================================

function [c, lam, ctim]=foubesco(...
  f,nord,nrts,nrquad,nft)
% [c,lam,cptim]=foubesco(f,nord,nrts,nrquad,nft)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Fourier-Bessel coefficients computed using the
% FFT
global besjrt
if nargin<5, nft=128; end
if nargin<4, nrquad=50; end
if nargin<3, nrts=10; end
if nargin<2, nord=10; end
if nargin==0, f='fbes'; end
tic; lam=besjrt(1:nord,1:nrts);
c=zeros(nord,nrts);
[dummy,r,w]=gcquad([],0,1,nrquad,1);
r=r(:)'; w=w(:)'; th=2*pi/nft*(0:nft-1)'
fmt=fft(feval(f,th,r));
fmt=fmt(1:nord,:).*repmat(r.*w,nord,1);
for n=1:nord
  for k=1:nrts
    lnk=lam(n,k);
    v=sum(fmt(n,:).*besselj(n-1,lnk*r));
    c(n,k)=4*v/nft/besselj(n,lnk).^2;
  end
end
c(1,:)=c(1,:)/2; cptim=toc;

%=============================================

function [u,tcpu]=tempsum(c,th,r,t,lam)
%
% [u,tsum]=tempsum(c,th,r,t,lam)
%
% This function sums a Fourier-Bessel series
% for transient temperature history in a circular
% cylinder with given initial conditions and
% zero temperature at the boundary. The series
% has the form
% u(theta,r,t)=sum({n=0:nord-1,k=1:nrts},...
% besselj(n,lam(n+1,k)*r)*real(...
% c(n+1,k)*exp(i*(n+1)*theta))*...
% exp(-lam(n+1,k)^2*t), where
% besselj(n-1,lon(n,k))=0 and
% [nord,nrts]=size(c)
%
% c - the series coefficients for the initial
% temperature distribution obtained using

© 2003 by CRC Press LLC
function foubesco
% th - vector or theta values between zero and 2*pi
% r - vector of radius values between zero and one
% lam - matrix of bessel function roots.
% If this argument is omitted, then function besjroot is called to compute the roots
% u - a three-dimensional array of function values where u(i,j,k) contains the temperature for theta(i), r(j), t(k)
% tcpu - computation time in seconds

tic; [nord,nrts]=size(c);
if nargin<5, lam=besjroot(0:nord-1,nrts); end
th=th(:); nth=length(th); r=r(:)'; nr=length(r);
nt=length(t); N=repmat((0:nord-1)',1,nrts);
N=N(:)'; c=c(:)'; lam=lam(:); lam2=-(lam.^2)';
u=zeros(nth,nr,nt); thmat=exp(i*th*N);
besmat=besselj(repmat(N',1,nr),lam*r);
for I=1:nt
C=c.*exp(lam2*t(I));
u(:,:,I)=real(thmat.*repmat(C,nth,1))*besmat;
end
tcpu=toc;

%=============================================%-------------
% function r=besjtabl(nordr,nrts)
% This function returns a table for roots of besselj(n,x)=0 accurate to about five digits.
% r(k,:) - contains the first 20 positive roots of besselj(k-1,x)=0; for k=1:21
% nordr - a vector of function orders lying between 0 and 20
% nrts - the highest root order not to exceed the twentieth positive root
if nargin==0, nordr=0:20; nrts=20; end
if max(nordr)>20 | nrts>20, r=nan; return; end
r=[2.4048 21.6415 40.7729 33.7758 53.7383 73.2731
© 2003 by CRC Press LLC
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>222</td>
<td>3.8317</td>
<td>22.9452</td>
<td>42.0679</td>
<td>35.3323</td>
<td>55.1847</td>
</tr>
<tr>
<td>223</td>
<td>5.1356</td>
<td>24.2339</td>
<td>43.3551</td>
<td>36.8629</td>
<td>56.6196</td>
</tr>
<tr>
<td>224</td>
<td>6.3801</td>
<td>25.5094</td>
<td>44.6349</td>
<td>38.3705</td>
<td>58.0436</td>
</tr>
<tr>
<td>225</td>
<td>7.5883</td>
<td>26.7733</td>
<td>45.9076</td>
<td>39.8577</td>
<td>59.4575</td>
</tr>
<tr>
<td>226</td>
<td>8.7715</td>
<td>28.0267</td>
<td>47.1740</td>
<td>41.3263</td>
<td>60.8617</td>
</tr>
<tr>
<td>227</td>
<td>9.9362</td>
<td>29.2706</td>
<td>48.4345</td>
<td>42.7784</td>
<td>62.2071</td>
</tr>
<tr>
<td>228</td>
<td>11.0864</td>
<td>30.5060</td>
<td>24.3525</td>
<td>44.2154</td>
<td>63.6441</td>
</tr>
<tr>
<td>229</td>
<td>12.2251</td>
<td>31.7334</td>
<td>25.9037</td>
<td>45.6384</td>
<td>65.0231</td>
</tr>
<tr>
<td>230</td>
<td>13.3543</td>
<td>32.9537</td>
<td>27.4206</td>
<td>47.0487</td>
<td>66.3943</td>
</tr>
<tr>
<td>231</td>
<td>14.4755</td>
<td>34.1672</td>
<td>28.9084</td>
<td>48.4475</td>
<td>67.7586</td>
</tr>
<tr>
<td>232</td>
<td>15.5898</td>
<td>35.3747</td>
<td>30.3710</td>
<td>49.8346</td>
<td>69.1159</td>
</tr>
<tr>
<td>233</td>
<td>16.6983</td>
<td>36.5764</td>
<td>31.8117</td>
<td>51.2120</td>
<td>70.4668</td>
</tr>
<tr>
<td>234</td>
<td>17.8014</td>
<td>37.7729</td>
<td>33.2330</td>
<td>52.5798</td>
<td>71.8113</td>
</tr>
<tr>
<td>235</td>
<td>18.9000</td>
<td>38.9699</td>
<td>34.6371</td>
<td>53.9382</td>
<td>73.1571</td>
</tr>
<tr>
<td>236</td>
<td>19.9944</td>
<td>40.1699</td>
<td>36.0257</td>
<td>55.2892</td>
<td>74.4939</td>
</tr>
<tr>
<td>237</td>
<td>21.0852</td>
<td>41.3699</td>
<td>37.4001</td>
<td>56.6319</td>
<td>75.8297</td>
</tr>
<tr>
<td>238</td>
<td>22.1725</td>
<td>42.5699</td>
<td>38.7618</td>
<td>57.9672</td>
<td>77.1655</td>
</tr>
<tr>
<td>239</td>
<td>23.2568</td>
<td>43.7699</td>
<td>40.1118</td>
<td>59.2953</td>
<td>78.4995</td>
</tr>
<tr>
<td>240</td>
<td>24.3383</td>
<td>45.0699</td>
<td>41.4511</td>
<td>60.6170</td>
<td>79.8335</td>
</tr>
<tr>
<td>241</td>
<td>25.4171</td>
<td>46.3699</td>
<td>42.7804</td>
<td>61.9323</td>
<td>81.1675</td>
</tr>
<tr>
<td>242</td>
<td>26.5598</td>
<td>47.6699</td>
<td>44.1006</td>
<td>63.2476</td>
<td>82.4965</td>
</tr>
<tr>
<td>243</td>
<td>27.6979</td>
<td>48.9699</td>
<td>45.4122</td>
<td>64.5530</td>
<td>83.8285</td>
</tr>
<tr>
<td>244</td>
<td>28.8317</td>
<td>50.2699</td>
<td>46.7158</td>
<td>65.8589</td>
<td>85.1585</td>
</tr>
<tr>
<td>245</td>
<td>29.9616</td>
<td>51.5699</td>
<td>48.0122</td>
<td>67.1644</td>
<td>86.4885</td>
</tr>
<tr>
<td>246</td>
<td>31.0852</td>
<td>52.8699</td>
<td>49.3001</td>
<td>68.4699</td>
<td>87.8185</td>
</tr>
<tr>
<td>247</td>
<td>32.2096</td>
<td>54.1699</td>
<td>50.5880</td>
<td>69.7752</td>
<td>89.1485</td>
</tr>
<tr>
<td>248</td>
<td>33.3331</td>
<td>55.4699</td>
<td>51.8660</td>
<td>71.0807</td>
<td>90.4785</td>
</tr>
<tr>
<td>249</td>
<td>34.4566</td>
<td>56.7699</td>
<td>53.1440</td>
<td>72.3860</td>
<td>91.8085</td>
</tr>
<tr>
<td>250</td>
<td>35.5790</td>
<td>58.0699</td>
<td>54.4220</td>
<td>73.6910</td>
<td>93.1385</td>
</tr>
<tr>
<td>251</td>
<td>36.6914</td>
<td>59.3699</td>
<td>55.6999</td>
<td>74.9960</td>
<td>94.4685</td>
</tr>
<tr>
<td>252</td>
<td>37.8038</td>
<td>60.6699</td>
<td>56.9779</td>
<td>76.2910</td>
<td>95.7985</td>
</tr>
<tr>
<td>253</td>
<td>38.9162</td>
<td>61.9699</td>
<td>58.2559</td>
<td>77.5860</td>
<td>97.1285</td>
</tr>
<tr>
<td>254</td>
<td>40.0286</td>
<td>63.2699</td>
<td>59.5339</td>
<td>78.8810</td>
<td>98.4585</td>
</tr>
<tr>
<td>255</td>
<td>41.1410</td>
<td>64.5699</td>
<td>60.8119</td>
<td>80.1760</td>
<td>99.7885</td>
</tr>
<tr>
<td>256</td>
<td>42.2534</td>
<td>65.8699</td>
<td>62.0899</td>
<td>81.4710</td>
<td>101.1185</td>
</tr>
<tr>
<td>257</td>
<td>43.3658</td>
<td>67.1699</td>
<td>63.3679</td>
<td>82.7660</td>
<td>102.4485</td>
</tr>
<tr>
<td>258</td>
<td>44.4782</td>
<td>68.4699</td>
<td>64.6459</td>
<td>84.0610</td>
<td>103.7785</td>
</tr>
<tr>
<td>259</td>
<td>45.5906</td>
<td>69.7699</td>
<td>65.9239</td>
<td>85.3560</td>
<td>105.1085</td>
</tr>
<tr>
<td>260</td>
<td>46.7030</td>
<td>71.0699</td>
<td>67.2019</td>
<td>86.6510</td>
<td>106.4385</td>
</tr>
<tr>
<td>261</td>
<td>47.8154</td>
<td>72.3699</td>
<td>68.4799</td>
<td>87.9460</td>
<td>107.7685</td>
</tr>
<tr>
<td>262</td>
<td>48.9278</td>
<td>73.6699</td>
<td>69.7579</td>
<td>89.2410</td>
<td>109.0985</td>
</tr>
<tr>
<td>263</td>
<td>50.0402</td>
<td>74.9699</td>
<td>71.0359</td>
<td>90.5360</td>
<td>110.4285</td>
</tr>
<tr>
<td>264</td>
<td>51.1526</td>
<td>76.2699</td>
<td>72.3139</td>
<td>91.8310</td>
<td>111.7585</td>
</tr>
<tr>
<td>265</td>
<td>52.2650</td>
<td>77.5699</td>
<td>73.5919</td>
<td>93.1260</td>
<td>113.0885</td>
</tr>
<tr>
<td>266</td>
<td>53.3774</td>
<td>78.8699</td>
<td>74.8699</td>
<td>94.4210</td>
<td>114.4185</td>
</tr>
</tbody>
</table>
9.10 Torsional Stresses in a Beam of Rectangular Cross Section

Elastic beams of uniform cross section are commonly used structural members. Evaluation of the stresses caused when beams undergo torsional moments depends on finding a particular type of complex valued function. This function is analytic inside the beam cross section and has its imaginary part known on the boundary [72]. The shear stresses $\tau_{XZ}$ and $\tau_{YZ}$ are obtained from the stress function $f(z)$ of the complex variable $z = x + iy$ according to

$$\frac{\tau_{XZ} - i\tau_{YZ}}{\mu\alpha} = f'(z) - iz$$
where \( \mu \) is the shear modulus and \( \alpha \) is the twist per unit length. In the case for a simply connected cross section, such as a rectangle or a semicircle, the necessary boundary condition is

\[
\text{imag}[f(z)] = \frac{1}{2} |z|^2
\]

at all boundary points. It can also be shown that the torsional moment causes the beam cross section to warp. The warped shape is given by the real part of \( f(z) \).

The geometry we will analyze is rectangular. As long as the ratio of side length remains fairly close to unity, \( f(z) \) can be well approximated by

\[
f(z) = i \sum_{j=1}^{n} c_j \left( \frac{z}{s} \right)^{2j+2}
\]

where \( c_1, \ldots, c_n \) are real coefficients computed to satisfy the boundary conditions in the least square sense. The parameter \( s \) is used for scaling to prevent occurrence of large numbers when \( n \) becomes large. We take a rectangle with sides parallel to the coordinate axes and assume side lengths of \( 2a \) and \( 2b \) for the horizontal and vertical directions, respectively. The scaling parameter will be chosen as the larger of \( a \) and \( b \). The boundary conditions state that for any point \( z_i \) on the boundary we should have

\[
\sum_{j=1}^{n} c_j \text{real} \left[ \left( \frac{z_i}{s} \right)^{2j+2} \right] = \frac{1}{2} |z_i|^2.
\]

Once the series coefficients are found, then shear stresses are computed as

\[
\frac{\tau_{XZ} - i\tau_{YZ}}{\mu \alpha} = -i \tilde{z} + 2i s^{-1} \sum_{j=2}^{n} (j-1) c_j \left( \frac{z}{s} \right)^{2j+3}
\]

A program was written to compute stresses in a rectangular beam and to show graphically the cross section warping and the dimensionless stress values. The program is short and the necessary calculations are almost self explanatory. It is worthwhile to observe, however, the ease with which MATLAB handles complex functions. Note how intrinsic function \texttt{linspace} is used to generate boundary data and \texttt{meshgrid} is used to generate a grid of complex values (see lines 50, 51, 72, 73, and 74 of function \texttt{recstrs}). The sample problem employs a rectangle of dimension 2 units by 4 units. The maximum stress occurs at the middle of the longest side. Figures 9.28 through 9.31 plot the results of this analysis.
Figure 9.28:  Warping of the Cross Section

Figure 9.29:  Total Shear Stress Surface
Figure 9.30: Total Stress Contours

Figure 9.31: Stress for $y = b/2$
MATLAB Example

Output from Torsion Example

>> rector;

=== TORSIONAL STRESS CALCULATION IN A RECTANGULAR ===
=== BEAM USING LEAST SQUARE APPROXIMATION ===

Input the lengths of the horizontal and the vertical sides
(make the long side horizontal)
> 3,2

Input the number of terms used in the stress function
(30 terms is usually enough)
> 30

Press [Enter] to plot
the warping surface

Press [[Enter]] to plot the
total stress surface

Press [Enter] to plot the
stress contours

Press [Enter] to plot the maximum
stress on a rectangle side

The Maximum Shear Stress is 1.6951
at x = 0 and y = 1

All Done

>>

Program rector

function rector
% Example: rector
% ~~~~~~~~~~~~~~~~
% This program uses point matching to obtain an
% approximate solution for torsional stresses
% in a Saint Venant beam having a rectangular
% cross section. The complex stress function is

© 2003 by CRC Press LLC
analytic inside the rectangle and has its real part equal to \( \text{abs}(z^2)/2 \) on the boundary. The problem is solved approximately using a polynomial stress function which fits the boundary condition in the least square sense. Surfaces and contour curves describing the stress and deformation pattern in the beam cross section are drawn.

User m functions required: recstrs

clear;
fprintf('
=== TORSIONAL STRESS CALCULATION');
fprintf(' IN A RECTANGULAR ===
');
fprintf('BEAM USING LEAST SQUARE APPROXIMATION ===
');
fprintf('
Input the lengths of the horizontal and the vertical sides
(make the long side horizontal)
');
u=input('> ? ','s'); u=eval(['[',u,']']);
a=u(1)/2; b=u(2)/2;
%
% The boundary conditions are approximated in terms of the number of least square points used along the sides
nsegb=100; nsega=ceil(a/b*nsegb);
nsega=fix(nsega/2); nsegb=fix(nsegb/2);
fprintf('
Input the number of terms used in the stress function
(30 terms is usually enough)
');
ntrms=input('> ? '); u=eval(['[',u,']']);
a=u(1)/2; b=u(2)/2;
%
% Define a grid for evaluation of stresses.
% Include the middle of each side.
nx=41; ny=fix(b/a*nx); ny=ny+1-rem(ny,2);
[c,phi,stres,z] = ... recstrs(a,nsega,b,nsegb,ntrms,nx,ny);
[smax,k]=max(abs(stres(:))); zmax=z(:);
zmax=zmax(k); xmax=abs(real(zmax));
ymax=abs(imag(zmax));
disp(''), disp(['The Maximum Shear ','Stress is ','num2str(smax)]);
disp(['at x = ','num2str(xmax),' and y = ','... num2str(ymax)]);
disp( ' ' ); disp( 'All Done' );

%=======================================================================

function [c,phi,stres,z]=...
recstrs(a,nsega,b,nsegb,ntrms,nxout,nyout)
% [c,phi,stres,z]=...
% recstrs(a,nsega,b,nsegb,ntrms,nxout,nyout)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function uses least square fitting to
% obtain an approximate solution for torsional
% stresses in a Saint Venant beam having a
% rectangular cross section. The complex stress
% function is analytic inside the rectangle
% and has its real part equal to abs(z*z)/2 on
% the boundary. The problem is solved
% approximately using a polynomial stress
% function which fits the boundary condition
% in the least square sense. The beam is 2*a
% wide parallel to the x axis and 2*b deep
% parallel to the y axis. The shear stresses
% in the beam are given by the stress formula:
% (tauza-i*tauzy)/(mu*alpha) = -i*conj(z)+f'(z)
% where
% f(z)=i*sum( c(j)*z^(2*j-2), j=1:ntrms )
% and c(j) are real.
% a,b          - half the side lengths of the
%    horizontal and vertical sides
% nsega,       - numbers of subintervals used to
% nsegb        - form the least square equations
% ntrms        - number of terms used in the
%    polynomial stress function
% nxout,       - number of grid points used to
%    evaluate output
% c            - coefficients defining the stress
%    function
% phi          - values of the membrane function
% stres        - array of complex stress values
% z            - complex point array at which
% stresses are found
% User m functions called: none
%----------------------------------------------

% Generate vector zbdry of boundary points
% for point matching.
zbdry=[a+i*b/nsega*(0:nsega-1)';
      i*b+a/nsegb*(nsegb:-1:0)'];

% Determine a scaling parameter used to
% prevent occurrence of large numbers when
% high powers of z are used
s=max(a,b);

% Form the least square equations to impose
% the boundary conditions.
neq=length(zbdry); amat=ones(neq,ntrms);
ztmp=(zbdry/s).^2; bvec=.5*abs(zbdry).^2;
for j=2:ntrms
  amat(:,j)=amat(:,j-1).*ztmp;
end

% Solve the least square equations.
amat=real(amat); c=pinv(amat)*bvec;

% Generate grid points to evaluate
% the solution.
xsid=linspace(-a,a,nxout);
ysid=linspace(-b,b,nyout);
xg=xsid; yg=ysid;
z=xg+i*yg; zz=(z/s).^2;

% Evaluate the warping function
phi=-imag(polyval(flipud(c),zz));

% Evaluate stresses and plot results
cc=(2*(1:ntrms)-2)'.*c;
stres=-i*conj(z)+i* ...
       polyval(flipud(cc),zz)./(z+eps*(z==0));

% Plot results
disp(' '), disp('Press [Enter] to plot')
disp(‘the warping surface’), pause
[pa,k]=max(abs(phi(:)));
Phi=a/4*sign(phi(k))/phi(k)*phi;
close, colormap(‘default’)
surfc(xg,yg,Phi)
title(‘Warping of the Cross Section’)
xlabel(‘x axis’), ylabel(‘y axis’)
zlabel(‘transverse warping’); axis(‘equal’)
shg, disp(‘ ’)
disp(‘Press [[Enter]] to plot the’)
disp(‘total stress surface’), pause
% print -deps warpsurf

surfc(xg,yg,abs(stres));
title(‘Total Shear Stress Surface’)
xlabel(‘x axis’); ylabel(‘y axis’)
zlabel(‘total stress’), axis(‘equal’), shg
disp(‘ ’), disp(‘Press [Enter] to plot the’)
disp(‘stress contours’), pause
% print -deps rectorst

contour(xg,yg,abs(stres),20); colorbar
title(‘Total Stress Contours’);
xlabel(‘x axis’); ylabel(‘y axis’)
shg, disp(‘ ’)
disp(‘Press [Enter] to plot the maximum’)
disp(‘stress on a rectangle side’), pause
% print -deps torcontu

plot(xsid,abs(stres(1,:)),’k’);
grid; ylabel(‘tangential stress’);
xlabel(‘position on a horizontal side’);
title(‘Stress for y = b/2’); shg
% print -deps torstsid
Chapter 10

Eigenvalue Problems and Applications

10.1 Introduction

Eigenvalue problems occur often in mechanics, especially linear system dynamics, and elastic stability. Usually nontrivial solutions are sought for homogeneous systems of differential equations. For a few simple systems like the elastic string, or a rectangular membrane, the eigenvalues and eigenfunctions can be determined exactly. More often, some discretization methods such as finite difference or finite element methods are employed to reduce the system to a linear algebraic form which is numerically solvable. Several eigenvalue problems analyzed in earlier chapters reduced easily to algebraic form where the function eig could immediately produce the desired results. The present chapter deals with several instances where reduction to eigenvalue problems is more involved. We will also make some comparisons of exact, finite difference, and finite element analyses. Among the physical systems studied are Euler beams and columns, two-dimensional trusses, and elliptical membranes.

10.2 Approximation Accuracy in a Simple Eigenvalue Problem

One of the simplest but useful eigenvalue problems concerns determining nontrivial solutions of

\[ y''(x) + \lambda^2 y(x) = 0, \; y(0) = y(1) = 0. \]

The eigenvalues and eigenfunctions are

\[ y_n = \sin(n\pi x), \; 0 \leq x \leq 1, \; \text{where} \; \lambda_n = n\pi, \; n = 1, 2, 3, \ldots \]

It is instructive to examine the answers obtained for this problem using finite differences and spline approximations. We introduce a set of node points defined by

\[ x_j = j\Delta, \; j = 0, 1, 2, \ldots, N + 1, \; \Delta = 1/(N + 1). \]
Then a finite difference description for the differential equation and boundary conditions is

\[ y_{j-1} - 2y_j + y_{j+1} + \omega^2 y_j = 0, \quad 1 \leq j \leq N, \quad y_0 = y_{N+1} = 0, \quad \omega = \Delta \lambda. \]

Solving the linear difference equation gives

\[ \lambda^d_n = 2(N + 1) \sin \left( \frac{\pi n}{2(N + 1)} \right), \quad n = 1, \ldots, N, \]

\[ y^d_j = \sin \left( \frac{\pi j n}{N + 1} \right), \quad n = 1, \ldots, N, \quad j = 0, \ldots, N + 1 \]

where the superscript \( d \) indicates a finite difference result. The ratio of the approximate eigenvalues to the exact eigenvalues is

\[ \lambda^d_n / \lambda_n = \sin \left( \frac{\pi n}{2(N + 1)} \right) / \left( \frac{\pi n}{2(N + 1)} \right). \]

So, for large enough \( M \), we get \( \lambda^d_1 / \lambda_1 = 1 \) and \( \lambda^d_N / \lambda_N = \frac{2}{\pi} \approx 0.63 \). The smallest eigenvalue is quite accurate, but the largest eigenvalue is too low by about thirty-seven percent. This implies that the finite difference method is not very good for computing high order eigenvalues. For instance, to get \( \lambda^d_{100} / \lambda_{100} = 0.999 \) requires a rather high value of \( N = 2027 \).

An alternate approach to the finite difference method is to use a series representation

\[ y(x) = \sum_{k=1}^{N} f_k(x) c_k \]

where the \( f_k(x) \) vanish at the end points. We then seek a least-squares approximate solution imposing

\[ \sum_{k=1}^{N} f_k''(\xi_j) c_k + \lambda^2 \sum_{k=1}^{N} f_k(\xi_j) c_k = 0 \]

for a set of collocation points \( \xi_j, j = 1 \ldots M \) with \( M \) taken much larger than \( N \). With the matrix form of the last equation denoted as \( B C + \lambda^2 A C = 0 \), we make the error orthogonal to the columns of matrix \( A \) and get the resulting eigenvalue problem

\[ \left(A \backslash B \right) C + \lambda^2 C = 0 \]

employing the generalized inverse of \( A \). A short program eigverr written to compare the accuracy of the finite difference and the spline algorithms produced Figure 10.1. The program is also listed. The spline approximation method gives quite accurate results, particularly if no more than half of the computed eigenvalues are used.

© 2003 by CRC Press LLC
Figure 10.1: Comparing an eigenvalue computation using the least squares method and a second order finite differences method
Program eigverr

function eigverr(nfd,nspl,kseg)

% This function compares two methods of computing eigenvalues corresponding to
% y''(x)+w^2*y(x)=0, y(0)=y(1)=0.
% Results are obtained using 1) finite differences
% and 2) cubic splines.

% nfd - number of interior points used for the finite difference equations
% nspl - number of interior points used for the spline functions.
% kseg - the number of interior spline points is kseg*(nspl+1)+nspl

if nargin==0, nfd=100; nspl=100; kseg=4; end
[ws,es]=spleig(nspl,kseg); [wd,ed]=findieig(nfd);
str=['COMPARING TWO METHODS FOR EIGENVALUES ',... 'OF Y''(X)+W^2*Y(X)=0, Y(0)=Y(1)=0'];
plot(1:nspl,es,'k-',1:nfd,ed,'k.);
title(str), xlabel('Eigenvalue Index'), Nfd=num2str(nfd);
Ns=num2str(nspl); M=num2str(nspl+(nspl+1)*kseg);
legend(["Using ",Ns, ' cubic splines and ',... ' M, least square points'"],... ["Using ",nfd,' finite differences points'],3)
grid on, shg
% print -deps eigverr

%==========================================

function [w,pcterr]=findieig(n)

% This function determines eigenvalues of
% y''(x)+w^2*y(x)=0, y(0)=y(1)=0
% The solution uses an n point finite difference approximation
if nargin==0, n=100; end
a=2*eye(n,n)-diag(ones(n-1,1),1)...
42: \[-\text{diag}(\text{ones}(n-1,1),-1)\];
43: \[ w=(n+1)*\sqrt{\text{sort} \left( \text{eig}(a) \right)}; \quad \text{we}=\pi*(1:n)'; \]
44: \[ \text{pcterr}=100*(w-we)./\text{we}; \]
45: %===================================================================
46: function \([w,pcterr]=\text{spleig}(n,nseg)\)
47: % \([w,pcterr]=\text{spleig}(n,nseg)\)
48: % This function determines eigenvalues of
49: % \(y''(x)+w^2y(x)=0, \quad y(0)=y(1)=0\)
50: % The solution uses \(n\) spline basis functions
51: % and \(nseg*(n+1)+n\) least square points
52: %===================================================================
53: if nargin==0, n=100; nseg=1; end
54: nls=(n+1)*nseg+n; xls=(1:nls)’/(nls+1);
55: a=zeros(nls,n); b=a;
56: for k=1:n
57: \[ a(:,k)=\text{splnf}(k,n,1,xls,2); \]
58: \[ b(:,k)=\text{splnf}(k,n,1,xls); \]
59: end
60: \[ w=\sqrt{\text{sort} \left( \text{eig}(-b\backslash a) \right)}; \quad \text{we}=\pi*(1:n)'; \]
61: \[ \text{pcterr}=100*(w-we)./\text{we}; \]
62: %===================================================================
63: function \(y=\text{splnf}(n,N,len,x,ideriv)\)
64: % \(y=\text{splnf}(n,N,len,x,ideriv)\)
65: % This function computes the spline basis
66: % functions and derivatives
67: xd=len/(N+1)*((0:N+1)'; yd=zeros(N+2,1);
68: \[ yd(n+1)=1; \]
69: if nargin<5, \(y=\text{spline}(xd,yd,x)\);
70: elseif ideriv==1, \(y=\text{splined}(xd,yd,x)\);
71: else, \(y=\text{splined}(xd,yd,x,2)\); end
72: \%===================================================================
73: \% function \(\text{val}=\text{splined}(xd,yd,x,if2)\)
74: \% See Appendix B
10.3 Stress Transformation and Principal Coordinates

The state of stress at a point in a three-dimensional continuum is described in terms of a symmetric $3 \times 3$ matrix $\mathbf{t} = [t(i,j)]$ where $t(i,j)$ denotes the stress component in the direction of the $x_i$ axis on the plane with it normal in the direction of the $x_j$ axis [9]. Suppose we introduce a rotation of axes defined by matrix $\mathbf{b}$ such that row $\mathbf{b}(i,:)$ represents the components of a unit vector along the new $\tilde{x}_i$ axis measured relative to the initial reference state. It can be shown that the stress matrix $\tilde{\mathbf{t}}$ corresponding to the new axis system can be computed by the transformation

$$\tilde{\mathbf{t}} = \mathbf{b} \mathbf{b}^T.$$ 

Sometimes it is desirable to locate a set of reference axes such that $\tilde{\mathbf{t}}$ is diagonal, in which case the diagonal components of $\mathbf{t}$ represent the extremal values of normal stress. This means that seeking maximum or minimum normal stress on a plane leads to the same condition as requiring zero shear stress on the plane. The eigenfunction operation

$$[\text{eigvecs}, \text{eigvals}] = \text{beig}([\mathbf{t}]);$$

applied to a symmetric matrix $\mathbf{t}$ produces an orthonormal set of eigenvectors stored in the columns of $\text{eigvecs}$, and a diagonal matrix $\text{eigvals}$ having the eigenvalues on the diagonal. These matrices satisfy

$$\text{eigvecs}^T \mathbf{t} \text{eigvecs} = \text{eigvals}.$$ 

Consequently, the rotation matrix $\mathbf{b}$ needed to transform to principal axes is simply the transpose of the matrix of orthonormalized eigenvectors. In other words, the eigenvectors of the stress tensor give the unit normals to the planes on which the normal stresses are extremal and the shear stresses are zero. The function prnstres performs the principal axis transformation.

10.3.1 Principal Stress Program

Function prnstres

1. function [pstress,pvecs]=prnstres(stress)
2. % [pstress,pvecs]=prnstres(stress)
3. % --------------------------------------------------------
4. %
5. % This function computes principal stresses
6. % and principal stress directions for a three-
7 % dimensional stress state.
8 %
9 % stress - a vector defining the stress components in the order
10 %      [sxx,syy,szz,sxy,sxz,syz]
11 %
12 % pstres - the principal stresses arranged in ascending order
13 %
14 % pvecs - the transformation matrix defining the orientation of the principal
15 %    axis system. The rows of this matrix define the surface normals to
16 %    the planes on which the extremal normal stresses act
17 %
18 % User m functions called: none
19 %
20 s=stress(:)';
21 s=[s([1 4 5]); s([4 2 6]); s([5 6 3])];
22 [pvecs,pstres]=eig(s);
23 [pstres,k]=sort(diag(pstres));
24 pvecs=pvecs(:,k)';
25 if det(pvecs)<0, pvecs(3,:)=-pvecs(3,:); end

10.3.2 Principal Axes of the Inertia Tensor

A rigid body dynamics application quite similar to principal stress analysis occurs in the kinetic energy computation for a rigid body rotating with angular velocity \( \omega = [\omega_x; \omega_y; \omega_z] \) about the reference origin [48]. The kinetic energy, \( K \), of the body can be obtained using the formula

\[
K = \frac{1}{2} \omega^T J \omega
\]

with the inertia tensor \( J \) computed as

\[
J = \iiint_V \rho [I r^T r - r r^T] \, dV,
\]

where \( \rho \) is the mass per unit volume, \( I \) is the identity matrix, and \( r \) is the Cartesian radius vector. The inertia tensor is characterized by a symmetric matrix expressed in component form as

\[
J = \iiint_V \begin{bmatrix}
  y^2 + z^2 & -xy & -xz \\
  -xy & x^2 + z^2 & -yz \\
  -xz & -yz & x^2 + y^2 \\
\end{bmatrix} \, dxdydz.
\]
Under the rotation transformation
\[ \tilde{r} = br \quad \text{with} \quad b^T b = I, \]
we can see that the inertia tensor transforms as
\[ \tilde{J} = bJb^T \]
which is identical to the transformation law for the stress component matrix discussed earlier. Consequently, the inertia tensor will also possess principal axes which make the off-diagonal components zero. The kinetic energy is expressed more simply as
\[ K = \frac{1}{2} (\omega_1^2 J_{11} + \omega_2^2 J_{22} + \omega_3^2 J_{33}) \]
where the components of \( \omega \) and \( J \) must be referred to the principal axes. The function \texttt{prnstres} can also be used to locate principal axes of the inertia tensor since the same transformations apply. As an example of principal axis computation, consider the inertia tensor for a cube of side length \( A \) and mass \( M \) which has a corner at \((0,0,0)\) and edges along the coordinate axes. The inertia tensor is found to be
\[ J = \begin{bmatrix} 2/3 & -1/4 & -1/4 \\ -1/4 & 2/3 & -1/4 \\ -1/4 & -1/4 & 2/3 \end{bmatrix} MA^2. \]
The computation
\[ [\text{pvl, pvc}] = \text{prnstres}([2/3, 2/3, 2/3, -1/4, -1/4, -1/4]); \]
produces the results
\[ \text{pvl} = \begin{bmatrix} 0.1667 \\ 0.9167 \\ 0.9167 \end{bmatrix}, \quad \text{pvc} = \begin{bmatrix} -0.5574 & -0.5574 & -0.5574 \\ -0.1543 & 0.7715 & -0.6172 \\ 0.8018 & -0.2673 & -0.5345 \end{bmatrix}. \]
This shows that the smallest possible inertial component equals \( 1/6 (\approx 0.1667) \) about the diagonal line through the origin while the maximal inertial moments of \( 11/12 (\approx 0.9167) \) occur about the axes normal to the diagonal.

### 10.4 Vibration of Truss Structures

Trusses are a familiar type of structure used in diverse applications such as bridges, roof supports, and power transmission towers. These structures can be envisioned as
a series of nodal points among which various axially loaded members are connected. These members are assumed to act like linearly elastic springs supporting tension or compression. Typically, displacement constraints apply at one or more points to prevent movement of the truss from its supports. The natural frequencies and mode shapes of two-dimensional trusses are computed when the member properties are known and the loads of interest arise from inertial forces occurring during vibration. A similar analysis pertaining to statically loaded trusses has been published recently [102].

Consider an axially loaded member of constant cross section connected between nodes $i$ and $j$ which have displacement components $(u_i, v_i)$ and $(u_j, v_j)$ as indicated in Figure 10.2. The member length is given by

$$
\ell = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2},
$$

and the member inclination is quantified by the trigonometric functions

$$
c = \cos \theta = \frac{x_j - x_i}{\ell} \quad \text{and} \quad s = \sin \theta = \frac{y_j - y_i}{\ell}.
$$

The axial extension for small deflections is

$$
\Delta = (u_j - u_i)c + (v_j - v_i)s.
$$

The axial force needed to extend a member having length $\ell$, elastic modulus $E$, and cross section area $A$ is given by

$$
P_{ij} = \frac{AE}{\ell} \Delta = \frac{AE}{\ell} \begin{bmatrix} -c, & -s, & c, & s \end{bmatrix} \begin{bmatrix} u_i \end{bmatrix}
$$

where

$$
\begin{bmatrix} u_i \end{bmatrix} = \begin{bmatrix} u_i; & v_i; & u_j; & v_j \end{bmatrix}
$$

is a column matrix describing the nodal displacements of the member ends. The corresponding end forces are represented by

$$
F_{ij} = [F_{ix}; F_{iy}; F_{jx}; F_{jy}] = P_{ij} \begin{bmatrix} -c, & -s, & c, & s \end{bmatrix},
$$

so that the end forces and end displacements are related by the matrix equation

$$
F_{ij} = K_{ij}U_{ij},
$$

where the element stiffness matrix is

$$
K_{ij} = \frac{AE}{\ell} \begin{bmatrix} -c; & -s; & c; & s \end{bmatrix} \begin{bmatrix} -c, & -s, & c, & s \end{bmatrix}.
$$

In regard to mass effects in a member, we will assume that any transverse motion is negligible and half of the mass of each member can be lumped at each end. Hence the mass placed at each end would be $A\rho\ell/2$ where $\rho$ is the mass per unit volume.
The deflection of a truss with \( n \) nodal points can be represented using a generalized displacement vector and a generalized nodal force vector:

\[
U = [u_1; v_1; u_2; v_2; \ldots; u_n; v_n], \quad F = [F_{1x}; F_{1y}; F_{2x}; F_{2y}; \ldots; F_{nx}; F_{ny}].
\]

When the contributions of all members in the network are assembled together, a global matrix relation results in the form

\[
F = KU
\]

where \( K \) is called the global stiffness matrix. Before we formulate procedures for assembling the global stiffness matrix, dynamical aspects of the problem will be discussed.

In the current application, the applied nodal forces are attributable to the acceleration of masses located at the nodes and to support reactions at points where displacement constraints occur. The mass concentrated at each node will equal half the sum of the masses of all members connected to the node. According to D’Alembert’s principle [48] a particle having mass \( m \) and acceleration \( \ddot{u} \) is statically equivalent to a force \( -m\ddot{u} \). So, the equation of motion for the truss, without accounting for support reactions, is

\[
KU = -M\ddot{U}
\]

where \( M \) is a global mass matrix given by

\[
M = \text{diag} ([m_1; m_1; m_2; m_2; \ldots; m_n; m_n])
\]

with \( m_i \) denoting the mass concentrated at the \( i \)’th node. The equation of motion \( M\ddot{U} + KU = 0 \) will also be subjected to constraint equations arising when some points are fixed or have roller supports. This type of support implies a matrix equation of the form \( CU = 0 \).
Natural frequency analysis investigates states-of-motion where each node of the structure simultaneously moves with simple harmonic motion of the same frequency. This means solutions are sought of the form

\[ U = X \cos(\omega t) \]

where \( \omega \) denotes a natural frequency and \( X \) is a modal vector describing the deflection pattern for the corresponding frequency. The assumed mode of motion implies \( \ddot{U} = -\lambda U \) where \( \lambda = \omega^2 \). We are led to an eigenvalue problem of the form

\[ KX = \lambda MX \]

with a side constraint \( CX = 0 \) needed to satisfy support conditions.

MATLAB provides the intrinsic functions `eig` and `null` which deal with the solution to this problem effectively. Using function `null` we can write

\[ X = QY \]

where \( Q \) has columns that are an orthonormal basis for the null space of matrix \( C \). Expressing the eigenvalue equation in terms of \( Y \) and multiplying both sides by \( Q^T \) gives

\[ K_\circ Y = \lambda M_\circ Y \]

where

\[ K_\circ = Q^T K Q \quad \text{and} \quad M_\circ = Q^T M Q. \]

It can be shown from physical considerations that, in general, \( K \) and \( M \) are symmetric matrices such that \( K \) has real non-negative eigenvalues and \( M \) has real positive eigenvalues. This implies that \( M_\circ \) can be factored as

\[ M_\circ = N^T N \]

where \( N \) is an upper triangular matrix. Then the eigenvalue problem can be rewritten as

\[ K_1 Z = \lambda Z \quad Y = N Z, \quad K_1 = (N^T)^{-1} K_\circ N^{-1}. \]

Because matrix \( K_1 \) will be real and symmetric, the intrinsic function `eig` generates orthonormal eigenvectors. The function `eigsym` used by program `trusvibs` produces a set of eigenvectors in the columns of \( X \) which satisfy generalized orthogonality conditions of the form

\[ X^T M X = I \quad \text{and} \quad X^T K X = \Lambda, \]

where \( \Lambda \) is a diagonal matrix containing the squares of the natural frequencies arranged in ascending order. The calculations performed in function `eigsym` illustrate the excellent matrix manipulative features that MATLAB embodies.

Before we discuss a physical example, the problem of assembling the global stiffness matrix will be addressed. It is helpful to think of all nodal displacements as if
they were known and then compute the nodal forces by adding the stiffness contributions of all elements. Although the total force at each node results only from the forces in members touching the node, it is better to accumulate force contributions on an element-by-element basis instead of working node by node. For example, a member connecting node $i$ and node $j$ will involve displacement components at row positions $2i - 1$, $2i$, $2j - 1$, and $2j$ in the global displacement vector and force components at similar positions in the generalized force matrix. Because principles of superposition apply, the stiffness contributions of individual members can be added, one member at a time, into the global stiffness matrix. This process is implemented in function `assemble` which also forms the mass matrix. First, selected points constrained to have zero displacement components are specified. Next the global stiffness and mass matrices are formed. This is followed by an eigenvalue analysis which yields the natural frequencies and the modal vectors. Finally the motion associated with each vibration mode is described by superimposing on the coordinates of each nodal point a multiple of the corresponding modal vector varying sinusoidally with time. Redrawing the structure produces an appearance of animated motion.

The complete program has several functions which should be studied individually for complete understanding of the methods developed. These functions and their purposes are summarized in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>trusvibs</td>
<td>reads data and guides interactive input to animate the various vibration modes</td>
</tr>
<tr>
<td>crossdat</td>
<td>function typifying the nodal and element data to define a problem</td>
</tr>
<tr>
<td>assemble</td>
<td>assembles the global stiffness and mass data matrices</td>
</tr>
<tr>
<td>elmsdf</td>
<td>forms the stiffness matrix and calculates the volume of an individual member</td>
</tr>
<tr>
<td>eige</td>
<td>forms the constraint equations implied when selected displacement components are set to zero</td>
</tr>
<tr>
<td>eigsym</td>
<td>solves the constrained eigenvalue problem pertaining to the global stiffness and mass matrices</td>
</tr>
<tr>
<td>trifacsm</td>
<td>factors a positive definite matrix into upper and lower global triangular parts</td>
</tr>
<tr>
<td>drawtrus</td>
<td>draws the truss in deflected positions</td>
</tr>
<tr>
<td>cubrange</td>
<td>a utility routine to determine a window for drawing the truss without scale distortion</td>
</tr>
</tbody>
</table>

The data in function `crossdat` contains the information for node points, element data, and constraint conditions needed to define a problem. Once the data values are read, mode shapes and frequencies are computed and the user is allowed to observe the animation of modes ordered from the lowest to the highest frequency. The number of modes produced equals twice the number of nodal points minus the number
of constraint conditions. The plot in Figure 10.3 shows mode eleven for the sample problem. This mode has no special significance aside from the interesting deflection pattern produced. The reader may find it instructive to run the program and select several modes by using input such as 3:5 or a single mode by specifying a single mode number.

Figure 10.3: Truss Vibration Mode Number 11
10.4.1  Truss Vibration Program

Program trusvibs

function trusvibs
% Example: trusvibs
% ~~~~~~~~~~~~~~~~~
% This program analyzes natural vibration modes
% for a general plane pin-connected truss. The
% direct stiffness method is employed in
% conjunction with eigenvalue calculation to
% evaluate the natural frequencies and mode
% shapes. The truss is defined in terms of a
% set of nodal coordinates and truss members
% connected to different nodal points. Global
% stiffness and mass matrices are formed. Then
% the frequencies and mode shapes are computed
% with provision for imposing zero deflection
% at selected nodes. The user is then allowed
% to observe animated motion of the various
% vibration modes.

% User m functions called:
%   eigsym, crossdat, drawtrus, eigc,
%   assemble, elmstf, cubrange

global x y inode jnode elast area rho idux iduy
kf=1; idux=[]; iduy=[]; disp('')
disp(['Modal Vibrations for a Pin ', ...
     'Connected Truss']); disp(' ');

% A sample data file defining a problem is
% given in crossdat.m
disp(['Give the name of a function which ', ...
     'creates your input data']);
disp(['Do not include .m in the name ', ...
     '(use crossdat as an example')]);
filename=input('>? ','s');
eval(filename); disp(' ');

% Assemble the global stiffness and
% mass matrices
[stiff,masmat]= ...
assemble(x,y,inode,jnode,area,elast,rho);

% Compute natural frequencies and modal vectors
% accounting for the fixed nodes
ifixed=[2*idux(:)-1; 2*iduy(:)];
[modvcs,eigval]=eigc(stiff,masmat,ifixed);
natfreqs=sqrt(eigval);

% Set parameters used in modal animation
nsteps=31; s=sin(linspace(0,6.5*pi,nsteps));
x=x(:); y=y(:); np=2*length(x);
bigxy=max(abs([x;y])); scafac=.05*bigxy;
highmod=size(modvcs,2); hm=num2str(highmod);

% Show animated plots of the vibration modes
while 1
    disp('Give the mode numbers to be animated?');
    disp(['Do not exceed a total of ',hm,...
         ' modes.']);
    if kf==1, disp(['Try 1:',hm]); kf=kf+1; end
    str=input('>? ','s');
    nmode=eval(['[',str,']']);
    nmode=nmode(find(nmode<=highmod));
    if sum(nmode)==0; break; end
    % Animate the various vibration modes
    hold off; clf; ovrsiz=1.1;
w=cubrange([x(:),y(:)],ovrsiz);
axis(w); axis('square'); axis('off'); hold on;
for kk=1:length(nmode) % Loop over each mode
    kkn=nmode(kk);
    titl=['Truss Vibration Mode Number ', ...
         num2str(kkn)];
    dd=modvcs(:,kkn); mdd=max(abs(dd));
dx=dd(1:2:np); dy=dd(2:2:np);
clf; pause(1);
    % Loop through several cycles of motion
    for jj=1:nsteps
        sf=scafac*s(jj)/mdd;
        xd=x+sf*dx; yd=y+sf*dy; clf;
        axis(w); axis('square'); axis('off');
drawtrus(xd,yd,inode,jnode); title(titl);
drawnow; figure(gcf);
    end
end
end
function crossdat
% [inode,jnode,elast,area,rho]=crossdat
% This function creates data for the truss vibration program. It can serve as a model for other configurations by changing the function name and data quantities
% Data set: crossdat
% ~~~~~~~~~~~~~~~~~~
% Data specifying a cross-shaped truss.
% %----------------------------------------------

global x y inode jnode elast area rho idux iduy

% Nodal point data are defined by:
% x - a vector of x coordinates
% y - a vector of y coordinates
x=10*[.5 2.5 1 2 0 1 2 3 0 1 2 3 1 2];
y=10*[ 0 0 1 1 2 2 2 2 3 3 3 3 4 4];

% Element data are defined by:
% inode - index vector defining the I-nodes
% jnode - index vector defining the J-nodes
% elast - vector of elastic modulus values
% area - vector of cross section area values
% rho - vector of mass per unit volume values
inode=[1 1 2 2 3 3 4 3 4 5 6 7 5 6 6 6 7 7 7 ... 8 9 10 11 10 11 10 11 13];
jnode=[3 4 3 4 4 6 6 7 7 6 7 8 9 10 10 ... 11 12 12 10 11 12 13 13 14 14 14];
elast=3e7*ones(1,28);
area=ones(1,28); rho=ones(1,28);

% Any points constrained against displacement
% are defined by:
% idux - indices of nodes having zero x-displacement
% iduy - indices of nodes having zero y-displacement

idux=[1 2]; iduy=[1 2];

function drawtrus(x,y,i,j)

% This function draws a truss defined by nodal coordinates defined in x,y and member indices defined in i,j.
% User m functions called: none

hold on;
for k=1:length(i)
    plot([x(i(k)),x(j(k))],y(i(k)),y(j(k))]);
end

function [vecs,eigvals]=eigc(k,m,idzero)

% [vecs,eigvals]=eigc(k,m,idzero)
% This function computes eigenvalues and eigenvectors for the problem
% k*x=eigval*m*x
% with some components of x constrained to equal zero. The imposed constraint is
% x(idzero(j))=0
% for each component identified by the index matrix idzero.

% k - a real symmetric stiffness matrix
% m - a positive definite symmetric mass matrix
% idzero - the vector of indices identifying components to be made zero
% vecs - eigenvectors for the constrained problem. If matrix k has dimension
% n by n and the length of idzero is
176  \% m (with m<n), then vecs will be a
177  \% set on n-m vectors in n space
178  \% eigvals - eigenvalues for the constrained
179  \% problem. These are all real.
180  \%
181  \% User m functions called: eigsym
182  \%----------------------------------------------
183  
184  n=size(k,1); j=1:n; j(idzero)=[];
185  c=eye(n,n); c(j,:)=[];
186  [vecs,eigvals]=eigsym((k+k')/2, (m+m')/2, c);
187  
188  \%=============================================
189  
190  function [evecs,eigvals]=eigsym(k,m,c)
191  \%
192  \% [evecs,eigvals]=eigsym(k,m,c)
193  \%----------------------------------------------
194  \% This function solves the constrained
195  \% eigenvalue problem
196  \% k*x=(lambda)*m*x, with c*x=0.
197  \% Matrix k must be real symmetric and matrix
198  \% m must be symmetric and positive definite;
199  \% otherwise, computed results will be wrong.
200  \%
201  \% k - a real symmetric matrix
202  \% m - a real symmetric positive
203  \% definite matrix
204  \% c - a matrix defining the constraint
205  \% condition c*x=0. This matrix is
206  \% omitted if no constraint exists.
207  \%
208  \% evecs - matrix of eigenvectors orthogonal
209  \% with respect to k and m. The
210  \% following relations apply:
211  \% evecs'*m*evecs=identity_matrix
212  \% evecs'*k*evecs=diag(eigvals).
213  \% eigvals - a vector of the eigenvalues
214  \% sorted in increasing order
215  \%
216  \% User m functions called: none
217  \%----------------------------------------------
218  
219  if nargin==3
220  q=null(c); m=q'*m*q; k=q'*k*q;
221  
222  © 2003 by CRC Press LLC
221: end
222: u=chol(m); k=u\k/u; k=(k+k')/2;
223: [evecs,eigvals]=eig(k);
224: [eigvals,j]=sort(diag(eigvals));
225: evecs=evecs(:,j); evecs=u\evecs;
226: if nargin==3, evecs=q*evecs; end
227:
228: function [stif,masmat]= ... 
229: assemble(x,y,id,jd,a,e,rho)
230: %
231: % [stif,masmat]=assemble(x,y,id,jd,a,e,rho)
232: % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
233: %
234: % This function assembles the global
235: % stiffness matrix and mass matrix for a
236: % plane truss structure. The mass density of
237: % each element equals unity.
238: %
239: % x,y      - nodal coordinate vectors
240: % id,jd    - nodal indices of members
241: % a,e      - areas and elastic moduli of members
242: % rho      - mass per unit volume of members
243: %
244: % stif     - global stiffness matrix
245: % masmat   - global mass matrix
246: %
247: % User m functions called: elmstf
248: %----------------------------------------------
249: 
250: numnod=length(x); numelm=length(a);
251: id=id(:); jd=jd(:);
252: stif=zeros(2*numnod); masmat=stif;
253: ij=[2*id-1,2*id,2*jd-1,2*jd];
254: for k=1:numelm, kk=ij(k,:);
255:     [stfk,volmk]= ... 
256:     elmstf(x,y,a(k),e(k),id(k),jd(k));
257:     stif(kk,kk)=stif(kk,kk)+stfk;
258:     masmat(kk,kk)=masmat(kk,kk)+ ... 
259:         rho(k)*volmk/2*eye(4,4);
260: end 
261: end
262:
263:
264: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
265: © 2003 by CRC Press LLC
function [k,vol]=elmstf(x,y,a,e,i,j)
% [k,vol]=elmstf(x,y,a,e,i,j)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function forms the stiffness matrix for
% a truss element. The member volume is also
% obtained.
%----------------------------------------------
xx=x(j)-x(i); yy=y(j)-y(i);
L=norm([xx,yy]); vol=a*L;
c=xx/L; s=yy/L; k=a*e/L*[-c;-s;c;s]*[-c,-s,c,s];

%=============================================
% function range=cubrange(xyz,ovrsiz)
% See Appendix B

10.5  Buckling of Axially Loaded Columns

Computing the buckling load and deflection curve for a slender axially loaded
column leads to an interesting type of eigenvalue problem. Let us analyze a column
of length $L$ subjected to a critical value of axial load $P$ just large enough to hold the
column in a deflected configuration. Reducing the load below the critical value will
allow the column to straighten out, whereas increasing the load above the buckling
value will result in a structural failure. To prevent sudden collapse of structures
using axially loaded members, designers must be able to calculate buckling loads
corresponding to various end constraints. We will present an analysis allowing the
flexural rigidity $EI$ to vary along the length. Four common types of end conditions
of interest are shown in Figure 10.4. For each of these systems we will assume that
the coordinate origin is at the left end of the column\(^1\) with $y(0) = 0$. Cases I and
II involve statically determinate columns. Cases III and IV are different because
unknown end reactions occur in the boundary conditions.

All four problems lead to a homogeneous linear differential equation subjected
to homogeneous boundary conditions. All of these cases possess a trivial solution
where $y(x)$ vanishes identically. However, the solutions of practical interest
involve a nonzero deflection configuration which is only possible when $P$ equals
the buckling load. Finite difference methods can be used to accurately approximate

\(^1\)Although columns are usually positioned vertically, we show them as horizontal for convenience.
I) Pinned-Pinned

\[ y = 0 \quad m = 0 \]

II) Free-Fixed

\[ y = 0 \quad y' = 0 \quad m = 0 \]

III) Pinned-Fixed

\[ y = 0 \quad m = 0 \]

IV) Fixed-Fixed

\[ y = 0 \quad y' = 0 \quad m = 0 \]

Figure 10.4: Buckling Configurations
the differential equation and boundary conditions. In this manner we obtain a linear algebraic eigenvalue problem subjected to side constraints characterized by an underdetermined system of linear simultaneous equations.

Consider a beam element relating the bending moment \( m \), the transverse shear \( v \), the axial load \( P \), and the transverse deflection \( y \) as shown in Figure 10.5. Equilibrium considerations imply

\[
v'(x) = 0, \quad m'(x) + Py'(x) = v.
\]

Since no transverse external loading acts on the column between the end supports, the shear \( v \) is constant. Differentiating the moment equation gives

\[
m''(x) + Py''(x) = 0.
\]

Furthermore, flexural deformation theory of slender elastic beams implies

\[
EIy''(x) = m(x),
\]

which leads to the following homogeneous differential equation governing the bending moment

\[
EIy''(x) + Pm(x) = 0.
\]

We need to find values of \( P \) allowing nontrivial solutions of this differential equation subject to the required homogeneous boundary conditions. The four types of end conditions shown in Figure 10.4 impose both deflection and moment conditions at the ends. Cases I and II can be formulated completely in terms of displacements because moment conditions evidently imply

\[
EIy''(x) = m = -Py.
\]

To handle cases III and IV, we need to relate the displacement and slope conditions at the ends to the bending moment. Let us denote the function \( 1/(EI) \) as \( k(x) \) so that

\[
y''(x) = k(x)m(x).
\]

Integration gives

\[
y'(x) = y'(0) + \int_0^x k(\xi)m(\xi) \, d\xi
\]
and

\[ y(x) = y(0) + y'(0)x + \int_0^x (x - \xi)k(\xi)m(\xi) \, d\xi. \]

The boundary conditions for the pinned-fixed case require that

a) \( y(0) = 0 \), \quad b) \( y'(L) = 0 \), \quad c) \( y(L) = 0 \).

Condition b) requires

\[ y'(0) = - \int_0^L k(\xi)m(\xi) \, d\xi, \]

whereas a) and c) combined lead to

\[ y(L) = y(0) - L \int_0^L km \, d\xi + \int_0^L (L - \xi)km \, d\xi. \]

Consequently for Cases III and IV the governing equation is

\[ EI\ddot{y}(x) + Pm(x) = 0. \]

The boundary conditions for Case III are

\[ m(0) = 0 \text{ and } \int_0^L xk(x)m(x) \, dx = 0. \]

The boundary conditions for Case IV are handled similarly. Since we must have \( y'(0) = y'(L) = 0 \) and \( y(0) = y(L) = 0 \), the conditions are

\[ \int_0^L k(x)m(x) \, dx = 0 \text{ and } \int_0^L xk(x)m(x) \, dx = 0. \]

The results for each case require a nontrivial solution of a homogeneous differential equation satisfying homogeneous boundary conditions as summarized in the table below.

Each of these boundary value problems can be transformed to linear algebraic form by choosing a set of evenly spaced grid points across the span and approximating \( y''(x) \) by finite differences. It follows from Taylor’s series that

\[ y''(x) = \frac{y(x - h) - 2y(x) + y(x + h)}{h^2} + O(h^2). \]

For sufficiently small \( h \), we neglect the truncation error and write

\[ y_j'' = \frac{y_{j-1} - 2y_j + y_{j+1}}{h^2} \]

where \( y_j \) is the approximation to \( y \) at \( x = x_j = jh \) for \( 1 \leq j \leq n \), where the stepsize \( h = L/(n + 1) \). Thus we have

\[ \left(\frac{EI}{h^2}\right)[y_{j-1} - 2y_j + y_{j+1}] + Py_j = 0 \]
### Differential Equation Conditions

<table>
<thead>
<tr>
<th>Case</th>
<th>Differential Equation</th>
<th>Boundary Conditions</th>
</tr>
</thead>
</table>
| I: pinned-pinned | $EIy''(x) + Py(x) = 0$ | $y(0) = 0$  
$y(L) = 0$ |
| II: free-fixed  | $EIy''(x) + Py(x) = 0$ | $y(0) = 0$  
$y'(L) = 0$ |
| III: pinned-fixed | $EIm''(x) + Pm(x) = 0$ | $m(0) = 0$  
$f_0^L k(x)m(x) \, dx = 0$ |
| IV: fixed-fixed  | $EIm''(x) + Pm(x) = 0$ | $f_0^L k(x)m(x) \, dx = 0$  
$f_0^L xk(x)m(x) \, dx = 0$ |

**Buckling Problem Summary**

For Cases I or II, and

$$
\frac{(EI)[m_{j-1} - 2m_j + m_{j+1}]}{h^2} + Pm_j = 0
$$

For Cases III or IV. At the left end, either $y$ or $m$ is zero in all cases. Case I also has $y(L) = y_{n+1} = 0$. Case II requires $y'(L) = 0$. This is approximated in finite difference form as

$$
y_{n+1} = \frac{4y_n - y_{n-1}}{3}
$$

which implies for Case II that

$$
y''_n = \frac{2(y_{n-1} - y_n)}{3h^2}.
$$

Cases III and IV are slightly more involved than I and II. The condition that

$$
\int_0^L \frac{mx}{EI} \, dx = 0
$$

can be formulated using the trapezoidal rule to give

$$
b_1 \ast [m_1, \ldots, m_n, m_{n+1}]^T = 0,
$$

where the asterisk indicates matrix multiplication involving a row matrix $b_1$ defined by

$$
b_1 = [1, 1, \ldots, 1, 1/2] \ast [x_1, x_2, \ldots, x_n, L] \ast [EI_1, \ldots, EI_n, EI_{n+1}] .
$$
Similarly, the condition
\[ \int_0^L \frac{m}{EI} \, dx = 0 \]
leads to
\[ b_2 \ast [m_1, \ldots, m_n]^T + \frac{1}{2} \left[ \frac{m_0}{EI_0} + \frac{m_{n+1}}{EI_{n+1}} \right] = 0 \]
with
\[ b_2 = \left[ \frac{1}{EI_1}, \ldots, \frac{1}{EI_n} \right]. \]
The first of these equations involving \( b_1 \) allows \( m_{n+1} \) to be eliminated in Case III, whereas the two equations involving \( b_1 \) and \( b_2 \) allow elimination of \( m_0 \) and \( m_{n+1} \) (the moments at \( x = 0 \) and \( x = L \)) for Case IV. Hence, in all cases, we are led to an eigenvalue problem typified as
\[ EI_j(-m_{j-1} + 2m_j - m_{j+1}) = \lambda m_j \]
with \( \lambda = h^2 P \), and we understand that the equations for \( j = 1 \) and \( j = n \) may require modification to account for pertinent boundary conditions. We are led to solve
\[ Am = \lambda m \]
where the desired buckling loads are associated with the smallest positive eigenvalue of matrix \( A \). Cases I and II lead directly to the deflection curve forms. However, Cases III and IV require that the deflection curve be computed from the trapezoidal rule as
\[ y'(x) = y'(0) + \int_0^x \frac{m}{EI} \, dx \]
and
\[ y(x) = y(0) + y'(0) + x \int_0^x \frac{m}{EI} \, dx - \int_0^x m x \frac{dx}{EI}. \]
The deflection curves can be normalized to make \( y_{max} \) equal unity. This completes the formulation needed in the buckling analysis for all four cases studied. These solutions have been implemented in the program described later in this section. An example, which is solvable exactly, will be discussed next to demonstrate that the finite difference formulation actually produces good results.

10.5.1 Example for a Linearly Tapered Circular Cross Section

Consider a column with circular cross section tapered linearly from diameter \( h_1 \) at \( x = 0 \) to diameter \( h_2 \) at \( x = L \). The moment of inertia is given by
\[ I = \frac{\pi d^4}{64}, \]
which leads to
\[ EI = E_o I_o \left( 1 + \frac{sx}{L} \right)^4, \]
where
\[ s = \frac{h_2 - h_1}{h_1}, \quad I_o = \frac{\pi h_1^4}{64} \]
and \( E_o \) is the elastic modulus which is assumed to have a constant value. The differential equation governing the moment in all cases (and for \( y \) in Case I or II) is
\[
(1 + \frac{sx}{L})^4 m''(x) + \frac{P}{E_o I_o} m(x) = 0.
\]
This equation can be reduced to a simpler form by making a change of variables. Let us replace \( x \) and \( m(x) \) by \( t \) and \( g(t) \) defined by
\[
t = \left(1 + \frac{sx}{L}\right)^{-1}, \quad g(t) = t m(x).
\]
The differential equation for \( g(t) \) is found to be
\[
g''(t) + \lambda^2 g(t) = 0 \quad \text{where} \quad \lambda = \frac{L}{|s|} \sqrt{\frac{P}{E_o I_o}}.
\]
Therefore,
\[
m(x) = \left(1 + \frac{sx}{L}\right) \left[ c_1 \sin \left( \frac{\lambda}{1 + \frac{sx}{L}} \right) + c_2 \cos \left( \frac{\lambda}{1 + \frac{sx}{L}} \right) \right]
\]
where \( c_1 \) and \( c_2 \) are arbitrary constants found by imposing the boundary conditions. We will determine these constants for Cases I, II, and III. Case IV can be solved similarly and is left as an exercise for the reader.

To deal with Cases I, II, and III it is convenient to begin with a solution that vanishes at \( x = 0 \). A function satisfying this requirement has the form
\[
m(x) = \left(1 + \frac{sx}{L}\right) \sin \left( \frac{\lambda}{1 + \frac{sx}{L}} \right).
\]
This equation can also represent the deflection curve for Cases I and II or the moment curve for Case III. Imposition of the remaining boundary conditions leads to an eigenvalue equation which is used to determine \( \lambda \) and the buckling load \( P \). The deflection curve for Case I is taken as
\[
y(x) = \left(1 + \frac{sx}{L}\right) \sin \left( \frac{\lambda}{1 + \frac{sx}{L}} \right)
\]
and the requirement that \( y(L) = 0 \) yields
\[
\frac{\lambda s}{1 + s} = \left( \frac{s}{1 + s} \right) \left( \frac{L}{s} \sqrt{\frac{P}{E_o I_o}} \right) = \pi.
\]
This means that the buckling load is
\[
P = \pi^2 E_o I_o \frac{1 + s^2}{L^2} \quad \text{where} \quad s = \frac{h_2 - h_1}{h_1}.
\]
Therefore the buckling load for the tapered column \((s \neq 0)\) is simply obtained by multiplying the buckling load for the constant cross section column \((s = 0)\) by a factor

\[
(1 + s)^2 = \left( \frac{h_2}{h_1} \right)^2.
\]

This is also true for Cases III and IV, but is not true for Case II. Let us derive the characteristic equation for Case III. The constraint condition for the pinned-fixed case requires

\[
\int_0^L x m(x) \frac{dx}{EI} = 0.
\]

So we need

\[
\int_0^L x \left(1 + \frac{sx}{L} \right)^{-3} \sin \left(\frac{\lambda}{1 + \frac{sx}{L}} \right) dx = 0.
\]

This equation can be integrated using the substitution \((1 + \frac{sx}{L})^{-1} = t\). This leads to a characteristic equation of the form

\[
\theta = \tan \theta , \quad \frac{\lambda s}{1 + s} = \frac{L}{1 + s} \sqrt{\frac{P}{E_o I_o}}.
\]

The smallest positive root of this equation is \(\theta = 4.4934\), which yields

\[
P = \frac{20.1906 E_o I_o}{L^2} (1 + s)^2 \text{ for Case III}.
\]

Further analysis produces

\[
P = \frac{4\pi^2 E_o I_o}{L^2} (1 + s)^2 \text{ for Case IV}.
\]

The characteristic equation for Case II can be obtained by starting with the Case I deflection equation and imposing the condition \(y'(L) = 0\). This leads to

\[
s \sin \theta + \theta \cos \theta = 0 , \quad \frac{L}{1 + s} \sqrt{\frac{P}{E_o I_o}}.
\]

When \(s = 0\), the smallest positive root of this equation is \(\theta = \pi/2\). Therefore, the buckling load (when \(s = 0\)) is

\[
P = \frac{\pi^2 E_o I_o}{4L^2}
\]

for Case II, and the dependence on \(s\) found in the other cases does not hold for the free-fixed problem.
10.5.2 Numerical Results

The function `colbuc`, which uses the above relationships, was written to analyze variable depth columns using any of the four types of end conditions discussed. The program allows a piecewise linear variation of $EI$. The program employs the function `lintrp` for interpolation and the function `trapsum` to perform trapezoidal rule integration. Comparisons were made with results presented by Beer and Johnston [9] and a comprehensive handbook on stability [19]. We will present some examples to show how well the program works. It is known that a column of length $L$ and constant cross section stiffness $E_o I_o$ has buckling loads of

$$\frac{\pi^2 E_o I_o}{L^2}, \frac{\pi^2 E_o I_o}{(2L)^2}, \frac{\pi^2 E_o I_o}{(0.6992L)^2}, \frac{\pi^2 E_o I_o}{(0.5L)^2}$$

for the pinned-pinned, the free-fixed, the pinned-fixed, and the fixed-fixed end conditions respectively. These cases were verified using the program `colbuc`. Let us illustrate the capability of the program to approximately handle a discontinuous cross section change. We analyze a column twenty inches long consisting of a ten inch section pinned at the outer end and joined to a ten inch long section which is considered rigid and fixed at the outer end. We use $E_o I_o = 1$ for the flexible section and $E_o I_o = 10000$ for the rigid section. This configuration should behave much like a pinned-fixed column of length 100 with a buckling load of $(\pi/6.992)^2 = 0.2019$.

Using 100 segments ($nseq=100$) the program yields a value of 0.1976, which agrees within 2.2% of the expected value. A graph of the computed deflection configuration is shown in Figure 10.6. The code necessary to solve this problem is:

```matlab
ei=[1 0; 1 10; 10000 10; 10000 20];
nseg=100; endc=3; len=20;
[p,y,x]=colbuc(len,ei,nseg,endc)
```

For a second example we consider a ten inch long column of circular cross section which is tapered from a one inch diameter at one end to a two inch diameter at the other end. We employ a fixed-fixed end condition and use $E_o = 1$. The theoretical results for this configuration indicate a buckling load of $\pi^3/400 = 0.07752$.

Using 100 segments the program produces a value of 0.07728, which agrees within 0.3% of the exact result. The code to generate this result utilizes function `eilt`:

```matlab
ei=eilt(1,2,10,101,1);
[p,y,x]=colbuc(10,ei,100,4);
```

The examples presented illustrate the effectiveness of using finite difference methods in conjunction with the intrinsic eigenvalue solver in MATLAB to compute buck-
ling loads. Furthermore, the provision for piecewise linear $EI$ variation provided in the program is adequate to handle various column shapes.

**Program Output and Code**

**Function colbuc**

```matlab
function [p,y,x]=colbuc(len,ei,nseg,endc)
% [p,y,x]=colbuc(len,ei,nseg,endc)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function determines the Euler buckling load for a slender column of variable cross section which can have any one of four constraint conditions at the column ends.
% len - the column length
% ei - the product of Young’s modulus and the cross section moment of inertia. This quantity is defined as a piecewise
```

© 2003 by CRC Press LLC
% linear function specified at one or
% more points along the length. ei(:,1)
% contains ei values at points
% corresponding to x values given in
% ei(:,2). Values at intermediate points
% are computed by linear interpolation
% using function lintrp which allows
% jump discontinuities in ei.
% nseg - the number of segments into which the
% column is divided to perform finite
% difference calculations.The stepsize h
% equals len/nseg.
% endc - a parameter specifying the type of end
% condition chosen.
% endc=1, both ends pinned
% endc=2, x=0 free, x=len fixed
% endc=3, x=0 pinned, x=len fixed
% endc=4, both ends fixed
% p - the Euler buckling load of the column
% x,y - vectors describing the shape of the
% column in the buckled mode. x varies
% between 0 and len. y is normalized to
% have a maximum value of one.
% User m functions called: lintrp, trapsum

if nargin==0;
ei=[1 0; 1 10; 1000 10; 1000 20];
nseg=100; endc=3; len=20;
end

% If the column has constant cross section,
% then ei can be given as a single number.
% Also, use at least 20 segments to assure
% that computed results will be reasonable.
if size(ei,1) < 2
  ei=[ei(1,1),0; ei(1,1),len];
end
nseg=max(nseg,30);

if endc==1
  % pinned-pinned case (y=0 at x=0 and x=len)
  str='Pinned-Pinned Buckling Load = ';
h=len/nseg; n=nseg-1; x=linspace(h,len-h,n);

© 2003 by CRC Press LLC
eiv=lintrp(ei(:,2),ei(:,1),x);
a=-diag(ones(n-1,1),1);
a=a+a'+diag(2*ones(n,1));
[yvecs,pvals]=eig(diag(eiv/h^2)*a);
pvals=diag(pvals);
% Discard any spurious nonpositive eigenvalues
j=find(pvals<=0);
if length(j)>0, pvals(j)=[]; yvecs(:,j)=[]; end
[p,k]=min(pvals); y=[0;yvecs(:,k);0];
[ym,j]=max(abs(y)); y=y/y(j); x=[0;x(:);len];
elseif endc==2
% free-fixed case (y=0 at x=0 and y’=0 at x=len)
str='Free-Fixed Buckling Load = ';
h=len/nseg; n=nseg-1; x=linspace(h,len-h,n);
eiv=lintrp(ei(:,2),ei(:,1),x);
a=-diag(ones(n-1,1),1);
a=a+a'+diag(2*ones(n,1));
% Zero slope at x=len implies
% y(n+1)=4/3*y(n)-1/3*y(n-1). This
% leads to y''(n)=(y(n-1)-y(n))*2/(3*h^2).
a(n,[n-1,n])=[-2/3,2/3];
[yvecs,pvals]=eig(diag(eiv/h^2)*a);
pvals=diag(pvals);
% Discard any spurious nonpositive eigenvalues
j=find(pvals<=0);
if length(j)>0, pvals(j)=[]; yvecs(:,j)=[]; end
[p,k]=min(pvals); y=yvecs(:,k);
y=[0;y;4*y(n)/3-y(n-1)/3];
[ym,j]=max(abs(y)); y=y/y(j); x=[0;x(:);len];
elseif endc==3
% pinned-fixed case
% (y=0 at x=0 and x=len, y’=0 at x=len)
str='Pinned-Fixed Buckling Load = ';
h=len/nseg; n=nseg; x=linspace(h,len,n);
eiv=lintrp(ei(:,2),ei(:,1),x);
a=-diag(ones(n-1,1),1);
a=a+a'+diag(2*ones(n,1));
% Use a five point backward difference approximation for the second derivative
% at x=len.
v=[35/12,-26/3,19/2,-14/3,11/12];
a(n,n:-1:n-4)=v; a=diag(eiv/h^2)*a;
% Form the equation requiring zero deflection
% at x=len.
b=x(:)’.*[ones(1,n-1),1/2]/eiv(:)’;
% Impose the homogeneous boundary condition
q=null(b); [z,pvals]=eig(q'*a*q);
pvals=diag(pvals);
% Discard any spurious nonpositive eigenvalues
k=find(pvals<=0);
if length(k)>0, pvals(k)=[]; z(:,k)=[]; end;
vecs=q*z; [p,k]=min(pvals); mom=[0;vecs(:,k)];
% Compute the slope and deflection from moment values.
yp=trapsum(0,len,mom./[1;eiv(:)]);
yp=yp-yp(n+1); y=trapsum(0,len,yp);
[ym,j]=max(abs(y)); y=y/y(j); x=[0;x(:)];
else
% fixed-fixed case
% (y and y’ both zero at each end)
str='Fixed-Fixed Buckling Load = ',
h=len/nseg; n=nseg+1; x=linspace(0,len,n);
eiv=lintrp(ei(:,2),ei(:,1),x);
a=-diag(ones(n-1,1),1);
a=a+a'+diag(2*ones(n,1));
% Use five point forward and backward difference approximations for the second derivatives at each end.
v=[-35/12,-26/3,19/2,-14/3,11/12];
a(1:5)=v; a(n,n:-1:n-4)=v;
a=diag(eiv/h^2)*a;
% Write homogeneous equations to make the slope and deflection vanish at x=len.
b=[1/2,ones(1,n-2),1/2]./eiv(:)';
b=[b;x(:)'.*b];
% Impose the homogeneous boundary conditions
q=null(b); [z,pvals]=eig(q'*a*q);
pvals=diag(pvals);
% Discard any spurious nonpositive eigenvalues
k=find(pvals<=0);
if length(k)>0, pvals(k)=[]; z(:,k)=[]; end;
vecs=q*z; [p,k]=min(pvals); mom=vecs(:,k);
% Compute the moment and slope from moment values.
yp=trapsum(0,len,mom./eiv(:));
y=trapsum(0,len,yp);
[ym,j]=max(abs(y)); y=y/y(j);
end
close;
plot(x,y); grid on;
xlabel('axial direction');
ylabel('transverse deflection');
title([str,num2str(p)]); figure(gcf);
print -deps buck

%=============================================%
% function v=trapsum(a,b,y,n)
% ~~~~~~~~~~~~~~~~~~
% This function evaluates:
% % integral(a=>x, y(x)*dx) for a<=x<=b
% % by the trapezoidal rule (which assumes linear
% % function variation between succesive function
% % values).
% % a,b - limits of integration
% % y - integrand which can be a vector valued
% % function returning a matrix such that
% % function values vary from row to row.
% % It can also be input as a matrix with
% % the row size being the number of
% % function values and the column size
% % being the number of components in the
% % vector function.
% % n - the number of function values used to
% % perform the integration. When y is a
% % matrix then n is computed as the number
% % of rows in matrix y.
% % v - integral value
% % User m functions called: none
%----------------------------------------------%

if isstr(y)
% y is an externally defined function
x=linspace(a,b,n); h=x(2)-x(1);
Y=feval(y,x); % Function values must vary in
% row order rather than column
% order or computed results
% will be wrong.
m=size(Y,2);
else
% y is column vector or a matrix
Y=y; [n,m]=size(Y); h=(b-a)/(n-1);
end
v=[zeros(1,m); ... 
    h/2*cumsum(Y(1:n-1,:)+Y(2:n,:))];

%=============================================  
% eilt(h1,h2,L,n,E)  
% ~~~~~~~~~~~~~~~~~~~~  
% This function computes the moment of inertia  
% along a linearly tapered circular cross  
% section and then uses that value to produce  
% the product EI.  
% h1,h2 - column diameters at each end  
% L    - column length  
% n    - number of points at which ei is  
%       computed  
% E    - Young’s modulus  
% ei   - vector of EI values along column  
% User m functions called: none  
%---------------------------------------------  
if nargin<5, E=1; end;
x=linspace(0,L,n)';
ei=E*pi/64*(h1+(h2-h1)/L*x).^4;
ei=[ei(:),x(:)];

%=============================================  
% function y=lintrp(xd,yd,x)  
% See Appendix B

© 2003 by CRC Press LLC
10.6 Accuracy Comparison for Euler Beam Natural Frequencies by Finite Element and Finite Difference Methods

Next we consider three different methods of natural frequency computation for a cantilever beam. Comparisons are made among results from: a) the solution of the frequency equation for the true continuum model; b) the approximation of the equations of motion using finite differences to replace the spatial derivatives; and c) the use of finite element methods yielding a piecewise cubic spatial interpolation of the displacement field. The first method is less appealing as a general tool than the last two methods because the frequency equation is difficult to obtain for geometries of variable cross section. Frequencies found using finite difference and finite element methods are compared with results from the exact model; and it is observed that the finite element method produces results that are superior to those from finite differences for comparable degrees of freedom. In addition, the natural frequencies and mode shapes given by finite elements are used to compute and animate the system response produced when a beam, initially at rest, is suddenly subjected to two concentrated loads.

10.6.1 Mathematical Formulation

The differential equation governing transverse vibrations of an elastic beam of constant depth is [69]

\[
EI \frac{\partial^4 Y}{\partial X^4} = -\rho \frac{\partial^2 Y}{\partial T^2} + W(X, T) \quad 0 \leq X \leq \ell, \quad T \geq 0
\]

where

- \(Y(X, T)\) – transverse displacement,
- \(X\) – horizontal position along the beam length,
- \(T\) – time,
- \(EI\) – product of moment of inertia and Young’s modulus,
- \(\rho\) – mass per unit length of the beam,
- \(W(X, T)\) – external applied force per unit length.

In the present study, we consider the cantilever beam shown in Figure 10.7, having end conditions which are

\[
Y(0, T) = 0, \quad \frac{\partial Y(0, T)}{\partial X} = 0, \quad EI \frac{\partial^2 Y(\ell, T)}{\partial X^2} = M_E(T), \quad \text{and} \quad EI \frac{\partial^3 Y(\ell, T)}{\partial X^3} = V_E(T).
\]

This problem can be expressed more concisely using dimensionless variables

\[
x = \frac{X}{\ell}, \quad y = \frac{Y}{\ell} \quad \text{and} \quad t = \sqrt{\frac{EI}{\rho}} \left(\frac{T}{\ell^2}\right).
\]
Then the differential equation becomes
\[
\frac{\partial^4 y}{\partial x^4} = -\frac{\partial^2 y}{\partial t^2} + w(x, t),
\]
and the boundary conditions reduce to
\[
y(0, t) = 0, \quad \frac{\partial y}{\partial x}(0, t) = 0, \quad \frac{\partial^2 y}{\partial x^2}(1, t) = m_e(t) \quad \text{and} \quad \frac{\partial^3 y}{\partial x^3}(1, t) = v_e(t)
\]
where
\[
w = \left(\frac{W\ell^3}{EI}\right), \quad m_e = \left(\frac{M_E\ell}{EI}\right) \quad \text{and} \quad v_e = \left(\frac{V_E\ell^2}{EI}\right).
\]
The natural frequencies of the system are obtained by computing homogeneous solutions of the form \(y(x, t) = f(x)\sin(\omega t)\) which exist when \(w = m_e = v_e = 0\). This implies
\[
\frac{d^4 f}{dx^4} = \lambda^4 f \quad \text{where} \quad \lambda = \sqrt{\omega},
\]
subject to
\[
f(0) = 0, \quad f'(0) = 0, \quad f''(1) = 0, \quad f'''(1) = 0.
\]
The solution satisfying this fourth order differential equation with homogeneous boundary conditions has the form
\[
f = [\cos(\lambda x) - \cosh(\lambda x)][\sin(\lambda) + \sinh(\lambda)] - [\sin(\lambda x) - \sinh(\lambda x)][\cos(\lambda) + \cosh(\lambda)],
\]
where \(\lambda\) satisfies the frequency equation
\[
p(\lambda) = \cos(\lambda) + 1/\cosh(\lambda) = 0.
\]
Although the roots cannot be obtained explicitly, asymptotic approximations exist for large \(n\):
\[
\lambda_n = (2k - 1)\pi/2.
\]
These estimates can be used as the starting points for finding approximate roots of
the frequency equation using Newton’s method:
\[ \lambda_{NEW} = \lambda_{OLD} - p(\lambda_{OLD})/p'(\lambda_{OLD}). \]

The exact solution will be used to compare related results produced by finite differ-
ence and finite element methods. First we consider finite differences. The following
difference formulas have a quadratic truncation error derivable from Taylor’s series
[1]:

\[ y'(x) = \frac{-y(x - h) + y(x + h)}{(2h)}, \]
\[ y''(x) = \frac{y(x - h) - 2y(x) + y(x + h)}{h^2}, \]
\[ y'''(x) = \frac{-y(x - 2h) + 2y(x - h) - 2y(x) + y(x + 2h)}{(2h)^2}, \]
\[ y''''(x) = \frac{y(x - 2h) - 4y(x - h) + 2y(x) - 4y(x + h)}{(2h)^3}. \]

The step-size is \( h = 1/n \) so that \( x_j = jh, \) \( 0 \leq j \leq n, \) where \( x_0 \) is at the left end
and \( x_n \) is at the right end of the beam. It is desirable to include additional fictitious
points \( x_{-1}, x_{n+1} \) and \( x_{n+2}. \) Then the left end conditions imply
\[ y_0 = y_1 \text{ and } y_{-1} = y_1, \]
and the right end conditions imply
\[ y_{n+1} = -y_{n-1} + 2y_n \text{ and } y_{n+2} = y_{n-2} - 4y_{n-1} + 4y_n. \]

Using these relations, the algebraic eigenvalue problem derived from the difference
approximation is

\[ 7y_1 - 4y_2 + y_3 = \tilde{\lambda}y_1, \]
\[ -4y_1 + 6y_2 - 4y_3 + y_4 = \tilde{\lambda}y_2, \]
\[ y_{j-2} - 4y_{j-1} + 6y_j - 4y_{j+1} + y_{j+2} = \tilde{\lambda}y_j, \quad 2 \leq j \leq n - 1, \]
\[ y_{n-3} - 4y_{n-2} + 5y_{n-1} - 2y_n = \tilde{\lambda}y_{n-1}, \]
\[ 2y_{n-2} - 4y_{n-1} + 2y_n = \tilde{\lambda}y_n, \]

where \( \tilde{\lambda} = h^4\lambda. \)

The finite element method leads to a similar problem involving global mass and
stiffness matrices [54]. When we consider a single beam element of mass \( m \) and
length \( \ell, \) the elemental mass and stiffness matrices found using a cubically varying
displacement approximation are

\[
M_e = \frac{m}{420} \begin{bmatrix}
156 & 22\ell & 54 -13\ell \\
22\ell & 4\ell^2 & 13\ell -3\ell^2 \\
54 & 13\ell & 156 -22\ell \\
-13\ell -3\ell^2 -22\ell & 4\ell^2
\end{bmatrix}, \quad
K_e = \frac{EI}{\ell^3} \begin{bmatrix}
6 & 3\ell & -6 & 3\ell \\
3\ell & 2\ell^2 & -3\ell & \ell^2 \\
-6 & -3\ell & 6 & -3\ell \\
3\ell & \ell^2 & -3\ell & 2\ell^2
\end{bmatrix},
\]

and the elemental equation of motion has the form
\[ M_e Y_e'' + K_e Y_e = F_e \]
where
\[ Y_e = [Y_1, Y_1', Y_2, Y_2']^T \]
\[ F_e = [F_1, M_1, F_2, M_2]^T \]
are generalized elemental displacement and force vectors. The global equation of motion is obtained as an assembly of element matrices and has the form
\[ MY'' + KY = F. \]
A system with \( N \) elements involves \( N + 1 \) nodal points. For the cantilever beam studied here both \( Y_0 \) and \( Y_0' \) are zero. So removing these two variables leaves a system of \( n = 2N \) unknowns. The solution of this equation in the case of a non-resonant harmonic forcing function will be discussed further. The matrix analog of the simple harmonic equation is
\[ MY'' + KY = F_1 \cos(\omega t) + F_2 \sin(\omega t) \]
with initial conditions
\[ Y(0) = Y_0 \text{ and } \dot{Y}(0) = V_0. \]
The solution of this differential equation is the sum of a particular solution and a homogeneous solution:
\[ Y = Y_p + Y_H, \]
where
\[ Y_H = Y_1 \cos(\omega t) + Y_2 \sin(\omega t) \]
with
\[ Y_j = (K - \omega^2 M)^{-1} F_j \quad j = 1, 2. \]
This assumes that \( K - \omega^2 M \) is nonsingular. The homogeneous equation satisfies the initial conditions
\[ Y_H(0) = Y_0 - Y_1, \quad \dot{Y}_H(0) = V_0 - \omega Y_2. \]
The homogeneous solution components have the form
\[ Y_{jH} = U_j \cos(\omega_j t + \phi_j) \]
where \( \omega_j \) and \( U_j \) are natural frequencies and modal vectors satisfying the eigenvalue equation
\[ KU_j = \omega_j^2 MU_j. \]
Consequently, the homogeneous solution completing the modal response is
\[ Y_H(t) = \sum_{j=1}^{n} U_j [\cos(\omega_j t) c_j + \sin(\omega_j t) d_j/\omega_j] \]
where \( c_j \) and \( d_j \) are computed to satisfy the initial conditions which require
\[ C = U^{-1}(Y_0 - Y_1) \text{ and } D = U^{-1}(V_0 - \omega Y_2). \]
The next section presents the MATLAB program. Natural frequencies from finite difference and finite element matrices are compared and modal vectors from the finite element method are used to analyze a time response problem.
10.6.2 Discussion of the Code

A program was written to compare exact frequencies from the original continuous beam model with approximations produced using finite differences and finite elements. The finite element results were also employed to calculate a time response by modal superposition for any structure that has general mass and stiffness matrices, and is subjected to loads which are constant or harmonically varying.

The code below is fairly long because various MATLAB capabilities are applied to three different solution methods. The following function summary involves nine functions, several of which were used earlier in the text.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Line</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbfreq</td>
<td></td>
<td>driver to input data, call computation modules, and print results</td>
</tr>
<tr>
<td>cbfrqwnm</td>
<td></td>
<td>function to compute exact natural frequencies by Newton's method for root calculation</td>
</tr>
<tr>
<td>cbfrqfdm</td>
<td></td>
<td>forms equations of motion using finite differences and calls eig to compute natural frequencies</td>
</tr>
<tr>
<td>cbfrqfem</td>
<td></td>
<td>uses the finite element method to form the equation of motion and calls eig to compute natural frequencies and modal vectors</td>
</tr>
<tr>
<td>frud</td>
<td></td>
<td>function which solves the structural dynamics equation by methods developed in Chapter 7</td>
</tr>
<tr>
<td>examplmo</td>
<td></td>
<td>evaluates the response caused when a downward load at the middle and an upward load at the free end are applied</td>
</tr>
<tr>
<td>animate</td>
<td></td>
<td>plots successive positions of the beam to animate the motion</td>
</tr>
<tr>
<td>plotsave</td>
<td></td>
<td>plots the beam frequencies for the three methods. Also plots percent errors showing how accurate finite element and finite difference methods are</td>
</tr>
<tr>
<td>inputv</td>
<td></td>
<td>reads a sequence of numbers</td>
</tr>
</tbody>
</table>

Table 10.2: Functions Used in the Beam Code

Several characteristics of the functions assembled for this program are worth examining in detail. The next table contains remarks relevant to the code.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Line</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td></td>
<td>Natural frequencies are printed along with error percentages. The output shown here has been extracted from the actual output to show only the highest and lowest frequencies.</td>
</tr>
</tbody>
</table>

continued on next page
Routine | Line | Operation
---|---|---
**cbfrqnwm** | 99 | Asymptotic estimates are used to start a Newton method iteration.
102-108 | | Root corrections are carried out for all roots until the correction to any root is sufficiently small.
**cbfrqfdm** | 135-136 | The equations of motion are formed without corrections for end conditions.
138-145 | | End conditions are applied.
149*150 | **eig** | Computes the frequencies.
**cbfrqfem** | 182-186 | Form elemental mass matrix.
189-192 | | Form elemental stiffness matrix.
198-201 | | Global equations of motion are formed using an element by element loop.
205 | | Boundary conditions are applied requiring zero displacement and slope at the left end, and zero moment and shear at the right end.
208-214 | | Frequencies and modal vectors are computed. Note that modal vector computation is made optional since this takes longer than only computing frequencies.
**frud** | | Compute time response by modal superposition. Theoretical details pertaining to this function appear in Chapter 7.
**examplmo** | 292-296 | The time step and maximum time for response calculation is selected.
300-301 | | Function **frud** is used to compute displacement and rotation response. Only displacement is saved.
304-307 | | Free end displacement is plotted.
314-319 | | A surface showing displacement as a function of position and time is shown.
**animate** | 324-326 | Function **animate** is called.
364-369 | | Window limits are determined.
373-381 | | Each position is plotted. Then it is erased before proceeding to the next position.
**plotsave** | | Plot and save graphs showing the frequencies and error percentages.

Table 10.3: Description of Code in Example

10.6.3 Numerical Results

The dimensionless frequency estimates from the finite difference and the finite element methods were compared for various numbers of degrees-of-freedom. Typical
program output for \( n = 100 \) is shown at the end of this section. The frequency results and error percentages are shown in Figures 10.8 and 10.9. It is evident that the finite difference frequencies are consistently low and the finite element results are consistently high. The finite difference estimates degrade smoothly with increasing order. The finite element frequencies are surprisingly accurate for \( \omega_k \) when \( k < n/2 \). At \( k = n/2 \) and \( k = n \), the finite element error jumps sharply. This peculiar error jump halfway through the spectrum has also been observed in [54]. The most important and useful result seen from Figure 10.9 is that in order to obtain a particular number of frequencies, say \( N \), which are accurate within 3.5%, it is necessary to employ more than \( 2N \) elements and keep only half of the predicted values.

The final result presented is the time response of a beam which is initially at rest when a concentrated downward load of five units is applied at the middle and a one unit upward load is applied at the free end. The time history was computed using function \texttt{frud}. Figure 10.10 shows the time history of the free end. Figure 10.11 is a surface plot illustrating how the deflection pattern changes with time. Finally, Figure 10.12 shows successive deflection positions produced by function \texttt{animate}. The output was obtained by suppressing the graph clearing option for successive configurations.
Figure 10.9: Cantilever Beam Frequency Error Percentages

Figure 10.10: Position of the Free End of the Beam versus Time
Cantilever Beam Deflection for Varying Position and Time

Figure 10.11: Beam Deflection History

Beam Animation

Figure 10.12: Beam Animation
MATLAB Example

Output from Example

>> cbfreq

CANTILEVER BEAM FREQUENCIES BY FINITE DIFFERENCE AND
FINITE ELEMENT APPROXIMATION

Give the number of frequencies to be computed
(use an even number greater than 2)
? > 100

<table>
<thead>
<tr>
<th>freq. number</th>
<th>freq. exact.</th>
<th>fdif. freq.</th>
<th>fd. pct.</th>
<th>felt. freq.</th>
<th>fe. pct.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.51602e+00</td>
<td>3.51572e+00</td>
<td>-0.008</td>
<td>3.51602e+00</td>
<td>0.000</td>
</tr>
<tr>
<td>2</td>
<td>2.20345e+01</td>
<td>2.20250e+01</td>
<td>-0.043</td>
<td>2.20345e+01</td>
<td>0.000</td>
</tr>
<tr>
<td>3</td>
<td>6.16972e+01</td>
<td>6.16414e+01</td>
<td>-0.090</td>
<td>6.16972e+01</td>
<td>0.000</td>
</tr>
<tr>
<td>4</td>
<td>1.20902e+02</td>
<td>1.20714e+02</td>
<td>-0.155</td>
<td>1.20902e+02</td>
<td>0.000</td>
</tr>
<tr>
<td>5</td>
<td>1.99860e+02</td>
<td>1.99386e+02</td>
<td>-0.237</td>
<td>1.99860e+02</td>
<td>0.000</td>
</tr>
<tr>
<td>6</td>
<td>2.98556e+02</td>
<td>2.97558e+02</td>
<td>-0.334</td>
<td>2.98556e+02</td>
<td>0.001</td>
</tr>
<tr>
<td>7</td>
<td>4.16991e+02</td>
<td>4.15123e+02</td>
<td>-0.448</td>
<td>4.16999e+02</td>
<td>0.002</td>
</tr>
<tr>
<td>8</td>
<td>5.55165e+02</td>
<td>5.51957e+02</td>
<td>-0.578</td>
<td>5.55184e+02</td>
<td>0.003</td>
</tr>
<tr>
<td>9</td>
<td>7.13079e+02</td>
<td>7.07918e+02</td>
<td>-0.724</td>
<td>7.13119e+02</td>
<td>0.006</td>
</tr>
<tr>
<td>10</td>
<td>8.90732e+02</td>
<td>8.82842e+02</td>
<td>-0.886</td>
<td>8.90809e+02</td>
<td>0.009</td>
</tr>
<tr>
<td>11</td>
<td>1.08812e+03</td>
<td>1.07655e+03</td>
<td>-1.064</td>
<td>1.08826e+03</td>
<td>0.013</td>
</tr>
<tr>
<td>12</td>
<td>1.30526e+03</td>
<td>1.28884e+03</td>
<td>-1.257</td>
<td>1.30550e+03</td>
<td>0.019</td>
</tr>
<tr>
<td>13</td>
<td>1.54213e+03</td>
<td>1.51950e+03</td>
<td>-1.467</td>
<td>1.54252e+03</td>
<td>0.026</td>
</tr>
<tr>
<td>14</td>
<td>1.79874e+03</td>
<td>1.76830e+03</td>
<td>-1.692</td>
<td>1.79937e+03</td>
<td>0.035</td>
</tr>
<tr>
<td>15</td>
<td>2.07508e+03</td>
<td>2.03497e+03</td>
<td>-1.933</td>
<td>2.07605e+03</td>
<td>0.047</td>
</tr>
<tr>
<td>16</td>
<td>2.37117e+03</td>
<td>2.31926e+03</td>
<td>-2.189</td>
<td>2.37261e+03</td>
<td>0.061</td>
</tr>
<tr>
<td>17</td>
<td>2.68700e+03</td>
<td>2.62088e+03</td>
<td>-2.461</td>
<td>2.68908e+03</td>
<td>0.077</td>
</tr>
<tr>
<td>18</td>
<td>3.02257e+03</td>
<td>2.93951e+03</td>
<td>-2.748</td>
<td>3.02551e+03</td>
<td>0.098</td>
</tr>
<tr>
<td>19</td>
<td>3.37787e+03</td>
<td>3.27486e+03</td>
<td>-3.050</td>
<td>3.38197e+03</td>
<td>0.121</td>
</tr>
<tr>
<td>20</td>
<td>3.75292e+03</td>
<td>3.62657e+03</td>
<td>-3.367</td>
<td>3.75851e+03</td>
<td>0.149</td>
</tr>
</tbody>
</table>

====== INTERMEDIATE LINES OF OUTPUT DELETED ======

<table>
<thead>
<tr>
<th>freq. number</th>
<th>freq. exact.</th>
<th>fdif. freq.</th>
<th>fd. pct.</th>
<th>felt. freq.</th>
<th>fe. pct.</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>7.90580e+04</td>
<td>3.88340e+04</td>
<td>-50.879</td>
<td>1.09328e+05</td>
<td>38.288</td>
</tr>
<tr>
<td>91</td>
<td>8.08345e+04</td>
<td>3.90347e+04</td>
<td>-51.710</td>
<td>1.11989e+05</td>
<td>38.541</td>
</tr>
<tr>
<td>92</td>
<td>8.26308e+04</td>
<td>3.92169e+04</td>
<td>-52.540</td>
<td>1.14512e+05</td>
<td>38.582</td>
</tr>
<tr>
<td>93</td>
<td>8.44468e+04</td>
<td>3.93804e+04</td>
<td>-53.367</td>
<td>1.16860e+05</td>
<td>38.384</td>
</tr>
<tr>
<td>94</td>
<td>8.62825e+04</td>
<td>3.95250e+04</td>
<td>-54.191</td>
<td>1.18999e+05</td>
<td>37.917</td>
</tr>
<tr>
<td>95</td>
<td>8.81380e+04</td>
<td>3.96507e+04</td>
<td>-55.013</td>
<td>1.20889e+05</td>
<td>37.159</td>
</tr>
<tr>
<td>96</td>
<td>9.00133e+04</td>
<td>3.97572e+04</td>
<td>-55.832</td>
<td>1.22496e+05</td>
<td>36.086</td>
</tr>
<tr>
<td>97</td>
<td>9.19082e+04</td>
<td>3.98445e+04</td>
<td>-56.648</td>
<td>1.23786e+05</td>
<td>34.684</td>
</tr>
<tr>
<td>98</td>
<td>9.38229e+04</td>
<td>3.99125e+04</td>
<td>-57.460</td>
<td>1.24730e+05</td>
<td>32.941</td>
</tr>
<tr>
<td>99</td>
<td>9.57574e+04</td>
<td>3.99611e+04</td>
<td>-58.268</td>
<td>1.25305e+05</td>
<td>30.857</td>
</tr>
<tr>
<td>100</td>
<td>9.77116e+04</td>
<td>3.99903e+04</td>
<td>-59.073</td>
<td>1.49694e+05</td>
<td>53.200</td>
</tr>
</tbody>
</table>

Evaluate the time response from two
concentrated loads. One downward at the
middle and one upward at the free end.

input the time step and the maximum time
(0.04 and 5.0) are typical. Use 0.0 to stop
Evaluate the time response resulting from a concentrated downward load at the middle and an upward end load.

input the time step and the maximum time (0.04 and 5.0) are typical. Use 0,0 to stop

Program cbfreq

function cbfreq

% Example: cbfreq
% ~~~~~~~~~~~~~~~
% This program computes approximate natural frequencies of a uniform depth cantilever beam using finite difference and finite element methods. Error results are presented which demonstrate that the finite element method is much more accurate than the finite difference method when the same matrix orders are used in computation of the eigenvalues.

% User m functions required:
% cbfrqsnwm, cbfrqfdm, cbfrqfem, frud,
% examplmo, beamanim, plotsave, inputv

clear, fprintf('

')
fprintf('CANTILEVER BEAM FREQUENCIES BY ')
fprintf('FINITE DIFFERENCE AND'
fprintf(...
fprintf('n               FINITE ELEMENT APPROXIMATION\n')

fprintf('
Give the number of frequencies '
fprintf('to be computed')
fprintf('
(use an even number greater ')
fprintf('than 2)\n', n=input('? > ');
if rem(n,2) ~= 0, n=n+1; end

% Exact frequencies from solution of
% the frequency equation
wex = cbfrqsnwm(n,1e-12);
% Frequencies for the finite difference solution
wfd = cbfrqfdm(n);

% Frequencies, modal vectors, mass matrix, and stiffness matrix from the finite element solution.
nelts=n/2; [wfe,mv,mm,kk] = cbfrqfem(nelts);
pefdm=(wfd-wex)./(.01*wex);
pefem=(wfe-wex)./(.01*wex);

nlines=17; nloop=round(n/nlines);
v=[([1:n]'),wex,wfd,pefdm,wfe,pefem];
disp(','), lo=1;
t1=[' freq. exact. fdif.' ...
    ', fd. pct.']);
t1=[t1,', felt. fe. pct.']));
t2=['number freq. freq.' ...
    ', error ']);
t2=[t2,', freq. error ']);
while lo < n
disp(t1),disp(t2)
    hi=min(lo+nlines-1,n);
    for j=lo:hi
        s1=sprintf('
 %4.0f %13.5e %13.5e', ...
            v(j,1),v(j,2),v(j,3));
        s2=sprintf(' %9.3f %13.5e %9.3f', ...
            v(j,4),v(j,5),v(j,6));
        fprintf([s1,s2])
    end
    fprintf('
Press [Enter] to continue

');
    pause;
    lo=lo+nlines;
end
plotsave(wex,wfd,pefdm,wfe,pefem)
nfe=length(wfe); nmidl=nfe/2;
if rem(nmidl,2)==0, nmidl=nmidl+1; end
x0=zeros(nfe,1); v0=x0; w=0;
f1=zeros(nfe,1); f2=f1; f1(nfe-1)=1;
f1(nmidl)=-5;
xsav=examplmo(mm,kk,f1,f2,x0,v0,wfe,mv);
close; fprintf('All Done
')

%=============================================
function z=cbfrqwnm(n,tol)
% z=cbfrqwnm(n,tol)
% Cantilever beam frequencies by Newton's method. Zeros of
% f(z) = cos(z) + 1/cosh(z)
% are computed.
% n - Number of frequencies required
% tol - Error tolerance for terminating the iteration
% z - Dimensionless frequencies are the squares of the roots of f(z)=0
% User m functions called: none
%----------------------------------------------
if nargin ==1, tol=1.e-5; end

% Base initial estimates on the asymptotic form of the frequency equation
zbegin=((1:n)-.5)'*pi; zbegin(1)=1.875; big=10;

% Start Newton iteration
while big > tol
    t=exp(-zbegin); tt=t.*t;
    f=cos(zbegin)+2*t./(1+tt);
    fp=-sin(zbegin)-2*t.*(1-tt)./(1+tt).^2;
    delz=-f./fp;
    z=zbegin+delz; big=max(abs(delz)); zbegin=z;
end
z=z.*z;

%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
function [wfindif,mat]=cbfrqfdm(n)
% [wfindif,mat]=cbfrqfdm(n)
% This function computes approximate cantilever beam frequencies by the finite difference method. The truncation error for the differential equation and boundary conditions are of order h^2.
% n - Number of frequencies to be computed
% wfindif - Approximate frequencies in dimensionless form
% mat - Matrix having eigenvalues which are the square roots of the frequencies
%
% User m functions called: none

Form the primary part of the frequency matrix
mat=3*diag(ones(n,1))-4*diag(ones(n-1,1),1)+
    diag(ones(n-2,1),2); mat=(mat+mat');

% Impose left end boundary conditions
% y(0)=0 and y'(0)=0
mat(1,[1:3])=[7,-4,1]; mat(2,[1:4])=[-4,6,-4,1];

% Impose right end boundary conditions
% y''(1)=0 and y'''(1)=0
mat(n-1,[n-3:n])=[1,-4,5,-2];
mat(n,[n-2:n])=[2,-4,2];

% Compute approximate frequencies and sort these values
w=eig(mat); w=sort(w); h=1/n;
wfindif=sqrt(w)/(h*h);

%----------------------------------------------

function [wfem,modvecs,mm,kk]= ...
    cbfrqfem(nelts,mas,len,ei)

% [wfem,modvecs,mm,kk]=
% cbfrqfem(nelts,mas,len,ei)
% Determination of natural frequencies of a uniform depth cantilever beam by the Finite Element Method.

% nelts - number of elements in the beam
% mas - total beam mass
% len - total beam length
% ei - elastic modulus times moment of inertia
%wfem - dimensionless circular frequencies
%modvecs - modal vector matrix
%mm,kk - reduced mass and stiffness matrices

% User m functions called: none
%----------------------------------------------

if nargin==1, mas=1; len=1; ei=1; end
n=nelts; le=len/n; me=mas/n;
c1=6/le^2; c2=3/le; c3=2*ei/le;

% element mass matrix
masselt=me/420* ...
[ 156, 22*le, 54, -13*le
 22*le, 4*le^2, 13*le, -3*le^2
 54, 13*le, 156, -22*le
-13*le, -3*le^2, -22*le, 4*le^2];

% element stiffness matrix
stifelt=c3*[ c1, c2, -c1, c2
 c2, 2, -c2, 1
-c1, -c2, c1, -c2
 c2, 1, -c2, 2];

ndof=2*(n+1); jj=0:3;
mm=zeros(ndof); kk=zeros(ndof);

% Assemble equations
for i=1:n
 j=2*i-1+jj; mm(j,j)=mm(j,j)+masselt;
kk(j,j)=kk(j,j)+stifelt;
end

% Remove degrees of freedom for zero
% deflection and zero slope at the left end.
mm=mm(3:ndof,3:ndof); kk=kk(3:ndof,3:ndof);

% Compute frequencies
if nargout ==1
 wfem=sqrt(sort(real(eig(mm\kk))));
else
 [modvecs,wfem]=eig(mm\kk);
[wfem,id]=sort(diag(wfem));
wfem=sqrt(wfem); modvecs=modvecs(:,id);
end

%=============================================% 
function [t,x]=...
frud(m,k,f1,f2,w,x0,v0,wn,modvc,h,tmax)
% [t,x]=frud(m,k,f1,f2,w,x0,v0,wn,modvc,h,tmax)
% This function employs modal superposition
% to solve
% m*x'' + k*x = f1*cos(w*t) + f2*sin(w*t)
% m,k - mass and stiffness matrices
% f1,f2 - amplitude vectors for the forcing function
% w - forcing frequency not matching any natural frequency component in wn
% wn - vector of natural frequency values
% x0,v0 - initial displacement and velocity vectors
% modvc - matrix with modal vectors as its columns
% h,tmax - time step and maximum time for evaluation of the solution
% t - column of times at which the solution is computed
% x - solution matrix in which row j is the solution vector at time t(j)
% User m functions called: none
%----------------------------------------------
t=0:h:tmax; nt=length(t); nx=length(x0);
wn=wn(:); wnt=wn*t;

% Evaluate the particular solution.
x12=(k-(w*w)*m)\[f1,f2];
x1=x12(:,1); x2=x12(:,2);
xp=x1*cos(w*t)+x2*sin(w*t);
% Evaluate the homogeneous solution.
cof=modvc*[x0-x1,v0-w*x2];
c1=cof(:,1); c2=(cof(:,2)./wn)';
xh=(modvc.*c1(ones(1,nx),:))*cos(wnt)+...
   (modvc.*c2(ones(1,nx),:))*sin(wnt);

% Combine the particular and
% homogeneous solutions.
t=t(:); x=(xp+xh)';

function x=examplmo(mm,kk,f1,f2,x0,v0,wfe,mv)
% x=examplmo(mm,kk,f1,f2,x0,v0,wfe,mv)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% Evaluate the response caused when a downward
% load at the middle and an upward load at the
% free end is applied.
% mm, kk - mass and stiffness matrices
% f1, f2 - forcing function magnitudes
% x0, v0 - initial position and velocity
% wfe - forcing function frequency
% mv - matrix of modal vectors
% User m functions called: frud, beamanim, inputv

w=0; n=length(x0); t0=0; x=[];
s1=['\nEvaluate the time response from two',...'
concentrated loads. One downward at the',...'
middle and one upward at the free end.'];
while 1
    fprintf(s1), fprintf('

')
    fprintf('Input the time step and ')fprintf('the maximum time ')fprintf('
(0.04 and 5.0) are typical.')fprintf(' Use 0,0 to stop
')fprintf('
')
    [h,tmax]=inputv;
    if norm([h,tmax])==0 | isnan(h), return, end
    disp('')
    [t,x]= ... frud(mm,kk,f1,f2,w,x0,v0,wfe,mv,h,tmax);

© 2003 by CRC Press LLC
x = x(:, 1:2:n-1); x = [zeros(length(t), 1), x];
[nt, nc] = size(x); hdist = linspace(0, 1, nc);
clf, plot(t, x(:, nc), 'k-')
title('Position of the Free End of the Beam')
xlabel('dimensionless time')
ylabel('end deflection'), figure(gcf)
disp('Press [Enter] for a surface plot of')
disp('transverse deflection versus x and t')
pause
print -deps endpos1
xc = linspace(0, 1, nc); zmax = 1.2 * max(abs(x(:))); clf,
surf(xc, t, x), view(30, 35)
colormap([1 1 1])
axis([0, 1, 0, tmax, -zmax, zmax])
xlabel('x axis'); ylabel('time')
zlabel('deflection')
title(['Cantilever Beam Deflection ' ...
      'for Varying Position and Time'])
figure(gcf);
print -deps endpos2
disp(' '), disp(['Press [Enter] to animate', ...
      'the beam motion'])
pause

titl = 'Cantilever Beam Animation';
xlab = 'x axis'; ylab = 'displacement';
beamanim(hdist, x, 0.1, titl, xlab, ylab), close
end

%=======================================================================
% function beamanim(x, u, t, pause, titl, xlab, ylab)
% See Appendix B

%=======================================================================
function plotsave(wex, wfd, pefd, wfe, pefem)
% % function plotsave(wex, wfd, pefd, wfe, pefem)
% % This function plots errors in frequencies
% % computed by two approximate methods.

© 2003 by CRC Press LLC
% wex    - exact frequencies
% wfd    - finite difference frequencies
% wfe    - finite element frequencies
% pefd, pefem - percent errors by both methods

% User m functions called: none
%----------------------------------------------
% plot results comparing accuracy
% of both frequency methods
w=[wex(:);wfd(:);wfd];
wm=min(w); wmax=max(w);
n=length(wex); wht=wmin+.001*(wmax-wmin);
j=1:n;

semilogy(j,wex,'k-',j,wfe,'k--',j,wfd,'k:');
title('Cantilever Beam Frequencies')
xlabel('frequency number')
ylabel('frequency values')
legend('Exact freq.','Felt. freq.','Fdif. freq.',2); figure(gcf)
disp(['Press [Enter] for a frequency ',...
'error plot']), pause
print -deps beamfrq1

plot(j,abs(pefd),'k--',j,abs(pefem),'k-')
title(['Cantilever Beam Frequency ' ...
'Error Percentages'])
xlabel('frequency number')
ylabel('percent frequency error')
legend('Fdif. pct. error','Felt. pct. error',4)
figure(gcf)
disp(['Press [Enter] for a transient ',...
'response calculation'])
pause
print -deps beamfrq2

%=============================================
% function varargout=inputv(prompt)
% See Appendix B
10.7 Vibration Modes of an Elliptic Membrane

10.7.1 Analytical Formulation

Examples using eigenvalues and modal functions of rectangular or circular membranes were presented in chapter 9. In this section we analyze modal vibrations of an elliptic membrane. In this case the natural frequencies and modal functions cannot be obtained easily in explicit form. The problem can be formulated in elliptical coordinates leading to Mathieu type differential equations [74]. Library routines to compute these functions are not widely available; so, a different approach is employed using least squares approximation and the MATLAB function `eig`. Consider a membrane with major and minor semi-diameters $a$ and $b$. The analytic function $z = h \cosh(\zeta)$ where $h = \sqrt{a^2 - b^2}$ and $\zeta = \xi + i \eta$ maps the rectangle defined by $0 \leq \xi \leq R = \tanh^{-1}(b/a)$, $-\pi \leq \eta \leq \pi$ onto the interior of the ellipse. This transformation takes lines of constant $\xi$ into a system of confocal ellipses and lines of constant $\eta$ into hyperbolas intersecting the ellipses orthogonally. The following function was used to produce the elliptic coordinate plot in Figure 10.13.

```matlab
function z = elipmap(a,b,neta,nxi)
    h=sqrt(a^2-b^2); R=atanh(b/a);
    [xi,eta]=meshgrid(...
linspace(0,R,nxi),linspace(-pi,pi,neta));
    z=h*cosh(xi+i*eta); x=real(z); y=imag(z);
    plot(x,y,'k',x',y','k')
    title('ELLIPTICAL COORDINATE SYSTEM')
    xlabel('x axis'), ylabel('y axis')
    axis equal, grid off, shg
```

Transforming the wave equation to $(\xi, \eta)$ coordinates gives

$$U_{\xi\xi} + U_{\eta\eta} = \frac{h^2}{2} [\cosh(2\xi) - \cos(2\eta)] U_{tt},$$

and assuming separable solutions of the form

$$U = f(\eta)g(\xi)\sin(\Omega t)$$

leads to

$$\frac{f''(\eta)}{f(\eta)} + \frac{g''(\xi)}{g(\xi)} = -\lambda [\cosh(2\xi) - \cos(2\eta)],$$

where $\lambda = \Omega^2 h^2 / 2$. So $f$ and $g$ are found to satisfy the following two Mathieu type differential equations:

$$f''(\eta) + [\alpha - \lambda \cos(2\eta)] f(\eta) = 0, \quad -\pi \leq \eta \leq \pi$$
and
\[ g''(\xi) - [\alpha - \lambda \cosh(2\xi)]g(\xi) = 0, \quad 0 \leq \xi \leq R \]
where the eigenvalue parameters \( \alpha \) and \( \lambda \) are determined to make \( f(\eta) \) have period \( 2\pi \) and make \( g(\xi) \) vanish at \( \xi = R \). The modal functions can be written in terms of Mathieu functions as products of the form
\[ ce(\eta, q)Ce(\xi, q) \]
for modes symmetric about the \( x \)-axis and
\[ se(\eta, \eta\overbar{)}Se(\xi, \eta) \]
for modes anti-symmetric about the \( x \)-axis. The functions \( ce \) and \( se \) are periodic Mathieu functions pertaining to the circumferential direction, while \( Ce \) and \( Se \) are modified Mathieu functions pertaining to the radial direction. The structure of these functions motivates using the following series approximation for the functions for even modes:
\[ f(\eta) = \sum_{k=1}^{N} \cos(\eta(k - 1)) a_k, \quad g(\xi) = \sum_{l=1}^{M} \cos\left(\frac{\pi \xi}{R} (l - 1/2)\right) b_l, \]
The analogous approximations for the modes anti-symmetric about the x-axis are:

\[ f(\eta) = \sum_{k=1}^{N} \sin(\eta k) a_k, \quad g(\xi) = \sum_{l=1}^{M} \sin(\frac{\pi \xi}{R} l) b_l. \]

Thus the expressions for both cases take the form:

\[ f(\eta) = \sum_{k=1}^{N} f_k(\eta) a_k \quad \text{and} \quad g(\xi) = \sum_{l=1}^{M} g_l(\xi) b_l. \]

Let us choose a set of collocation points \( \eta_i, i = 1, \ldots, n \), and \( \xi_j, j = 1, \ldots, m \). Then substituting the series approximation for \( f(\eta) \) into the differential equation gives the following over-determined system of equations:

\[ \sum_{k=1}^{N} f_k''(\eta_i) a_k + \alpha \sum_{k=1}^{N} f_k(n_i) a_k - \lambda \cos(2\eta_i) \sum_{k=1}^{N} f_k(\eta_i) a_k = 0, \quad i = 1, \ldots, n. \]

Denote \( F \) as the matrix having \( f_k(\eta_i) \) as the element in row \( i \) and column \( k \). Then multiplying the last equation on the left by the generalized inverse of \( F \) gives a matrix equation of the form

\[ CA + \alpha A - \lambda DA = 0, \]

where \( A \) is a column matrix consisting of the coefficients \( a_k \). A similar equation results when the series for \( g(\xi) \) is substituted into the differential equation for the radial direction. It reduces to

\[ EB - \alpha B + \lambda GB = 0. \]

The parameter \( \alpha \) can be eliminated from the last two equations to yield a single eigenvalue equation

\[ WE' + CW = \lambda (-WG' + DW) \]

where \( W = AB' \), and the tic mark indicates matrix transposition. By addressing the two-dimensional array \( W \) in terms of a single index, the eigenvalues \( \lambda \) and the modal multipliers defined by \( W \) can be computed using the function \texttt{eig}. Then the values of the other eigenvalue parameter \( \alpha \) can also be obtained using the known \( \lambda, W \) combinations. The mathematical developments just given are implemented below in a program which animates the various natural frequency vibration modes for an elliptic membrane.

### 10.7.2 Computer Formulation

The program \texttt{elipfreq} was written to compute frequencies and mode shapes for an elliptic membrane. The primary data input includes the ellipse semi-diameters, a flag indicating whether even modes, odd modes, or both are desired, the number of
least squares points used, and the number of terms used in the approximation series. Natural frequencies and data needed to produce modal surfaces are returned. The program also animates the various mode shapes arranged in the order of increasing frequency. The modules employed are described in the following table.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>elipfreq</td>
<td>reads data, calls other computational modules, and outputs modal plots</td>
</tr>
<tr>
<td>frqsimpl</td>
<td>forms the matrix approximations of the Mathieu equations and calls eigenrec to generate frequencies and mode shapes</td>
</tr>
<tr>
<td>eigenrec</td>
<td>solves the rectangular eigenvalue problem</td>
</tr>
<tr>
<td>plotmode</td>
<td>generates animated plots of the modal functions</td>
</tr>
<tr>
<td>modeshap</td>
<td>computes modal function shapes using the approximating function series</td>
</tr>
<tr>
<td>funcxi</td>
<td>approximating series functions in the xi variable</td>
</tr>
<tr>
<td>funceta</td>
<td>approximating series functions in the eta variable</td>
</tr>
</tbody>
</table>

The accuracy of the formulation developed above was assessed by 1) comparison with circular membrane frequencies known in terms of Bessel function roots and 2) results obtained from the commercial PDE toolbox from MathWorks employing triangular finite element analysis. The elliptic coordinate formulation is singular for a circular shape, but a nearly circular shape with \( a = 1 \) and \( b = 0.9999 \) causes no numerical difficulty. Figure 10.14 shows how well frequencies from elipfreq with \( \text{nlsq}=[200,200] \) and \( \text{nfun}=30,30 \) compare with the roots of \( J_n(r) \). The first fifty frequencies were accurate to within 0.8 percent and the first one hundred frequencies were accurate to within 5 percent. The function pdetool from the PDE toolbox was also used to compute circular membrane frequencies with a quarter circular shape and 2233 node points. The first two hundred even mode frequencies from this model were accurate to within 1 percent for the first one hundred frequencies and to within 7 percent for the first 200 frequencies. Since the function pdetool would probably give comparable accuracy for an elliptic membrane, results from elipfreq were compared with those from pdetool using an ellipse with \( a = 1 \) and \( b = 0.5 \). The percent difference between the frequencies from the two methods appears in Figure 10.15. This comparison suggests that the first fifty frequencies produced by elipfreq for the elliptic membrane are probably accurate to within about 2 percent.

The various modal surfaces of an elliptic membrane have interesting shapes. The program elipfreq allows a sequence of modes to be exhibited by selecting vectors of frequency numbers such as 1:10 or 10:2:20. Two typical shapes are shown in Figures 10.16 and 10.17. The particular modes shown have no special significance besides their esthetic appeal. A listing of some interactive computer output and the source code for elipfreq follows.
Figure 10.14: Comparing Elipfreq Results with Bessel Function Roots

Figure 10.15: Comparing Elipfreq Results with PDE Toolbox
Figure 10.16: Surface for Anti-Symmetric Mode Number 98
Figure 10.17: Surface for Symmetric Mode Number 99
Interactive Input-Output for Program elipfreq

>> elipfreq;

VIBRATION MODE SHAPES AND FREQUENCIES
OF AN ELLIPTIC MEMBRANE

Input the major and minor semi-diameters > ? 1,.5

Select the modal form option
1<=>even, 2<=>odd, 3<=>both > ? 1

The computation takes awhile. Please wait.

Computation time  = 44.1 seconds.
Number of modes   = 312
Highest frequency = 116.979

Press return to see modal plots.

Give a vector of mode indices (try 10:2:20)
enter 0 to stop > ? 1

Give a vector of mode indices (try 10:2:20)
enter 0 to stop > ? 2:6

Give a vector of mode indices (try 10:2:20)
enter 0 to stop > ? [20 25 30]

Give a vector of mode indices (try 10:2:20)
enter 0 to stop > ? 0

>>

Elliptic Membrane Program

function [frqs,modes,indx,x,y,alpha,cptim]=elipfreq(...
a,b,type,nlsq,nfuns,noplot)

% [frqs,modes,indx,x,y,alpha,cptim]=elipfreq(...
% a,b,type,nlsq,nfuns,noplot)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function computes natural frequencies and mode
% shapes for an elliptical membrane. Modes that are
% symmetrical or anti-symmetrical about the x axis are
% included. An approximate solution is obtained using

© 2003 by CRC Press LLC
% a separation of variables formulation in elliptical
% coordinates.

% a,b  - the ellipse major and minor semi-
% diameters along the x and y axes
% nlsq  - two-component vector giving the number
% of least square points in the eta and
% xi directions
% nfuns - two-component vector giving the number of
% functions used to solve the differential
% equations for the eta and xi directions.
% type  - use 1 for even modes symmetric about the
% x-axis. Use 2 for odd modes anti-
% symmetric about the x-axis. Use 3 to
% combine both even and odd modes.
% frqs  - a vector of natural frequencies
% arranged in increasing order.
% modes - a three dimensional array in which
% modes(:,:,j) defines the modal
% deflection surface for frequency
% frqs(j).
% indx  - a vector telling whether each
% mode is even (1) or odd (2)
% x,y  - curvilinear coordinate arrays of
% points in the membrane where modal
% function values are computed.
% alpha - a vector of eigenvalue parameters in
% the Mathieu equation: u''(eta)+(alpha-lambda*cos(2*eta))*u(eta)=0
% where lambda=(h*freq)^2/2 and
% h=atanh(b/a)
% cptim - the cpu time in seconds used to
% form the equations and solve for
% eigenvalues and eigenvectors
% noplot - enter any value to skip mode plots
%
% User m functions called:
% frqsimpl eigenrec plotmode
% modeshap funcxi funceta

if nargin==0
    disp(' ')
    disp('VIBRATION MODE SHAPES AND FREQUENCIES')
    disp(' OF AN ELLIPTIC MEMBRANE')

© 2003 by CRC Press LLC
disp(' ')
nlsq=[300,300]; nfuns=[25,25];
v=input(['Input the major and minor ',... 
'semi-diameters > ? '],'s');
v=eval(['[',v,',']']); a=v(1); b=v(2); disp(' ')
disp('Select the modal form option')
type=input(... 
'1<=>even, 2<=>odd, 3<=>both > ? ');
disp(' ')
disp(['The computation takes awhile.',... 
' PLEASE WAIT.'])
end
if type ==1 | type==2 % Even or odd modes
[frqs,modes,x,y,alpha,cptim]=frqsimpl(...
a,b,type,nlsq,nfuns);
indx=ones(length(frqs),1)*type;
else % Both modes
[frqs,modes,x,y,alpha,cptim]=frqsimpl(...
a,b,1,nlsq,nfuns);
indx=ones(length(frqs),1);
[frqso,modeso,x,y,alphao,cpto]=frqsimpl(...
a,b,2,nlsq,nfuns);
frqs=[frqs;frqso]; alpha=[alpha;alphao];
modes=cat(3,modes,modeso);
indx=[indx;2*ones(length(frqso),1)];
[frqs,k]=sort(frqs); modes=modes(:,:,k);
indx=indx(k); cptim=cptim+cpto;
end
if nargin==6, return, end

% Plot a sequence of modal functions
neig=length(frqs);
disp(' '), disp(['Computation time = ',... 
num2str(sum(cptim)),' seconds.'])
disp(['Number of modes = ',num2str(neig)]);
disp(['Highest frequency = ',... 
num2str(frqs(end))]); disp(' ')
disp('Press return to see modal plots.')
pause, plotmode(a,b,x,y,frqs,modes,indx)
%
%==============================================
function [frqs, Modes, x, y, alpha, cptim] = frqsimpl(...
    a, b, type, nlsq, nfuns)

% [frqs, Modes, x, y, alpha, cptim] = frqsimpl(...
%    a, b, type, nlsq, nfuns)
% % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% % a, b - ellipse major and minor semi-diameters
% % type - numerical values of one or two for modes
% % symmetric or anti-symmetric about the x axis
% % nlsq - vector [neta, nxi] giving the number of least
% % square points used for the eta and xi
% % directions
% % nfuns - vector [meta, mxi] giving the number of
% % approximating functions used for the eta and
% % xi directions
% % frqs - natural frequencies arranged in increasing
% % order
% % Modes - modal surface shapes in the ellipse
% % x, y - coordinate points in the ellipse
% % alpha - vector of values for the eigenvalues in the
% % Mathieu differential equation:
% % u''(eta) + (alpha - lambda * cos(2*eta)) * u(eta) = 0
% % cptim - vector of computation times
% % User m functions called: funceta funcxi
% % eigenrec modeshap

if nargin == 0
    a = cosh(2); b = sinh(2); type = 1;
    nlsq = [200, 200]; nfuns = [30, 30];
end

h = sqrt(a^2 - b^2); R = atanh(b/a); neta = nlsq(1); alpha = [];
nxi = nlsq(2); meta = nfuns(1); mxi = nfuns(2);
eta = linspace(0, pi, neta)'; xi = linspace(0, R, nxi)';
[Xi, Eta] = meshgrid(xi, eta); z = h * cosh(Xi + i*Eta);
x = real(z); y = imag(z); cptim = zeros(1, 3);

da = tic; [Veta, A] = funceta(meta, type, eta);
A = Veta \ [A, repmat(cos(2*eta), 1, meta) * Veta];
B = A(:, meta + 1:end); A = A(:, 1:meta);

da = tic; [Vxi, B] = funcxi(mxi, type, xi);
B = B \ [B, repmat(cos(2*xi), 1, mxi) * B];
B = B(:, mxi + 1:end); B = B(:, 1:mxi);

% Form the Mathieu equation for the circumferential
% direction as: A*E + alpha*E - lambda*B*E = 0
tic; [Veta, A] = funceta(meta, type, eta);
A = Veta \ [A, repmat(cos(2*eta), 1, meta) * Veta];
B = A(:, meta + 1:end); A = A(:, 1:meta);

da = tic; [Vxi, B] = funcxi(mxi, type, xi);
B = B \ [B, repmat(cos(2*xi), 1, mxi) * B];
B = B(:, mxi + 1:end); B = B(:, 1:mxi);

% Form the modified Mathieu equation for the radial
% direction as: P*F - alpha*F + lambda*Q*F = 0
145: \[ [V_{xi}, P] = \text{funcxi}(a, b, mxi, type, xi); \]
146: \[ P = V_{xi} \backslash [P, \text{repmat(cosh(2*xi), 1, mxi)} \ast V_{xi}]; \]
147: \[ Q = P(:, mxi+1:end); P = P(:, 1:mxi); \]
148: \[ \text{cptim}(1) = \text{toc}; \text{tic} \]
149:
150: % Solve the eigenvalue problem. This takes most
151: % of the computation time
152: \[ [\text{frqs}, \text{modes}] = \text{eigenrec}(P', A, -Q', B); \]
153: % Keep only half of the modes and frequencies
154: \[ \text{nmax} = \text{fix(length(frqs)/2)}; \text{frqs} = \text{frqs(1:nmax)}; \]
155: \[ \text{modes} = \text{modes}(:, :, 1:nmax); \text{cptim}(2) = \text{toc}; \]
156:
157: % Compute values of the second eigenvalue
158: % parameter in Mathieu's equation
159: \[ \text{alpha} = \text{zeros}(1, \text{nmax}); \text{tic}; \]
160: \[ \text{s} = \text{size(modes)}; \text{s} = \text{s(1:2)}; V_{xi} = V_{xi}'; \]
161:
162: % Obtain the modal surface shapes
163: \[ \text{Neta} = 91; \text{Nxi} = 25; \text{Modes} = \text{zeros}(	ext{Neta}, \text{Nxi}, \text{nmax}); \]
164: % for k=1:nmax
165: \[ \text{Mk} = \text{modes}(:, :, k); [\text{dmk}, K] = \text{max(abs(Mk(:))}); \]
166: \[ [I, J] = \text{ind2sub}(s, K); \text{Ej} = \text{Mk}(:, J); \]
167: \[ \text{alpha}(k) = (B(I, :) \ast \text{Ej} \ast \text{frqs}(k) - A(I, :) \ast \text{Ej}) / \text{Mk}(K); \]
168: \[ [\text{Modes}(:, :, k), x, y] = \text{modeshap}(a, b, type, \text{Mk}, \text{Nxi}, \text{Neta}); \]
169: \[ \text{end} \]
170: \[ \text{frqs} = \text{sqrt}(2 \ast \text{frqs}) / h; \text{cptim}(3) = \text{toc}; \]
171:
172: %=================================================================
173:
174: function [\text{eigs, vecs, Amat, Bmat}] = \text{eigenrec}(A, B, C, D)
175: % [eigs, vecs, Amat, Bmat] = \text{eigenrec}(A, B, C, D)
176: % Solve a rectangular eigenvalue problem of the
177: % form: \[ X \ast A + B \ast X = \lambda \ast (X \ast C + D \ast X) \]
178: %
179: % A, B, C, D - square matrices defining the problem.
180: % A and C have the same size. B and D
181: % have the same size.
182: % eigs - vector of eigenvalues
183: % vecs - array of eigenvectors where vecs(:, :, j)
184: % contains the rectangular eigenvector
185: % for eigenvalue eigs(j)
186: % Amat,
187: % Bmat - matrices that express the eigenvalue
188: % problem as Amat*V=\lambda*Bmat*V
189:  

© 2003 by CRC Press LLC
n=size(B,1); m=size(A,2); s=[n,m]; N=n*m;
for i=1:n
    IK=sub2ind(s,i*ones(1,m),km);
    Bikn=B(i,kn); Dikn=D(i,kn);
    for j=1:m
        I=sub2ind(s,i,j);
        Amat(I,IK)=A(km,j)'; Bmat(I,IK)=C(km,j)';
        KJ=sub2ind(s,kn,j*ones(1,n));
        Amat(I,KJ)=Amat(I,KJ)+ Bikn;
        Bmat(I,KJ)=Bmat(I,KJ)+ Dikn;
    end
[end
[vecs,eigs]=eig(Bmat\Amat);
[eigs,k]=sort(diag(eigs));
vecs=reshape(vecs(:,k),n,m,N);

%===========================================

function plotmode(a,b,x,y,eigs,modes,indx)

% This function makes animated plots of the
% mode shapes of an elliptic membrane for
% various frequencies
% a,b - major and minor semi-diameters
% x,y - arrays of points defining the
% curvilinear coordinate grid
% eigs - vector of sorted frequencies
% modes - array of modal surfaces for
% the corresponding frequencies
% indx - vector of indices designating
% each mode as even (1) or odd (2)

range=[-a,a,-b,b,-a,a];
nf=25; ft=cos(linspace(0,4*pi,nf));
boa=[', B/A = ',num2str(b/a,4));
while 1
    jlim=[];
    while isempty(jlim), disp(' ')
        disp(['Give a vector of mode ',...
            'indices (try 10:2:20) > ? '])
        jlim=input('(input 0 to stop > ? ');
    end
© 2003 by CRC Press LLC
if any(jlim==0)
    disp(' '), disp('All done'), break, end
for j=jlim
    if indx(j)==1, type='EVEN'; f=1;
    else, type = 'ODD'; f=-1; end
    u=a/2*modes(:,:,j);
    for kk=1:nf
        surf(x,y,ft(kk)*u)
        axis equal, axis(range)
        xlabel('x axis'), ylabel('y axis')
        zlabel('u(x,y)')
        title([type,' MODE ',num2str(j),...
        ', OMEGA = ',num2str(eigs(j),4),boa])
        %colormap([127/255 1 212/255])
        colormap([1 1 0])
        drawnow, shg
    end
    pause(1);
end

%==================================================================
function [u,x,y]=modeshap(...
    a,b,type,modemat,nxi,neta,H)
    % [u,x,y]=modeshap(a,b,type,modemat,nxi,neta,H)
    % This function uses the eigenvectors produced by
    % the rectangular eigenvalue solver to form modal
    % surface shapes in cartesian coordinates.
    % a,b      - major and minor semi-diameters
    % type     - 1 for even, 2 for odd
    % modemat  - eigenvector matrix output by eigenrec
    % nxi,neta - number of radial and circumferential
    % coordinate values
    % H        - maximum height of the modal surfaces.
    % The default value is one.
    % u,x,y     - modal surface array and corresponding
    % cartesian coordinate matrices. u(:,::j)
    % gives the modal surface for the j'th
    % natural frequency.
    if nargin<7, H=1; end
if nargin<6, neta=81; end; if nargin<5, nxi=22; end
h=sqrt(a^2-b^2); r=atanh(b/a); x=[]; y=[];
xi=linspace(0,r,nxi); eta=linspace(-pi,pi,neta);
if nargout>1
    [Xi,Eta]=meshgrid(xi,eta); z=h*cosh(Xi+i*Eta);
    x=real(z); y=imag(z);
end
[Neta,Nxi]=size(modemat);
mateta=funceta(Neta,type,eta);
matxi=funcxi(a,b,Nxi,type,xi);
u=mateta*modemat*matxi'; [umax,k]=max(abs(u(:)));
u=H/u(k)*u;

function [f,f2]=funcxi(a,b,n,type,xi)
    % [f,f2]=funcxi(a,b,n,type,xi)
    % This function defines the approximating functions
    % for the radial direction
    % a,b - ellipse major and minor half-diameters
    % n - number of series terms used
    % type - 1 for even valued, 2 for odd valued
    % xi - vector of radial coordinate values
    % f,f2 - matrix of function and second derivative values

    xi=xi(:); nxi=length(xi); R=atanh(b/a);
    if type==1, N=pi/R*(1/2:n); f=cos(xi*N);
    else, N=pi/R*(1:n); f=sin(xi*N); end
    f2=-repmat(N.^2,nxi,1).*f;

function [f,f2]=funceta(n,type,eta)
    % [f,f2]=funceta(n,type,eta)
    % This function defines the approximating functions
    % for the circumferential direction
    % n - number of series terms used
    % type - 1 for even valued, 2 for odd valued
    % xi - vector of circumferential coordinate values
    % f,f2 - matrix of function and second derivative
325. \% values
326:
327. eta=eta(:); neta=length(eta);
328. if type==1, N=0:n-1; f=cos(eta*N);
329. else, N=1:n; f=sin(eta*N); end
330. f2=-repmat(N.^2,neta,1).*f;
Chapter 11

Bending Analysis of Beams of General Cross Section

11.1 Introduction

Elastic beams are important components in many types of structures. Consequently methods to analyze the shear, moment, slope, and deflection in beams with complex loading and general cross section variation are of significant interest. A typical beam of the type considered is shown in Figure 11.1. The study of Euler beam theory is generally regarded as an elementary topic dealt with in undergraduate engineering courses. However, simple analyses presented in standard textbooks usually do not reveal difficulties encountered with statically indeterminate problems and general geometries [115]. Finite element approximations intended to handle arbitrary problems typically assume a piecewise constant depth profile and a piecewise cubic transverse deflection curve. This contradicts even simple instances such as a constant depth beam subjected to a linearly varying distributed load which actually leads to a deflection curve which is a fifth order polynomial. Exact solutions of more involved problems where the beam depth changes linearly, for example, are more complicated. Therefore, an exact analysis of the beam problem is desirable to handle depth variation, a combination of concentrated and distributed loads, and static indeterminacy providing for general end conditions and multiple in-span supports. The current formulation considers a beam carrying any number of concentrated loads and linearly varying distributed loads. The equations for the shear and moment in the beam are obtained explicitly. Expressions for slope and deflection are formulated for evaluation by numerical integration allowing as many integration steps as necessary to achieve high accuracy. A set of simultaneous equations imposing desired constraints at the beam ends and at supports is solved for support reactions and any unknown end conditions. Knowledge of these quantities then allows evaluation of internal load and deformation quantities throughout the beam. The analytical formulation is implemented in a program using a concise problem definition specifying all loading, geometry, and constraint conditions without reference to beam elements or nodal points as might be typical in a finite element formulation. The program and example problem are discussed next.
11.1.1 Analytical Formulation

Solution of beam problems utilizes some mathematical idealizations such as a concentrated load, which implies infinite load intensity acting over an infinitesimal area. Also of importance are linearly varying distributed loads, or ramp loads. Treatment of these entities is facilitated by use of singularity functions [9]. The singularity function of order $n$ is denoted by $<x-x_0>_n$ and is defined as

$$<x-x_0>_n = \begin{cases} 
0, & x < x_0 \\
(x-x_0)^n, & x \geq x_0.
\end{cases}$$

For $n \geq 0$, the function satisfies

$$\int_{0}^{x} <x-x_0>_n \, dx = \frac{<x-x_0>_{n+1}}{n+1}.$$ 

The special case where $n = -1$ is appropriate for describing a concentrated load. The term $<x-x_0>_{-1}^{-1}$ means the limit as $\epsilon \to 0$ of the following function

$$<x-x_0>_{-1}^{-1} = \begin{cases} 
0, & x < x_0, \\
\frac{1}{\epsilon}, & x_0 \leq x \leq (x_0 + \epsilon), \\
0, & x > (x_0 + \epsilon).
\end{cases}$$

Consequently, in the limit as $\epsilon$ approaches zero the integral becomes

$$\int_{0}^{x} <x-x_0>_{-1}^{-1} \, dx = <x-x_0>_0^-.$$ 

Analyzing the loads and deformations in the beam requires computation of the shear, moment, slope, and deflection designated as $v(x), m(x), y'(x),$ and $y(x)$. The beam lies in the range $0 \leq x \leq L$. A total of four end conditions are imposed at $x = 0$ and $x = L$. Normally, two conditions will be specified at each end; so, two unknown conditions applicable at $x = 0$ need to be found during the solution process. Along with the end conditions, interior supports may exist at $x = r_j, 1 \leq j \leq N_s$. Displacements $y_j$ will occur at supports, and the reactions $R_j$, as well as four end
conditions, needed to cause the deflections will have to be determined during the analysis. Within the beam span, the applied loading will consist of known external loads described as \( w_e(x) \) and the support reactions. Fundamentals of Euler beam theory developed in standard textbooks \[9, 102\] imply the following differential and integral relations:

I) Load

\[
\begin{align*}
  v'(x) &= w_e(x) + \sum_{j=1}^{N_s} R_j < x - r_j >^{-1}; \\
  m'(x) &= v,
\end{align*}
\]

II) Shear

\[
\begin{align*}
  v(x) &= v_0 + v_e(x) + \sum_{j=1}^{N_s} R_j < x - r_j >^0; \\
  v_e(x) &= \int_{0}^{x} w_e(x) \, dx;
\end{align*}
\]

III) Moment and Second Derivative

\[
\begin{align*}
  m(x) &= m_0 + v_0 x + m_e(x) + \sum_{j=1}^{N_s} R_j < x - r_j >^1; \\
  m_e(x) &= \int_{0}^{x} v_e(x) \, dx, \\
  y''(x) &= k(x) \left[ m_0 + v_0 x + m_e(x) + \sum_{j=1}^{N_s} R_j < x - r_j >^1 \right], \\
  k(x) &= \frac{1}{E(x) I(x)};
\end{align*}
\]

IV) Slope

\[
\begin{align*}
  y'(x) &= y'_0 + m_0 \int_{0}^{x} k(x) \, dx + v_0 \int_{0}^{x} x k(x) \, dx + \\
                  & \quad \int_{0}^{x} k(x) m_e(x) \, dx + \sum_{j=1}^{N_s} R_j \int_{0}^{x} < x - r_j >^1 k(x) \, dx;
\end{align*}
\]
V) Deflection

\[ y(x) = y_0 + y'_0 x + m_0 \int_0^x \int_0^x k(x) \, dx \, dx + \]

\[ v_0 \int_0^x \int_0^x x k(x) \, dx \, dx + \int_0^x \int_0^x k(x) m_e(x) \, dx \, dx + \]

\[ \sum_{j=1}^{N_s} R_j \int_0^x \int_0^x <x-r_j>^1 k(x) \, dx \, dx \]

where \( E(x)I(x) \) is the product of the Young’s modulus and the cross section moment of inertia, \( y_0, y'_0, v_0, m_0, \) are the left-end values of the deflection, slope, shear and moment respectively. The property \( k(x) \) will be spatially variable unless \( EI \) is constant, which yields the following simple formulas

\[ EIy'(x) = EIy'_0 + m_0 x + \frac{v_0 x^2}{2} + \int_0^x m_e(x) \, dx + \frac{1}{2} \sum_{j=1}^{N_s} R_j <x-r_j>^2, \]

\[ EIy(x) = EI (y_0 + y'_0 x) + \frac{m_0 x^2}{2} + \frac{v_0 x^3}{6} + \int_0^x m_e(x) \, dx + \]

\[ \frac{1}{6} \sum_{j=1}^{N_s} R_j <x-r_j>^3. \]

The external loading conditions employed here can handle most practical situations. It is assumed that several concentrated loads \( F_j \) act at positions \( f_j, 1 \leq j \leq N_f \). Distributed loads are described by linearly varying ramp loads. A typical ramp load starts at position \( p_j \) with intensity \( P_j \) and varies linearly to magnitude \( Q_j \) at position \( q_j \). The ramp load is zero unless \( p_j \leq x \leq q_j \). A total of \( N_r \) ramp loads may be present. Instances where \( P_j = Q_j \) can also occur, implying a uniformly distributed load. The general external loading chosen can be represented as

\[ w_e(x) = \sum_{j=1}^{N_f} F_j <x-f_j>^{-1} + \]

\[ \sum_{j=1}^{N_r} \left[ P_j <x-p_j>^0 -Q_j <x-q_j>^0 + \right. \]

\[ S_j \left( <x-p_j>^1 - <x-q_j>^1 \right) \]

where

\[ S_j = \frac{Q_j - P_j}{q_j - p_j} \]

© 2003 by CRC Press LLC
and each summation extends over the complete range of pertinent values. Similarly, integration using the properties of singularity functions yields

\[ v_e(x) = \sum_{j=1}^{N_f} F_j < x - f_j >^0 + \]

\[ \sum_{j=1}^{N_r} \left[ P_j < x - p_j >^1 - Q_j < x - q_j >^1 + \right. \]

\[ \frac{S_j}{2} \left( < x - p_j >^2 - < x - q_j >^2 \right) \]

and

\[ m_e(x) = \sum_{j=1}^{N_f} F_j < x - f_j >^1 + \]

\[ \sum_{j=1}^{N_r} \left[ \frac{P_j}{2} < x - p_j >^2 - \frac{Q_j}{2} < x - q_j >^2 + \right. \]

\[ \frac{S_j}{6} \left( < x - p_j >^3 - < x - q_j >^3 \right) \]

The single and double integrals given earlier involving \( m_e(x) \) and \( k(x) \) can easily be evaluated exactly when \( EI \) is constant, but these are not needed here. Since \( k(x) \) will generally be spatially variable in the target problem set, the integrations to compute \( y'(x) \) and \( y(x) \) are best performed numerically. Leaving the number of integration increments as an independent parameter allows high accuracy evaluation of all integrals whenever this is desirable. Typically, problems using several hundred integration points only require a few seconds to solve using a personal computer.

Completing the problem solution requires formulations and solution of a system of simultaneous equations involving \( v_0, m_0, y'_0, y_0, R_1, \ldots, R_{N_s} \). The desired equations are created by specifying the displacement constraints at the supports, as well as four of eight possible end conditions. To present the equations more concisely the following notation is adopted:

\[ \int_0^x k(x) \, dx = K_1(x), \quad \int_0^x \int_0^x k(x) \, dx \, dx = K_2(x), \]

\[ \int_0^x x k(x) \, dx = L_1(x), \quad \int_0^x \int_0^x x k(x) \, dx \, dx = L_2(x), \]

\[ \int_0^x m_e(x) k(x) \, dx = I_1(x), \quad \int_0^x \int_0^x m_e(x) k(x) \, dx \, dx = I_2(x), \]

\[ \int_0^x < x - r_j >^1 k(x) \, dx = J_1(x, r_j) \]

\[ \int_0^x \int_0^x < x - r_j >^1 k(x) \, dx \, dx = J_2(x, r_j), \]

\[ \int_0^x \int_0^x < x - r_j >^1 k(x) \, dx \, dx = J_2(x, r_j), \]

\[ \int_0^x \int_0^x < x - r_j >^1 k(x) \, dx \, dx = J_2(x, r_j), \]
and it is evident from their definitions that both $J_1(x, r_j)$ and $J_2(x, r_j)$ both equal zero for $x \leq r_j$.

At a typical support location $r_1$, the deflection will have an imposed value $y_1$. Consequently, the displacement constraints require

$$y_0 + r_1 y'_0 + K_2(r_1) m_0 + L_2(r_1) v_0 + \sum_{j=1}^{N_s} J_2(r_1, r_j) R_j = y_1 - I_2(r_1)$$

for $1 \leq i \leq N_s$. The remaining four end conditions can specify any legitimate combination of conditions yielding a unique solution. For example, a beam cantilevered at $x = 0$ and pin supported at $x = L$ would require $y(0) = 0$, $y'(0) = 0$, $m(L) = 0$, and $y(L) = 0$. In general, conditions imposed at $x = 0$ have an obvious form since only $v_0$, $m_0$, $y_0$, or $y'_0$ are explicitly involved. To illustrate a typical right end condition, let us choose slope, for example. This yields

$$y_0 + y'_0 + K_1(L) m_0 + L_1(L) v_0 + \sum_{j=1}^{N_s} J_1(L, r_j) R_j = y'(L) - I_1(L).$$

Equations for other end conditions have similar form, and all eight possibilities are implemented in the computer program listed at the end of the chapter. Once the reactions and any initially unknown left-end conditions have been determined, load and deformation quantities anywhere in the beam can be readily found.

### 11.1.2 Program to Analyze Beams of General Cross Section

A program to solve general beam problems was written which tabulates and plots the shear, moment, slope, and deflection. The driver program vdb defines the data, calls the analysis functions, and outputs the results. Six functions that implement the methods given in this section were written. Understanding the program details can best be achieved by studying the code closely. The program was checked extensively using examples from several texts and reference books. The three span beam having parabolically tapered haunches shown in Figure 11.2 was analyzed previously by Arbabi and Li [5]. The program vdb was used to analyze the same problem and produces the results in Figure 11.3, which agree well with the paper.

We believe that the computer program is general enough to handle a wide variety of practical problems. Some readers may want to extend the program by adding interactive input or input from a data file. Such a modification is straightforward.

### 11.1.3 Program Output and Code

**Output from Arbabi and Li Example**

Analysis of a Variable Depth Elastic Beam

---

Title: Problem from Arbabi and Li
Figure 11.2: Parabolic Beam from Arbabi and Li

Figure 11.3: Results for Arbabi and Li Example
Beam Length: 3
Number of integration segments: 301
Print frequency for results: 10

Interior Supports: (2)
<table>
<thead>
<tr>
<th>#</th>
<th>X-location</th>
<th>Deflection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0000e+000</td>
<td>0.0000e+000</td>
</tr>
<tr>
<td>2</td>
<td>2.0000e+000</td>
<td>0.0000e+000</td>
</tr>
</tbody>
</table>

Concentrated Forces: (1)
<table>
<thead>
<tr>
<th>#</th>
<th>X-location</th>
<th>Force</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.0000e-001</td>
<td>-1.0000e+000</td>
</tr>
</tbody>
</table>

Ramp loads: (1)
<table>
<thead>
<tr>
<th>#</th>
<th>X-start</th>
<th>Load</th>
<th>X-end</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0000e+000</td>
<td>-1.0000e+000</td>
<td>2.0000e+000</td>
<td>-1.0000e+000</td>
</tr>
</tbody>
</table>

End conditions:
<table>
<thead>
<tr>
<th>End Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>left slope</td>
<td>0.0000e+000</td>
</tr>
<tr>
<td>left deflection</td>
<td>0.0000e+000</td>
</tr>
<tr>
<td>right slope</td>
<td>0.0000e+000</td>
</tr>
<tr>
<td>right deflection</td>
<td>0.0000e+000</td>
</tr>
</tbody>
</table>

EI values are specified
<table>
<thead>
<tr>
<th>#</th>
<th>X-start</th>
<th>EI-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000e+000</td>
<td>7.9976e+000</td>
</tr>
<tr>
<td>2</td>
<td>1.0101e-002</td>
<td>7.5273e+000</td>
</tr>
<tr>
<td>3</td>
<td>2.0202e-002</td>
<td>7.0848e+000</td>
</tr>
<tr>
<td>4</td>
<td>3.0303e-002</td>
<td>6.6688e+000</td>
</tr>
<tr>
<td>5</td>
<td>4.0404e-002</td>
<td>6.2776e+000</td>
</tr>
</tbody>
</table>

Material deleted for publication
| 296 | 2.9596e+000 | 6.2776e+000 |
| 297 | 2.9697e+000 | 6.6688e+000 |
| 298 | 2.9798e+000 | 7.0848e+000 |
| 299 | 2.9899e+000 | 7.5273e+000 |
| 300 | 3.0000e+000 | 7.9976e+000 |

Solution time was 0.55 secs.

Reactions at Internal Supports:
<table>
<thead>
<tr>
<th>X-location</th>
<th>Reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0782e+000</td>
</tr>
<tr>
<td>2</td>
<td>4.7506e-001</td>
</tr>
</tbody>
</table>

Table of Results:
<table>
<thead>
<tr>
<th>X-location</th>
<th>Shear</th>
<th>Moment</th>
<th>Theta</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.2494e-001</td>
<td>-1.7415e-001</td>
<td>0.0000e+000</td>
<td>0.0000e+000</td>
</tr>
<tr>
<td>0.1</td>
<td>5.2494e-001</td>
<td>-1.2166e-001</td>
<td>-2.4859e-003</td>
<td>-1.1943e-004</td>
</tr>
<tr>
<td>0.2</td>
<td>5.2494e-001</td>
<td>-6.9164e-002</td>
<td>-5.3262e-003</td>
<td>-5.0966e-004</td>
</tr>
<tr>
<td>0.3</td>
<td>5.2494e-001</td>
<td>-1.6670e-002</td>
<td>-7.4251e-003</td>
<td>-1.1612e-003</td>
</tr>
<tr>
<td>0.4</td>
<td>5.2494e-001</td>
<td>3.5824e-002</td>
<td>-6.5761e-003</td>
<td>-1.8965e-003</td>
</tr>
<tr>
<td>0.5</td>
<td>-4.7506e-001</td>
<td>8.8318e-002</td>
<td>-5.5680e-004</td>
<td>-2.3003e-003</td>
</tr>
<tr>
<td>0.6</td>
<td>-4.7506e-001</td>
<td>4.0812e-002</td>
<td>5.6976e-003</td>
<td>-1.9998e-003</td>
</tr>
<tr>
<td>0.7</td>
<td>-4.7506e-001</td>
<td>6.6940e-003</td>
<td>7.1119e-003</td>
<td>-1.3258e-003</td>
</tr>
<tr>
<td>0.8</td>
<td>-4.7506e-001</td>
<td>5.4200e-002</td>
<td>5.6441e-003</td>
<td>-6.7385e-004</td>
</tr>
<tr>
<td>0.9</td>
<td>-4.7506e-001</td>
<td>1.0171e-001</td>
<td>3.3302e-003</td>
<td>-2.2402e-004</td>
</tr>
<tr>
<td>1.0</td>
<td>6.0309e-001</td>
<td>-1.4921e-001</td>
<td>1.2242e-003</td>
<td>-2.4286e-004</td>
</tr>
<tr>
<td>1.1</td>
<td>5.0309e-001</td>
<td>-9.3903e-002</td>
<td>-7.9439e-004</td>
<td>2.3707e-005</td>
</tr>
<tr>
<td>1.2</td>
<td>4.0309e-001</td>
<td>-4.8593e-002</td>
<td>-2.8814e-003</td>
<td>-1.6165e-004</td>
</tr>
<tr>
<td>1.3</td>
<td>3.0309e-001</td>
<td>-1.3284e-002</td>
<td>-4.3574e-003</td>
<td>-5.3250e-004</td>
</tr>
<tr>
<td>1.4</td>
<td>2.0309e-001</td>
<td>1.2025e-002</td>
<td>-4.8833e-003</td>
<td>-9.8078e-004</td>
</tr>
<tr>
<td>1.5</td>
<td>1.0309e-001</td>
<td>2.7334e-002</td>
<td>-2.3015e-003</td>
<td>-1.3242e-003</td>
</tr>
<tr>
<td>1.6</td>
<td>3.0918e-003</td>
<td>3.2643e-002</td>
<td>6.5694e-004</td>
<td>-1.4078e-003</td>
</tr>
<tr>
<td>1.7</td>
<td>-9.6908e-002</td>
<td>2.7953e-002</td>
<td>3.0625e-003</td>
<td>-1.2125e-003</td>
</tr>
<tr>
<td>1.8</td>
<td>-1.9691e-001</td>
<td>1.3262e-002</td>
<td>4.1954e-003</td>
<td>-8.3907e-004</td>
</tr>
<tr>
<td>1.9</td>
<td>-2.9691e-001</td>
<td>-1.1429e-002</td>
<td>4.2843e-003</td>
<td>-4.0860e-004</td>
</tr>
<tr>
<td>2.0</td>
<td>7.8151e-002</td>
<td>-4.6120e-002</td>
<td>3.8358e-003</td>
<td>-1.1102e-016</td>
</tr>
<tr>
<td>2.1</td>
<td>7.8151e-002</td>
<td>-3.8305e-002</td>
<td>3.1202e-003</td>
<td>3.5021e-004</td>
</tr>
<tr>
<td>2.2</td>
<td>7.8151e-002</td>
<td>-3.0490e-002</td>
<td>2.0801e-003</td>
<td>6.1308e-004</td>
</tr>
<tr>
<td>2.3</td>
<td>7.8151e-002</td>
<td>-2.2675e-002</td>
<td>7.2881e-004</td>
<td>7.5555e-004</td>
</tr>
<tr>
<td>2.4</td>
<td>7.8151e-002</td>
<td>-1.4860e-002</td>
<td>-6.9898e-004</td>
<td>7.5597e-004</td>
</tr>
<tr>
<td>2.5</td>
<td>7.8151e-002</td>
<td>-7.0445e-003</td>
<td>-1.7447e-003</td>
<td>6.2865e-004</td>
</tr>
<tr>
<td>2.6</td>
<td>7.8151e-002</td>
<td>7.7058e-004</td>
<td>-2.0539e-003</td>
<td>4.3228e-004</td>
</tr>
<tr>
<td>2.7</td>
<td>7.8151e-002</td>
<td>8.5857e-003</td>
<td>-1.7105e-003</td>
<td>2.4008e-004</td>
</tr>
<tr>
<td>2.8</td>
<td>7.8151e-002</td>
<td>1.6401e-002</td>
<td>-1.0840e-003</td>
<td>9.9549e-005</td>
</tr>
<tr>
<td>2.9</td>
<td>7.8151e-002</td>
<td>2.4216e-002</td>
<td>-4.7454e-004</td>
<td>2.2493e-005</td>
</tr>
<tr>
<td>3.0</td>
<td>7.8151e-002</td>
<td>3.2031e-002</td>
<td>-4.4409e-016</td>
<td>-2.2204e-16</td>
</tr>
</tbody>
</table>

**Variable Depth Beam Program**

```matlab
function vdb

% Example: vdb
% ~~~~~~~~~~~~
%
% This program calculates the shear, moment, slope, and deflection of a variable depth
% indeterminate beam subjected to complex loading and general end conditions. The
% input data are defined in the program statements below.
%
% User m functions required:
% bmvardep, extload, lintrp, oneovrei,
% sngf, trapsum
```

© 2003 by CRC Press LLC
clear all; Problem=1;
if Problem == 1
    Title=['Problem from Arbabi and Li'];
    Printout=10; % Output frequency
    BeamLength=3; % Beam length
    NoSegs=301; % # of beam divisions for integration
    ExtForce= [-1]; ExtForceX=[.5];
    ExtRamp=[-1 -1 1 2];
    % External concentrated loads and location
    % External ramp loads and range
    % Interior supports: initial displacement and location
    IntSupX= [1; 2]; IntSupDelta=[0; 0];
    % End (left and right) conditions
    EndCondVal= [0; 0; 0; 0]; % magnitude
    EndCondFunc=[3; 4; 3; 4];
    % 1=left end,2=right end
    EndCondEnd= [1; 1; 2; 2];
    % EI or beam depth specification
    EIorDepth=1; % 1=EI values specified
    % 2=depth values specified
    if EIorDepth == 1
        % Discretize the parabolic haunch for the three spans
        Width=1; E=1; a=0.5^2; Npts=100;
        h1=0.5; k1=1; x1=linspace(0,1,Npts);
        h2=1.5; k2=1; x2=linspace(1,2,Npts);
        h3=2.5; k3=1; x3=linspace(2,3,Npts);
        y1=(x1-h1).^2/a+k1; y2=(x2-h2).^2/a+k2;
        y3=(x3-h3).^2/a+k3;
        EIx=[x1 x2 x3]'; h=[y1 y2 y3]';
        EIvalue=E*Width/12*h.^3;
        mn=min(EIvalue); EIvalue=EIvalue./mn;
    else
        % Beam width and Young’s modulus
        BeamWidth=[]; BeamE=[]; Depth=[]; DepthX=[];
    end
else
    Title=['From Timoshenko and Young,’... , p 434, haunch beam'];
    Printout=12; NoSegs=144*4+1; BeamLength=144;
ExtForce=[]; ExtForceX=[];
ExtRamp=[-1 -1 0 108];
IntSupX=[36; 108]; IntSupDelta=[0; 0];
EndCondVal=[0; 0; 0; 0];
EndCondFunc=[2; 4; 2; 4];
EndCondEnd=[1; 1; 2; 2]; EIorDepth=2;
if EIorDepth == 1
  EIvalue=[]; EIx=[];
else
  BeamWidth=[1]; BeamE=[1];
  % Discretize the parabolic sections
  a=36^2/5; k=2.5; h1=0; h2=72; h3=144;
  N1=36; N2=72; N3=36;
  x1=linspace( 0, 36,N1); y1=(x1-h1).^2/a+k;
  x2=linspace( 36,108,N2); y2=(x2-h2).^2/a+k;
  x3=linspace(108,144,N3); y3=(x3-h3).^2/a+k;
  Depth=[y1 y2 y3]; DepthX=[x1 x2 x3];
  % Comparison values
  I=BeamWidth*Depth.^3/12; Imin=min(I); L1=36;
  k1=BeamE*min(I)/L1; k2=k1/2; k3=k1;
  t0=10.46/k1; t1=15.33/k1; t2=22.24/k1; 
  t3=27.95/k1;
  fprintf('

Values from reference');
  fprintf('
 Theta (x= 0): %12.4e',t0);
  fprintf('
 Theta (x= 36): %12.4e',t1);
  fprintf('
 Theta (x=108): %12.4e',t2);
  fprintf('
 Theta (x=144): %12.4e
',t3);
end
end

% Load input parameters into matrices
Force=[ExtForce,ExtForceX];
NoExtForce=length(ExtForce);
[NoExtRamp,ncol]=size(ExtRamp);
IntSup=[IntSupDelta,IntSupX];
NoIntSup=length(IntSupX);
EndCond=[EndCondVal,EndCondFunc,EndCondEnd];
if EIorDepth == 1
  BeamProp=[]; NoEIorDepths=length(EIx);
  EIdata=[EIvalue EIx];
else
  BeamProp=[BeamWidth BeamE];
  NoEIorDepths=length(DepthX);
  EIdata=[Depth DepthX];
end

© 2003 by CRC Press LLC
\begin{verbatim}
% more on

% Output input data
label1=['shear ';'moment ';' ... 
'slope ';'deflection'];
label2=['left ';'right '];
fprintf('

Analysis of a Variable Depth Elastic Beam');
fprintf('
--------------------------------');

');
fprintf('
Beam Length: %g', BeamLength);
fprintf('
Number of integration segments: %g', NoSegs);
fprintf('
Print frequency for results: %g', Printout);
fprintf('

');
fprintf('
Beam Length: %g', BeamLength);
fprintf('
Number of integration segments: %g', NoSegs);
fprintf('
Print frequency for results: %g', Printout);
fprintf('

');

if NoIntSup > 0
 fprintf('
 | # X-location Deflection');
 fprintf('
 | --- ------------ ------------');
 for i=1:NoIntSup
  fprintf('
 |%4.0f %12.4e %12.4e', i,IntSup(i,2),IntSup(i,1));
 end
end

if NoExtForce > 0
 fprintf('
 | # X-location Force');
 fprintf('
 | --- ------------ ------------');
 for i=1:NoExtForce
  fprintf('
 |%4.0f %12.4e %12.4e', i,Force(i,2),Force(i,1));
 end
end

fprintf('

');
end

fprintf('
');
end

fprintf('
');
end

fprintf('
');
end

\end{verbatim}
fprintf(' X-end Load');
fprintf('n | --- ------------ ------------');
fprintf(' ------------ ------------');
for i=1:NoExtRamp
fprintf('n |%4.0f %12.4e %12.4e ', ... 
i,ExtRamp(i,3),ExtRamp(i,1));
fprintf('%12.4e %12.4e', ... 
ExtRamp(i,4),ExtRamp(i,2));
end
End conditions:
fprintf('n | End Function Value');
fprintf('n | ------ ---------- ------------
');
for i=1:4
j=EndCond(i,3); k=EndCond(i,2);
strg=sprintf(' %12.4e',EndCond(i,1));
disp([' | ' label2(j,:) label1(k,:) strg]);
end
if EIorDepth == 1
fprintf('nEI values are specified');
fprintf('n | # X-start EI-value');
fprintf('n | --- ------------ ------------');
for i=1:NoEIorDepths
fprintf('n |%4.0f %12.4e %12.4e', ... 
i,EIdata(i,2),EIdata(i,1));
end
else
fprintf('nDepth values are specified for '); fprintf('rectangular cross section');
fprintf('n | Beam width: %12.4e', ... 
BeamProp(1));
fprintf('n | Young''s modulus: %12.4e', ... 
BeamProp(2));
fprintf('n |');
fprintf('n | # X-start Depth');
fprintf('n | --- ------------ ------------');
for i=1:NoEIorDepths
fprintf('n |%4.0f %12.4e %12.4e', ... 
i,EIdata(i,2),EIdata(i,1));
end
end
disp(' ');% Begin analysis
x=linspace(0,BeamLength,NoSegs)'; t=clock;
[V,M,Theta,Delta,Reactions]=...
bmvardep(NoSegs,BeamLength,Force,ExtRamp,...
   EndCond,IntSup,EIdata,BeamProp);
t=etime(clock,t);

% Output results
disp('');
disp(['Solution time was ',num2str(t),' secs.']);
if NoIntSup > 0
   fprintf('
Reactions at Internal Supports:');
   fprintf('
 | X-location Reaction');
   fprintf('
 | ------------ ------------');
   for i=1:NoIntSup
       fprintf('
 | %12.8g %12.4e',...
           IntSup(i,2),Reactions(i));
   end
end
fprintf('

Table of Results:');
fprintf('
 | X-location Shear');
fprintf(' Moment');
fprintf(' Theta Delta');
fprintf('
 | ----------- ------------ ');
fprintf('------------');
fprintf(' ------------ ------------');
if Printout > 0
   for i=1:Printout:NoSegs
       fprintf('
 |%12.4g %12.4e %12.4e',...
           x(i),V(i),M(i));
       fprintf(' %12.4e %12.4e',Theta(i),Delta(i));
   end
   disp(' ');
else
   i=1; j=NoSegs;
   fprintf('
 |%12.4g %12.4e %12.4e',...
          x(i),V(i),M(i));
   fprintf(' %12.4e %12.4e',Theta(i),Delta(i));
   fprintf('
 |%12.8g %12.4e %12.4e',...
          x(j),V(j),M(j));
   fprintf(' %12.4e %12.4e',Theta(j),Delta(j));
end
fprintf('

');
subplot(2,2,1);
plot(x,V,'k-'); grid; xlabel('x axis');
ylabel('Shear'); title('Shear Diagram');
subplot(2,2,2);
plot(x,M,'k-'); grid; xlabel('x axis');
ylabel('Moment'); title('Moment Diagram')

subplot(2,2,3);
plot(x,Theta,'k-'); grid; xlabel('x axis');
ylabel('Slope'); title('Slope Curve');

subplot(2,2,4);
plot(x,Delta,'k-'); grid; xlabel('y axis');
ylabel('Deflection'); title('Deflection Curve'); subplot

drawnow; figure(gcf)

%print -deps vdb

% more off

%===================================================================

function [V,M,Theta,Delta,Reactions]= ...
 bmvardep(NoSegs,BeamLength,Force,ExtRamp, ... 
 EndCond,IntSup,EIdata,BeamProp)

% [V,M,Theta,Delta,Reactions]=bmvardep ...
(NoSegs,BeamLength,Force,ExtRamp,EndCond, ... 
 IntSup,EIdata,BeamProp)

% This function computes the shear, moment, 
% slope, and deflection in a variable depth 
% elastic beam having specified end conditions, 
% intermediate supports with given 
% displacements, and general applied loading, 
% allowing concentrated loads and linearly 
% varying ramp loads.

% NoSegs - number of beam divisions for 
% integration
% BeamLength - beam length
% Force - matrix containing the magnitudes 
% and locations for concentrated 
% loads
% ExtRamp - matrix containing the end 
% magnitudes and end locations 
% for ramp loads
% EndCond - matrix containing the type of 
% end conditions, the magnitudes, 
% and whether values are for the
% left or right ends
% IntSup - matrix containing the location
% and delta for interior supports
% EIdata - either EI or depth values
% BeamProp - either null or beam widths
% V - vector of shear values
% M - vector of moment values
% Theta - vector of slope values
% Delta - vector of deflection values
% Reactions - reactions at interior supports
%
% User m functions required:
% oneovrei, extload, sngf, trapsum
%----------------------------------------------

if nargin < 8, BeamProp=[]; end
% Evaluate function value coordinates and 1/EI
x=linspace(0,BeamLength,NoSegs)';
k=oneovrei(x,EIdata,BeamProp);

% External load contributions to shear and
% moment interior to span and at right end
[ve,me]=extload(x,Force,ExtRamp);
[vv,mm]=extload(BeamLength,Force,ExtRamp);

% Deflections and position of interior supports
ns=size(IntSup,1);
if ns > 0
  ysprt=IntSup(:,1); r=IntSup(:,2);
  snf=sngf(x,r,1);
else
  ysprt=[]; r=[]; snf=zeros(NoSegs,0);
end

% Form matrix governing y''(x)
smat=kk(:,ones(1,ns+3)).* ...
  [x,ones(NoSegs,1),snf,me];

% Integrate twice to get slope and deflection
% matrices
smat=trapsum(0,BeamLength,smat);
ymat=trapsum(0,BeamLength,smat);

% External load contributions to
% slope/deflection at the right end
ss = smat(NoSegs, ns + 3); yy = ymat(NoSegs, ns + 3);

% Equations to solve for left end conditions
% and internal reactions
ns4 = ns + 4; j = 1:4; a = zeros(ns4, ns4);
b = zeros(ns4, 1); js = 1:ns; js4 = js + 4;

% Account for four independent boundary
% conditions. Usually two conditions will be
% imposed at each end.
for k = 1:4
    val = EndCond(k, 1); typ = EndCond(k, 2);
    wchend = EndCond(k, 3);
    if wchend == 1
        b(k) = val; row = zeros(1, 4); row(typ) = 1;
        a(k, j) = row;
    else
        if typ == 1 % Shear
            a(k, j) = [1, 0, 0, 0]; b(k) = val - vv;
            if ns > 0
                a(k, js4) = sngf(BeamLength, r, 0);
            end
        elseif typ == 2 % Moment
            a(k, j) = [BeamLength, 1, 0, 0]; b(k) = val - mm;
            if ns > 0
                a(k, js4) = sngf(BeamLength, r, 1);
            end
        elseif typ == 3 % Slope
            a(k, j) = [smat(NoSegs, 1:2), 1, 0];
            b(k) = val - ss;
            if ns > 0
                a(k, js4) = smat(NoSegs, 3:ns + 2);
            end
        else % Deflection
            a(k, j) = [ymat(NoSegs, 1:2), BeamLength, 1];
            b(k) = val - yy;
            if ns > 0
                a(k, js4) = ymat(NoSegs, 3:ns + 2);
            end
        end
    end
end

% Interpolate to assess how support deflections
% are affected by end conditions, external
% loads, and support reactions.
if ns>0
  a(js4,1)=interp1(x,ymat(:,1),r);
a(js4,2)=interp1(x,ymat(:,2),r);
a(js4,3)=r; a(js4,4)=ones(ns,1);
  for j=1:ns-1
    a(j+5:ns+4,j+4)= interp1(x,ymat(:,j+2),r(j+1:ns));
  end
end
b(js4)=ysprt-interp1(x,ymat(:,ns+3),r);
% Solve for unknown reactions and end conditions
c=a\b; v0=c(1); m0=c(2); s0=c(3); y0=c(4);
Reactions=c(5:ns+4);

% Compute the shear, moment, slope, deflection
% for all x
if ns > 0
  V=v0+ve+sngf(x,r,0)*Reactions;
  M=m0+v0*x+me+sngf(x,r,1)*Reactions;
  Theta=s0+smat(:,ns+3)+smat(:,1:ns+2)* ... 
      [v0;m0;Reactions];
  Delta=y0+s0*x+ymat(:,ns+3)+ ... 
      ymat(:,1:ns+2)*[v0;m0;Reactions];
else
  Reactions=[]; V=v0+ve; M=m0+v0*x+me;
  Theta=s0+smat(:,ns+3)+smat(:,1:2)*[v0;m0];
  Delta=y0+s0*x+ymat(:,ns+3)+ ... 
      ymat(:,1:2)*[v0;m0];
end

%=============================================
% This function computes the shear, moment,
% slope, and deflection in a uniform depth
% Euler beam which is loaded by a series of
% concentrated loads and ramp loads. The values

© 2003 by CRC Press LLC
% of shear, moment, slope and deflection all
% equal zero when x=0.

% x - location along beam
% Force - concentrated force matrix
% ExtRamp - distributed load matrix

% V - shear
% M - moment
% EITheta - slope
% EIDelta - deflection

% User m functions required: sngf

%---------------------------------------------

nf=size(Force,1); nr=size(ExtRamp,1);

nx=length(x); V=zeros(nx,1); M=V;

EITheta=V; EIDelta=V;

% Concentrated load contributions

if nf > 0
    F=Force(:,1); f=Force(:,2); V=V+sngf(x,f,0)*F;
    M=M+sngf(x,f,1)*F;
    if nargout > 2
        EITheta=EITheta+sngf(x,f,2)*(F/2);
        EIDelta=EIDelta+sngf(x,f,3)*(F/6);
    end
end

% Ramp load contributions

if nr > 0
    P=ExtRamp(:,1); Q=ExtRamp(:,2);
    p=ExtRamp(:,3); q=ExtRamp(:,4);
    S=(Q-P)./(q-p); sp2=sngf(x,p,2);
    sq2=sngf(x,q,2); sp3=sngf(x,p,3);
    sq3=sngf(x,q,3); sp4=sngf(x,p,4);
    sq4=sngf(x,q,4);
    V=V+sngf(x,p,1)*P-sngf(x,q,1)* ... % Shear
        Q+(sp2-sq2)*(S/2);
    M=M+sp2*(P/2)-sq2*(Q/2)+ ... % Moment
        (sp3-sq3)*(S/6);
    if nargout > 2
        EITheta=EITheta+sp3*(P/6)- ... % EI*Theta
            sq3*(Q/6)+(sp4-sq4)*(S/24);
        EIDelta=EIDelta+sp4*(P/24)- ... % EI*Delta
            sq4*(Q/24)+(sngf(x,p,5)- ... 
                sngf(x,q,5))*(S/120);
end
function val=oneovrei(x,EIdata,BeamProp)
% [val]=oneovrei(x,EIdata,BeamProp)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function computes 1/EI by piecewise
% linear interpolation through a set of data
% values.
%
% x - location along beam
% EIdata - EI or depth values
% BeamProp - null or width values
% val - computed value for 1/EI
% User m functions required: none
%----------------------------------------------

if size(EIdata,1) < 2 % uniform depth case
    v=EIdata(1,1);
    EIdata=[v,min(x);v,max(x)];
end
if ( nargin > 2 ) & ( sum(size(BeamProp)) > 0)
% Compute properties assuming the cross
% section is rectangular and EIdata(:,1)
% contains depth values
    width=BeamProp(1); E=BeamProp(2);
    EIdata(:,1)=E*width/12*EIdata(:,1).^3;
end
val=1./lintrp(EIdata(:,2),EIdata(:,1),x);

function y=sngf(x,x0,n)
% y=sngf(x,x0,n)
% ~~~~~~~~~~~~~~~~~
% This function computes the singularity
% function defined by
% y=(x-x0)^n for n=0,1,2,...
% User m functions required: none
%----------------------------------------------
if nargin < 3, n=0; end
x=x(:); nx=length(x); x0=x0(:)'; n0=length(x0);
x=x(:,ones(1,n0)); x0=x0(ones(nx,1),:); d=x-x0;
s=(d>=zeros(size(d))); v=d.*s;
if n==0
    y=s;
else
    y=v;
    for j=1:n-1; y=y.*v; end
end
%=============================================

function v=trapsum(a,b,y,n)
% v=trapsum(a,b,y,n)
% ~~~~~~~~~~~~~~~~~~
% This function evaluates:
% integral(a=>x, y(x)*dx) for a<=x<=b
% by the trapezoidal rule (which assumes linear
% function variation between succeasive function
% values).
% a,b - limits of integration
% y - integrand that can be a vector-valued
% function returning a matrix such that
% function values vary from row to row.
% It can also be input as a matrix with
% the row size being the number of
% function values and the column size
% being the number of components in the
% vector function.
% n - the number of function values used to
% perform the integration. When y is a
% matrix then n is computed as the number
% of rows in matrix y.
% v - integral value

% 2003 by CRC Press LLC
% User m functions called: none
%-------------------------------------------------------------

if isstr(y)
  % y is an externally defined function
  x=linspace(a,b,n)'; h=x(2)-x(1);
  Y=feval(y,x); % Function values must vary in
  % row order rather than column
  % order or computed results
  % will be wrong.
  m=size(Y,2);
else
  % y is column vector or a matrix
  Y=y; [n,m]=size(Y); h=(b-a)/(n-1);
end

v=[zeros(1,m); ...
  h/2*cumsum(Y(1:n-1,:)+Y(2:n,:))];

%=============================================

% function y=lintrp(xd,yd,x)
% See Appendix B
Chapter 12

Applications of Analytic Functions

12.1 Properties of Analytic Functions

Complex valued functions of a single complex variable are useful in various disciplines such as physics and numerical approximation theory. The current chapter summarizes a number of attractive properties of analytic functions and presents some applications in which MATLAB is helpful. Excellent textbooks presenting the theory of analytic functions [18, 75, 119] are available which fully develop various theoretical concepts employed in this chapter. Therefore, only the properties which may be helpful in subsequent discussions are included.

12.2 Definition of Analyticity

We consider a complex valued function

\[ F(z) = u(x, y) + iv(x, y), \quad z = x + iy \]

which depends on the complex variable \( z \). The function \( F(z) \) is analytic at point \( z \) if it is differentiable in the neighborhood of \( z \). Differentiability requires that the limit

\[
\lim_{|\Delta z| \to 0} \left[ \frac{F(z + \Delta z) - F(z)}{\Delta z} \right] = F'(z)
\]

exists independent of how \( |\Delta z| \) approaches zero. Necessary and sufficient conditions for analyticity are continuity of the first partial derivatives of \( u \) and \( v \) and satisfaction of the Cauchy-Riemann conditions (CRC)

\[
\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y}, \quad \frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}
\]

These conditions can be put in more general form as follows. Let \( n \) denote an arbitrary direction in the \( z \)-plane and let \( s \) be the direction obtained by a 90° counterclockwise rotation from the direction of \( n \). The generalized CRC are:

\[
\frac{\partial u}{\partial n} = \frac{\partial v}{\partial s}, \quad \frac{\partial u}{\partial s} = -\frac{\partial v}{\partial n}
\]
Satisfaction of the CRC implies that both \( u \) and \( v \) are solutions of Laplace’s equation

\[
\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0
\]

and

\[
\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} = 0
\]

These functions are called harmonic. Functions related by the CRC are also said to be harmonic conjugates. When one function \( u \) is known, its harmonic conjugate \( v \) can be found within an additive constant by using

\[
v = \int dv = \int \frac{\partial v}{\partial x} dx + \int \frac{\partial v}{\partial y} dy = \int \left(- \frac{\partial u}{\partial y} dx + \frac{\partial u}{\partial x} dy\right) + \text{constant}
\]

Harmonic conjugates also have the properties that curves \( u = \text{constant} \) and \( v = \text{constant} \) intersect orthogonally. This follows because \( u = \text{constant} \) implies \( \frac{\partial u}{\partial n} = 0 \) in a direction tangent to the curve. However \( \frac{\partial u}{\partial n} = \frac{\partial v}{\partial s} \) so \( v = \text{constant} \) along a curve intersecting \( u = \text{constant} \) orthogonally.

Sometimes it is helpful to regard a function of \( x \) and \( y \) as a function of \( z = x + iy \) and \( \bar{z} = x - iy \). The inverse is \( x = (z + \bar{z})/2 \) and \( y = (z - \bar{z})/(2i) \). Chain rule differentiation applied to a general function \( \phi \) yields

\[
\frac{\partial \phi}{\partial x} = \frac{\partial \phi}{\partial z} + \frac{\partial \phi}{\partial \bar{z}}, \quad \frac{\partial \phi}{\partial y} = i \frac{\partial \phi}{\partial z} - i \frac{\partial \phi}{\partial \bar{z}}
\]

so that

\[
\left( \frac{\partial}{\partial x} - i \frac{\partial}{\partial y} \right) \phi = 2 \frac{\partial \phi}{\partial z}, \quad \left( \frac{\partial}{\partial x} + i \frac{\partial}{\partial y} \right) \phi = 2 \frac{\partial \phi}{\partial \bar{z}}
\]

So Laplace’s equation becomes

\[
\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 4 \frac{\partial^2 \phi}{\partial z \partial \bar{z}} = 0
\]

It is straightforward to show the condition that a function \( F \) be an analytic function of \( z \) is expressible as

\[
\frac{\partial F}{\partial \bar{z}} = 0
\]

It is important to note that most of the functions routinely employed with real arguments are analytic in some part of the \( z \)-plane. These include:

\[ z^n, \sqrt{z}, \log(z), e^z, \sin(z), \cos(z), \arctan(z), \]

to mention a few. The real and imaginary parts of these functions are harmonic and they arise in various physical applications. The integral powers of \( z \) are especially significant. We can write

\[ z = re^{i\theta}, \quad r = \sqrt{x^2 + y^2}, \quad \theta = \tan^{-1} \left( \frac{y}{x} \right) \]
and get
\[ z^n = u + iv, \quad u = r^n \cos(n\theta), \quad v = r^n \sin(n\theta) \]
The reader can verify by direct differentiation that both \( u \) and \( v \) are harmonic.

Points where \( F(z) \) is nondifferentiable are called singular points and these are categorized as isolated or nonisolated. Isolated singularities are termed either poles or essential singularities. Branch points are the most common type of nonisolated singularity. Singular points and their significance are discussed further below.

12.3 Series Expansions

If \( F(z) \) is analytic inside and on the boundary of an annulus defined by \( a \leq |z - z_0| \leq b \) then \( F(z) \) is representable in a Laurent series of the form
\[
F(z) = \sum_{n=-\infty}^{\infty} a_n(z - z_0)^n, \quad a \leq |z - z_0| \leq b
\]
where
\[
F(z) = \frac{1}{2\pi i} \int_L \frac{F(t) \, dt}{(t - z_0)^{n+1}}
\]
and \( L \) represents any closed curve encircling \( z_0 \) and lying between the inner circle \( |z - z_0| = a \) and the outer circle \( |z - z_0| = b \). The direction of integration along the curve is counterclockwise. If \( F(z) \) is also analytic for \( |z - z_0| < a \), the negative powers in the Laurent series drop out to give Taylor’s series
\[
F(z) = \sum_{n=0}^{\infty} a_n(z - z_0)^n, \quad |z - z_0| \leq b
\]

Special cases of the Laurent series lead to classification of isolated singularities as poles or essential singularities. Suppose the inner radius can be made arbitrarily small but nonzero. If the coefficients below some order, say \(-m\), vanish but \( a_{-m} \neq 0 \), we classify \( z_0 \) as a pole of order \( m \). Otherwise, we say \( z_0 \) is an essential singularity.

Another term of importance in connection with Laurent series is \( a_{-1} \), the coefficient of \((z - z_0)^{-1}\). This coefficient, called the residue at \( z_0 \), is sometimes useful for evaluating integrals.

12.4 Integral Properties

Analytic functions have many useful integral properties. One of these properties that concerns integrals around closed curves is:
**Cauchy-Goursat Theorem:** If \( F(z) \) is analytic at all points in a simply connected region \( R \), then
\[
\int_L F(z) \, dz = 0
\]
for every closed curve \( L \) in the region.

An immediate consequence of this theorem is that the integral of \( F(z) \) along any path between two end points \( z_1 \) and \( z_2 \) is independent of the path (this only applies for simply connected regions).

### 12.4.1 Cauchy Integral Formula

If \( F(z) \) is analytic inside and on a closed curve \( L \) bounding a simply connected region \( R \) then
\[
F(z) = \frac{1}{2\pi i} \int_L \frac{F(t) \, dt}{t - z}
\]
for \( z \) inside \( L \)
\[
F(z) = 0
\]
for \( z \) outside \( L \)

The Cauchy integral formula provides a simple means for computing \( F(z) \) at interior points when its boundary values are known. We refer to any integral of the form
\[
I(z) = \frac{1}{2\pi i} \int_L \frac{F(t) \, dt}{t - z}
\]
as a Cauchy integral, regardless of whether \( F(t) \) is the boundary value of an analytic function. \( I(z) \) defines a function analytic in the complex plane cut along the curve \( L \). When \( F(t) \) is the boundary value of a function analytic inside a closed curve \( L \), \( I(z) \) is evidently discontinuous across \( L \) since \( I(z) \) approaches \( F(z) \) as \( z \) approaches \( L \) from the inside but gives zero for an approach from the outside. The theory of Cauchy integrals for both open and closed curves is extensively developed in Muskhelishvili’s texts [72, 73] and is used to solve many practical problems.

### 12.4.2 Residue Theorem

If \( F(z) \) is analytic inside and on a closed curve \( L \) except at isolated singularities \( z_1, z_2, \ldots, z_n \) where it has Laurent expansions, then
\[
\int_L F(z) \, dz = 2\pi i \sum_{j=1}^{j=n} B_j
\]
where \( B_j \) is the residue of \( F(z) \) at \( z = z_j \). In the instance where \( z_i \) is a pole of order \( m \), the residue can be computed as
\[
a_{-1} = \frac{1}{(m - 1)!} \left\{ \frac{d^{m-1}}{dz^{m-1}} [F(z)(z - z_i)^m] \right\}_{z \to z_i}
\]

© 2003 by CRC Press LLC
12.5 Physical Problems Leading to Analytic Functions

Several physical phenomena require solutions involving real valued functions satisfying Laplace’s equation. Since an analytic function has harmonic real and imaginary parts, a harmonic function can often be expressed concisely as the real part of an analytic function. Useful tools such as Taylor series can yield effective computational devices. One of the simplest practical examples involves determining a function \( u \) harmonic inside the unit disk \( |z| \leq 1 \) and having boundary values described by a Fourier series. In the following equations, and in subsequent articles, we will often refer to a function defined inside and on the unit circle in terms of polar coordinates as \( u(r, \theta) \) while we may, simultaneously, think of it as a function of the complex variable \( z = r\sigma \) where \( \sigma = e^{i\theta} \). Hence we write the boundary condition for the circular disk as

\[
 u(1, \theta) = \sum_{n=-\infty}^{\infty} c_n \sigma^n, \quad \sigma = e^{i\theta}
\]
with \( c_{-n} = \bar{c}_n \) because \( u \) is real. The desired function can be found as

\[
 u(r, \theta) = \text{real} \left( F(z) \right)
\]

where

\[
 F(z) = c_0 + 2 \sum_{n=1}^{\infty} c_n z^n, \quad |z| \leq 1
\]

This solution is useful because the Fast Fourier Transform (FFT) can be employed to generate Fourier coefficients for quite general boundary conditions, and the series for \( F(z) \) converges rapidly when \( |z| < 1 \). This series will be employed below to solve both the problem where boundary values are given (the Dirichlet problem) and where normal derivative values are known on the boundary (the Neumann problem). Several applications where analytic functions occur are mentioned below.

12.5.1 Steady-State Heat Conduction

The steady-state temperature distribution in a homogeneous two-dimensional body is harmonic. We can take \( u = \text{Real} [F(z)] \). Boundary curves where \( u = \text{constant} \) lead to conditions

\[
 F(z) + \overline{F(z)} = \text{constant}
\]

in the complex plane. Boundary curves insulated to prevent transverse heat flow lead to \( \frac{\partial u}{\partial n} = 0 \), which implies

\[
 F(z) - \overline{F(z)} = \text{constant}
\]
12.5.2 Incompressible Inviscid Fluid Flow

Some flow problems for incompressible, nonviscous fluids involve velocity components obtainable in terms of the first derivative of an analytic function. A complex velocity potential $F(z)$ exists such that

$$u - iv = F'(z)$$

At impermeable boundaries the flow normal to the boundary must vanish which implies

$$F(z) - \overline{F(z)} = \text{constant}.$$  

Furthermore, a uniform flow field with $u = U, v = V$ is easily described by

$$F(z) = (U - iV)z$$

12.5.3 Torsion and Flexure of Elastic Beams

The distribution of stresses in a cylindrical elastic beam subjected to torsion or bending can be computed using analytic functions [90]. For example, in the torsion problem shear stresses $\tau_{xz}$ and $\tau_{yz}$ can be sought as

$$\tau_{xz} - i\tau_{yz} = \mu \varepsilon |f'(z) - iz\bar{z}|$$

and the condition of zero traction on the lateral faces of the beam is described by

$$f(z) - \overline{f(z)} = iz\bar{z}$$

If the function $z = \omega(\zeta)$ which maps $|\zeta| \leq 1$ onto the beam cross section is known, then an explicit integral formula solution can be written as

$$f(\zeta) = \frac{1}{2\pi} \int_{|\sigma|=1} \frac{\omega(\sigma)\overline{\omega(\sigma)}d\sigma}{\sigma - \zeta}$$

Consequently, the torsion problem for a beam of simply connected cross section is represented concisely in terms of the function which maps a circular disk onto the cross section.

12.5.4 Plane Elastostatics

Analyzing the elastic equilibrium of two-dimensional bodies satisfying conditions of plane stress or plane strain can be reduced to determining two analytic functions. The formulas to find three stress components and two displacement components are more involved than the ones just stated. They will be investigated later when stress concentrations in a plate having a circular or elliptic hole are discussed.
12.5.5 Electric Field Intensity

Electromagnetic field theory is concerned with the field intensity $\epsilon$ which is described in terms of the electrostatic potential $E$ such that

$$E = E_x + i E_y = -\frac{\partial \phi}{\partial x} - i \frac{\partial \phi}{\partial y}$$

where $\phi$ is a harmonic function at all points not occupied by charge. Consequently a complex electrostatic potential $\Omega(z)$ exists such that

$$E = -\Omega'(z)$$

The electromagnetic problem is analogous to inviscid incompressible fluid flow problems. We will also find that harmonic functions remain harmonic under the geometry change of a conformal transformation, which will be discussed later. This produces interesting situations where solutions for new problems can sometimes be derived by simple geometry changes.

12.6 Branch Points and Multivalued Behavior

Before specific types of maps are examined, we need to consider the concept of branch points. A type of singular point quite different from isolated singularities such as poles arises when a singular point of $F(z)$ cannot be made the interior of a small circle on which $F(z)$ is single valued. Such singularities are called branch points and the related behavior is typified by functions such as $\sqrt{z-z_0}$ and $\log(z-z_0)$.

To define $p = \log(z - z_0)$, we accept any value $p$ such that $e^p$ produces the value $z - z_0$. Using polar form we can write

$$(z - z_0) = |z - z_0|e^{i(\theta + 2\pi k)}$$

where $\theta = \arg(z - z_0)$ with $k$ being any integer. Taking

$$p = \log |z - z_0| + i(\theta + 2\pi k)$$

yields an infinity of values all satisfying $e^p = z - z_0$. Furthermore, if $z$ traverses a counterclockwise circuit around a circle $|z - z_0| = \delta$, $\theta$ increases by $2\pi$ and $\log(z - z_0)$ does not return to its initial value. This shows that $\log(z - z_0)$ is discontinuous on a path containing $z_0$. A similar behavior is exhibited by $\sqrt{z-z_0}$, which changes sign for a circuit about $|z - z_0| = \delta$.

Functions with branch points have the characteristic behavior that the relevant functions are discontinuous on contours enclosing the branch points. Computing the function involves selection among a multiplicity of possible values. Hence $\sqrt{4}$ can equal $+2$ or $-2$, and choosing the proper value depends on the functions involved.
For sake of definiteness MATLAB uses what are called principal branch definitions such that
\[ \sqrt{z} = |z|^{1/2} e^{i\theta/2} , \quad -\pi < \theta = \tan^{-1} \left( \frac{y}{x} \right) \leq \pi \]
and
\[ \log(z) = \log |z| + i\theta \]

The functions defined this way have discontinuities across the negative real axis. Furthermore, \( \log(z) \) becomes infinite at \( z = 0 \).

Dealing carelessly with multivalued functions can produce strange results. Consider the function
\[ p = \sqrt{z^2 - 1} \]
which will have discontinuities on lines such that \( z^2 - 1 = -|h| \), where \( h \) is a general parameter. Discontinuity trouble occurs when
\[ z = \pm \sqrt{1 - |h|} \]
Taking \( 0 \leq |h| \leq 1 \) gives a discontinuity line on the real axis between \(-1\) and \(+1\), and taking \( |h| > 1 \) leads to a discontinuity on the imaginary axis. Figure 12.1 illustrates the odd behavior exhibited by \( \text{sqr}(z^2 - 1) \). The reader can easily verify that using
\[ \text{sqr}(z-1) \cdot \text{sqr}(z+1) \]
defines a different function that is continuous in the plane cut along a straight line between \(-1\) and \(+1\).

Multivalued functions arise quite naturally in solutions of boundary value problems, and the choices of branch cuts and branch values are usually evident from physical circumstances. For instance, consider a steady-state temperature problem for the region \( |z| < 1 \) with boundary conditions requiring
\[ u(1, \theta) = 1 , \quad 0 < \theta < \pi \quad \text{and} \quad \frac{\partial u(1, \theta)}{\partial r} = 0 , \quad \pi < \theta < 2\pi. \]

It can be shown that the desired solution is
\[ u = \text{real} \left\{ \frac{1}{\pi i} \left[ \log(z + 1) - \log(z - 1) \right] \right\} + \frac{3}{2} \]
where the logarithms must be defined so \( u \) is continuous inside the unit circle and \( u \) equals 1/2 at \( z = 0 \). Appropriate definitions result by taking
\[ -\pi < \arg(z + 1) \leq \pi , \quad 0 \leq \arg(z - 1) \leq 2\pi \]
MATLAB does not provide this definition intrinsically; so, the user must handle each problem individually when branch points arise.
12.7 Conformal Mapping and Harmonic Functions

A transformation of the form

\[ x = x(\xi, \eta) \quad , \quad y = y(\xi, \eta) \]

is said to be conformal if the angle between intersecting curves in the \((\xi, \eta)\) plane remains the same for corresponding mapped curves in the \((x, y)\) plane. Consider the transformation implied by \( z = \omega(\zeta) \) where \( \omega \) is an analytic function of \( \zeta \). Since

\[ dz = \omega'(\zeta) \, d\zeta \]

it follows that

\[ |dz| = |\omega'(\zeta)| \cdot |d\zeta| \quad \text{and} \quad \text{arg}(dz) = \text{arg}(\omega'(\zeta)) + \text{arg}(d\zeta) \]

This implies that the element of length \( |d\zeta| \) is stretched by a factor of \( |\omega'(\zeta)| \) and the line element \( d\zeta \) is rotated by an angle \( \arg[\omega'(\zeta)] \). The transformation is conformal at all points where \( \omega'(\zeta) \) exists and is nonzero.

Much of the interest in conformal mapping results from the fact that harmonic functions remain harmonic under a conformal transformation. To see why this is true, examine Laplace’s equation written in the form

\[ \nabla^2_{xy} u = 4 \frac{\partial^2 u}{\partial z \partial \bar{z}} = 0 \]

© 2003 by CRC Press LLC
For a conformal map we have

\[ z = \omega(\zeta), \quad \bar{z} = \overline{\omega(\zeta)} \]

\[ \frac{\partial u}{\partial z} = \frac{1}{\omega'(\zeta)} \frac{\partial u}{\partial \zeta}, \quad \frac{\partial u}{\partial \bar{z}} = \frac{\overline{\omega'(\zeta)}}{\overline{\omega'(\zeta)}} \frac{\partial u}{\partial \bar{\zeta}} \]

Since \( z \) depends only on \( \zeta \) and \( \bar{z} \) depends only on \( \bar{\zeta} \) we find that

\[ \nabla_{xy}^2 u = 4 \frac{1}{\omega'(\zeta)\omega'(\zeta)} \frac{\partial^2 u}{\partial \zeta^2} = \frac{1}{|\omega'(\zeta)|^2} \nabla_{\xi\eta}^2 u \]

It follows that

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{implies} \quad \frac{\partial^2 u}{\partial \xi^2} + \frac{\partial^2 u}{\partial \eta^2} = 0 \]

wherever \( \omega'(\zeta) \neq 0 \). The transformed differential equation in the new variables is identical to that of the original differential equation. Hence, when \( u(x, y) \) is a harmonic function of \( (x, y) \), then \( u(x(\xi, \eta), y(\xi, \eta)) \) is a harmonic function of \( (\xi, \eta) \), provided \( \omega(\zeta) \) is an analytic function. This is a remarkable and highly useful property. Normally, changing the independent variables in a differential equation changes the form of the equation greatly. For instance, with the polar coordinate transformation

\[ x = r \cos(\theta), \quad y = r \sin(\theta) \]

the Laplace equation becomes

\[ \nabla^2 u = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2} = 0 \]

The appearance of this equation is very different from the Cartesian form because \( x+i y \) is not an analytic function of \( r+i \theta \). On the other hand, using the transformation

\[ z = \log(\zeta) = \log(|\zeta|) + i \arg(\zeta) \]

gives

\[ \nabla_{xy}^2 u = (\zeta) \nabla_{\xi\eta}^2 u \]

and \( \nabla_{xy}^2 u = 0 \) implies \( \nabla_{\xi\eta}^2 u = 0 \) at points other than \( \zeta = 0 \) or \( \zeta = \infty \).

Because solutions to Laplace’s equation are important in physical applications, and such functions remain harmonic under a conformal map, an analogy between problems in two regions often can be useful. This is particularly attractive for problems where the harmonic function has constant values or zero normal gradient on critical boundaries. An instance pertaining to inviscid fluid flow about an elliptic cylinder will be used later to illustrate the harmonic function analogy. In the subsequent sections we discuss several transformations and their relevant geometrical interpretation.
12.8 Mapping onto the Exterior or the Interior of an Ellipse

We will examine in some detail the transformation

\[ z = \left( \frac{a + b}{2} \right) \zeta + \left( \frac{a - b}{2} \right) \zeta^{-1} = R(\zeta + m\zeta^{-1}) \quad , \quad \zeta \geq 1 \]

where \( R = (a + b)/2 \) and \( m = (a - b)/(a + b) \). The derivative

\[ z'(\zeta) = R(1 - m\zeta^{-2}) \]

becomes nonconformal when \( z'(\zeta) = 0 \) or \( \zeta = \pm \sqrt{m} \). For sake of discussion, we temporarily assume \( a \geq b \) to make \( \sqrt{m} \) real rather than purely imaginary. A circle \( \zeta = \rho_0 e^{i\theta} \) transforms into

\[ x + iy = R(\rho_0 + m\rho_0^{-1}) \cos(\theta) + iR(\rho_0 - m\rho_0^{-1}) \sin(\theta) \]

yielding an ellipse. When \( \rho_0 = 1 \) we get \( x = a \cos(\theta), y = b \sin(\theta) \). This mapping function is useful in problems such as inviscid flow around an elliptic cylinder or stress concentration around an elliptic hole in a plate. Furthermore, the mapping function is easy to invert by solving a quadratic equation to give

\[ \zeta = \frac{z + \sqrt{(z - \alpha)(z + \alpha)}}{a + b} \quad , \quad \alpha = \sqrt{a^2 - b^2} \]

The radical should be defined to have a branch cut on the \( x \)-axis from \(-\alpha\) to \( \alpha \) and to behave like \(+z\) for large \(|z|\). Computing the radical in MATLAB as

\[ \text{sqrt}(z-alpha).*\text{sqrt}(z+alpha) \]

works fine when \( \alpha \) is real because MATLAB uses

\[-\pi < \arg(z \pm \alpha) \leq \pi\]

and the sign change discontinuities experienced by both factors on the negative real axis cancel to make the product of radicals continuous. However, when \( a < b \) the branch points occur at \( \pm z_0 \) where \( z_0 = i\sqrt{b^2 - a^2} \), and a branch cut is needed along the imaginary axis. We can give a satisfactory definition by requiring

\[ -\frac{\pi}{2} < \arg(z \pm z_0) \leq \frac{3\pi}{2} \]

The function \texttt{elipinvr} provided below handles general \( a \) and \( b \).

Before leaving the problem of ellipse mapping we mention the fact that mapping the interior of a circle onto the interior of an ellipse is rather complicated but can be formulated by use of elliptic functions [75]. However, a simple solution to compute boundary point correspondence between points on the circle and points on the ellipse

© 2003 by CRC Press LLC
appears in [52]. This can be used to obtain mapping functions in rational form which
are quite accurate. The function elipdplt produces the mapping. Results showing
how a polar coordinate grid in the \(\zeta\)-plane maps onto a two to one ellipse appears
in Figure 12.2. In these examples and other similar ones, grid networks in polar
coordinates always use constant radial increments and constant angular increments.
Only the region corresponding to \(0.3 \leq |\zeta| \leq 1\) and \(0 \leq \arg(\zeta) \leq \frac{\pi}{2}\) is shown. Note
that the distortion of line elements at different points of the grid is surprisingly large.
This implies that the stretching effect, depending on \(|\omega'(\zeta)|\), varies more than might
at first be expected.

Often it is desirable to see how a rectangular or polar coordinate grid distorts
under a mapping transformation. This is accomplished by taking the point arrays and
simultaneously plotting rows against rows and columns against columns as computed
by the following function gridview which works for general input arrays \(x, y\). If the
input data are vectors instead of arrays, then the routine draws a single curve instead
of a surface. When gridview is executed with no input, it generates the plot in
Figure 12.3 which shows how a polar coordinate grid in the \(\zeta\)-plane maps under the
transformation

\[ z = R \left( \frac{\zeta + m}{\zeta} \right) \]

The new grid consists of a system of confocal ellipses orthogonally intersecting a
system of hyperbolas.
Figure 12.2: Mapping $|z| < 1$ onto an Elliptic Disk
Figure 12.3: Circular Annulus Mapped onto an Elliptic Annulus
12.8.1 Program Output and Code

Function sqrtsurf

function sqrtsurf

% sqrtsurf
% ~~~~~~~~
%
%
% This function illustrates the discontinuity
% in the function w=sqrt(z^2-1).
%
xx=linspace(-2,2,41); [x,y]=meshgrid(xx,xx);
z=x+i*y; w=sqrt(z.*z-1); close
surf(x,y,imag(w)); view(-40,50);
xlabel(’real axis’); ylabel(’imaginary axis’);
zlabel(’imag( sqrt( z^2-1 ) )’);
title([’Discontinuous Surface for imag( sqrt( z^2-1 ) )’]);
grid on; figure(gcf);
%print -deps sqrtsurf

Function elipinvr

function zeta=elipinvr(a,b,z)
%
% zeta=elipinvr(a,b,z)
% ~~~~~~~~~~~~~~~
%
%
% This function inverts the transformation
% z=(a+b)/2*zeta+(a-b)/2/zeta which maps
% abs(zeta)>=1 onto (x/a)^2+(y/b)^2 >= 1
%
% a  - semi-diameter on x-axis
% b  - semi-diameter on y-axis
% z  - array of complex values
%
% zeta - array of complex values for the
%        inverse mapping function
%
% User m functions called: none

© 2003 by CRC Press LLC
19 \( z_0 = \sqrt{a^2 - b^2}; \) \( ab = a + b; \)
20 if \( a = b \)
21 \( zeta = z/a; \)
22 elseif \( a > b \) \% branch cut along the real axis
23 \( zeta = (z + \sqrt{z - z0} \cdot \sqrt{z + z0})/ab; \)
24 else \% branch cut along the imaginary axis
25 \( ap = \text{angle}(z + z0); \) \( ap = ap + 2\pi \cdot (ap < -\pi/2); \)
26 \( am = \text{angle}(z - z0); \) \( am = am + 2\pi \cdot (am < -\pi/2); \)
27 \( zeta = (z + \sqrt{abs(z^2 - z0^2)} \cdot \exp(... \)
28 \( i/2 \cdot (ap + am))/ab; \)
29 end

Function elipdplt

1 function \([z,a,b] = \text{elipdplt}(rx,ry)\)
2 \% \([z,a,b] = \text{elipdplt}(rx,ry)\)
3 \% -----------------------------------
4 \% This function plots contour lines showing
5 \% how a polar coordinate grid in a circular
6 \% disk maps onto an elliptic disk.
7 \%
8 \% User m functions called: elipdisk, gridview
9 if nargin==0, rx=2; ry=1; end
10 \( zeta = \text{linspace}(.3,1,12) \cdot \ldots \)
11 \( \exp(i \cdot \text{linspace}(0,\pi/2,61)); \)
12 \([z,a,b] = \text{elipdisk}(zeta,rx,ry); \)
13 \( x = \text{real}(z); \) \( y = \text{imag}(z); \)
14 \text{gridview}(x,y,'x axis','y axis',...)
15 'Mapping abs(ZETA)<1 onto an Elliptic Disk');
16 \text{colormap([1 1 1]); shg}
17 \text{print -deps elipdisk}
18 \%
19 function \text{gridview}(x,y,xlabl,ylabl,titl)
20 \% \text{gridview}(x,y,xlabl,ylabl,titl)
21 \% -----------------------------------
22 \% This function views a surface from the top
to show the coordinate lines of the surface.

It is useful for illustrating how coordinate
lines distort under a conformal transformation.
Calling gridview with no arguments depicts the
mapping of a polar coordinate grid map under
a transformation of the form
\[ z = R \cdot (\zeta + m/zeta). \]

The data define a curve
\[
\begin{align*}
\text{xlabl} &= \text{real axis}; \\
\text{ylabl} &= \text{imaginary axis}; \\
\text{titl} &= \text{title for the graph}
\end{align*}
\]
User m functions called: cubrange

%----------------------------------------------
% close
if nargin<5
xlabl='real axis'; ylabl='imaginary axis';
titl='';
end

% Default example using \( z = R \cdot (\zeta + m/zeta) \)
if nargin==0
\[
\begin{align*}
\zeta &= \text{linspace}(1,3,10) \cdot \exp(i \cdot \text{linspace}(0,2 \cdot \text{pi},81)); \\
a &= 2; b &= 1; R &= (a+b)/2; m &= (a-b)/(a+b); \\
z &= R \cdot (\zeta + m./zeta); x &= \text{real}(z); y &= \text{imag}(z); \\
titl &= ['Circular Annulus Mapped onto an ', ... \\
      'Elliptical Annulus'];
\end{align*}
\]
end

range=cubrange([x(:),y(:)],1.1);

% The data define a curve
if size(x,1)==1 | size(x,2)==1
plot(x,y,'-k'); xlabel(xlabl); ylabel(ylabl); \\
title(titl); axis('equal'); axis(range); \\
grid on; figure(gcf);
if nargin==0
print -deps gridviewl
end
% The data define a surface
else
\[
\begin{align*}
\text{plot}(x,y, 'k-', x',y', 'k-')
\end{align*}
\]
xlabel(xlabl); ylabel(ylabl); title(titl);
axis('equal'); axis(range); grid on;
figure(gcf);
if nargin==0
    print -deps gridview
end

e==---------------------------------------------
function [z,a,b]=elipdisk(zeta,rx,ry)
%
% [z,a,b]=elipdisk(zeta,rx,ry)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%
% This function computes a rational function
% mapping abs(zeta)<=1 onto an elliptical disk
% defined by (x/rx)^2+(y/ry)^2<=1. Boundary
% points are computed using theory from
% P. Henrici, Applied Complex Analysis,
% Vol 3, p391.
% The rational function approximation has the
% form:
% z=\sum(a(j)*zeta^(2*j-1)) /
% (1+\sum(b(j)*zeta^(2*j)));
%
% zeta - matrix of points with abs(zeta)<=1
% rx,ry - ellipse semidiameters on x and y
% axes
%
% z - points into which zeta maps
% a,b - coefficients in the rational
% function defining the map
%
% User m functions called: ratcof
%----------------------------------------------
ntrms=100; ntheta=251;
tau=(0:2*pi/ntheta:2*pi)';
ep=(rx-ry)/(rx+ry);
z=exp(i*tau); z=z+ep*conj(z);
j=1:ntrms; ep=ep.^j; ep=ep./(j.*(1+ep.*ep));
theta=tau+2*( sin((2*tau+pi)*j).*ep');
zt=exp(i*theta); z=rx/max(real(z))*zt;
[a,b]=ratcof(zt.^2, z./zt, 8);
a=fix(real(1e8*a))/1e8; b=fix(real(1e8*b))/1e8;
af=flipeud(a(:)); bf=flipeud([1;b(:)]);
zta2=zeta.^2;
z=zeta.*polyval(af,zta2)./polyval(bf,zta2);

function [a,b]=ratcof(xdata,ydata,ntop,nbot)
% [a,b]=ratcof(xdata,ydata,ntop,nbot)
% %---------------------------------------------
%
% Determine a and b to approximate ydata as
% a rational function of the variable xdata.
% The function has the form:
% y(x) = sum(1=>ntop) ( a(j)*x^(j-1) ) /
% ( 1 + sum(1=>nbot) ( b(j)*x^(j)) )
% xdata,ydata - input data vectors (real or
% complex)
% nttop,nbot - number of series terms used in
% the numerator and the
% denominator.
%
% User m functions called: none
%---------------------------------------------

ydata=ydata(:); xdata=xdata(:);
if nargin==3, nbot=ntop; end;
x=ones(m,ntop+nbot); x(:,ntop+1)=-ydata.*xdata;
for i=2:ntop, x(:,i)=xdata.*x(:,i-1); end
for i=2:nbot
    x(:,i+ntop)=xdata.*x(:,i+ntop-1);
end
ab=x\ydata;
a=ab(1:ntop); b=ab(ntop+1:ntop+nbot);

%==============================================
% function range=cubrange(xyz,ovrsiz)
% See Appendix B

© 2003 by CRC Press LLC
12.9 Linear Fractional Transformations

The mapping function defined by

\[ w = \frac{az + b}{cz + d} \]

is called a linear fractional, or bilinear, transformation where \( a, b, c, \) and \( d \) are constants. It can be inverted to yield

\[ z = \frac{-dw + b}{cw - a} \]

If \( c \) is zero the transformation is linear. Otherwise, we can divide out \( c \) to get

\[ w = \frac{Az + B}{z + D} \]

The three remaining constants can be found by making three points in the \( z \)-plane map to three given points in the \( w \)-plane. Note that \( z = \infty \) maps to \( w = A \) and \( z = -D \) maps to \( w = \infty \).

The transformation has the attractive property that circles or straight lines map into circles or straight lines. An equation defining a circle or straight line in the \( z \)-plane has the form

\[ Pz\overline{z} + Qz + \overline{Q}\overline{z} + S = 0 \]

where \( P \) and \( S \) are real. A straight line is obtained when \( P \) is zero. Expressing \( z \) in terms of \( w \) and clearing fractions leads to an equation of the form

\[ P_0w\overline{w} + Q_0w + \overline{Q}_0\overline{w} + S_0 = 0 \]

which defines a circle in the \( w \)-plane when \( P_0 \) is nonzero. Otherwise, a straight line in the \( w \)-plane results.

Determining the bilinear transformation to take three \( z \)-points to three \( w \)-points is straightforward except for special cases. Let

\[ Z = [z_1; z_2; z_3] \quad \text{and} \quad W = [w_1; w_2; w_3] \]

If \( \det([Z, W, \text{ones}(3, 1)]) \) vanishes then a linear transformation with \( c = 0 \) and \( d = 1 \) applies. If \( z = \infty \) maps to \( w_1 \) we take \( a = w_1, \ c = 1 \). If \( z = z_1 \) maps to \( w = \infty \) we take \( c = 1, \ d = -z_1 \). In the usual situation we simply write \( w(z + D) = Az + B \) and solve the system

\[ [Z, \text{ones}(3, 1), -W] * [A; B; D] = W * Z \]
Function \texttt{linfrac}, used to compute the coefficients in the transformation, is provided at the end of this section. Points at infinity are handled by including \( \infty \) (represented in MATLAB by \texttt{inf}) as a legitimate value in the components of \( z \) or \( w \). For example, the transformation \( w = (2z + 3)/(z - 1) \) takes \( z = \infty \) to \( w = 2 \), \( z = 1 \) to \( w = \infty \), and \( z = 1 + i \) to \( w = 2 - 5i \). The expression

\[
\texttt{cz=linfrac([inf,1,1+i],[2,inf,2-5i]);}
\]

produces the coefficients in the transformation. Similarly, the transformation is inverted by

\[
\texttt{cw=linfrac([2,inf,2-5i],[inf,1,1+i]);}
\]

or equivalently by

\[
\texttt{cw=linfrac([0,1,2i],[-1.5,-4,-0.25-1.25i]);}
\]

Another type of problem of interest in connection with a known bilinear transformation is to find the circle or straight line into which a given circle or straight line maps. Function \texttt{crc2crc} performs this task. The coefficients \( c \) are given along with three points lying on a circle or a straight line. Then parameters \( w_0, r_0 \) pertaining to the \( w \)-plane are computed. If parameter \texttt{type} equals 1, then \( w_0 \) and \( r_0 \) specify the center and radius of a circle. Otherwise, \( w_0 \) and \( r_0 \) are two points defining a straight line.

The linear fractional transformation can be used to map an eccentric annulus such as that in Figure 12.4 onto a concentric annulus. Suppose a region \( 1 \leq |z| \leq R \) is to be mapped onto the region defined by

\[
|w| \geq R_1 \quad , \quad |w - w_0| \leq R_0
\]

The radius \( R \) and mapping coefficients \( c \) can be obtained by solving a system of non-linear simultaneous equations. Function \texttt{eccentric} accomplishes the task. A function call of

\[
[c,r]=eccentric(0.25,-0.25,1);
\]

produces

\[
w = \frac{3.4821z + 0.25}{z + 13.9282} \quad , \quad R = 3.7321
\]

and the plot in Figure 12.4 shows the mapped image of a polar coordinate grid using
constant radial and angular increment in the $z$ plane.

To demonstrate the utility of the transformation just discussed, consider the problem of determining the steady-state temperature field in an eccentric annulus with the inner and outer boundaries held at $u_1$ and $u_0$, respectively. The temperature field will be a harmonic function that remains harmonic under a conformal transformation. The related problem for the concentric annulus has the simple form

$$u = u_1 + \frac{(u_0 - u_1) \ln(r)}{\ln(R)}, \quad 1 \leq r \leq R$$

By analogy, expressing $r = |z|$ in terms of $w$ gives the temperature distribution at points in the $w$-plane.
12.9.1 Program Output and Code

Function linfrac

```matlab
function c=linfrac(z,w)
% This function determines the linear fractional transformation to map any three points in the z-plane into any three points in the w plane. Not more than one point in either the z or w plane may be located at infinity.
%
% z - vector of complex values [z1,z2,z3]
% w - vector of complex values [w1,w2,w3]
% c - vector defining the bilinear transformation
% w=(c(1)*z + c(2))/(c(3)*z + c(4))
%
% User m functions called: none
%----------------------------------------------

z=z(:); w=w(:); c=ones(4,1); k=find(z==inf); j=find(w==inf); kj=[k;j];
%
% z and w both contain points at infinity
if length(kj)==2
    c(1)=w(k); c(4)=-z(j); w(kj)=[]; z(kj)=[];
    c(2)=(w-c(1))*z+w*c(4);
    return
end
%
% z=inf maps to a finite w point
if ~isempty(k) & isempty(j)
    c(1)=w(k); z(k)=[]; w(k)=[];
    c([2 4])=[1;1],[-w\([w-c(1)].*z];
    return
end
%
% a finite z point maps to w = infinity
```

© 2003 by CRC Press LLC
if isempty(j) & isempty(k)
    c(4)=-z(j); z(j)=[]; w(j)=[];
    c([1 2])=[z,[1;1]]\[w.*(z+c(4))];
    return
end

% case where all points are finite
mat=[z,ones(3,1),-w];

% case for a general transformation
if det(mat)~=0
    c([1 2 4])=mat\[w.*z];
% case where transformation is linear
else
    c(3)=0; c([1 2])=[z,ones(3,1)]\w;
end

Function crc2crc

function [w0,r0,type]=crc2crc(c,z)

% This function determines the circle or
% straight line into which a circle or straight
% line maps under a linear fractional
% transformation.
%
% c - coefficients defining a linear
% fractional transformation
% w=(c(1)*z+c(2))/(c(3)*z*c(4))
% where c(2)*c(3)-c(1)*c(4) is nonzero
% z - a vector of three complex values
% lying on a circle or a straight line
%
% w0 - center of a circle in the w plane
% if type=1, or a point on a straight line if type=2
% r0 - radius of a circle in the w plane
% if type=1, or a point on a straight line if type=2
% type - equals 1 to denote a circle or 2 to
function [c,r]=ecentric(ri,wo,ro,nopl)
%
% [c,r]=ecentric(ri,wo,ro,nopl)
%
% This function determines the bilinear
% transformation which maps the region
% 1<=abs(z)<=r onto an eccentric annulus

© 2003 by CRC Press LLC
% defined by
% \abs(w) >= ri & \abs(w-wo) <= ro
%
% The coefficients c in the transformation
% \( w = \frac{c(1)z + c(2)}{c(3)z + c(4)} \)
% must be found as well as the outer radius r
% of the annulus in the z plane.
%
% ri - radius of inner circle \( \abs(w) = ri \)
% wo - center of outer circle \( \abs(w-wo) = ro \)
% ro - radius of outer circle
%
% c - coefficients in the mapping function
% r - radius of outer circle \( \abs(z) = r \)
% nopl - no plot is given if nopl is input
%
% User m functions called: gridview

if nargin==0, ri=.25; wo=-.25; ro=1; end

if wo~0
    c1=(wo+ro)/ri; c2=(wo-ro)/ri; c3=2/(c1+c2);
    c4=(c2-c1)/(c1+c2); c5=c3-c1-c1*c4; c6=1-c1*c3;
    rt=sqrt(c5^2-4*c4*c6);
    r1=(-c5+rt)/(2*c4); r2=(-c5-rt)/(2*c4);
    r=max([r1,r2]); d=c3+c4*r; c=[ri*d;ri;1;d];
else
    c=[ri;0;0;1]; r=ro/ri;
end
if nargin > 3, return, end

% Show the region onto which a polar coordinate
% grid in the z-plane maps.
% z=linspace(1,r,20)*exp(i*linspace(0,2*pi,81));
% w=(c(1)*z+c(2))/(c(3)*z+c(4));
% titl=['Concentric Annulus Mapped onto ','...'
%       'Eccentric Annulus'];
gridview(real(w),imag(w),...'
'real axis','imaginary axis',titl); shg
% print -deps ecentric
12.10 Schwarz-Christoffel Mapping onto a Square

The Schwarz-Christoffel transformation [75] provides integral formulas defining transformations to map the interior of a circle onto the interior or exterior of a polygon. Special cases obtained by allowing selected vertices to pass to infinity lead to a variety of results [58]. In general situations, evaluating the parameters and integrals in the Schwarz-Christoffel transformation is difficult and requires use of special software [35]. We will examine only two cases: a) where the interior of a circle is mapped onto the interior of a square, and b) where the exterior of a circle is mapped onto the exterior of a square. The function

\[ \zeta = C \int_0^\zeta (1 + t^4)^{-1/2} \, dt, \]

where \( C \) is a scaling constant, maps \(|\zeta| \leq 1\) inside the square defined by \((|x| \leq 1) \cap (|y| \leq 1)\).

Expanding this radical by the binomial expansion and integrating gives

\[ \zeta = c \sum_{n=0}^{\infty} (-1)^n \left[ \frac{\Gamma(n + \frac{1}{2})}{n!(4n + 1)} \right] \zeta^{1+4n}, \quad |\zeta| \leq 1 \]

A reasonably good approximation to the mapping function can be obtained by taking several hundred terms in the mapping function and adjusting the constant \( c \) to make \( \zeta = 1 \) match \( z = 1 \). This series expansion converges slowly and rounds the corners of the square because the derivative of the mapping function behaves like \((\zeta - \zeta_0)^{-1/2}\) at \( \zeta_0 = \pm e \pm i \pi/4 \).

The transformation to map \(|\zeta| \geq 1\) onto the square exterior defined by \((|x| \geq 1) \cup (|y| \geq 1)\)

has the form

\[ z = c_0 \int_1^\zeta (1 + t^{-4})^{1/2} \, dt + c_1, \]

where \( c_0 \) and \( c_1 \) are arbitrary constants. Using the binomial expansion again and term by term integration leads to

\[ z = c \sum_{n=0}^{\infty} (-1)^n \left[ \frac{\Gamma(n - \frac{1}{2})}{n!(4n - 1)} \right] \zeta^{1-4n}, \quad |\zeta| \geq 1 \]

The function swcsqmap provides both interior and exterior polynomial maps. Once again, truncating the series after a specified number of terms and making \( \zeta = 1 \) map to \( z = 1 \) gives an approximate mapping function which converges much more
rapidly than the series for the interior problem. Rounding of the square corners is
greatly reduced because the mapping function derivative behaves like $\left(\zeta - \zeta_o\right)^{1/2}$ at
$\zeta_o = \pm e^{\pm i\pi/4}$. Figure 12.5 illustrates results produced by the ten term series for
both interior and exterior regions. Using rational functions to produce better results
than polynomials was discussed earlier in Chapter 3. The function squarat, which
provides both interior and exterior maps, appears below.

It should be noted that inverting a mapping function $z = \omega(\zeta)$ to get $\zeta = g(z)$
explicitly is often difficult, if not impossible. For example, consider the form

$$z = \frac{\zeta(a + b\zeta^4 + c\zeta^8)}{1 + d\zeta^4 + e\zeta^8}, \quad |\zeta| \leq 1$$

which requires solving the polynomial

$$c\zeta^9 - e\zeta^8 + b\zeta^5 - d\zeta^4 + a\zeta - z = 0$$

and picking the root inside or on the unit circle. Although the MATLAB function
roots efficiently factors polynomials with complex coefficients, inverting the mapping
function for hundreds or thousands of values can be time consuming.
Figure 12.5: Square Maps Using a 10-term Series
12.10.1  Program Output and Code

Function swcsq10

1 function swcsq10
2  \% Example: swcsq10
3  \% -------------------
4  \%
5  \% This example demonstrates square map
6  \% approximations pertaining to truncated
7  \% Schwarz-Christoffel transformations.
8  \%
9  \% User m functions called:  swcsqmap, gridview
10:
11  zeta=linspace(0.2,1,8)’* ...
12    exp(i*linspace(0,pi/2,61));
13  [z,a]=swcsqmap(zeta,10);
14  subplot(211)
15   gridview(real(z),imag(z),’x axis’,’y axis’, ... 
16     [’Interior Map of a Square Using ’, ... 
17       ’ a 10-term Series’]);
18  subplot(212)
19  zeta=linspace(1,1.25,8)’* ...
20    exp(i*linspace(0,pi/2,61));
21  [z0,a]=swcsqmap(zeta,10,1);
22  gridview(real(z0),imag(z0),’x axis’,’y axis’, ... 
23     [’Exterior Map of a Square Using ’, ... 
24      ’ a 10-term Series’]);
25  print -deps sqrplt10
26  subplot
27:
28  \%=================================================================================
29:
30 function [z,a]=swcsqmap(zeta,ntrms,ifout)
31  \%
32  \% [z,a]=swcsqmap(zeta,ntrms,ifout)
33  \% ------------------------------
34  \%
35  \% This function evaluates power series
36  \% approximations for mapping either the inside
37  \% of a circle onto the inside of a square, or
38  \% mapping the outside of a circle onto the
39  \% outside of a square. The Schwarz-Christoffel
40  \% integrals defining the mapping functions are

© 2003 by CRC Press LLC
% expanded in Taylor series and are truncated
% to produce approximations in the following
% polynomial forms:

% For the interior problem:
%  z=sum(a(n)*zeta^((4*n-3),n=1:ntrms)
% 
% For the exterior problem:
%  z=sum(a(n)*zeta^((-4*n+5),n=1:ntrms)
%
% The side length of the square is adjusted
% to equal 2.

% zeta - complex values where the mapping
% function is evaluated
% ntrms - number of terms used in the
% truncated series
% ifout - a parameter omitted if an interior
% map applies. ifout can have any
% value (such as 1) to show that an
% exterior map is to be performed.

% z - values of the mapping function
% a - coefficients in the mapping series

% User m functions called: none

n=0:ntrms-2;
if nargin==2 % recursion formula for mapping
  p1=(n+1/2)./(n+1); p2=(n+1/4)./(n+5/4);
else % recursion formula for mapping
  p1=(n-1/2)./(n+1); p2=(n-1/4)./(n+3/4);
end
a=[1,cumprod(-p1.*p2)]; a=a(:)/sum(a);
z4=zeta.^4;
if nargin ==3, z4=1./z4; end;
z=zeta.*polyval(flipud(a(:)),z4);

% function gridview(x,y,xlabl,ylabl,titl)
% See Appendix B

Function squarat

function [z,a,b]=squirat(zeta,ifout)
%
% [z,a,b]=squirat(zeta,ifout)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function maps either the interior of a
% circle onto the interior of a square, or maps
% the exterior of a circle onto the exterior of
% a square using a rational function having the
% approximate form:
% z(zeta) = zeta *
% Sum(a(j)*zeta^4^j)/(1+Sum(b(j)*zeta^4^j),
% where zeta^4=zeta^4 for an interior problem,
% or zeta^4=zeta^(-4) for an exterior problem.
% zeta - matrix of complex values such that
% abs(zeta)<=1 for an interior map,
% or abs(zeta)>=1 for an exterior map
% ifout - parameter present in the call list
% only when an exterior mapping is
% required
% z - matrix of values of the mapping
% a,b - coefficients of the polynomials
% defining the rational mapping
% User m functions called: none
%----------------------------------------------

zeta4=zeta.^4;

if nargin==1 % map interior on interior
    a=[ 1.07835, 1.37751,-0.02642, -0.09129, ...
        0.13460,-0.15763, 0.07430, 0.14858, ...]
12.11 Determining Harmonic Functions in a Circular Disk

The problem of determining a function that is harmonic for $|z| < 1$ and satisfies certain boundary conditions can be analyzed effectively using series methods. In problems pertaining to the unit circle, it is often convenient to consider a function $u$, in polar coordinates, and write $u(r, \theta)$. Simultaneously, we may wish to think in terms of the related complex variable $z = r \sigma$ where $\sigma = e^{i\theta}$. Three basic problems will be considered.

I) Dirichlet Problem

$$\nabla^2 u = 0 , \quad |z| < 1$$

$$u(1, \theta) = f(\theta) , \quad 0 \leq \theta \leq 2\pi$$

We assume $f(\theta)$ is a real piecewise continuous function expandable in a Fourier series as

$$f(\theta) = \sum_{n=-\infty}^{\infty} f_n \sigma^n , \quad f_{-n} = \overline{f_n}$$

Then $u$ is given by the series

$$u = f_0 + 2 \text{real} \left( \sum_{n=1}^{\infty} f_n z^n \right) , \quad |z| \leq 1$$

II) Neumann Problem

$$\nabla^2 u = 0 , \quad |z| < 1$$

$$\frac{\partial u(1, \theta)}{\partial r} = g(\theta) , \quad 0 \leq \theta \leq 2\pi$$
We assume that the gradient function $g$ is expandable in a Fourier series as

$$g(\theta) = \sum_{n=-\infty}^{\infty} g_n \sigma^n, \quad g_{-n} = \overline{g_n}.$$ 

The solution only exists if the integral of $g(\theta)$ with respect to arc length around the boundary is zero. Hence, when

$$g_0 = \frac{1}{2\pi} \int_0^{2\pi} g(\theta) d\theta = 0,$$

then the series solution is

$$u = 2 \text{Re} \left( \sum_{n=1}^{\infty} \left( \frac{g_n}{n} \right) z^n \right) + c, \quad |z| \leq 1$$

where $c$ is an arbitrary real constant.

III) Mixed Problem

In the third type of problem the function value is specified on one part of the boundary and the normal gradient is specified on the remainder. In the general situation a solution can be constructed by methods using Cauchy integrals \[73\]. Only a simple case will be examined here. We require

$$\nabla^2 u = 0 \quad |z| < 1$$

$$u(1, \theta) = f(\theta), \quad \theta_1 < \theta < \theta_2$$

$$\frac{\partial u(1, \theta)}{\partial r} = g(\theta), \quad \theta_2 < \theta < (2\pi + \theta_1)$$

For convenience use the notation

$$L : \quad z = e^{i\theta}, \quad \theta_1 < \theta < \theta_2$$

$$L' : \quad z = e^{i\theta}, \quad \theta_2 < \theta < (2\pi + \theta_1)$$

The mixed problem can be reduced to a case where $g$ is zero by first solving a Neumann problem for a harmonic function $v$ such that

$$\frac{\partial v}{\partial r} = g(\theta), \quad z \in L'$$

$$\frac{\partial v}{\partial r} = -\frac{f^{2\pi+\theta_1}}{\theta_2 - \theta_1} g(\theta) d\theta, \quad z \in L$$

Then we replace $f(\theta)$ by $f(\theta) - v(1, \theta)$ to get a problem where

$$u = f(\theta) - v(1, \theta), \quad z \in L$$
\[ \frac{\partial u}{\partial r} = 0 \quad , \quad z \in L' \]

The complete solution then equals the sum of \( u \) and \( v \). Consequently, no loss of generality results in dealing with the problem

\[ u = f \quad , \quad z \in L \]
\[ \frac{\partial u}{\partial r} = 0 \quad , \quad z \in L' \]

Consider the function

\[ R(z) = \sqrt{(z-a)(z-b)} \quad , \quad a = e^{i\theta_1} \quad , \quad b = e^{i\theta_2} \]

defined in the complex plane cut along \( L \). We choose the branch of \( R \) satisfying

\[ R(0) = e^{i(\theta_1+\theta_2)/2} \]

The solution to the mixed boundary value problem can be expressed as

\[ u = \text{real}( \frac{R(z)}{\pi i} \int_{L} \frac{f(t) \, dt}{R^+(t)(t-z)} ) \quad , \quad t = e^{i\theta} \quad , \quad \theta_1 < \theta < \theta_2 \]

where \( R^+(t) \) means the boundary value of \( R(z) \) on the inside of the arc. As an example take

\[ \theta_1 = -\frac{\pi}{2} \quad , \quad \theta_2 = \frac{\pi}{2} \]
\[ R(z) = \sqrt{z^2 + 1} \quad , \quad R(0) = 1 \]
\[ u = \cos(\theta) \quad , \quad -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2} \]

Carrying out the integration gives

\[ u = \text{real}( F(z) ) \]

where

\[ F(z) = \frac{z + z^{-1} + (1 - z^{-1})\sqrt{z^2 + 1}}{2} \quad , \quad |z| \leq 1 \]

and the square root equals +1 at \( z = 0 \). This function is employed as a test case in subsequent calculations. The exact solution is evaluated in function `mbvtest`.

### 12.11.1 Numerical Results

The function `lapcrel` solves either Dirichlet or Neumann problems for the unit disk. The boundary values are specified as piecewise linear functions of the polar angle. Then function `lintrp` is used to obtain a dense set of boundary values which are transformed by the FFT to produce coefficients in the series solution. When
lapcercel is executed with no input data, a Dirichlet problem is solved having the boundary condition

\[ u(1, \theta) = 1 + \frac{\cos(16\theta)}{10}, \quad -\frac{\pi}{2} < \theta < \frac{\pi}{2} \]

\[ u(1, \theta) = \frac{\cos(16\theta)}{10}, \quad \frac{\pi}{2} < \theta < \frac{3\pi}{2} \]

This chosen boundary condition produces the interesting surface plot shown in Figure 12.6 where the solution was evaluated on a polar coordinate grid employing constant radial and angular increments.

The mixed boundary value problem is more difficult to handle than the Dirichlet or Neumann problems because numerical evaluation of the Cauchy integral must be performed cautiously. As \( z \) approaches a point on \( L \), the integrand becomes singular. Theoretical developments involving Cauchy principal value integrals and the Plemelj formulas are needed to handle this situation thoroughly [73]. Even when \( z \) is close to the boundary, large integrand magnitude may cause inaccurate numerical integration. Furthermore, the integrand will have square root type singularities at the ends of \( L \) unless \( f(a) = f(b) = 0 \). Regularization procedures that can cope fully with these difficulties [26] will not be investigated in this text. Instead a simplified approach is presented.

The function cauchint was written to evaluate a contour integral involving a general density function \( f(\zeta) \) defined on a curve \( L \) of general shape. We consider

\[ F(z) = \frac{1}{2\pi i} \int_L \frac{f(\zeta) d\zeta}{\zeta - z} \]

with both the density function \( f \) and the shape of \( L \) being defined using cubic spline interpolation. A set of points

\[ [\zeta_1, \zeta_2, \ldots, \zeta_m], \quad \zeta = \xi + i\eta \]

lying on \( L \), along with boundary values

\[ [f(\zeta_1), f(\zeta_2), \ldots, f(\zeta_m)] = [f_1, f_2, \ldots, f_m] \]

are given. Spline functions \( \zeta(t), f(t) \) are defined for \( 1 \leq t \leq m \) such that

\[ \zeta(j) = \zeta_j \quad \text{and} \quad f(j) = f_j \quad j = 1, 2, \ldots, n \]

The integrand in parametric form becomes

\[ F(z) = \frac{1}{2\pi i} \int_1^n \frac{f(t) [\xi'(t) + i\eta'(t)] dt}{\zeta(t) - z} \]

and this integral is evaluated using function gcquad which computes Gaussian base points and weight factors using eigenvalue methods. It should be remembered that
when $z$ is a point on the contour of integration, the integrand has a first order singularity. Hence, procedures to regularize the integrand would be needed to achieve accurate numerical integration in such cases.

Function cauchst was employed to produce an approximate solution of the problem cited above. A surface plot of the exact solution appears in Figure 12.7. A plot of the difference between the exact and approximate solutions for $0 \leq r \leq 0.99$ is shown in Figure 12.8. This error is about three orders-of-magnitude smaller than the maximum function values in the solution. The reader can verify that using $r = 0.999$ and $-\pi/2 < \theta < \pi/2$ leads to much larger errors. The authors have found function cauchint to be helpful if proper caution is exercised for results involving points near the boundary.

© 2003 by CRC Press LLC
Figure 12.7: Approximate Solution to a Mixed Boundary Value Problem

Figure 12.8: Difference Between Exact and Approximate Solutions
12.11.2 Program Output and Code

Function lapcrl

function [u,r,th]=lapcrl ...
    (bvtyp,bvdat,rvec,thvec,nsum)

% [u,r,th]=lapcrl(bvtyp,bvdat,rvec,thvec,nsum)

% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

% This function solves Laplace’s equation inside a circle of unit radius. Either a Dirichlet problem or a Neumann problem can be analyzed using boundary values defined by piecewise linear interpolation of data specified in terms of the polar angle.

% bvtyp - parameter determining what type of boundary value problem is solved. If bvtyp equals one, boundary data specify function values and a Dirichlet problem is solved. Otherwise, the boundary data specify values of normal gradient, and a Neumann problem is solved if, in accord with the existence conditions for this problem, the average value of gradient on the boundary is zero (negligibly small in an approximate solution).

% bvdat - a matrix of boundary data. Each bvdat(j,:) gives a function value and polar angle (in degrees) of a data point used by function lintrp to linearly interpolate for all other boundary values needed to generate the solution.

% rvec,thvec - vectors of radii and polar coordinate values used to form a polar coordinate grid of points inside the unit circle. No values of r exceeding unity are allowed.

% nsum - the number of terms summed in the
series expansion of the analytic function which has \( u \) as its real part. Typically, no more than one hundred terms are needed to produce a good solution.

\( u \) - values of the harmonic function evaluated at a set of points on a polar coordinate grid inside the unit circle.

\( r, \theta \) - the grid of polar coordinate values in which the function is evaluated

User m functions called: lintrp

Default test case solves a Dirichlet problem for a function having the following exact solution:

\[-1/2+\text{imag}(\log((z-i)/(z+i))/\pi)+\text{real}(z^{16})/10\]

if nargin ==0

\begin{verbatim}
72: thfft=linspace(0,2*pi*(nfft-1)/nfft,nfft);
73: if nargin<5, nsum=200; end;
74: nsum=min(nsum,nfft/2-1);
75: fbuf=bvdat(:,1); thbuf=pi/180*bvdat(:,2);
76: nev=size(bvdat,1); nr=length(rvec);
77: nth=length(thvec); neval=nr*nth;
78: [R,Th]=meshgrid(rvec,pi/180*thvec);
79: r=R(:); th=Th(:);
80: end;
\end{verbatim}

Check for any erroneous points outside the unit circle

\begin{verbatim}
81: if length(kout)>0
82: rvec=rvec(kout);
83: kout=find(rvec>1); nout=length(kout);
84: if length(kout)>0
85: \end{verbatim}
print('Input data are incorrect. The ');
print('following r values lie outside the ');
print('unit circle:'); disp(rvec(kout'));
return
end

if botyp==1 % Solve a Dirichlet problem
% Check for points on the boundary where
% function values are known. Interpolate
% these directly
konbd=find(r==1); onbndry=length(konbd);
if onbndry > 0
  u(konbd)=lintrp(thbv,fbv,th(konbd));
end

% Evaluate the series solution
kinsid=find(r<1); inside=length(kinsid);
if inside > 0
  a=fft(lintrp(thbv,fbv,thfft));
a=a(1:nsum)/(nft/2);
a(1)=a(1)/2; Z=r(kinsid).*exp(i*th(kinsid));
  u(kinsid)=real(polyval(flipud(a(:)),Z));
end

titl= ...
'Dirichlet Problem Inside the Unit Circle';
else % Solve a Neumann problem
gbv=lintrp(thbv,fbv,thfft);
a=fft(gbv)/(nft/2);
erchek=abs(a(1))/sum(abs(gbv));
if erchek>1e-3
  disp(' ');
disp('ERROR DUE TO NONZERO AVERAGE VALUE');
disp('OF NORMAL GRADIENT ON THE BOUNDARY. ');
disp('CORRECT THE INPUT DATA AND RERUN.');
disp(' '); u=[]; r=[]; th=[]; return;
end
a=a(2:nsum)./(1:nsum-1'); z=r.*exp(i*th);
u=real(polyval(flipud([0;a(:)]),Z));
titl='Neumann Problem Inside the Unit Circle';
end

u=reshape(u,nth,nr); r=R; th=Th;
surf(r.*cos(th), r.*sin(th), u);
xlabel('x axis'); ylabel('y axis');
zlabel('function u'); title(titl);
colormap('default');
grid on; figure(gcf);

% function y=lintrp(xd,yd,x)
% See Appendix B

Function cauchtst

function u=cauchtst(z,nquad)

% This function solves a mixed boundary value problem for the interior of a circle by numerically evaluating a Cauchy integral.

% z - matrix of complex coordinates where function values are computed
% nquad - order of Gauss quadrature used to perform numerical integration
% u - computed values of the approximate solution

% User m functions called: cauchint, mbvtest, gcquad, splined

if nargin<2, nquad=50; end; nbdat=61;
if nargin==0
    z=linspace(0,.99,10)'*exp(i*linspace(0,2*pi,91));
end

th=linspace(-pi/2,pi/2,nbdat); zb=exp(i*th);
fb=sqrt(zb-i).*sqrt(zb+i); fb(1)=1; fb(nbdat)=1;
28: fb=cos(th)./fb; fb(1)=0; fb(end)=0;
29: F=cauchint(fb,zb,z,nquad);
30: F=F.*sqrt(z-i).*sqrt(z+i); u=2*real(F);
31: surf(real(z),imag(z),u); xlabel('x axis');
32: ylabel('y axis'); zlabel('Solution Value')
33: title([ 'Approximate Solution to ', ...
34: 'a Mixed Boundary Value Problem']);
35: grid on; figure(gcf); %gra(.4);
36: fprintf([ '
Press [Enter] to solution error
']);
37: pause
38: %print -deps caucher1
39: uexact=mbvtest(z,1); udif=u-uexact;
40: clf; surf(real(z),imag(z),udif);
41: title([ 'Difference Between Exact and ', ...
42: 'Approximate Solutions']);
43: xlabel('x axis'); ylabel('y axis');
44: zlabel('Solution Error')
45: grid on; figure(gcf); %gra(.4)
46: %print -deps caucher2
47: %============================================= 
48: function u=mbvtest(z,noplot)
49: %
50: % u=mbvtest(z,noplot)
51: %-------------------
52: % This function determines a function which is
53: % harmonic for abs(z)<1 and satisfies at r=1,
54: % u=cos(theta), -pi/2<theta<pi/2
55: % du/dr=0, pi/2<theta<3*pi/2
56: % The solution only applies for points inside
57: % or on the unit circle.
58: %
59: % z - matrix of complex values where the
60: % solution is computed.
61: % noplot - option set to one if no plot is
62: % requested, otherwise option is not
63: % required.
64: %
65: % u - values of the harmonic function
66: % defined inside the unit circle
67: %
68: % User m functions called: none
69: %
70: %
71: %
72: %
73: %
© 2003 by CRC Press LLC
if nargin==0
    noplot=0;
    z=linspace(0,1,10)* ... 
    exp(i*linspace(0,2*pi,81));
end
[n,m]=size(z); z=z(:); u=1/2*ones(size(z));
k=find(abs(z)>0); Z=z(k);
U=(Z+1./Z+(1-1./Z).*sqrt(Z-i).*sqrt(Z+i))/2;
u(k)=real(U); u=reshape(u,n,m);
if nargin==1 | noplot==0
    z=reshape(z,n,m);
surf(real(z),imag(z),u); xlabel('x axis');
ylabel('y axis');
title(['Mixed Boundary Value Problem ', ... 
    'for a Circular Disk']);
    grid; figure(gcf); %gra(.4), pause
end

function F=cauchint(fb,zb,z,nquad)

% This function numerically evaluates a Cauchy integral of the form:
% F(z)=1/(2*pi*i)*Integral(f(t)/(t-z)*dt)
% where t denotes points on a curve in the complex plane. The boundary curve is defined by spline interpolation through data points zb lying on the curve. The values of f(t) are also specified by spline interpolation through values fb corresponding to the points zb. Numerical evaluation of the integral is performed using a composite Gauss formula of arbitrary order.
% fb - values of density function f
% at point on the curve
% zb - points where fb is given. The 
% number of values of zb must be 
% adequate to define the curve 
% accurately.
% Z - a matrix of values at which the 
% Cauchy integral is to be evaluated.
% If any of the z-values lie on path 
% of integration or too close to the 
% path of integration, incorrect 
% results will be obtained.
% nquad - the order of Gauss quadrature 
% formula used to perform numerical 
% integration
% F - The value of the Cauchy integral 
% corresponding to matrix argument z 
% User m functions called: gcquad splined 

%----------------------------------------------

n=length(fb); [nr,nc]=size(z); z=z(:).';
t=1:n;
[dummy,bp,wf]=gcquad('',1,n,nquad,n-1);
fq=spline(t,fb,bp); zq=spline(t,zb,bp);
zqd=splined(t,zb,bp); nq=length(fq);
fq=fq(:).*zqd(:);
bdrylen=sum(abs(zq(2:nq)-zq(1:nq-1)));
closnes=1e100; bigz=max(abs(z));
for j=1:nq
  closnes=min([closnes,abs(zq(j)-z)]);
end
if closnes/bdrylen<.01 | closnes/bigz<.01
  disp(' WARNING! SOME DATA VALUES ARE ',...
  'EITHER NEAR OR ON');
  disp(['THE BOUNDARY. COMPUTED RESULTS ',...
  'MAY BE INACCURATE']); disp(' ')
end
F=wf(:)'*fq(:,ones(1,nz))./(zq(:,ones(1,nz))...
  -z(ones(nq,1),:)));
F=reshape(F,nr,nc)/(2*pi*i);

%================================================

© 2003 by CRC Press LLC
12.12 Inviscid Fluid Flow around an Elliptic Cylinder

This section analyzes inviscid flow around an elliptic cylinder in an infinite field. Flow around a circular cylinder is treated first. Then the function conformally mapping the exterior of a circle onto the exterior of an ellipse is used in conjunction with the invariance of harmonic functions under a conformal transformation. Results describing the elliptic cylinder flow field for uniform velocity components at infinity are presented.

Let us solve for the flow around a circular cylinder in the region $|\zeta| \geq 1$, $\zeta = \xi + i\eta$ with the requirement that the velocity components at infinity have constant values

$$u = U, \quad v = V$$

where $(u, v)$ are the horizontal and vertical components of velocity. These components are derivable from a potential function $\phi$ such that

$$u = \frac{\partial \phi}{\partial \xi}, \quad v = \frac{\partial \phi}{\partial \eta}$$

where $\phi$ is a harmonic function. The velocity normal to the cylinder boundary must be zero. This requires that the function $\psi$, the harmonic conjugate of $\phi$, must be constant on the boundary. The constant can be taken as zero without loss of generality. In terms of the complex velocity potential

$$f(\zeta) = \phi + i\psi$$

we need

$$f(\zeta) - f^*(\zeta) = 0 \quad \text{on} \quad |\zeta| = 1$$

The velocity field is related to the complex velocity potential by

$$u - iv = f'(\zeta)$$

so the flow condition at infinity is satisfied by

$$f(\zeta) = p\zeta + O(1) \quad \text{where} \quad p = U - iV$$
A Laurent series can be used to represent \( f(\zeta) \) in the form

\[
f(\zeta) = p\zeta + a_0 + \sum_{n=1}^{\infty} a_n\zeta^{-n}
\]

Imposition of the boundary condition on the cylinder surface requiring

\[
f(\sigma) - \bar{f}(\sigma) = 0 \quad \text{where} \quad \sigma = e^{i\theta}
\]

leads to

\[
p\sigma + a_0 + \sum_{n=1}^{\infty} a_n\sigma^{-n} - \bar{p}\sigma^{-1} - \sum_{n=1}^{\infty} a_n\bar{\sigma}^n = 0
\]

Taking \( a_0 = 0, a_1 = \bar{p}, \) and \( a_n = 0, n \geq 2 \) satisfies all conditions of the problem and yields

\[
f(\zeta) = p\zeta + \bar{p}\zeta^{-1}
\]

as the desired complex potential function giving the velocity field as

\[
u - iv = f'(\zeta) = p - \bar{p}\zeta^{-2} \quad |\zeta| \geq 1
\]

Now consider flow about an elliptic cylinder lying in the \( z \)-plane. If the velocity at infinity has components \((U, V)\) then we need a velocity potential \( F(z) \) such that

\[
F'(\infty) = U - iV
\]

This is nearly the same problem as was already solved in the \( \zeta \)-plane except that

\[
dF = \omega'(\zeta)\frac{dF}{d\zeta} = \frac{1}{\omega'(\zeta)} \frac{dF}{d\zeta}
\]

where \( \omega(\zeta) \) is the mapping function

\[
z = \omega(\zeta) = R(\zeta + m\zeta^{-1}) \quad , \quad R = \frac{a + b}{2} \quad , \quad m = \frac{a - b}{a + b}
\]

In terms of \( \zeta \) we would need

\[
F = p\zeta + \bar{p}\zeta^{-1} \quad , \quad p = R(U - iV)
\]

Consequently, the velocity potential for the elliptic cylinder problem expressed in terms of \( \zeta \) is

\[
F = p\zeta + \bar{p}\zeta^{-1} \quad , \quad p = R(U - iV)
\]

and the velocity components in the \( z \)-plane are given by

\[
u - iv = \frac{1}{\omega'(\zeta)} [p - \bar{p}\zeta^{-2}] = \frac{(U - iV) - (U - iV)\zeta^{-2}}{1 - m\zeta^{-2}}.
\]
To get values for a particular choice of $z$ we can use the inverse mapping function

$$
\zeta = \frac{z + \sqrt{z^2 - 4mR^2}}{2R}
$$

to eliminate $\zeta$ or we can compute results in terms of $\zeta$.

To complete our discussion of this flow problem we will graph the lines characterizing the directions of flow. The velocity potential $F = \phi + i\psi$ satisfies

$$
\begin{align*}
    u &= \frac{\partial \phi}{\partial x} = \frac{\partial \psi}{\partial y}, \quad v = \frac{\partial \phi}{\partial y} = -\frac{\partial \psi}{\partial x}
\end{align*}
$$

so a curve tangent to the velocity field obeys

$$
\frac{dy}{dx} = \frac{v}{u} = -\frac{\partial \psi/\partial x}{\partial \psi/\partial y}
$$

or

$$
\frac{\partial \psi}{\partial x} dx + \frac{\partial \psi}{\partial y} dy = 0, \quad \psi = \text{constant}
$$

Consequently, the flow lines are the contours of function $\psi$, which is called the stream function. The function we want to contour does not exist inside the ellipse, but we can circumvent this problem by computing $\psi$ in the ellipse exterior and then setting $\psi$ to zero inside the ellipse. The function \texttt{elipcy} analyzes the cylinder flow and produces the accompanying contour plot shown in Figure 12.9.
Figure 12.9: Elliptic Cylinder Flow Field for Angle = 30°

12.12.1 Program Output and Code

Function elipcy

function [x,y,F]=elipcy(a,n,rx,ry,ang)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function computes the flow field around an elliptic cylinder. The velocity direction at infinity is arbitrary.
% a - defines the region -a<x<a, -a<y<a within which the flow field is computed
% n - this determines the grid size which uses n by n points
% rx,ry - major and minor semi-diameters of the ellipse lying on the x and y axes,
% respectively
17  % ang - the angle in degrees which the
18  % velocity at infinity makes with the
19  % x axis
20  %
21  %
22  % x,y - matrices of points where the velocity
23  % potential is computed
24  % F - matrix of complex velocity potential
25  % inside the ellipse, where the
26  % potential is actually not defined
27  %
28  % User m functions called: none
29  
30  % default data for a 2 by 1 ellipse
31  if nargin==0
32      a=5; n=81; rx=2; ry=1; ang=30;
33  end
34  
35  % Compute a square grid in the z plane.
36  ar=pi/180*ang; p=(rx+ry)/2*exp(-i*ar);
37  cp=conj(p); d=linspace(-a,a,n);
38  [x,y]=meshgrid(d,d); m=sqrt(rx^2-ry^2);
39  
40  % Obtain points in the zeta plane outside
41  % the ellipse
42  z=x(:)+i*y(:); k=find((x/rx).^2+(y/ry).^2>=1);
43  Z=z(k); zeta=(Z+sqrt(Z-m).*sqrt(Z+m))/(rx+ry);
44  F=zeros(n*n,1);
45  
46  % Evaluate the potential for a circular
47  % cylinder
48  F(k)=p*zeta+cp./zeta; F=reshape(F,n,n);
49  
50  % Contour the stream function to show the
51  % direction of flow
52  clf; contourf(x(1,:),y(:,1),abs(imag(F)),30);
53  axis('square'); zb=exp(i*linspace(0,2*pi,101));
54  xb=rx*real(zb); yb=ry*imag(zb);
55  xb(end)=xb(1); yb(end)=yb(1);
56  hold on; fill(xb,yb,[127/255 1 212/255]);
57  xlabel('x axis'); ylabel('y axis');
58  title(['Elliptic Cylinder Flow Field for ', ...
59     'Angle = ',num2str(ang),', Degrees']);
Torsional Stresses in a Beam Mapped onto a Unit Disk

Torsional stresses in a cylindrical beam can be computed from an integral formula when the function \( z = \omega(\zeta) \) mapping the unit disk, \( |\zeta| \leq 1 \), onto the beam cross section is known [90]. The complex stress function

\[
f(\zeta) = \frac{1}{2\pi} \int_{\gamma} \frac{\omega(\sigma)\omega(\sigma)}{\sigma - \zeta} \, d\sigma + \text{constant},
\]

where \( \gamma \) denotes the unit circle, can be evaluated exactly by contour integration in some cases. However, an approach employing series methods is easy to implement and gives satisfactory results if enough series terms are taken. When \( \omega(\zeta) \) is a polynomial, \( f(\zeta) \) is a polynomial of the same order as \( \omega(\zeta) \). Furthermore, when \( \omega(\zeta) \) is a rational function, residue calculus can be employed to compute \( f(\zeta) \) exactly, provided the poles of \( \varpi(1/\zeta) \) can be found. A much simpler approach is to use the FFT to expand \( \omega(\sigma)\omega(\sigma) \) in a complex Fourier series and write

\[
\omega(\sigma)\omega(\sigma) = \sum_{n=-\infty}^{\infty} c_n \sigma^n, \quad \sigma = e^{i\theta}
\]

Then the complex stress function is

\[
f(\zeta) = i \sum_{n=1}^{\infty} c_n \zeta^n + \text{constant}
\]

where the constant has no influence on the stress state. The shear stresses relative to the curvilinear coordinate system are obtainable from the formula

\[
\frac{\tau_{\rho Z} - i\tau_{\alpha Z}}{\mu} = \frac{[f'(\zeta) - i\omega(\zeta)\omega'(\zeta)]}{|\zeta\omega'(\zeta)|} \zeta
\]

where \( \mu \) is the shear modulus and \( \varepsilon \) is the angle of twist per unit length. The capital \( Z \) subscript on shear stresses refers to the direction of the beam axis normal to the \( xy \) plane rather than the complex variable \( z = x + iy \). The series expansion gives

\[
f'(\zeta) = i \sum_{n=1}^{\infty} n c_n \zeta^{n-1}
\]

and this can be used to compute stresses. Differentiated series expansions often converge slowly or may even be divergent. To test the series expansion solution, a

© 2003 by CRC Press LLC
Torsional Shear Stresses on a Square Cross Section

Figure 12.10: Torsional Shear Stresses on a Square Cross Section

rational function mapping $|\zeta| < 1$ onto a square defined by $|x| \leq 1$ and $|y| \leq 1$ was employed. Function mapsqr which computes $z(\zeta)$ and $z'(\zeta)$ is used by function torstres to evaluate stresses in terms of $\zeta$. A short driver program runtors evaluates stresses on the boundary for $x = 1, 0 \leq y \leq 1$. Stresses divided by the side length of 2 are plotted and results produced from a highly accurate solution [90] are compared with values produced using 800 terms in $f(\zeta)$. Results depicted in Figure 12.10 show that the error in maximum shear stress was only 0.44% and the torsional stiffness was accurate within 0.05%. The numerical solution gives a nonzero stress value for $y = 1$, which disagree with the exact solution. This error is probably due more to the mapping function giving slightly rounded corners than to slow convergence of the series solution. Even though the differentiated series converges slowly, computation time is still small. The reader can verify that using 1500 terms reduces the boundary stress oscillations to negligible magnitude and produces a maximum stress error of 0.03%. Although taking 1500 terms to achieve accurate results seems excessive, less than 400 nonzero terms are actually involved because geometrical symmetry implies a series increasing in powers of four. For simplicity and generality, no attempt was made to account for geometrical symmetry exhibited by a particular mapping function. It appears that a series solution employing a mapping function is a viable computational tool to deal with torsion problems.
12.13.1 Program Output and Code

Program runtors

function runtors(ntrms)
% Example: runtors(ntrms)
% ~~~~~~~~~~~~~~~
% Example showing torsional stress computation
% for a beam of square cross section using
% conformal mapping and a complex stress
% function.
% ntrms - number of series terms used to
% represent abs(w(zeta))^2
% User m functions called: torstres, mapsqr

% Generate zeta values defining half of a side
theta=linspace(0,pi/4,501); zeta=exp(i*theta);
if nargin==0, ntrms=800; end

% Compute stresses using an approximate rational
% function mapping function for the square
[tr,ta,z,c,C]=torstres('mapsqr',zeta,ntrms,4*1024);

% Results from the exact solution
n=1:2:13;
tmexact=1-8/pi^2*sum(1./(n.^2.*cosh(n*pi/2)));
err=abs(ta(1)/2-tmexact)*100/tmexact;
stfexct=16/3-1024/pi^5*sum(tanh(pi/2*n)./n.^5);
stfaprx=8/3-pi*sum((1:ntrms)'.*
    abs(C(2:ntrms+1)).^2);
ster=100*abs(stfaprx-stfexct)/stfexct;

% Plot circumferential and normal stresses at
% the boundary
th=180/pi*theta;
clf; plot(imag(z),tr/2,'k:',imag(z),ta/2,'k-')
xlabel('y distance along the side');
ylabel('shear stresses at the boundary');
title(['Torsional Shear Stresses on a ',...
    'Square Cross Section']);
[
'text(.05,.40,...
[Max Shear Stress = ',num2str(max(ta)/2))];
'text(.05,.34,...
[Number of Series Terms = ',num2str(ntrms)];
'text(.05,.28,...
[Maximum Stress Error = ',num2str(err),'%']);
'text(.05,.22,'Stiffness Factor Error = ',...
num2str(ster),'%']);
legend('Radial shear stress',...'
'Tangential shear stress');
figure(gcf);
%disp('Use mouse to locate legend block');
%disp('Press [Enter] when finished');
%print -deps torsion

---------------------------------------------

function [trho,talpha,z,c,C]= ...
torstres(mapfun,zeta,ntrms,nft)
%
% [trho,talpha,z,c,C]= ...
torstres(mapfun,zeta,ntrms,nft)
% ---------------------------------------------
%
% This function computes torsional stresses in
% a beam such that abs(zeta)<=1 is mapped onto
% the beam cross section by a function named
% mapfun.
%
% trho - torsional stresses in directions
% normal to the lines into which
abs(zeta)=const map. These values

© 2003 by CRC Press LLC
% should be zero at the boundary
% corresponding to abs(zeta)=1.
% talpha - torsional stresses in directions
% tangent to the curves into which
% abs(zeta)=const map. The maximum
% value of shear stress always occurs
% at some point on the boundary defined
% by abs(zeta)=1.
% z - values of z where stresses are
% computed
% c - coefficients in the series expansion
% of the complex stress function
% C - complex Fourier coefficients of
% z.*conj(z) on the boundary of the
% beam cross section

% User m functions called: none
%----------------------------------------------

if nargin<4, nft=4096; end;
if nargin<3, ntrms=800; end

% Compute boundary values of the mapping
% function needed to construct the complex
% stress function
zetab=exp(i*linspace(0,2*pi*(nft-1)/nft,nft));
zb=feval(mapfun,zetab); zb=zb(:);

% Evaluate z and z'(zeta) at other
% desired points
[z,zp]=feval(mapfun,zeta);

% Compute Fourier coefficients for the complex
% stress function and its derivative
C=fft(zb.*conj(zb))/nft;
c=i*C(2:ntrms+1).*(1:ntrms)';
fp=polyval(flipud(c),zeta);

% Evaluate stresses relative to the curvilinear
% coordinate system
tcplx=zeta./abs(zeta.*zp).*(fp-i*conj(z).*zp);

% trho is the radial shear stress that should
% vanish at the boundary
trho=real(tcplx);
talpa is the circumferential stress which
gives the maximum stress of interest at the
boundary
talpha=-imag(tcplx);

% This function maps the interior of a circle
% onto the interior of a square using a rational
% function of the approximate form:
% z(zeta)=zeta*Sum(a(j)* ...
% zeta4^(j-1)/(1+Sum(b(j)*zeta4^(j-1))
% where zeta4=zeta^4
% zeta - matrix of complex values such that
% abs(zeta)<=1
% z,zp - matrices of values of the mapping
% function and its first derivative
% User m functions called: none

a=[ 1.07835, 1.37751, -0.02642, -0.09129, ...
  0.13460, -0.15763, 0.07430, 0.14858, ...
  0.01878, -0.00354 ];
b=[ 1.37743, 0.07157, -0.11085, 0.12778, ...
 -0.13750, 0.05313, 0.14931, 0.02683, ...
 -0.00350, -0.000120 ];

zeta4=zeta.^4; p=zeta.*polyval(flipud(a),zeta4);
q=polyval(flipud([1;b]),zeta4); z=p./q;

% Exit if the derivative of z is not needed
if nargout==1, return, end

% evaluate z'(zeta)
Two-dimensional problems in linear elastostatics of homogeneous bodies can be analyzed with the use of analytic functions. The primary quantities of interest are cartesian stress components $\tau_{xx}$, $\tau_{yy}$, and $\tau_{xy}$ and displacement components $u$ and $v$. These can be expressed as

$$\tau_{xx} + \tau_{yy} = 2[\Phi(z) + \overline{\Phi(z)}]$$

$$-\tau_{xx} + \tau_{yy} + 2i\tau_{xy} = 2[\Phi'(z) + \Psi(z)]$$

$$2\mu(u + iv) = \kappa\phi(z) - z\Phi(z) - \psi(z)$$

$$\phi(z) = \int \Phi(z)\,dz$$

$$\psi(z) = \int \Psi(z)\,dz$$

where $\mu$ is the shear modulus and $\kappa$ depends on Poisson’s ratio $\nu$ according to $\kappa = 3 - 4\nu$ for plane strain or $\kappa = (3 - \nu)/(1 + \nu)$ for plane stress. The above relations are known as the Kolosov-Muskhelishvili formulas [73] and they have been used to solve many practical problems employing series or integral methods. Bodies such as a circular disk, a plate with a circular hole, and a circular annulus can be handled for quite general boundary conditions. Solutions can also be developed for geometries where a rational function is known that maps the interior of a circle onto the desired geometry. Furthermore, complex variable methods provide the most general techniques available for solving a meaningful class of mixed boundary value problems such as contact problems typified by pressing a rigid punch into a half plane.

Fully understanding all of the analyses presented in [72, 73] requires familiarity with contour integration, conformal mapping, and multivalued functions. However, some of the closed form solutions given in these texts can be used without extensive background in complex variable methods or the physical concepts of elasticity theory. With that perspective let us examine the problem of computing stresses in an infinite plate uniformly stressed at infinity and having a general normal stress $N(\theta)$ and tangential shear $T(\theta)$ applied to the hole. We will use the general solution of Muskhelishvili [72] to evaluate stresses anywhere in the plate with particular interest on stress concentrations occurring around the hole. The stress functions $\Psi$ and $\Phi$

1Chapter 20.
can be represented as follows

\[ \Phi(z) = -\frac{1}{2\pi i} \int_{\gamma} \frac{(N + iT)\,d\sigma}{\sigma - z} + \alpha + \beta z^{-1} + \delta z^{-2}, \quad \sigma = e^{i\theta} \]

where \( \gamma \) denotes counterclockwise contour integration around the boundary of the hole and the other constants are given by

\[ \alpha = \frac{\tau_{xx}^\infty + \tau_{yy}^\infty}{4}, \quad \delta = -\frac{\tau_{xx}^\infty + \tau_{yy}^\infty + 2i\tau_{xy}^\infty}{2} \]

\[ \beta = -\frac{\kappa}{1 + \kappa} \frac{1}{2\pi} \int_0^{2\pi} (N + iT)e^{i\theta} \, d\theta \]

Parameters \( \alpha \) and \( \delta \) depend only on the components of stress at infinity, while \( \beta \) is determined by the force resultant on the hole caused by the applied loading. The quantity \( N + iT \) is the boundary value of radial stress \( \tau_{rr} \) and shear stress \( \tau_{r\theta} \) in polar coordinates. Hence

\[ N + iT = \tau_{rr} + iT_{r\theta}, \quad |z| = 1 \]

The transformation formulas relating Cartesian stresses \( \tau_{xx}, \tau_{yy}, \tau_{xy} \) and polar coordinate stresses \( \tau_{rr}, \tau_{r\theta}, \tau_{r\theta} \) are

\[ \tau_{rr} + \tau_{r\theta} = \tau_{xx} + \tau_{yy}, \quad -\tau_{rr} + \tau_{r\theta} + 2i\tau_{r\theta} = (-\tau_{xx} + \tau_{yy} + 2i\tau_{xy})e^{2i\theta} \]

Let us assume that \( N + iT \) is expandable in a Fourier series of the form

\[ N + iT = \sum_{n=-\infty}^{\infty} c_n \sigma^n, \quad \sigma = e^{i\theta} \]

where \( c_n \) can be obtained by integration as

\[ c_n = \frac{1}{2\pi} \int_0^{2\pi} (N + iT)\sigma^{-n} \, d\theta \]

or we can compute the approximate coefficients more readily using the FFT.

The stress function \( \Psi(z) \) is related to \( \Phi(z) \) according to

\[ \Psi = \frac{1}{z^2} \Phi \left( \frac{1}{\bar{z}} \right) - \frac{d}{dz} \left[ \frac{1}{z} \Phi(z) \right], \quad |z| \geq 1 \]

Substituting the complex Fourier series into the integral formula for \( \Phi \) gives

\[ \Phi = -\sum_{n=0}^{\infty} c_n z^n + \alpha + \beta z^{-1} + \delta z^{-2}, \quad |z| \leq 1 \]

\[ \Phi = \sum_{n=1}^{\infty} c_{-n} z^{-n} + \alpha + \beta z^{-1} + \delta z^{-2}, \quad |z| \geq 1 \]
which has the form
\[ \Phi = \sum_{n=0}^{\infty} a_n z^{-n} , \quad |z| \geq 1 \]

These two relations then determine \( \Psi \) as
\[ \Psi = \bar{\delta} + \bar{\beta} z^{-1} + (\alpha + a_0 - c_0) z^{-2} + \sum_{n=3}^{\infty} \left( (n - 1) a_{n-2} - c_{n-2} \right) z^{-n} \]

The last equation has the form
\[ \Psi = \sum_{n=0}^{\infty} b_n z^{-n} , \quad |z| \geq 1 \]

where the coefficients \( b_n \) are obtainable by comparing coefficients of corresponding powers in the two series. Hence, the series expansions of functions \( \Phi(z) \) and \( \Psi(z) \) can be generated in terms of the coefficients \( c_n \) and the stress components at infinity. The stresses can be evaluated by using the stress functions. Displacements can also be obtained by integrating \( \Phi \) and \( \Psi \), but this straightforward calculation is not discussed here.

The program \textit{runplate} was written to evaluate the above formulas by expanding \( N + iT \) using the FFT. Truncating the series for harmonics above some specified order, say \( np \), gives approximations for \( \Phi(z) \) and \( \Psi(z) \), which exactly represent the solution corresponding to the boundary loading defined by the truncated Fourier series. Using the same approach employed in Chapter 6 we can define \( N \) and \( T \) as piecewise linear functions of the polar angle \( \theta \).

The program utilizes several routines described in the table below.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{runplate}</td>
<td>define ( N ), ( T ), stresses at infinity, ( z )-points where results are requested, and the number of series terms used.</td>
</tr>
<tr>
<td>\textit{platecrc}</td>
<td>computes series coefficients defining the stress functions.</td>
</tr>
<tr>
<td>\textit{strfun}</td>
<td>evaluates ( \Phi ), ( \Psi ), and ( \Phi' ).</td>
</tr>
<tr>
<td>\textit{cartstrs}</td>
<td>evaluates Cartesian stresses for given values of ( z ) and the stress functions.</td>
</tr>
<tr>
<td>\textit{rec2polr}</td>
<td>transforms from Cartesian stresses to polar coordinate stresses.</td>
</tr>
<tr>
<td>\textit{pollip}</td>
<td>simplified interface to function \textit{polyval}.</td>
</tr>
</tbody>
</table>

The program solves two sample problems. The first one analyzes a plate having no loading on the hole, and stresses at infinity given by \( \tau_{yy}^\infty = 1 \), \( \tau_{xx}^\infty = \tau_{xy}^\infty = 0 \). Figure 12.11 shows that the circumferential stress on the hole varies between \(-1\) and
Figure 12.11: Stress Concentration around a Circular Hole in a Plate

3. producing a stress concentration factor of three due to the presence of the hole. The second problem applies a sinusoidally varying normal stress on the hole while the stresses at infinity are zero. Taking

\[
T=0; \, t_i=[0,0,0];
\]
\[
\text{th} = \text{linspace}(0,2\pi,81);
\]
\[
N = [\cos(4*\text{th}), \, 180/\pi*\text{th}];
\]

gives the results depicted in Figure 12.12. Readers may find it interesting to investigate how stresses around the hole change with different combinations of stress at infinity and normal stress distributions on the hole.
Figure 12.12: Harmonic Loading on a Circular Hole in a Plate
12.14.1 Program Output and Code

Program runplate

```matlab
function runplate(WhichProblem)
% Example: runplate(WhichProblem)
% ~~~~~~~~~~~~~~~~~
% Example to compute stresses around a
% circular hole in a plate using the
% Kolosov-Muskhelishvili method.
% User m functions required:
% platecr, strfun, cartstrs,
% rec2polr, polflip, lintrp

if nargin==0
    titl=['Stress Concentration Around a ', ...
          'Circular Hole in a Plate'];
    N=0; T=0; ti=[0,1,0]; kapa=2; np=50;
    Nn='N = 0'; Tt='T = 0';
    rz=linspace(1,3,20)'; tz=linspace(0,2*pi,81);
    z=rz*exp(i*tz); x=real(z); y=imag(z);
    viewpnt=[-40,10];
else
    titl=['Harmonic Loading on a Circular', ...
          ' Hole in a Plate'];
    th=linspace(0,2*pi,81)';
    N=[cos(4*th),180/pi*th];
    Nn='N = cos(4*theta)'; Tt='T = 0';
    T=0; ti=[0,0,0]; kapa=2; np=10;
    rz=linspace(1,2,10)'; tz=linspace(0,2*pi,81);
    z=rz*exp(i*tz); x=real(z); y=imag(z);
    viewpnt=[-20,20];
end

default('
STRESSES IN A PLATE WITH A CIRCULAR HOLE')
default('
Stress components at infinity are: '); default('%g',ti);
default('
Normal stresses on the hole are defined by ',Nn);
default('
Tangential stresses on the hole are defined by ',Tt))
```

© 2003 by CRC Press LLC
fprintf('Elastic constant kappa equals: ')
fprintf('%s',num2str(kapa));
fprintf('Highest harmonic order used is: ')
fprintf('%s',num2str(np));
[a,b,c]=platecrc(N,T,ti,kapa,np);
fprintf('\n');
fprintf('The Kolosov-Muskhelishvili stress functions have the series forms:');
fprintf('nPhi=sum(a(k)*z^(-k+1), k=1:np+1)');
fprintf('nPsi=sum(b(k)*z^(-k+1), k=1:np+3)');
fprintf('n');
fprintf('Coefficients defining stress Phi are:\n');
disp(a(:));
fprintf('Coefficients defining stress Psi are:\n');
disp(b(:));
%
Evaluate the stress functions
[Phi,Psi,Phip]=strfun(a,b,z);
%
Compute the Cartesian stresses and the principal stresses
[tx,ty,txy,pt1,pt2]=cartstrs(z,Phi,Psi,Phip);
theta=angle(z./abs(z)); x=real(z); y=imag(z);
[tr,tt,trt]=rec2polr(tx,ty,txy,theta);
pmin=num2str(min([pt1(:);pt2(:)]));
pmax=num2str(max([pt1(:);pt2(:)]));
disp(...
['Minimum Principal Stress = ',num2str(pmin)]);
disp(...
['Maximum Principal Stress = ',num2str(pmax)]);
fprintf('nPress [Enter] for a surface ');
fprintf('in the plate\n'); input('','s'); clf;
close; colormap('hsv');
surf(x,y,tt); xlabel('x axis'); ylabel('y axis'); zlabel('Circumferential Stress');
title(titl); grid on; view(viewpnt); figure(gcf);
%if nargin==0, print -deps strconc1
%else, print -deps strconc2; end
fprintf('All Done\n');
%=============================================  
function [a,b,c]=platecrc(N,T,ti,kapa,np)
%  
% [a,b,c]=platecrc(N,T,ti,kapa,np)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%  
% This function computes coefficients in the
% series expansions that define the Kolosov-
% Muskhelishvili stress functions for a plate
% having a circular hole of unit radius. The
% plate is uniformly stressed at infinity. On
% the surface of the hole, normal and tangential
% stress distributions N and T defined as
% piecewise linear functions are applied.
%
% N   - a two column matrix with each row
% containing a value of normal stress
% and polar angle in degrees used to
% specify N as a piecewise linear
% function of the polar angle. Step
% discontinuities can be included by
% using successive values of N with the
% same polar angle values. The data
% should cover the range of theta from
% 0 to 360. N represents boundary values
% of the polar coordinate radial stress.
% A single constant value can be input
% when N is constant (including zero
% if desired).
% T   - a two column matrix defining values of
% the polar coordinate shear stress on
% the hole defined as a piecewise linear
% function. The points where function
% values of T are specified do not need
% to be the same as as those used to
% specify N. Input a single constant
% when T is constant on the boundary.
% ti   - vector of Cartesian stress components
% [tx,ty,txy] at infinity.
% kapa - a constant depending on Poisson’s ratio
% nu.
%       kapa=3-4*nu for plane strain
%       kapa=(3-nu)/(1+nu) for plane stress

© 2003 by CRC Press LLC
When the resultant force on the hole is zero, then kappa has no effect on the solution.

np - the highest power of exp(i*theta) used in the series expansion of N+i*T. This should not exceed 255.

a - coefficients in the series expansion defining the stress function

\[ \Phi = \sum a(k) z^{-k+1}, \quad k=1:np+1 \]

b - coefficients in the series expansion defining the stress function

\[ \Psi = \sum b(k) z^{-k+1}, \quad k=1:np+3 \]

User m functions called: lintrp

% Handle case of constant boundary stresses
if length(N(:))==1; N=[N,0;N,360]; end
if length(T(:))==1; T=[T,0;T,360]; end

% Expand the boundary stresses in a Fourier series
f=pi/180; nft=512; np=min(np,nft/2-1);
theta=linspace(0,2*pi*(nft-1)/nft,nft);

% Interpolate linearly for values at the Fourier points
Nft=lintrp(f*N(:,2),N(:,1),theta);
Tft=lintrp(f*T(:,2),T(:,1),theta);
c=fft(Nft(:)+i*Tft(:))/nft;

% Evaluate auxiliary parameters in the series solutions
alp=(ti(1)+ti(2))/4; bet=-kapa*c(nft)/(1+kapa);
sig=(-ti(1)+ti(2)-2*i*ti(3))/2;

% Generate a and b coefficients using the Fourier coefficients of N+i*T.
a=zeros(np+1,1); b=zeros(np+3,1); j=(1:np)';
a(j+1)=c(nft+1-j); a(1)=alp;
a(2)=bet+c(nft); a(3)=sig+c(nft-1);
j=(3:np+2)'; b(j+1)=(j-1).*a(j-1)-conj(c(j-1));
b(1)=conj(sig); b(2)=conj(bet);
b(3)=alp+a(1)-conj(c(1));
Discard any negligibly small high order coefficients.

tol=max(abs(N(:);T(:);ti(:)))/1e4;
ka=max(find(abs(a)>tol));
if isempty(ka), a=0; else, a(ka+1:np+1)=[]; end
kb=max(find(abs(b)>tol));
if isempty(kb), b=0; else, b(kb+1:np+3)=[]; end

%=============================================

function [Phi,Psi,Phip]=strfun(a,b,z)
% % [Phi,Psi,Phip]=strfun(a,b,z)
% % % This function evaluates the complex stress functions Phi(z) and Psi(z)
% % as well as the derivative function Phi'(z)
% % using series coefficients determined from function platecrc. The calculation also
% % uses a function polflip defined such that polflip(a,z)=polyval(flipud(a(:)),z).
% % a,b - series coefficients defining Phi and Psi
% % z - matrix of complex values
% % Phi,Psi - complex stress function values
% % Phip - derivative Phi'(z)
% % User m functions called: polflip
% %==============================================

zi=1./z; np=length(a); a=a(:);
Phi=polflip(a,zi); Psi=polflip(b,zi);
Phip=-polflip((1:np-1).*a(2:np),zi)./z.^2;

%==============================================

function [tx,ty,txy,tp1,tp2]= ...
  cartstrs(z,Phi,Psi,Phip)
% % [tx,ty,txy,tp1,tp2]=cartstrs(z,Phi,Psi,Phip)
This function uses values of the complex stress functions to evaluate Cartesian stress components relative to the x,y axes.

- **z** - matrix of complex values where stresses are required
- **Phi, Psi** - matrices containing complex stress function values
- **Phip** - values of \( \Phi'(z) \)

- **tx, ty, txy** - values of the Cartesian stress components for the x,y axes
- **tp1, tp2** - values of maximum and minimum principal stresses

User m functions called: none

---

```matlab
A = 2 * real(Phi); B = conj(z) .* Phip + Psi;
C = A - B; R = abs(B);
tx = real(C); ty = 2 * A - tx; txy = -imag(C);
%----------------------------------------------
```

---

This function transforms Cartesian stress components \( tx, ty, txy \) to polar coordinate stresses \( tr, tt, trt \).

- **tx, ty, txy** - matrices of Cartesian stress components
- **theta** - a matrix of polar coordinate values. This can also be a single value if all stress components are rotated by the same angle.

- **tr, tt, trt** - matrices of polar coordinate stresses

---

© 2003 by CRC Press LLC
266 \% User m functions called: none
267 \%---------------------------------------------------------------
268
269: if length(theta(:))==1
270: \hspace{1cm} theta=theta*ones(size(tx)); end
271: a=(tx+ty)/2;
272: b=((tx-ty)/2-i*txy).*exp(2*i*theta);
273: c=a+b; tr=real(c); tt=2*a-tr; trt=-imag(c);
274:
275: \%==============================================================
276:
277: function y=polflip(a,x)
278: \%
279: \% y=polflip(a,x)
280: \% ~~~~~~~~~~~~~~
281: \%
282: \%
283: \% This function evaluates polyval(a,x) with
284: \% the order of the elements reversed.
285: \%
286: \%---------------------------------------------------------------
287: y=polyval(a(end:-1:1),x);
288:
289: \%================================================================
290:
291: \% function y=lintrp(xd,yd,x)
292: \% See Appendix B

12.14.2 Stressed Plate with an Elliptic Hole

This chapter is concluded with an example using conformal mapping in elasticity theory. We discussed earlier the useful property that harmonic functions remain harmonic under a conformal transformation. However, linear elasticity leads to the biharmonic Airy stress function which satisfies

\[
\left[ \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right] U = 0
\]

Unfortunately, a conformal transformation \( x + iy = \omega(\xi + i\eta) \) does not imply

\[
\left[ \frac{\partial^2}{\partial \xi^2} + \frac{\partial^2}{\partial \eta^2} \right] U = 0
\]

even though the mapping function has the trivial linear form \( z = c_1 \xi + c_0 \). Consequently, the analogy employed in the ideal flow problem is not applicable in linear
elasticity. This does not preclude use of conformal mapping in elasticity, but we encounter equations of very different structure in the mapped variables. We will examine that problem enough to illustrate the kind of differences involved. Let a mapping function \( z = \omega(\zeta) \) define curvilinear coordinate lines in the \( z \)-plane. A polar coordinate grid corresponding to \( \arg(\zeta) = \text{constant} \) and \( |\zeta| = \text{constant} \) maps into curves we term \( \rho \) lines and \( \alpha \) lines, respectively. Plotting of such lines was demonstrated previously with function gridview (mapping the exterior of a circle onto the exterior of an ellipse). It can be shown that curvilinear coordinate stresses \( \tau_{\rho\rho}, \tau_{\alpha\alpha}, \tau_{\rho\alpha} \) are related to cartesian stresses according to

\[
\tau_{\rho\rho} + \tau_{\alpha\alpha} = \tau_{xx} + \tau_{yy} \quad , \quad -\tau_{\rho\rho} + \tau_{\alpha\alpha} + 2i\tau_{\rho\alpha} = h(-\tau_{xx} + \tau_{yy} + 2i\tau_{xy})
\]

where

\[
h = \frac{\zeta\omega'(\zeta)}{\zeta\omega''(\zeta)}
\]

Muskhelishvili [72] has developed a general solution for a plate with an elliptic hole allowing general boundary tractions. Here we use one solution from his text which employs the mapping function \( z = \omega(\zeta) = R\left(\zeta + \frac{m}{\zeta}\right) \) and the stress functions

\[
\phi(z) = \int \Phi(z) \, dz \quad \psi(z) = \int \Psi(z) \, dz
\]

When \( \zeta \) is selected as the primary reference variable, we have to perform chain rule differentiation and write

\[
\Phi(z) = \frac{\phi'(\zeta)}{\omega'(\zeta)} \quad \Psi(z) = \frac{\psi'(\zeta)}{\omega'(\zeta)}
\]

\[
\Phi'(z) = \frac{\omega'(\zeta)\phi''(\zeta) - \omega''(\zeta)\phi'(\zeta)}{\omega'(\zeta)^3}
\]

in order to compute stresses in terms of the \( \zeta \)-variable. Readers unaccustomed to using conformal mapping in this context should remember that there is no stress state in the \( \zeta \)-plane comparable to the analogous velocity components which can be envisioned in a potential flow problem. We are simply using \( \zeta \) as a convenient reference variable to analyze physical stress and displacement quantities existing only in the \( z \)-plane.

Suppose the infinite plate has an elliptic hole defined by

\[
\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 = 1
\]

© 2003 by CRC Press LLC
and the hole is free of applied tractions. The stress state at infinity consists of a tension $p$ inclined at angle $\lambda$ with the $x$-axis. The stress functions relating to that problem are found to be ([72], page 338)

\[ \phi(\zeta) = b\zeta + \frac{c}{\zeta} \]

\[ \psi(\zeta) = d\zeta + \frac{e}{\zeta} + \frac{f\zeta}{\zeta^2 - m} , \quad |\zeta| \geq 1 \]

\[ a = e^{2\lambda} , \quad b = \frac{pr}{4} , \quad c = b(2a - m) \]

\[ d = -\frac{pra}{2} , \quad e = -\frac{pra}{2m} , \quad f = \frac{pr(m + \frac{1}{m})}{2} \]

Clearly these functions have no obvious relation to the simpler results shown earlier for a plate with a circular hole. The function eliphole computes curvilinear coordinate stresses in the $z$-plane expressed in terms of the $\zeta$-variable. When $\lambda = \pi/2$, the plate tension acts along the $y$-axis and the maximum circumferential stress occurs at $z = r_x$ corresponding to $\zeta = 1$. A surface plot produced by eliphole for the default data case using $r_x = 2$ and $r_y = 1$ is shown in Figure 12.13. It is also interesting to graph $\tau_{\alpha\alpha}/\tau_{yy}$ as a function of $r_x/r_y$. The program elpmaxst produces the plot in Figure 12.14 showing that the circumferential stress concentration increases linearly according to

\[ \frac{\tau_{\alpha\alpha}}{p} = 1 + 2 \left( \frac{r_x}{r_y} \right) \]

which can also be verified directly from the stress functions.
Figure 12.13: Circumferential Stress around an Elliptical Hole

Figure 12.14: Stress Concentration around an Elliptical Hole
12.14.3 Program Output and Code

Program elpmaxst

function elpmaxst
    % Example: elpmaxst
    % ~~~~~~~~~~~~~~~~~
    % MATLAB example to plot the stress
    % concentration around an elliptic hole
    % as a function of the semi-diameter ratio.
    %
    % User m functions required: eliphole
    %
    rx=2; ry=1; p=1; ang=90; ifplot=1;
    zeta=linspace(1,2,11)*
        exp(i*linspace(0,2*pi,121));
    eliphole(rx,ry,p,ang,zeta,1);

    r=linspace(1.001,10,19); tamax=zeros(size(r));
    for j=1:19
        [tr,tamax(j)]=eliphole(r(j),1,1,90,1);
    end
    plot(r,tamax,'-',r,tamax,'o');
    title(['Stress Concentration Around an ', ... 
        'Elliptical Hole']);
    xlabel(['ratio ( max diameter ) / ', ... 
        '( min diameter )']);
    ylabel(['( max circumferential stress ) / ',... 
        '( plate tension at infinity )']);
    grid on; figure(gcf);
    %print -deps elpmaxst
    %============================================= 

function [tr,ta,tra,z]=eliphole...
    (rx,ry,p,ang,zeta,ifplot)
    %
    % [tr,ta,tra,z]=eliphole(rx,ry,p,ang,...
    % zeta,ifplot)
    % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    %
    % This function determines curvilinear
    % coordinate stresses around an elliptic hole
% in a plate uniformly stressed at infinity.

% rx,ry - ellipse semidiameters on the x and
% y axes
% p   - values of uniaxial tension at
% infinity
% ang - angle of inclination in degrees
% of the tensile stress at infinity
% zeta - curvilinear coordinate values for
% which stresses are evaluated
% ifplot - optional parameter that is given
% a value if a surface plot of the
% circumferential stress is desired

% tr   - tensile stress normal to an
% elliptical coordinate line
% ta   - tensile stress in a direction
% tangential to the elliptical
% coordinate line
% tra  - shear stress complementary to the
% normal stresses
% z    - points in the z plane where
% stresses are computed

% User m functions called: none

%--------------------------------------------------------------

if nargin<6, ifplot=0; end
if nargin==0

rx=2; ry=1; p=1; ang=90; ifplot=1;
zeta=linspace(1,2,11)'* ...
   exp(i*linspace(0,2*pi,121));
end

r=(rx+ry)/2; m=(rx-ry)/(rx+ry);
z=r*(zeta+m./zeta); zeta2=zeta.^2;
zeta3=zeta.^3; wp=r*(1-m./zeta2);
wpp=2*r*m./zeta3; a=exp(2*i*pi/180*ang);
b=p*r/4; c=b*(2*a-m); d=-p*r/2*conj(a);
e=-p*r/2*a/m; f=p*r/2*(m+1/m)*(a-m);
phip=b-c./zeta2; phipp=2*c./zeta3;
h=wp.*zeta; h=h./conj(h);
Phi=phip./wp; Phipz=(wp.*phipp-wpp.*phip)./wp.^3;
Psi=(d-e./zeta2-f*(zeta2+m)./(zeta2-m).^2)./wp;
A=2*real(Phi); B=(conj(z).*Phipz+Psi).*h;
C=A-B; tr=real(C); ta=2*A-tr; tra=imag(B);
if ifplot>0
  colormap('gray'); brighten(.95);
surf(real(z),imag(z),ta);
xlabel('x axis'); ylabel('y axis');
zlabel('circumferential stress');
title(['Circumferential Stress Around ', ...
       'an Elliptical Hole']);
grid on; figure(gcf); input('','s');
end
Chapter 13

Nonlinear Optimization Applications

13.1 Basic Concepts

Optimization problems occur for a diverse range of topics. Perhaps the simplest type of optimization problem involves a scalar function of several variables. For example, the cost of a product having several ingredients may need to be minimized. This problem can be represented by a function \( F(x) \) which depends on the vector \( x = [x_1; x_2; \ldots; x_n] \) in \( n \)-dimensional space. Function \( F \) is called the objective function and cases where the independent variables \( x_i \) can vary arbitrarily are considered unconstrained. Most problems have constraints requiring \( x_i \) to remain within given bounds or satisfy other functional equations. Different analysis procedures exist for solving problems depending on whether they are linear or nonlinear, constrained or unconstrained. General solutions are available to handle linear objective functions with linear equality and inequality constraints. The discipline devoted to such problems is known as linear programming \[41\] and applications involving thousands of independent variables can be analyzed. \(^1\) Although this class of linear problems is important, it does not offer the versatility of methods used to address nonlinear problems (which are more compute intensive for problems of similar dimensionality). \(^2\) The material in this chapter addresses nonlinear problems with a few independent variables which are either constrained or restricted to lie within bounds of the form

\[
a_i \leq x_i \leq b_i.
\]

This type of constraint can be satisfied by taking

\[
x_i = a_i + (b_i - a_i) \sin^2(z_i)
\]

and letting \( z_i \) vary arbitrarily. The MATLAB intrinsic functions \texttt{fminbnd} and \texttt{fminsearch} are employed for solving this class of problems. The following five examples are presented to illustrate the nature of nonlinear optimization methods:

1. Computing the inclination angle necessary to cause a projectile to strike a stationary distant object;

\(^1\)High dimensionality linear problems should always be solved using the appropriate specialized software.  

\(^2\)The MathWorks markets an “Optimization Toolbox” intended to satisfy a number of specialized optimization needs.
2. Finding parameters of a nonlinear equation to closely fit a set of data values;

3. Determining components of end force on a statically loaded cable necessary to make the endpoint assume a desired position;

4. Computing the shape of a curve between two points such that a smooth particle slides from one end to the other in the minimum time;

5. Determining the closest points on two surfaces.

Before addressing specific problems, some of the general concepts of optimization will be discussed.

The minimum of an unconstrained differentiable function

\[ F(x_1, x_2, \ldots, x_n) \]

will occur at a point where the function has a zero gradient. Thus the condition

\[ \frac{\partial F}{\partial x_i} = 0, \quad 1 \leq i \leq n \]

leads to \( n \) nonlinear simultaneous equations. Such systems often have multiple solutions, and a zero gradient indicates either a maximum, or a minimum, or a saddle point. No reliable general methods currently exist to obtain all solutions to a general system of nonlinear equations. However, practical situations do occur where one unique point providing a relative minimum is expected. In such cases \( F(x) \) is called unimodal and we seek \( x_0 \) which makes

\[ F(x_0) < F(x_0 + \Delta) \text{ for } |\Delta| > 0. \]

Most unconstrained nonlinear programming software starts from an initial point and searches iteratively for a point where the gradient vanishes. Multimodal, or non-unimodal, functions can sometimes be solved by initiating searches from multiple starting points and using the best result obtained among all the searches. Since situations such as false convergence are fairly common with nonlinear optimization methods, results obtained warrant greater scrutiny than might be necessary for linear problems.

The intrinsic MATLAB functions \texttt{fminbnd} and \texttt{fminsearch} are adequate to address many optimization problems. Readers should study the documentation available for \texttt{fminbnd}, which performs a one-dimensional search within specified limits, and \texttt{fminsearch}, which performs an unconstrained multi-dimensional search starting from a user selected point. Both functions require objective functions of acceptable syntactical form. Various options controlling convergence tolerances and function evaluation counts should be studied to insure that the parameter choices are appropriately defined.
13.2 Initial Angle for a Projectile

In Chapter 8, equations of motion for motion of a projectile with atmospheric drag were formulated and a function \texttt{traject} producing a solution \(y(x)\) passing through \((x, y) = (0, 0)\) with arbitrary inclination was developed. The solution is generated for \(0 \leq x \leq x_f\) assuming the initial velocity is large enough for the projectile to reach \(x_f\). Therefore, program execution terminates if \(dx/dt\) goes to zero. In order to hit a target at position \((x_f, y_f)\), the starting angle of the trajectory must be selected iteratively because the equations of motion cannot be solved exactly (except for the undamped case). With the aid of an optimization method we calculate \(|y(x_f) - y_f)|\) and minimize this quantity (described in function \texttt{missdis} which has the firing angle as its argument). Function \texttt{fminbnd} seeks the angle to minimize the “miss” distance. Program \texttt{runtraj} illustrates the solution to the problem described and Figure 13.1 shows the trajectory required for the projectile to strike the object.

Depending on the starting conditions, zero, one, or two solutions exist to cause the “miss” distance to approach zero. Function \texttt{fminbnd} terminates at either a local minimum or at one of the search limits. The reader will need to examine how the initial data correlate to the final answers. For example, if the projectile misses the target by a significant amount, the initial projectile velocity was not large enough to reach the target.
Program Output and Code

Trajectory Analysis Program

function runtraj
% Example: runtraj
% ~~~~~~~~~~~~~~~~
% This program integrates the differential equations governing two-dimensional motion of a projectile subjected to gravity loading and atmospheric drag proportional to the velocity squared. The initial inclination angle needed to hit a distant target is computed repeatedly and function fmin is employed to minimize the square of the distance by which the target is missed. The optimal value of the miss distance is zero and the optimum angle will typically be found unless the initial velocity is too small and the horizontal velocity becomes zero before the target is passed. The initial velocity of the projectile must be large enough to make the problem well posed. Otherwise, the program will terminate with an error message.
% User m functions called: missdis, traject, projcteq

clear all;
global Vinit Gravty Cdrag Xfinl Yfinl

vinit=600; gravty=32.2; cdrag=0.002;
xfinl=1000; yfinl=100;
disp(’ ’);
disp(’SEARCH FOR INITIAL INCLINATION ANGLE ’);
disp(’TO MAKE A PROJECTILE STRIKE A DISTANT OBJECT’);
disp(’ ’);
disp([’Initial velocity = ’,num2str(vinit)]);
disp([’Gravity constant = ’,num2str(gravty)]);
disp([’Drag coefficient = ’,num2str(cdrag)]);
disp([’Coordinates of target = (’, ...
num2str(xfinl),',',...  
num2str(yfinl),')'); disp('');

% Replicate input data as global variables
Vinit=vinit; Gravty=gravty; Cdrag=cdrag;
Xfinl=xfinl; Yfinl=yfinl;

% Perform the minimization search
fstart=180/pi*atan(yfinl/xfinl); fend=75;
disp('Please wait for completion of the')
disp('minimization search');
bestang=fminbnd(@missdis,fstart,fend);

% Display final results
[y,x,t]=traject ...
(bestang,vinit,gravty,cdrag,xfinl);
dmiss=abs(yfinl-y(length(y))); disp('')
disp(['Final miss distance is ', ...
num2str(dmiss),', when the']);
disp(['initial inclination angle is ', ...
num2str(bestang),...  
', degrees'])};

%====================================================================

function [dsq,x,y]=missdis(angle)
%
% [dsq,x,y]=missdis(angle)
% ------------------------
%
% This function is used by fminbnd. It returns 
% an error measure indicating how much the 
% target is missed for a particular initial 
% inclination angle of the projectile.
%
% angle - the initial inclination angle of 
% the projectile in degrees
%
% dsq - the square of the difference between 
% Yfinal and the final value of y found 
% using function traject.
%
% Several global parameters (Vinit, Gravty, 
% Cdrag, Xfinl) are passed to missdis by the
% driver program runtraj.
% User m functions called: traject
%----------------------------------------------

 global Vinit Gravty Cdrag Xfinl Yfinl
 [y,x,t]=traject ...
 (angle,Vinit,Gravty,Cdrag,Xfinl,1);
 dsq=(y(length(y))-Yfinl)^2;
 %=============================================

function [y,x,t]=traject ...
 (angle,vinit,gravty,cdrag,xfinl,noplot)

 % This function integrates the dynamical
 % equations for a projectile subjected to
 % gravity loading and atmospheric drag
 % proportional to the square of the velocity.
 % angle - initial inclination of the
 % projectile in degrees
 % vinit - initial velocity of the projectile
 % (muzzle velocity)
 % gravty - the gravitational constant
 % cdrag - drag coefficient specifying the
 % drag force per unit mass which
 % equals cdrag*velocity^2.
 % xfinl - the projectile is fired toward the
 % right from x=0. xfinl is the
 % largest x value for which the
 % solution is computed. The initial
 % velocity must be large enough that
 % atmospheric damping does not reduce
 % the horizontal velocity to zero
 % before xfinl is reached. Otherwise
 % an error termination will occur.
 % noplot - plotting of the trajectory is
 % omitted when this parameter is
 % given an input value
% y,x,t - the y, x and time vectors produced by integrating the equations of motion

% Global variables:
% grav, - two constants replicating gravity and dragc cdrag, for use in function projcteq
% vtol - equal to vinit/1e6, used in projcteq to check whether the horizontal velocity has been reduced to zero

% User m functions called: projcteq
%----------------------------------------------

global grav dragc vtol

% Default data case generated when input is null
if nargin ==0
angle=45; vinit=600; gravty=32.2;
cdrag=0.002; xfinl=1000;
end;

% Assign global variables and evaluate initial velocity
grav=gravty; dragc=cdrag; ang=pi/180*angle;
vtol=vinit/1e6;
z0=[vinit*cos(ang); vinit*sin(ang); 0; 0];

% Integrate the equations of motion defined in function projcteq
deoptn=odeset('RelTol',1e-6);
[x,z]=ode45('projcteq',[0,xfinl],z0,deoptn);

y=z(:,3); t=z(:,4); n=length(x);
xf=x(n); yf=y(n);

% Plot the trajectory curve
if nargin < 6
plot(x,y,'k-',xf,yf,'ko');
xlabel('x axis'); ylabel('y axis');
title(["Projectile Trajectory for ", ... 'Velocity Squared Drag'"]);
axis('equal'); grid on; figure(gcf);
%print -deps trajplot
function zp=projcteq(x,z)

% This function defines the equation of motion
% for a projectile loaded by gravity and
% atmospheric drag proportional to the square
% of the velocity.

% x - the horizontal spatial variable
% z - a vector containing [vx; vy; y; t];
% zp - the derivative dz/dx which equals
% [vx'(x); vy'(x); y'(x); t'(x)];

% Global variables:
% grav - the gravity constant
% dragc - the drag coefficient divided by
% gravity
% vtol - a global variable used to check
% whether vx is zero

% User m functions called: none

global grav dragc vtol
vx=z(1); vy=z(2); v=sqrt(vx^2+vy^2);

% Check to see whether drag reduced the
% horizontal velocity to zero before the
% xfinl was reached.
if abs(vx) < vtol
disp(' ');
disp('***************************');
disp('ERROR in function projcteq. The ');
disp(' initial velocity of the projectile');
disp(' was not large enough for xfinal to');
disp(' be reached.');
disp('EXECUTION IS TERMINATED.');

© 2003 by CRC Press LLC
13.3 Fitting Nonlinear Equations to Data

Often an equation of known form is needed to approximately fit some given data values. An equation \( y(t) \) to fit \( m \) data values \((t_i, y_i)\) might be sought from an equation expressible as

\[
y = f(a_1, a_2, \ldots, a_n, t)
\]

where \( n \) parameters \( a_1, a_2, \ldots, a_n \) are needed to minimize the least squares error

\[
\epsilon(a_1, a_2, \ldots, a_n) = \sum_{j=1}^{n} [y_j - f(a_1, a_2, \ldots, a_n, t_j)]^2.
\]

The smallest possible error would be zero when the equation passes exactly through all the data values. Function \( \epsilon \) can be minimized with an optimizer such as \texttt{fminsearch}, or conditions seeking a zero gradient of \( \epsilon \) which require

\[
\frac{\partial \epsilon}{\partial a_i} = 2 \sum_{j=1}^{n} [f(a_1, a_2, \ldots, a_n, t_j) - y_j] \left( \frac{\partial f}{\partial a_i} \right)
\]

can be written. Note that the problem of minimizing a function and the problem of solving a set of nonlinear simultaneous equations are closely related. Solving large systems of nonlinear equations is difficult. Therefore, data fitting by use of function minimization procedures is typically more effective.

The formulation assuming \( y \) depends on a single independent variable could just as easily have involved several independent variables \( x_1, x_2, \ldots, x_N \), which would yield an equation of the form

\[
y = f(a_1, a_2, \ldots, a_n, x_1, x_2, \ldots, x_N).
\]

For instance, we might choose the simplest useful equation depending linearly on the independent variables

\[
y = \sum_{k=0}^{N} x_k a_k
\]

where \( x_0 = 1 \). The least squares error can be expressed as

\[
\epsilon(a_0, a_1, \ldots, a_n) = \sum_{j=1}^{n} \left[ y_j - \sum_{k=0}^{N} x_{jk} a_k \right]^2
\]
where $X_{jk}$ means the value of the $k^{th}$ independent variable at the $j^{th}$ data point. The condition that $\epsilon$ have a zero gradient gives

$$
\sum_{k=0}^{N} \left[ \sum_{j=1}^{n} X_{j1} X_{jk} \right] a_k = \sum_{j=1}^{n} X_{j1} y_j, \quad 1 \leq t \leq N.
$$

This linear system can be solved using traditional methods. Since the multiple indices in the equation are slightly cryptic, expressing the relationship in matrix notation is helpful. We get

$$
Y \approx X A
$$

where

$$
Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad X = [1, X_1, X_2, \ldots, X_N], \quad A = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix}
$$

with $X_i$ being the column matrix $[x_{i1}, x_{i2}, \ldots, x_{in}]$ and the first column of $X$ contains all ones. The requirement to minimize $\epsilon$ is simply

$$(X^T X) A = X^T Y$$

and MATLAB produces the desired solution using

```
A=X\Y;
```

Although taking $y$ as a linear function of parameters $a_0, a_1, \ldots, a_N$ produces solvable linear equations, the general situation yields nonlinear equations, and a minimization search procedure has greater appeal. We conclude this section with an example employing a minimization search.

Consider an experiment where data values $(t_i, y_i)$ are expected to conform to the transient response of a linear harmonic oscillator governed by the differential equation

$$m_0 \ddot{y} + c_0 \dot{y} + k_0 y = 0.$$

This equation has a solution representable as

$$y = a_1 e^{-|a_2| t} \cos(|a_3| t + a_4)$$

where $|a_2|$ makes the response decay exponentially and $|a_3|$ assures that the damped natural frequency is positive. Minimizing the error function

$$
\epsilon(a_1, a_2, a_3, a_4) = \sum_{j=1}^{n} \left[ y_j - a_1 e^{-|a_2| t_j} \cos(|a_3| t_j + a_4) \right]^2
$$

requires a four-dimensional search.
Approx. equation is \( y = a \times \exp(b \times t) \times \cos(c \times t + d) \)

\[ a = 1.4961 \quad b = -0.098761 \quad c = 2.4993 \quad d = 0.62104 \]

Figure 13.2: Data Approximating \( y = 1.5 \exp(-0.1t) \cos(2.5t + \pi/4) \)

The program vibfit tests data deviating slightly from an equation employing specific values of \( a_1, a_2, a_3, a_4 \). Then function fminsearch is used to verify whether the coefficients can be recovered from the data points. Figure 13.2 shows the data values and the equation resulting from the nonlinear least square fit. The results produced are quite acceptable.

Program Output and Code

Program vibfit

```matlab
function vibfit

% Example: vibfit
% ~~~~~~~~~~~~~~~
% This program illustrates use of the Nelder Mead multi-dimensional function minimization method to determine an equation for \( y(t) \) which depends nonlinearly on several parameters chosen to closely fit known data
```
% values. The program minimizes the sum of the
% squares of error deviations between the data
% values and results produced by the chosen
% equation. The example pertains to the time
% response curve characterizing free vibrations
% of a damped linear harmonic oscillator.
%
% User m functions called: vibfun
%
% Make the data vectors global to allow
% access from function vibfun
global timdat ydat

echo off;
disp(' '); disp('CHOOSING PARAMETERS');
disp('IN THE NONLINEAR EQUATION'); disp('Y = A*EXP(B*T)*COS(C*T+D)'); disp('TO OBTAIN THE BEST FIT TO GIVEN DATA'); fprintf('
Press [Enter] to list function\n'); fprintf('vibfun which is to be minimized\n'); pause;

% Generate a set of data to be fitted by a
% chosen equation.
a=1.5; b=-.1; c=2.5; d=pi/5;
timdat=0:.2:20;
ydat=a*exp(b*timdat).*cos(c*timdat+d);

% Add some random noise to the data
ydat=ydat+.1*(-.5+rand(size(ydat)));

% Function vibfun defines the quantity to be
% minimized by a search using function fmins.
disp(''); disp('The function to be minimized is:');
type vibfun.m; disp(' '); disp('The input data will be plotted next.'); disp('Press [Enter] to continue'); pause;
plot(timdat,ydat,'k.'); title('Input Data'); xlabel('time'); ylabel('y axis'); grid off; figure(gcf);
input('','s');

% Initiate the four-dimensional search
x=fminsearch(@vibfun,[1 1 1 1]);

% Check how well the computed parameters
% fit the data.
aa=x(1); bb=-abs(x(2)); cc=abs(x(3)); dd=x(4);
as=num2str(aa); bs=num2str(bb);
cs=num2str(cc); ds=num2str(dd);
ttrp=0:.05:20;
ytrp=aa*exp(bb*ttrp).*cos(cc*ttrp+dd);
disp(’ ’);
disp(’Press [Enter] to see how well’);
disp(’the equation fits the data’); pause;
plot(ttrp,ytrp,’k-’,timdat,ydat,’k.’);
str1=[’Approx. equation is y =’, ’
’ ’a*exp(b*t)*cos(c*t+d)’];
str2=[’a =’,as,’ b =’,bs,’ c =’,cs,’ d =’,ds]

%=============================================
% This function evalautes the least square
% error for a set of vibration data. The data
% vectors timdat and ydat are passed as global
% variables. The function to be fitted is:
% y=a*exp(b*t)*cos(c*t+d)
% x - a vector defining a,b,c and d
% z - the square of the norm for the vector
% of error deviations between the data and
% results the equation gives for current
% parameter values
101: % User m functions called: none
102: %----------------------------------------------
103: 104: global timdat ydat
105: a=x(1); b=-abs(x(2)); c=abs(x(3)); d=x(4);
106: z=a*exp(b*timdat).*cos(c*timdat+d);
107: z=norm(z-ydat)^2;

13.4 Nonlinear Deflections of a Cable

We will now present an optimization procedure to determine the static equilibrium position of a perfectly flexible inextensible cable having given end positions and a known distributed load per unit length. If \( \mathbf{R}(s) \) is the position of any point on the cable as a function of arc length \( 0 \leq s \leq L \), then the internal tension at position \( s \) is

\[
T(s) = F_e + \int_s^L q(s) \, ds
\]

with \( q(s) \) being the applied force per unit length and \( F_e \) being the support force at \( s = L \). The end force to produce a desired end deflection has to be determined in the analysis. However, the end deflection resulting from any particular choice of end force can be computed by observing that the tangent to the deflection curve will point along the direction of the cable tension. This means

\[
\frac{d\mathbf{R}}{ds} = \frac{T(s)}{|T(s)|}
\]

and

\[
\mathbf{R}(s) = \int_0^s \frac{T(t) \, dt}{|T(t)|} = \int_0^s \frac{F_e + \int_s^L q \, ds}{|F_e + \int_s^L q \, ds|} \, ds
\]

where \( \mathbf{R}(0) = 0 \) is taken as the position at the starting end. The deflection at \( s = L \) will have some specified position \( \mathbf{R}_e \) so that requiring \( \mathbf{R}(L) = \mathbf{R}_e \) gives a vector equation depending parametrically on \( F_e \). Thus, we need to solve three nonlinear simultaneous equations in the Cartesian components of force \( F_e \): A reasonable analytical approach is to employ an optimization search to minimize \( |\mathbf{R}(L) - \mathbf{R}_e| \) in terms of the components of \( F_e \).

The procedure described for a cable with continuous loading extends easily to a cable having several rigid links connected at frictionless joints where arbitrary concentrated forces are applied. The function \texttt{cabledef} evaluates the position of each joint when the joint forces and outer end force are given. With the end force on the last link treated as a parameter, function \texttt{endf} computes an error measure \( |\mathbf{F}(L) - \mathbf{R}_E|^2 \) to be minimized using function \texttt{fminsearch}. The optimization search seeks the components of \( F_e \) needed to reduce the error measure to zero. Specifying a sensible problem obviously requires that \( |\mathbf{R}_e| \) must not exceed the total length of all members.
in the chain. Initiating the search with a randomly generated starting force leads to a final force produced by \texttt{fminsearch}, which is then employed in another call to \texttt{cabldefl} to determine and plot the final deflection position as shown in Figure 13.3. Using a random initial guess for the end force was done to show that choosing bad starting data, insufficiently stringent convergence tolerances, or too few allowable function iterations can sometimes produce erroneous results. This emphasizes the need to always examine the results from nonlinear search procedures to assure that satisfactory answers are obtained.

**Program Output and Code**

**Program cablsolv**

```matlab
function [r,t,pends]=cablsolv(Len,P,Rend)
% [r,t,pends]=cablsolv(Len,P,Rend)
% ----------------------------------------
% This function computes the equilibrium
% position for a cable composed of rigid
% weightless links with loads applied at the
% frictionless joints. The ends of the cable
% are assumed to have a known position.
```

© 2003 by CRC Press LLC
% Len - a vector containing the lengths
%   Len(1), ..., Len(n)
% P - matrix of force components applied
%   at the interior joints. P(:,i) contains the Cartesian components of
%   the force at joint i.
% Rend - the coordinate vector giving the
%   position of the outer end of the last
%   link, assuming the outer end of the
%   first link is at [0,0,0].
% r - a matrix with rows giving the
%   computed equilibrium positions of all
%   ends
% t - a vector of tension values in the
%   links
% pends - a matrix having two rows which
%   contain the force components acting
%   at both ends of the chain to maintain
%   equilibrium

% User m functions called: endfl, cabldefl

if nargin < 3
% Example for a ten link cable with vertical
% and lateral loads
Len=1.5*ones(10,1); Rend=[10,0,0];
P=ones(9,1)*[0,-2,-1];
end

global len p rend
len=Len; rend=Rend; p=P; tol=sum(Len)/1e8;

% Start the search with a random force applied
% at the far end

% Perform several searches to minimize the
% length of the vector from the outer end of
% the last link to the desired position Rend
% where the link is to be connected to a
% support. The final end force should reduce
% the deflection error to zero if the search
% is successful.

opts=optimset('tolx',tol,'tolfun',tol,...

© 2003 by CRC Press LLC
maxfunevals', 2000);
endval = realmax;

% Iterate several times to avoid false
% convergence
for k = 1:5
    p0 = 10 * max(abs(p(:))) * rand(size(p, 2), 1);
    [pendk, endvalk, exitflag] = ...
        fminsearch(@endfl, p0, opts);
    if endvalk < endval
        pend = pendk(:); endval = endvalk;
    end
end

% Use the computed end force to obtain the
% final deflection. Also return the
% support forces.
[r, t, pstart] = cabldefl(len, [p; pend']);
x = r(:, 1); y = r(:, 2); z = r(:, 3);
pend = [pstart(:)'; pend(:)'];

% Plot the deflection curve of the cable
plot3(x, y, z, 'k-', x, y, z, 'ko'); xlabel('x axis');
ylabel('y axis'); zlabel('z axis');
title('Deflection Shape for a Loaded Cable');
axis('equal'); grid on; figure(gcf);
print -deps defcable

%=============================================

function enderr = endfl(pend)
% enderr = endfl(pend)
% ------------------------
% This function computes how much the
% position of the outer end of the last link
% deviates from the desired position when an
% arbitrary force pend acts at the cable end.
% pend - vector of force components applied
% at the outer end of the last link
% enderr - the deflection error defined by the
% square of the norm of the vector
from the computed end position and
the desired end position. This error
should be zero for the final
equilibrium position

User m functions called: cabldefl

----------------------------------------------

% Pass the lengths, the interior forces and the
% desired position of the outer end of the last
% link as global variables.
global len p rend

% use function cabldefl to compute the
% desired error
r=cabldefl(len,[p;pend(:)']);
rlast=r(size(r,1),:);
d=rlast(:)-rend(:); enderr=d'*d;

=================================================================================

function [r,t,pbegin]=cabldefl(len,p)
% [r,t,pbegin]=cabldefl(len,p)
% This function computes the static equilibrium
% position for a cable of rigid weightless
% links having concentrated loads applied at
% the joints and the outside of the last link.
% The outside of the first link is positioned
% at the origin.
% len - a vector of link lengths
% len(1), ..., len(n)
% p - a matrix with rows giving the
% force components acting at the
% interior joints and at the outer
% end of the last link
% r - matrix having rows which give the
% final positions of each node
% t - vector of member tensions
% pbegin - force acting at the outer end of
% the first link to achieve
\begin{verbatim}
147:  % equilibrium
148:  %
149:  % User m functions called: none
150:  %--------------------------------------------------------------
151:  n=length(len); len=len(:); nd=size(p,2);
152:  % Compute the forces in the links
153:  T=flipud(cumsum(flipud(p)));
154:  t=sqrt(sum((T.^2)'));
155:  % Obtain the deflections of the outer ends
156:  % and the interior joints
157:  r=cumsum(T./t(:,ones(1,nd)).*len(:,ones(1,nd)));
158:  r=[zeros(1,nd);r]; pbegin=-t(1)*r(2,:)/len(1);
\end{verbatim}

13.5 Quickest Time Descent Curve (the Brachistochrone)

The subject of variational calculus addresses methods to find a function producing the minimum value for an integral depending parametrically on the function. Typically, we have a relationship of the form

\[ I(y) = \int_{x_1}^{x_2} G(x, y, y'(x)) \, dx \]

where values of \( y \) at \( x = x_1 \) and \( x = x_2 \) are known, and \( y(x) \) for \( x_1 < x < x_2 \) is sought to minimize \( I \). A classical example in this subject is determining a curve starting at \((0, 0)\) and ending at \((a, b)\) so that a smooth particle will slide from one end to the other in the shortest possible time. Let \( X \) and \( Y \) be measured positive to the right and downward. Then the descent time for frictionless movement along the curve will be

\[ t = \frac{1}{\sqrt{2g}} \int_0^a \sqrt{1 + \left(\frac{Y''(X)}{Y(X)}\right)^2} \, dX, \quad Y(0) = 0, \ Y(a) = b. \]

This problem is solved in various advanced calculus books.\(^3\) The curve is a cycloid expressed in parametric form as

\[ X = k[\theta - \sin(\theta)], \ Y = k[1 - \cos(\theta)] \]

---

\(^3\)Weinstock [105] provides an excellent discussion of the brachistochrone problem using calculus of variation methods.
where $0 < \theta < \theta_f$. Values of $\theta_f$ and $k$ are found to make $x(\theta_f) = a$ and $Y(\theta_f) = b$. The exact descent time is

$$t_{\text{best}} = \theta_f \sqrt{\frac{k}{y}}$$

which is significantly smaller than the descent time for a straight line, which is

$$t_{\text{line}} = \sqrt{\frac{2(a^2 + b^2)}{gb}}.$$

Two functions, \texttt{brfaltim} and \texttt{bracifun}, are used to compute points on the brachistochrone curve and evaluate the descent time.

The main purpose of this section is to illustrate how optimization search can be used to minimize an integral depending parametrically on a function. The method used chooses a set of base points through which an interpolation curve is constructed to specify the function. Using numerical integration gives a value for the integral. Holding the $x$ values for the interpolation points constant and allowing the $y$ values to vary expresses the integral as a function of the $y$ values at a finite number of points. Then a multi-dimensional search function such as \texttt{fminsearch} can optimize the choice of $Y$ values. Before carrying out this process for the brachistochrone problem it is convenient to change variables so that $x = X/a$ and

$$Y(X) = b[x + y(x)] , \ 0 \leq x \leq 1,$$

with

$$y(0) = y(1) = 0.$$

Then the descent integral becomes

$$t = \frac{a}{\sqrt{2gb}} \int_0^1 \sqrt{\frac{1 + (b/a)^2[1 + y'(x)]^2}{x + y}} \ dx.$$

For any selected set of interpolation points, functions \texttt{spline} and \texttt{splined} can evaluate $y(x)$ and $y'(x)$ needed in the integrand, and function \texttt{gcquad} can be used to perform Gaussian integration. An optimization search employing \texttt{fminsearch} produces the curve heights yielding an approximation to the brachistochrone as shown in Figure 13.4.
Descent time (secs)
Approximate: 1.76
Exact: 1.7774
Error tolerance: 0.0001

Figure 13.4: Brachistochrone Curve for \( \frac{a}{b} = 3 \)
Program Output and Code

Program brachist

function brachist

% Example: brachist
% ~~~~~~~~~~~~~~~~~
% This program determines the shape of a
% smooth curve down which a particle can slide
% in minimum possible time. The analysis
% employs a piecewise cubic spline to define
% the curve by interpolation using a fixed set
% of base point positions. The curve shape
% becomes a function of the heights chosen at
% the base points. These heights are determined
% by computing the descent time as a function
% of the heights and minimizing the descent
% time by use of an optimization program. The
% Nelder and Mead unconstrained search
% procedure is used to compute the minimum.
%
% User m functions called:
% chbpts, brfaltim, fltim, gcquad,
% bracifun, splined

global cbp cwf cofs n xc yc a b b_over_a ...
  grav nparts nquad nfcls

fprintf(...
  'BRACHISTOCHRONE DETERMINATION BY NONLINEAR');
fprintf('
OPTIMIZATION SEARCH 
');
fprintf(['
Please wait. The ',...
'calculation takes a while.']);

% Initialize
a=30; b=10; grav=32.2; nparts=1; nquad=50;
tol=1e-4; n=6; b_over_a = b/a;
[dummy,cbp,cwf]=gcquad('’,0,1,nquad,nparts);
xc=chbps(0,1,n); xc=xc(:);
y0=5*sin(pi*xc); xc=[0;xc;1];

% Calculate results from the exact solution
[texact,xexact,yexact]=brfaltim(a,b,grav,100);
% Perform minimization search for approximate solution
opts=optimset('tolx',tol,'tolfun',tol);
[yfmin,fmin,flag,outp] = ...
    fminsearch(@fltim,y0,opts);

% Evaluate final position and approximate descent time
Xfmin=xc; Yfmin=Xfmin+[0;yfmin(:);0];
% tfmin=a/sqrt(2*grav*b)*fltim(yfmin(:));
tfmin=a/sqrt(2*grav*b)*fmin;
nfcls=1+outp.funcCount;

% Summary of calculations
fprintf('
Brachistochrone Summary');
fprintf('
-----------------------');
fprintf('
Number of function calls: %g',nfcls);
fprintf('
Descent time: %g',tfmin), fprintf('
')

% Plot results comparing the approximate and exact solutions
xplot=linspace(0,1,100);
yplot=spline(Xfmin,Yfmin,xplot);
plot(xplot,-yplot,'-',Xfmin,-Yfmin,'o', ...%
    xexact/a,-yexact/b,'--');
xlabel('x/a'); ylabel('y/b'); % grid
title(['Brachistochrone Curve for ', ...%
    'a/b = ',num2str(a/b)]);
text(.5,.1, 'Descent time (secs)');
text(.5,-.175,[ 'Approximate: ',num2str(tfmin)]);
text(.5,-.25, ['Exact: ',num2str(texact)]);
text(.5,-.325, ...%
    sprintf('Error tolerance: %g',tol));
legend('Approximate Curve', 'Computed Points','Exact Curve',3);
figure(gcf);
print -deps brachistograph

%============================================

function [tfall,xbrac,ybrac]=brfaltim ...
    (a,b,grav,npts)
% [tfall,xbrac,ybrac]=brfaltim(a,b,grav,npts)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
%
% This function determines the descent time
% and a set of points on the brachistochrone
% curve passing through (0,0) and (a,b).
% The curve is a cycloid expressible in
% parametric form as
% \[ x = k \cdot (\theta - \sin(\theta)), \]
% \[ y = k \cdot (1 - \cos(\theta)) \quad \text{for} \quad 0 \leq \theta \leq \theta_f \]
% where \( \theta_f \) is found by solving the equation
% \[ \frac{b}{a} = \frac{1 - \cos(\theta_f)}{(\theta_f - \sin(\theta_f))}. \]
% Once \( \theta_f \) is known then \( k \) is found from
% \[ k = \frac{a}{\theta - \sin(\theta)}. \]
% The exact value of the descent time is given
% by
% \[ tfall = \sqrt{\frac{k}{g}} \cdot \theta_f \]
% where \( a, b \) are final values of \((x,y)\) on the curve
% \( \text{grav} \) is the gravity constant
% \( npts \) is the number of points computed on
% the curve
% \( tfall \) is the time required for a smooth
% particle to slide along the curve
% from \((0,0)\) to \((a,b)\)
% \( xbrac \) is \( x \) points on the curve with \( x \)
% increasing to the right
% \( ybrac \) is \( y \) points on the curve with \( y \)
% increasing downward
% \( \text{User m functions called: none} \)
%----------------------------------------------
% brfn=inline(’cos(th)-1+cof*(th-sin(th))’,’th’,’cof’);
ba=b/a; [th,fval,flag]=fzero(...
    brfn,[.01,10],optimset('fzero'),ba);

k=a/(th-sin(th)); tfall=sqrt(k/grav)*th;
if nargin==4
    thvec=(0:npts-1)'*(th/(npts-1));
    xbrac=k*(thvec-sin(thvec));
    ybrac=k*(1-cos(thvec));
end

%=============================================
function x=chbpts(xmin,xmax,n)
%
% x=chbpts(xmin,xmax,n)
% ~~~~~~~~~~~~~~~~~~~~~
% Determine n points with Chebyshev spacing
% between xmin and xmax.
% User m functions called: none
%----------------------------------------------

x=(xmin+xmax)/2+((xmin-xmax)/2)*cos(pi/n*((0:n-1)'+.5));

%=============================================
function t=fltim(y)
%
% t=fltim(y)
% ~~~~~~~
%
% This function evaluates the time descent
% integral for a spline curve having heights
% stored in y.
%
% y - vector defining the curve heights at
%     interior points corresponding to base
%     positions in xc
%
% t - the numerically integrated time descent
%     integral evaluated by use of base points
%     cbp and weight factors cwf passed as
%     global variables

© 2003 by CRC Press LLC
13.6 Determining the Closest Points on Two Surfaces

Determining the closest points on two surfaces arises in applications such as robotic collision avoidance and container packing. Many types of surfaces can be parameterized using two curvilinear coordinates; so, the problem reduces to a four dimensional search to minimize the length of a line from a point on one surface to a point on the other surface. We call this the proximity problem and will consider typical instances involving two circular cylinders arbitrarily positioned in space as illustrated by the test examples of Figure 13.5. This application illustrates that, despite the apparent simplicity of this problem, convergence difficulties can occur with minimization search procedures, and several runs may be needed to get correct results.

An elementary way to analyze the proximity of two surfaces is to describe each surface by a grid of points and find the smallest element in a matrix describing the distance from point $i$ of the first surface to a point $j$ of the second surface. Large array
Figure 13.5: Geometry for Four Test Cases
dimensions can occur since a typical 100 by 100 surface grid involves 10,000 points and 30,000 coordinate values. The adjacency matrix for two surfaces, each using 10,000 points, has one hundred million points and would consume 2400 megabytes of memory when stored unpartitioned. However, memory limitations can be overcome by processing a few points at a time. In the program given below, a function \texttt{surf2surf} is presented to perform exhaustive search. It works well for the cylinder to cylinder problem and also handles some special cases. Since points and space curves are degenerate examples of surfaces, \texttt{surf2surf} can solve problems like obtaining the point on a curve closest to an arbitrary point in space.

For surfaces described by equations of the form $r(s_1, s_2)$ and $R(s_3, s_4)$, the proximity problem can be treated by minimizing $\|r - R\|^2$ as a function of $[s_1, s_2, s_3, s_4]$. In this context, let us discuss briefly the concepts used in function \texttt{fminsearch} based on the flexible polyhedron search procedure developed by Nelder and Mead [\textsuperscript{1}]. The search employs a polyhedron having $n + 1$ corners in $n$ space, which are initially aggregated about a starting point $x_0$. A sequence of moves repeatedly replaces corners at which the objective function has maximum values, with new corners corresponding to smaller values. Ultimately, the polyhedron is reduced in size and contracts to a point where the objective function is perceived to have a relative minimum. The algorithm embodied in \texttt{fminsearch} is useful but it sometimes gives false convergence. This experience led the authors to implement, for comparison purposes, a somewhat shorter version of the Nelder and Mead algorithm given in function \texttt{nelmed} shown in the following program \texttt{cylclose}. This program is designed to solve four test problems using functions \texttt{fminsearch}, \texttt{nelmed}, or \texttt{surf2surf}. Both implementations of the flexible polyhedron search are vulnerable to false convergence; so, it is necessary to initiate the search several times using random starting points. By making enough trials so that the same best result is obtained several times, reasonable confidence in the answers can be achieved. Furthermore, the program shows images of the cylinders and connecting minimum distance lines. These images can be rotated interactively to observe the validity of results. In the test cases considered, about eight trials was sufficient to produce the same best results at least twice. Some results showing computer output for case 4 are typical.

cylclose(1);

CASE 4 USING FUNCTION NELMED

<table>
<thead>
<tr>
<th>Trial Number</th>
<th>Minimum Distance</th>
<th>Function Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.915</td>
<td>163</td>
</tr>
<tr>
<td>2</td>
<td>1.916</td>
<td>161</td>
</tr>
<tr>
<td>3</td>
<td>1.710</td>
<td>207</td>
</tr>
<tr>
<td>4</td>
<td>2.293</td>
<td>156</td>
</tr>
<tr>
<td>5</td>
<td>1.710</td>
<td>154</td>
</tr>
<tr>
<td>6</td>
<td>2.165</td>
<td>139</td>
</tr>
<tr>
<td>7</td>
<td>2.165</td>
<td>122</td>
</tr>
<tr>
<td>8</td>
<td>1.710</td>
<td>182</td>
</tr>
</tbody>
</table>

© 2003 by CRC Press LLC
The analysis used FUNCTION NELMED
Shortest Distance = 1.710
Function Evaluations = 1284
Compute Time = 4.450 secs
cylclose(1);

CASE 4 USING FUNCTION FMINSEARCH
Trial Minimum Function
Number Distance Evaluations
1 1.710 223
2 2.293 472
3 2.293 693
4 2.293 295
5 2.165 286
6 2.165 585
7 1.710 265
8 1.915 231

The analysis used FUNCTION FMINSEARCH
Shortest Distance = 1.710
Function Evaluations = 3050
Compute Time = 10.930 secs
cylclose(3);

CASE 4 USING EXHAUSTIVE SEARCH
Shortest Distance = 1.729
Function Evaluations = 546
Compute Time = 0.440 secs

Note that incorrect answers were obtained repeatedly by fminsearch and nelmed, whereas exhaustive search gave the fastest and most reliable solution. Readers interested in exploring the convergence problems occurring with the Nelder and Mead search will find it instructive to run program cylclose to observe the variations in results produced from randomly chosen starting points. This example problem shows clearly that, unless the best result among a number of trials is taken, an incorrect answer may occur.

13.6.1 Discussion of the Computer Code

Program cylclose uses minimization search to determine the closest points on two arbitrarily positioned circular cylinders. Three solution methods are provided using functions fminsearch, nelmed, or surf2surf. Four test cases are included, and other
geometries can be analyzed by modifying data lines in function `cylclose`. The various modules in the program are listed in the following table.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Line</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cylclose</td>
<td>1-155</td>
<td>several functions are called to plot the geometry and perform the minimization search</td>
</tr>
<tr>
<td>cylpoint</td>
<td>159-178</td>
<td>gives the position of a point on a cylinder surface</td>
</tr>
<tr>
<td>decyl2cyl</td>
<td>182-197</td>
<td>computes the distance between points on two cylinders</td>
</tr>
<tr>
<td>cylfigs</td>
<td>201-244</td>
<td>plots the geometries for four data cases</td>
</tr>
<tr>
<td>plot2cyls</td>
<td>248-276</td>
<td>plots the geometry for two cylinders</td>
</tr>
<tr>
<td>cylpts</td>
<td>280-300</td>
<td>generates a grid of points on a cylinder surface</td>
</tr>
<tr>
<td>crnrpts</td>
<td>304-321</td>
<td>generates a dense set of points in an increasing set of data set</td>
</tr>
<tr>
<td>ortbas</td>
<td>325-332</td>
<td>creates orthonormal base vectors needed to define cylinder geometry</td>
</tr>
<tr>
<td>nelmed</td>
<td>336-475</td>
<td>function which performs the Nelder-Mead search</td>
</tr>
<tr>
<td>surf2surf</td>
<td>479-513</td>
<td>uses discrete search to compute closest points on two surfaces defined by coordinate grids. Large grids can be handled by calling function <code>srf2surf</code></td>
</tr>
<tr>
<td>srf2surf</td>
<td>517-534</td>
<td>uses discrete search to compute closest points on two surfaces defined by coordinate grids.</td>
</tr>
<tr>
<td>rads</td>
<td>538-550</td>
<td>gives base radii for example problems</td>
</tr>
</tbody>
</table>

Program `cylclose`

```matlab
function [dbest,r,R] = cylclose(srchtype,...
   ntrials,sidlen,tolx,tolf)
% [dbest,r,R] = cylclose(srchtype,ntrials,...
%   sidlen,tolx,tolf)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This program locates the points closest
% together on the surfaces of two circular
% cylinders arbitrarily positioned in space.
% A four-dimensional unconstrained search
% is performed using functions NELMED,
% FMINSEARCH, or SURF2SURF. The quantity
```

© 2003 by CRC Press LLC
% minimized is the square of the distance
% from an arbitrary point on one cylinder
% to an arbitrary point on the other cylinder.
% The search parameters specify axial and
% circumferential coordinates of points on
% the cylinder surfaces.
%
% srchtype - selects the solution method. Use
% 1, 2, or 3 for NELMED, FMINSEARCH,
% or SURF2SURF
% ntrials - Number of times the solution is
% repeated to avoid false
% convergence
% sidlen - initial polyhedron side length
% tolx - Convergence tolerance on solution
% vector
% tolf - Convergence tolerance on function
% value
%
% User m functions called:
% cylpoint, dcy12cyl, cylfigs, plot2cyls
% cylpts, cornrpts, ortbasis, nelmed,
% surfmany, surf2surf, srf2srf, rads

if nargin<5, tolf=1e-4; end
if nargin<4, tolx=1e-2; end
if nargin<3, sidlen=.5; end
if nargin<2, ntrials=8; end
if nargin<1, srchtype=1; end

if srchtype==1
  fname='FUNCTION NELMED';
elseif srchtype==2
  fname='FUNCTION FMINSEARCH';
else
  fname='EXHAUSTIVE SEARCH';
end

disp(' ')
disp(' CYLINDER PROXIMITY ANALYSIS')
disp(' USING A FOUR-DIMENSIONAL SEARCH')
cylfigs, drawnow, disp(' '), dumy=input(...
'Press return to begin the search','s');
close; ncases=4;
for jcase=1:ncases
  disp(’ ’), disp([’CASE ’,...
      num2str(jcase),’ USING ’,fname])

% Define several data cases
switch jcase
  case 1
    rad=1; len=3; r0=[4,0,0]; v=[0,0,1];
    Rad=1; Len=3; R0=[0,0,1]; V=[0,0,1];
  case 2
    rad=1; len=3; r0=[4,0,0]; v=[3,0,4];
    Rad=1; Len=3; R0=[0,0,4]; V=[0,3,4];
  case 3
    rad=1; len=5; r0=[4,0,0]; v=[-4,0,3];
    Rad=1; Len=5; R0=[0,0,1]; V=[0,0,1];
  case 4
    rad=1; len=4*sqrt(2); r0=[4,0,0];
    v=[-1,1,0];
    Rad=1; Len=3; R0=[0,0,-2]; V=[0,0,-1];
end

% Create data parameters used repeatedly
% during the search process
% First cylinder
dat=cumsum([0;rad;len;rad]);
dat=dat/max(dat); zdat=[dat,[0;0;len;len]];
rdat=[dat,[0;rad;rad;0]]; m=ortbasis(v);
% Second cylinder
dat=cumsum([0;Rad;Len;Rad]);
dat=dat/max(dat); Zdat=[dat,[0;0;Len;Len]];
Rdat=[dat,[0;Rad;Rad;0]]; M=ortbasis(V);
% Make several searches starting from
% randomly chosen points and keep
% the best answer obtained
ntotal=0; ntype=zeros(1,5); disp(’ ’)
tic; dbest=realmax; opts=optimset;
if srchtype<3
  disp(’Trial Minimum Function’) disp(’Number Distance Evaluations’) for k=1:ntrials
    winit=2*pi*rand(4,1);
if srchtype==1 % Search using nelmed
    [w,fmin,nvals,ntyp]=nelmed(@dcyl2cyl,...
    winit,sidlen,tolx,tolf,2000,0,...
    r0,m,rdat,zdat,R0,M,Rdat,Zdat);
elseif srchtype==2 % Search using fminsearch
    [w,fmin,xflag,outp]=fminsearch(@dcyl2cyl,...
    winit,opts,r0,m,rdat,zdat,R0,M,Rdat,Zdat);
    nvals=outp.funcCount; ntyp=zeros(1,5);
end
    dk=sqrt(dcyl2cyl(w,r0,m,rdat,zdat,...
    R0,M,Rdat,Zdat));
    fprintf('%4i %8.3f %7i
',k,dk,nvals)
    if dk<dbest, dbest=dk; W=w; end
end
    w=W; r=cylpoint(w(1),w(2),r0,m,rdat,zdat);
    R=cylpoint(w(3),w(4),R0,M,Rdat,Zdat);
    t=toc;
    fprintf(['The analysis used ',fname,'
'])
    if srchtype==1
        fprintf(['Reflect Expand Contract
%4i %7i %9i %7i
'],ntype(2),...
        ntype(3),ntype(4),ntype(5))
    end
else
    dplot=0.3; tic;
    [x,y,z,X,Y,Z]=plot2cyls(rad,len,r0,v,...
    Rad,Len,R0,V,dplot,' '); close;
    [dbest,r,R]=surf2surf(x,y,z,X,Y,Z);
    ntotal=length(x)*length(X); t=toc;
end
    fprintf...
    ['Shortest Distance = %8.3f
',...
    'Function Evaluations = %8i
',...
    'Compute Time = %8.3f secs
']
    dbest,ntotal,t)
    n=1; Rr=repmat(R,1,n+1)+(r-R)*(0:n)/n;
    hold off; clf,
    titl=['CASE ',num2str(jcase),' USING ',fname];
    dplot=0.3; plot2cyls(...
    rad,len,r0,v,Rad,Len,R0,V,dplot,titl);
    colormap([1 1 0]), hold on,
    plot3(Rr(1,:),Rr(2,:),Rr(3,:),'linewidth',2)
    title([titl,' : DISTANCE = ',...
© 2003 by CRC Press LLC
num2str(dbest),', CPU TIME = ',...
num2str(t),', SECS']
rotate3d on, shg, disp(' ')
disp('Rotate the figure or press')
disp('return to continue')
dumy=input(' ',s); close
end

function r=cylpoint(w1,w2,r0,m,rdat,zdat)
% r=cylpoint(w1,w2,v,r0,m,rdat,zdat)
% ----------------------------------------
% This function computes the position of a
% point on the surface of a circular cylinder
% arbitrarily positioned in space. The argument
% list parameters have the following form,
% where rad means cylinder radius, and len
% means cylinder length.
% b=2*rad+len;
% zdat=[[0,0]; [rad/b, 0];
%     [(rad+len)/b, len];[ 1, len]];
% rdat=zdat; rdat(2,2)=rad;
% rdat(3,2)=rad; rdat(4,2)=0;
% u=2*pi*sin(w1)^2; v=sin(w2)^2;
% z=interp1(zdat(:,1),zdat(:,2),v);
% rho=interp1(rdat(:,1),rdat(:,2),v);
% x=rho*cos(u); y=rho*sin(u);
% r=r0(:)+m*[x;y;z];
%
% function dsqr=dcyl2cyl(...
% w,r0,m,rdat,zdat,R0,M,Rdat,Zdat)
% %----------------------------------------
% % This function computes the square of the
% % distance between generic points on the
% % surfaces of two circular cylinders in three
% % dimensions.
% % User m functions called: cylpoint

© 2003 by CRC Press LLC
global fcount
fcount=fcount+1;
r=cylpoint(w(1),w(2),r0,m,rdat,zdat);
R=cylpoint(w(3),w(4),R0,M,Rdat,Zdat);

dsqr=norm(r-R)^2;

%===========================================

function cylfigs
% cylfigs
% ~~~~
% This function plots the geometries
% pertaining to four data cases used
% to test closest proximity problems
% involving two circular cylinders
%
% User m functions called: plot2cyls

w=rads; p=1:2; q=3:4; s=5:6; t=7:8;

rad=1; len=3; r0=[4,0,0]; v=[0,0,1];
Rad=1; Len=3; R0=[0,4,0]; V=[0,0,1];
d=.4; subplot(2,2,1)
[x,y,z,X,Y,Z]=plot2cyls(rad,len,r0,v,Rad,Len,...
    R0,V,d,'
    CASE 1'); hold on
plot3(w(p,1),w(p,2),w(p,3),'linewidth',2')
hold off

rad=1; len=3; r0=[4,0,0]; v=[3,0,4];
Rad=1; Len=3; R0=[0,4,0]; V=[0,3,4];
d=.4; subplot(2,2,2);
[x,y,z,X,Y,Z]=plot2cyls(rad,len,r0,v,Rad,Len,...
    R0,V,d,'
    CASE 2'); hold on
plot3(w(q,1),w(q,2),w(q,3),'linewidth',2')
hold off

rad=1; len=5; r0=[4,0,0]; v=[-4,0,3];
Rad=1; Len=5; R0=[0,4,0]; V=[0,0,1];
d=.4; subplot(2,2,3)
[x,y,z,X,Y,Z]=plot2cyls(rad,len,r0,v,Rad,Len,...
    R0,V,d,'
    CASE 3'); hold on
plot3(w(s,1),w(s,2),w(s,3),'linewidth',2')
hold off

rad=1; len=4*sqrt(2); r0=[4,0,0]; v=[-1,1,0];
Rad=1; Len=3; R0=[0,0,-2]; V=[0,0,-1];
d=.4; subplot(2,2,4);
[x,y,z,X,Y,Z]=plot2cyls(rad,len,r0,v,Rad,Len,...
R0,V,d,'CASE 4'); hold on
plot3(w(t,1),w(t,2),w(t,3),'linewidth',2')
hold off, subplot
% print -deps cylclose

%====================================================================

function [x,y,z,X,Y,Z]=plot2cyls(...
    rad,len,r0,vc,Rad,Len,R0,Vc,d,titl)
% [x,y,z,X,Y,Z]=plot2cyls(rad,len,r0,vc,Rad,...
% Len,R0,Vc,d,titl)
% ==============================================================
% This function generates point grids on the
% surfaces of two circular cylinders and plots
% both cylinders together
% %
% % User m functions called: cornrpts surfmany
% %
% if nargin==0
%    titl='TWO CYLINDERS';
%    rad=1; len=3; r0=[4,0,0]; vc=[3,0,4];
%    Rad=1; Len=3; R0=[0,4,0]; Vc=[0,3,4]; d=.2;
% end
% if isempty(titl), titl=' '; end
% u=2*rad+len; v=2*pi*rad;
% nu=ceil(u/d); nv=ceil(v/d);
% u=cornrpts([0,rad,rad+len,u],nu)/u;
% v=linspace(0,1,nv);
% [x,y,z]=cylpts(u,v,rad,len,r0,vc);
% U=2*Rad+Len; V=2*pi*Rad;
% Nu=ceil(U/d); Nv=ceil(V/d);
% U=cornrpts([0,Rad,Rad+Len,U],Nu)/U;
% V=linspace(0,1,Nv);
% [X,Y,Z]=cylpts(U,V,Rad,Len,R0,Vc);
% surfmany(x,y,z,X,Y,Z), title(titl)
% colormap([1 1 0]), shg
%
%====================================================================

function [x,y,z]=cylpts(...
    axial,circum,rad,len,r0,vectax)
% [x,y,z]=cylpts(axial,circum,rad,len,r0,vectax)
% This function computes a grid of points on the 
% surface of a circular cylinder 
% User m functions called: ortbasis

U=2*rad+len; u=U*axial(:); n=length(u); 
v=2*pi*circum(:)'; m=length(v); 
ud=[0,rad,rad+len,U]; 
r=interp1(ud,[0,rad,rad,0],u); 
z=interp1(ud,[0,0,len,len],u); 
x=r*cos(v); y=r*sin(v); z=repmat(z,1,m); 
% w=basis(vectax)*[x(:,y(:,z(:))]; 
w=ortbasis(vectax)*[x(:,y(:,z(:))]; 

x=r0(1)+reshape(w(1,:),n,m); 
y=r0(2)+reshape(w(2,:),n,m); 
z=r0(3)+reshape(w(3,:),n,m); 

function v=cornrpts(u,N)
% v=cornrpts(u,N)
% This function generates approximately N 
% points between min(u) and max(u) including 
% all points in u plus additional points evenly 
% spaced in each successive interval. 
% u - vector of points 
% N - approximate number of output points 
% v - vector of points in increasing order

u=sort(u(:))'; np=length(u); 
d=u(np)-u(1); v=u(1); 
for j=1:np-1 
dj=u(j+1)-u(j); nj=max(1,fix(N*dj/d)); 
v=[v,[u(j)+dj/nj*(1:nj)]]; 
end 

function mat=ortbasis(v)
% mat=ortbasis(v)
% ----------------
% This function generates a rotation matrix
% having v(:)/norm(v) as the third column

v=v(:)/norm(v); mat=[null(v'),v];
if det(mat)<0, mat(:,1)=-mat(:,1); end

%===========================================

function [xmin,fmin,m,ntype]=nelmed(...
    F,x0,dx,epsx,epsf,M,ifpr,varargin)
% [xmin,fmin,m,ntype]=nelmed(...
%     F,x0,dx,epsx,epsf,M,ifpr,varargin)
% This function performs multidimensional
% unconstrained function minimization using the
% direct search procedure developed by
% J. A. Nelder and R. Mead. The method is
% described in various books such as:
% 'Nonlinear Optimization', by M. Avriel
% %
% % F - objective function of the form
% % F(x,p1,p2,...) where x is vector
% % in n space and p1,p2,... are any
% % auxiliary parameters needed to
% % define F
% % x0 - starting vector to initiate
% % the search
% % dx - initial polyhedron side length
% % epsx - convergence tolerance on x
% % epsf - convergence tolerance on
% % function values
% % M - function evaluation limit to
% % terminate search
% % ifpr - when this parameter equals one,
% % different stages in the search
% % are printed
% % varargin - variable length list of parameters
% % which can be passed to function F
% % xmin - coordinates of the smallest
% % function value
% % fmin - smallest function value found
% % m - total number of function
% % evaluations made
% % ntype - a vector containing
% [ninit,nrefl,nexpn,ncontr,nshrnk]

© 2003 by CRC Press LLC
which tells the number of reflections, expansions, contractions, and shrinkages performed

User m functions called: objective function

if isempty(ifpr), ifpr=0; end
if isempty(M), M=500; end;
if isempty(epsf), epsf=1e-5; end
if isempty(epsx), epsx=1e-5; end

% Initialize the simplex array
x0=x0(:); n=length(x0); N=n+1; f=zeros(1,N);
x=repmat(x0,1,N)+[zeros(n,1),dx*eye(n,n)];
for k=1:N
  f(k)=feval(F,x(:,k),varargin{:});
end

ninit=N; nrefl=0; nexpn=0; ncontr=0;

Erx=realmax; Erf=realmax;
alpha=1.0; % Reflection coefficient
beta=0.5; % Contraction coefficient
gamma=2.0; % Expansion coefficient

% Top of the minimization loop
while Erx>epsx | Erf>epsf
  [f,k]=sort(f); x=x(:,k);
  if m>M, xmin=x(:,1); fmin=f(1); return; end
  % Generate the reflected point and function value
  c=sum(x(:,1:n),2)/n; xr=c+alpha*(c-x(:,N));
  fr=feval(F,xr,varargin{:}); m=m+1;
  nrefl=nrefl+1;
  if ifpr==1, fprintf(’ :RFL
’); end
  if fr<f(1)
% Expand and take best from expansion
% or reflection
xe=c+gamma*(xr-c);
fe=feval(F,xe,varargin{:});
m=m+1; nexpn=nexpn+1;
if ifpr==1, fprintf(’ :EXP \n’); end

if fr<fe
  % The reflected point was best
  f(N)=fr; x(:,N)=xr;
else
  % The expanded point was best
  f(N)=fe; x(:,N)=xe;
end

elseif fr<=f(n) % In the middle zone
  f(N)=fr; x(:,N)=xr;
else
  % Reflected point exceeds the second
  % highest value so either use contraction
  % or shrinkage
  if fr<f(N)
    xx=xr; ff=fr;
  else
    xx=x(:,N); ff=f(N);
  end
  xc=c+beta*(xx-c);
  fc=feval(F,xc,varargin{:});
m=m+1; ncontr=ncontr+1;
if fc<=ff
  % Accept the contracted value
  x(:,N)=xc; f(N)=fc;
  if ifpr==1, fprintf(’ :CNT \n’); end
else
  % Shrink the simplex toward
  % the best point
  x=(x+repmat(x(:,1),1,N))/2;
  for j=2:N
    f(j)=feval(F,x(:,j),varargin{:});
  end
  m=m+n; nshrnk=nshrnk+n;
if ifpr==1, fprintf(’ :SHR \n’); end
end

% Evaluate parameters to check convergence
favg=sum(f)/N; Erf=sqrt(sum((f-favg).^2)/n);
xcent=sum(x,2)/N; xdif=x-repmat(xcent,1,N);
Erx=max(sqrt(sum(xdif.^2)));
end % Bottom of the optimization loop

xmin=x(:,1); fmin=f(1);
ntype=[ninit,nrefl,nexpn,ncontr,nshrnk];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [d,r,R]=surf2surf(x,y,z,X,Y,Z,n)
% [d,r,R]=surf2surf(x,y,z,X,Y,Z,n)
% % This function determines the closest points on two
% % surfaces and the distance between these points. It
% % is similar to function srf2srf except that large
% % arrays can be processed.
% %
% % x,y,z - arrays of points on the first surface
% % X,Y,Z - arrays of points on the second surface
% % d - the minimum distance between the surfaces
% % r,R - vectors containing the coordinates of the
% % nearest points on the first and the
% % second surface
% % n - length of subvectors used to process the
% % data arrays. Sending vectors of length
% % n to srf2srf and taking the best of the
% % subresults allows processing of large
% % arrays of data points
% % User m functions used: srf2srf

if nargin<7, n=500; end
N=prod(size(x)); M=prod(size(X)); d=realmax;
kN=max(1,floor(N/n)); kM=max(1,floor(M/n));
for i=1:kN
    i1=1+(i-1)*n; i2=min(i1+n,N); i12=i1:i2;
    xi=x(i12); yi=y(i12); zi=z(i12);
    for j=1:kM
        [d1,r1,R1]=srf2srf(xi,yi,zi,X,Y,Z,n);
        if d1<d, d=d1; r=r1; R=R1; end
    end
end
```matlab
j1=1+(j-1)*n; j2=min(j1+n,M); j12=j1:j2;
[dij,rij,Rij]=srf2srf(xi,yi,zi,X(j12),Y(j12),Z(j12));
if dij<d, d=dij; r=rij; R=Rij; end
eend

%=================================================================================
function [d,r,R]=srf2srf(x,y,z,X,Y,Z)
% [d,r,R]=srf2srf(x,y,z,X,Y,Z)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function determines the closest points on two surfaces and the distance between these points.
% x,y,z - arrays of points on the first surface
% X,Y,Z - arrays of points on the second surface
% d - the minimum distance between the surfaces
% r,R - vectors containing the coordinates of the nearest points on the first and the second surface

x=x(:); y=y(:); z=z(:); n=length(x); v=ones(n,1);
X=X(:)'; Y=Y(:)'; Z=Z(:)'; N=length(X); h=ones(1,N);
d2=(x(:,h)-X(v,:)).^2; d2=d2+(y(:,h)-Y(v,:)).^2; d2=d2+(z(:,h)-Z(v,:)).^2;
[u,i]=min(d2); [d,j]=min(u); i=i(j); d=sqrt(d);
r=[x(i);y(i);z(i)]; R=[X(j);Y(j);Z(j)];

%=================================================================================
function R=rads
% R=rads
% Radii for the problem solutions

R=[...
0.7045 3.2903 0.8263
3.2932 0.7074 0.8295
0.7783 3.4977 0.3767
3.4994 0.7800 0.3755
0.0026 3.0000 2.9934
0.0028 1.0000 3.0001
0.7034 0.7107 -2.0000
1.5139 1.5320 -0.7382];

%=================================================================================
```

© 2003 by CRC Press LLC
% surfmany(x1,y1,z1,x2,y2,z2,x3,y3,z3,...
% xnxn,yn,zn)
% See Appendix B
# Appendix A

## List of MATLAB Routines with Descriptions

Table A.1: Description of MATLAB Programs and Selected Functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>finance</td>
<td>1</td>
<td>Financial analysis program illustrating programming methods.</td>
</tr>
<tr>
<td>inputv</td>
<td>1</td>
<td>Function to read several data items on one line.</td>
</tr>
<tr>
<td>polyplot</td>
<td>2</td>
<td>Program comparing polynomial and spline Interpolation.</td>
</tr>
<tr>
<td>squarrun</td>
<td>2</td>
<td>Program illustrating conformal mapping of a square.</td>
</tr>
<tr>
<td>squarmap</td>
<td>2</td>
<td>Function for Schwarz-Christoffel mapping of a circular disk inside a square.</td>
</tr>
<tr>
<td>cubrange</td>
<td>2</td>
<td>Function to compute data range limits for 2D or 3D data.</td>
</tr>
<tr>
<td>pendulum</td>
<td>2</td>
<td>Program showing animated large oscillations of a pendulum.</td>
</tr>
<tr>
<td>animpen</td>
<td>2</td>
<td>Function showing pendulum animation.</td>
</tr>
<tr>
<td>smdplot</td>
<td>2</td>
<td>Program to animate forced motion of a spring-mass-damper system.</td>
</tr>
<tr>
<td>smdsolve</td>
<td>2</td>
<td>Function to solve a constant coefficient linear second order differential equation with a harmonic forcing function.</td>
</tr>
<tr>
<td>strngrun</td>
<td>2</td>
<td>Program animating wave motion in a string with given initial deflection.</td>
</tr>
<tr>
<td>strngwav</td>
<td>2</td>
<td>Function to compute deflections of a vibrating string.</td>
</tr>
<tr>
<td>animate</td>
<td>2</td>
<td>Function to show animation of a vibrating string.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Routine</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>splinerr</td>
<td>2</td>
<td>Program showing differential geometry properties of a space curve.</td>
</tr>
<tr>
<td>curvprpsp</td>
<td>2</td>
<td>Function using spline interpolation to compute differential properties of a space curve.</td>
</tr>
<tr>
<td>splined</td>
<td>2</td>
<td>Function to compute first or second derivatives of a cubic spline.</td>
</tr>
<tr>
<td>srfex</td>
<td>2</td>
<td>Program illustrating combined plotting of several surfaces.</td>
</tr>
<tr>
<td>frus</td>
<td>2</td>
<td>Function to compute points on a frustum.</td>
</tr>
<tr>
<td>surfmany</td>
<td>2</td>
<td>Function to plot several functions together without distortion.</td>
</tr>
<tr>
<td>rgdbodmo</td>
<td>2</td>
<td>Program illustrating 3D rigid body rotation and translation.</td>
</tr>
<tr>
<td>rotatran</td>
<td>2</td>
<td>Function to perform coordinate rotation.</td>
</tr>
<tr>
<td>membran</td>
<td>3</td>
<td>Program illustrating static deflection of a membrane.</td>
</tr>
<tr>
<td>mbvprun</td>
<td>3</td>
<td>Program to solve a mixed boundary value problem for a circular disk.</td>
</tr>
<tr>
<td>makratsq</td>
<td>3</td>
<td>Program showing conformal mapping of a square using rational functions.</td>
</tr>
<tr>
<td>ratcof</td>
<td>3</td>
<td>Function to compute coefficients for rational function interpolation.</td>
</tr>
<tr>
<td>raterp</td>
<td>3</td>
<td>Function to evaluate a rational function using coefficients from function raterp.</td>
</tr>
<tr>
<td>strdyneq</td>
<td>3</td>
<td>Program to solve the structural dynamics equation using eigenvalue-eigenvector methods.</td>
</tr>
<tr>
<td>fhrmck</td>
<td>3</td>
<td>Function to solve a linear second order matrix differential equation having a harmonic forcing function.</td>
</tr>
<tr>
<td>recmemfr</td>
<td>3</td>
<td>Program illustrating use of functions null and eig to compute rectangular membrane frequencies.</td>
</tr>
<tr>
<td>Routine</td>
<td>Chapter</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>multimer</td>
<td>3</td>
<td>Program comparing execution of intrinsic MATLAB matrix multiplication and slow Fortran style using loops.</td>
</tr>
<tr>
<td>lintrp</td>
<td>4</td>
<td>Function for piecewise linear interpolation allowing finite jump discontinuities.</td>
</tr>
<tr>
<td>curvprop</td>
<td>4</td>
<td>Program to compute the length and area bounded by a curve defined by spline interpolation.</td>
</tr>
<tr>
<td>spcof</td>
<td>4</td>
<td>Function to compute spline interpolation coefficients used by function spterp.</td>
</tr>
<tr>
<td>spterp</td>
<td>4</td>
<td>Function to interpolate, differentiate, and integrate a cubic spline having general end conditions.</td>
</tr>
<tr>
<td>powermat</td>
<td>4</td>
<td>Function used by functions spcof and spterp.</td>
</tr>
<tr>
<td>splineq</td>
<td>4</td>
<td>Function to interpolate, integrate, and differentiate using the intrinsic function spline.</td>
</tr>
<tr>
<td>splincof</td>
<td>4</td>
<td>Function that computes coefficients used by spline to handle general end conditions.</td>
</tr>
<tr>
<td>matlbdat</td>
<td>4</td>
<td>Program that draws the word MATLAB using a spline.</td>
</tr>
<tr>
<td>finitdif</td>
<td>4</td>
<td>Program to compute finite difference formulas.</td>
</tr>
<tr>
<td>findifco</td>
<td>4</td>
<td>Function to compute finite difference formulas for derivatives of arbitrary order.</td>
</tr>
<tr>
<td>simpson</td>
<td>5</td>
<td>Function using Simpson’s rule to integrate an exact function or one defined by spline interpolation.</td>
</tr>
<tr>
<td>gcquad</td>
<td>5</td>
<td>Function to perform composite Gauss integration of arbitrary order, and return the base points and weight factors.</td>
</tr>
<tr>
<td>quadtest</td>
<td>5</td>
<td>Program comparing the performance of gcquad and quadl for several test functions.</td>
</tr>
<tr>
<td>Routine</td>
<td>Chapter</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>areaprog</td>
<td>5</td>
<td>Program to compute area, centroidal coordinates and inertial properties of general areas bounded by spline curves.</td>
</tr>
<tr>
<td>apro</td>
<td>5</td>
<td>Function to compute geometrical properties of general areas.</td>
</tr>
<tr>
<td>volrevol</td>
<td>5</td>
<td>Program to compute geometrical properties of partial volumes of revolution bounded by spline curves.</td>
</tr>
<tr>
<td>volrev</td>
<td>5</td>
<td>Function to compute geometrical properties of partial volumes of revolution.</td>
</tr>
<tr>
<td>rotasurf</td>
<td>5</td>
<td>Function to plot a partial surface of revolution.</td>
</tr>
<tr>
<td>ropesymu</td>
<td>5</td>
<td>Program using numerical and symbolic computation to evaluate geometrical properties of a rope shaped solid.</td>
</tr>
<tr>
<td>ropedraw</td>
<td>5</td>
<td>Function to draw a twisted rope shaped surface.</td>
</tr>
<tr>
<td>twistprop</td>
<td>5</td>
<td>Function using symbolic computation to obtain geometrical properties.</td>
</tr>
<tr>
<td>srfv</td>
<td>5</td>
<td>Function to compute geometrical properties of a solid specified by general surface coordinates.</td>
</tr>
<tr>
<td>polhdrun</td>
<td>5</td>
<td>Program to produce geometrical properties and a surface plot of an arbitrary polyhedron.</td>
</tr>
<tr>
<td>polhedron</td>
<td>5</td>
<td>Function for geometrical properties of a polyhedron.</td>
</tr>
<tr>
<td>polyxy</td>
<td>5</td>
<td>Function for geometrical properties of a polygon.</td>
</tr>
<tr>
<td>sqrtquadtest</td>
<td>5</td>
<td>Program using quad and gcquad to evaluate integrals having square root type singularities at the integration end points.</td>
</tr>
<tr>
<td>quadqsqrt</td>
<td>5</td>
<td>Function applying gcquad to integrals having square root type singularities.</td>
</tr>
<tr>
<td>quadlsqrt</td>
<td>5</td>
<td>Function applying quad to integrals having square root type singularities.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Routine</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>triplint</td>
<td>5</td>
<td>Program applying Gauss quadrature to evaluate a triple integral with variable integration limits.</td>
</tr>
<tr>
<td>plotjrun</td>
<td>6</td>
<td>Program to compute and plot integer order Bessel functions using the FFT.</td>
</tr>
<tr>
<td>runimpv</td>
<td>6</td>
<td>Program using the FFT to analyze earthquake data.</td>
</tr>
<tr>
<td>fouapprox</td>
<td>6</td>
<td>Function for Fourier series approximation of a general function.</td>
</tr>
<tr>
<td>fouseris</td>
<td>6</td>
<td>Program to plot truncated Fourier series expansions of general functions.</td>
</tr>
<tr>
<td>fousum</td>
<td>6</td>
<td>Function to sum a Fourier series and include coefficient smoothing.</td>
</tr>
<tr>
<td>cablinea</td>
<td>7</td>
<td>Program showing modal superposition analysis of a swinging cable.</td>
</tr>
<tr>
<td>udfre vib</td>
<td>7</td>
<td>Function computing undamped response of a second order matrix differential equation with general initial conditions.</td>
</tr>
<tr>
<td>strdynrk</td>
<td>7</td>
<td>Function using ode45 to solve a second order matrix differential equation.</td>
</tr>
<tr>
<td>deislner</td>
<td>7</td>
<td>Program comparing implicit second and fourth order integrators which use fixed stepsize.</td>
</tr>
<tr>
<td>mckde2i</td>
<td>7</td>
<td>Function to solve a matrix ODE using a second order fixed stepsize integrator.</td>
</tr>
<tr>
<td>mckde4i</td>
<td>7</td>
<td>Function to solve a matrix ODE using a fourth order fixed stepsize integrator.</td>
</tr>
<tr>
<td>rkdestab</td>
<td>8</td>
<td>Program to plot stability zones for Runge-Kutta integrators.</td>
</tr>
<tr>
<td>prun</td>
<td>8</td>
<td>Program illustrating ode45 response calculation of an inverted pendulum.</td>
</tr>
<tr>
<td>toprun</td>
<td>8</td>
<td>Program for dynamic response of a spinning top.</td>
</tr>
<tr>
<td>traject</td>
<td>8</td>
<td>Program for a projectile trajectory.</td>
</tr>
<tr>
<td>cablenl</td>
<td>8</td>
<td>Program illustrating animated nonlinear dynamic response for a multi-link cable of rigid links.</td>
</tr>
</tbody>
</table>

*continued on next page*
## Advanced Math and Mechanics Applications Using MATLAB

<table>
<thead>
<tr>
<th>Routine</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plotmotn</td>
<td>8</td>
<td>Function to animate the dynamic response of a cable.</td>
</tr>
<tr>
<td>sprchan</td>
<td>8</td>
<td>Program for animated nonlinear dynamics of an elastic cable shaken at both ends.</td>
</tr>
<tr>
<td>laplacec</td>
<td>9</td>
<td>Program using Fourier series to solve the Laplace equation in a rectangle having general boundary conditions.</td>
</tr>
<tr>
<td>recseris</td>
<td>9</td>
<td>Function to compute a harmonic function and gradient components in a rectangular region.</td>
</tr>
<tr>
<td>stringft</td>
<td>9</td>
<td>Program for Fourier series solution and animated response for a string with given initial displacement.</td>
</tr>
<tr>
<td>forcemove</td>
<td>9</td>
<td>Program for response of a string subjected to a moving concentrated load.</td>
</tr>
<tr>
<td>membwave</td>
<td>9</td>
<td>Program animating the response of a rectangular or circular membrane subjected to an oscillating concentrated force.</td>
</tr>
<tr>
<td>besjroot</td>
<td>9</td>
<td>Function to compute a table of integer order Bessel function roots.</td>
</tr>
<tr>
<td>membanim</td>
<td>9</td>
<td>Function to show animated membrane response.</td>
</tr>
<tr>
<td>bemimpact</td>
<td>9</td>
<td>Program showing wave propagation in a simply supported beam subjected to an oscillating end moment.</td>
</tr>
<tr>
<td>beamanim</td>
<td>9</td>
<td>Function to animate the motion of a vibrating beam.</td>
</tr>
<tr>
<td>pilevibs</td>
<td>9</td>
<td>Program illustrating the response of a pile embedded in an oscillating elastic foundation.</td>
</tr>
<tr>
<td>slabheat</td>
<td>9</td>
<td>Program for heat conduction in a slab having sinusoidally varying end temperature.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Routine</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>heatcyln</td>
<td>9</td>
<td>Program analyzing transient heat conduction in a circular cylinder.</td>
</tr>
<tr>
<td>tempstdy</td>
<td>9</td>
<td>Function for the steady-state temperature in a circular cylinder with general boundary conditions.</td>
</tr>
<tr>
<td>foubesco</td>
<td>9</td>
<td>Function to compute coefficients in a Fourier-Bessel series.</td>
</tr>
<tr>
<td>besjtabl</td>
<td>9</td>
<td>Function giving a table of integer order Bessel function roots.</td>
</tr>
<tr>
<td>rector</td>
<td>9</td>
<td>Program to compute torsional stresses in a beam of rectangular cross section.</td>
</tr>
<tr>
<td>eigverr</td>
<td>10</td>
<td>Program comparing eigenvalues of a second order differential equation computed using finite difference methods and using collocation with spline interpolation.</td>
</tr>
<tr>
<td>prnstres</td>
<td>10</td>
<td>Function to compute principal stresses and principal directions for a symmetric second order stress tensor.</td>
</tr>
<tr>
<td>trusvibs</td>
<td>10</td>
<td>Program to compute and show animation of the natural vibration modes of a general pin connected truss.</td>
</tr>
<tr>
<td>drawtruss</td>
<td>10</td>
<td>Function to draw the deflection modes of a truss.</td>
</tr>
<tr>
<td>eigsym</td>
<td>10</td>
<td>Function solving the constrained eigenvalue problem associated with an elastic structure fixed as selected points.</td>
</tr>
<tr>
<td>elmstf</td>
<td>10</td>
<td>Function to form mass and stiffness matrices of a pin connected truss.</td>
</tr>
<tr>
<td>colbuc</td>
<td>10</td>
<td>Program to compute buckling loads of a variable depth column with general end conditions.</td>
</tr>
<tr>
<td>cbfreq</td>
<td>10</td>
<td>Program comparing cantilever beam natural frequencies computed by exact, finite difference, and finite element methods.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Routine</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbfrqnwm</td>
<td>10</td>
<td>Function to compute exact cantilever beam frequencies.</td>
</tr>
<tr>
<td>cbfrqfdm</td>
<td>10</td>
<td>Function to compute cantilever beam frequencies using finite difference methods.</td>
</tr>
<tr>
<td>cbfrqfem</td>
<td>10</td>
<td>Function to compute cantilever beam frequencies using the finite element method.</td>
</tr>
<tr>
<td>elipfreq</td>
<td>10</td>
<td>Program for natural frequencies and animation of the mode shapes of an elliptic membrane.</td>
</tr>
<tr>
<td>frqsimpl</td>
<td>10</td>
<td>Function to compute elliptic membrane natural frequencies and mode shapes.</td>
</tr>
<tr>
<td>eigenrec</td>
<td>10</td>
<td>Function to solve a rectangular eigenvalue problem of the form: $XA + BX = \lambda(XC + DX)$.</td>
</tr>
<tr>
<td>plotmode</td>
<td>10</td>
<td>Function to plot the mode shapes of the membrane.</td>
</tr>
<tr>
<td>vdb</td>
<td>11</td>
<td>Program to compute shear, moment, slope, and deflection in a variable depth multi-support beam with general external loading conditions.</td>
</tr>
<tr>
<td>extload</td>
<td>11</td>
<td>Function to compute load and deformation quantities for distributed and concentrated loading on a beam.</td>
</tr>
<tr>
<td>sngf</td>
<td>11</td>
<td>Singularity function used to describe beam loads.</td>
</tr>
<tr>
<td>trapsum</td>
<td>11</td>
<td>Trapezoidal rule function used to integrate beam functions.</td>
</tr>
<tr>
<td>sqrtsurf</td>
<td>12</td>
<td>Function used to illustrate branch cut discontinuities for an analytic function.</td>
</tr>
<tr>
<td>elipinvr</td>
<td>12</td>
<td>Function to invert the function mapping the exterior of a circle onto the exterior of an ellipse.</td>
</tr>
<tr>
<td>elipdplt</td>
<td>12</td>
<td>Program showing grid lines for conformal mapping of a circular disk onto an elliptic disk.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Routine</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>elipdisk</td>
<td>12</td>
<td>Function mapping an elliptic disk onto a circular disk.</td>
</tr>
<tr>
<td>gridview</td>
<td>12</td>
<td>Function to plot a curvilinear coordinate grid.</td>
</tr>
<tr>
<td>linfrac</td>
<td>12</td>
<td>Function to perform linear fractional transformations.</td>
</tr>
<tr>
<td>crc2crc</td>
<td>12</td>
<td>Function analyzing mapping of circles and straight lines under a linear frac-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tional transformation.</td>
</tr>
<tr>
<td>eccentric</td>
<td>12</td>
<td>Function to determine a concentric annulus which maps onto a given eccentric</td>
</tr>
<tr>
<td></td>
<td></td>
<td>annulus.</td>
</tr>
<tr>
<td>swcsq10</td>
<td>12</td>
<td>Program illustrating both interior and exterior maps regarding a circle and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a square.</td>
</tr>
<tr>
<td>squarat</td>
<td>12</td>
<td>Rational function map taking the inside of a circle onto the interior of a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>square or the exterior of a square onto the exterior of a square.</td>
</tr>
<tr>
<td>swcsqmap</td>
<td>12</td>
<td>Function using truncated series expansions in relation to circle to square</td>
</tr>
<tr>
<td></td>
<td></td>
<td>maps.</td>
</tr>
<tr>
<td>lapcrcl</td>
<td>12</td>
<td>Program solving the Laplace equation in a circular disk for either Dirichlet</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or Neumann boundary conditions.</td>
</tr>
<tr>
<td>cauchtst</td>
<td>12</td>
<td>Program using a Cauchy integral to solve a mixed boundary value problem for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a circular disk.</td>
</tr>
<tr>
<td>cauchint</td>
<td>12</td>
<td>Function to numerically evaluate a Cauchy integral.</td>
</tr>
<tr>
<td>elipcyl</td>
<td>12</td>
<td>Program illustrating inviscid fluid flow about an elliptic cylinder in an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>infinite stream.</td>
</tr>
<tr>
<td>runtors</td>
<td>12</td>
<td>Program using a Cauchy integral and conformal mapping to compute torsio-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nal stresses in a beam.</td>
</tr>
<tr>
<td>runplate</td>
<td>12</td>
<td>Program using complex stress functions to compute stresses in a plate with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a circular hole.</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Routine</th>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>platecrc</td>
<td>12</td>
<td>Function computing series coefficients for complex stress functions pertaining to a plate with a circular hole.</td>
</tr>
<tr>
<td>strfun</td>
<td>12</td>
<td>Function to evaluate stress functions phi and psi.</td>
</tr>
<tr>
<td>cartstrs</td>
<td>12</td>
<td>Function using complex stress functions to evaluate Cartesian stress components.</td>
</tr>
<tr>
<td>rec2polr</td>
<td>12</td>
<td>Function transforming stress components from Cartesian to polar coordinates.</td>
</tr>
<tr>
<td>elipmaxst</td>
<td>12</td>
<td>Program using conformal mapping and complex stress functions to compute stress in a plate with an elliptic hole.</td>
</tr>
<tr>
<td>runtraj</td>
<td>13</td>
<td>Program using one-dimensional search to optimize a projectile trajectory.</td>
</tr>
<tr>
<td>vibfit</td>
<td>13</td>
<td>Program using multi-dimensional search to fit a nonlinear equation to vibration response data.</td>
</tr>
<tr>
<td>cablsolv</td>
<td>13</td>
<td>Program to compute large deflection static equilibrium of a loaded cable.</td>
</tr>
<tr>
<td>brachist</td>
<td>13</td>
<td>Program to determine a minimum time descent curve (brachistochrone).</td>
</tr>
<tr>
<td>cylclose</td>
<td>13</td>
<td>Program using multi-dimensional search to find the closest points on two adjacent circular cylinders.</td>
</tr>
<tr>
<td>surf2surf</td>
<td>13</td>
<td>Function using exhaustive search to find the closest points on two surfaces.</td>
</tr>
<tr>
<td>nelmed</td>
<td>13</td>
<td>Function similar to fminsearch which implements the Nelder and Mead algorithm for multi-dimensional search.</td>
</tr>
</tbody>
</table>
Appendix B

Selected Utility and Application Functions

Function animate

```matlab
function animate(x,y,titl,tim,trace)

% animate(x,y,titl,tim,trace)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function performs animation of a 2D curve
% x,y - arrays with columns containing curve positions
% for successive times. x can also be a single
% vector if x values do not change. The animation
% is done by plotting (x(:,j),y(:,j)) for
% j=1:size(y,2).
% titl- title for the graph
% tim - the time in seconds between successive plots
%
if nargin<5, trace=0; else, trace=1; end;
if nargin<4, tim=.05; end
if nargin<3, trac=''; end; [np,nt]=size(y);
if min(size(x))==1, j=ones(1,nt); x=x(:);
else, j=1:nt; end; ax=newplot;
if trace, XOR='none'; else, XOR='xor'; end
r=[min(x(:)),max(x(:)),min(y(:)),max(y(:))];
axis('equal') % Needed for an undistorted plot
axis(r), % axis('off')
curve = line('color','k','linestyle','-',
'erase',XOR, 'xdata',[],'ydata',[]);
xlabel('x axis'), ylabel('y axis'), title(titl)
for k = 1:nt
    set(curve,'xdata',x(:,j(k)),'ydata',y(:,k))
    if tim>0, pause(tim), end, drawnow, shg
end
```

© 2003 by CRC Press LLC
Function aprop

```matlab
function [p,zplot]=aprop(xd,yd,kn)
  %
  % [p,zplot]=aprop(xd,yd,kn)
  % ~~~~~~~~~~~~~~~~~~~~~~~~~
  % This function determines geometrical properties
  % of a general plane area bounded by a spline
  % curve
  %
  % xd,yd - data points for spline interpolation
  % with the boundary traversed in counter-
  % clockwise direction. The first and last
  % points must match for boundary closure.
  % kn - vector of indices of points where the
  % slope is discontinuous to handle corners
  % like those needed for shapes such as a
  % rectangle.
  % p - the vector [a,xcg,ycg,axx,axy,ayy]
  % containing the area, centroid coordinates,
  % moment of inertia about the y-axis,
  % product of inertia, and moment of inertia
  % about the x-axis.
  % zplot - complex vector of boundary points for
  % plotting the spline interpolated geometry.
  % The points include the numerical quadrature
  % points interspersed with data values.
  %
  % User functions called: gcquad, curve2d
  if nargin==0
    td=linspace(0,2*pi,13); kn=[1,13];
    xd=cos(td)+1; yd=sin(td)+1;
  end
  nd=length(xd); nseg=nd-1;
  [dum,bp,wf]=gcquad([],1,nd,6,nseg);
  [z,zplot,zp]=curve2d(xd,yd,kn,bp);
  w=[ones(size(z)), z, z.*conj(z), z.^2]*...;
  repmat(imag(conj(z).*zp),1,4);
  v=(wf'*w)./ [2,3,8,8]; vr=real(v); vi=imag(v);
  p=[vr(1:2),vi(2),vr(3)+vr(4),vi(4),vr(3)-vr(4)];
  p(2)=p(2)/p(1); p(3)=p(3)/p(1);
```

Function besjroot

1: function rts=besjroot(norder,nrts,tol)
2: %
3: % rts=besjroot(norder,nrts,tol)
4: % ----------------------------------
5: % This function computes an array of positive roots
6: % of the integer order Bessel functions besselj of
7: % the first kind for various orders. A chosen number
8: % of roots is computed for each order
9: % norder - a vector of function orders for which
10: % roots are to be computed. Taking 3:5
11: % for norder would use orders 3, 4, and 5.
12: % nrts - the number of positive roots computed for
13: % each order. Roots at x=0 are ignored.
14: % rts - an array of roots having length(norder)
15: % rows and nrts columns. The element in
16: % column k and row i is the k’th root of
17: % the function besselj(norder(i),x).
18: % tol - error tolerance for root computation.
19: if nargin<3, tol=1e-5; end
20: jn=inline('besselj(n,x)', 'x', 'n');
21: N=length(norder); rts=ones(N,nrts)*nan;
22: opt=optimset('TolFun',tol,'TolX',tol);
23: for k=1:N
24:   n=norder(k); xmax=1.25*pi*(nrts-1/4+n/2);
25:   xsrch=.1:pi/4:xmax; fb=besselj(n,xsrch);
26:   nf=length(fb); K=find(fb(1:nf-1).*fb(2:nf)<=0);
27:   if length(K)<nrts
28:     disp('Search error in function besjroot')
29:     rts=nan; return
30:   else
31:     K=K(1:nrts);
32:     for i=1:nrts
33:       interval=xsrch(K(i):K(i)+1);
34:       rts(k,i)=fzero(jn,interval,opt,n);
35:     end
36:   end
37: end

© 2003 by CRC Press LLC
Function cubrange

function range=cubrange(xyz,ovrsiz)
% This function determines limits for a square or cube shaped region for plotting data values
% in the columns of array xyz to an undistorted scale
% xyz - a matrix of the form [x,y] or [x,y,z] where x,y,z are vectors of coordinate points
% ovrsiz - a scale factor for increasing the window size. This parameter is set to one if only one input is given.
% range - a vector used by function axis to set window limits to plot x,y,z points undistorted. This vector has the form [xmin,xmax,ymin,ymax] when xyz has only two columns or the form [xmin,xmax,ymin,ymax,zmin,zmax] when xyz has three columns.

if nargin==1, ovrsiz=1; end
pmin=min(xyz); pmax=max(xyz); pm=(pmin+pmax)/2;
pd=max(ovrsiz/2*(pmax-pmin));
if length(pmin)==2
  range=pm([1,1,2,2])+pd*[-1,1,-1,1];
else
  range=pm([1 1 2 2 3 3])+pd*[-1,1,-1,1,1,1];
end

Function curve2d

function [z,zplot,zp]=curve2d(xd,yd,kn,t)
%
% [z,zplot,zp]=curve2d(xd,yd,kn,t)
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function generates a spline curve through
% given data points with corners (slope dis-
% continuities) allowed as selected points.
% xd,yd - real data vectors of length nd
% defining the curve traversed in
% counterclockwise order.
% kn - vectors of point indices, between one
% and nd, where slope discontinuities
% occur
% t - a vector of parameter values at which
% points on the spline curve are
% computed. The components of t normally
% range from one to nd, except when t is
% a negative integer,-m. Then t is
% replaced by a vector of equally spaced
% values using m steps between each
% successive pair of points.
% z - vector of points on the spline curve
% corresponding to the vector t
% zplot - a complex vector of points suitable
% for plotting the geometry
% zp - first derivative of z with respect to
% t for the same values of t as is used
% to compute z
% User m functions called: splined

nd=length(xd); zd=xd(:)+i*yd(:); td=(1:nd)';
if isempty(kn), kn=[1;nd]; end
kn=sort(kn(:)); if kn(1)==1, kn=[1;kn]; end
if kn(end)==nd, kn=[kn;nd]; end
N=length(kn)-1; m=round(abs(t(1)));
if -t(1)==m, t=linspace(1,nd,1+N*m)'; end
z=[]; zp=[]; zplot=[];
for j=1:N
k1=kn(j); k2=kn(j+1); K=k1:k2;
k=find(k1<=t & t<k2);
if j==N, k=find(k1<=t & t<=k2); end
if ~isempty(k)
zk=spline(K,zd(K),t(k)); z=[z,zk];
zplot=[zplot,zd(k1);zk];
if nargout==3
zp=[zp; splined(K, zd(K), t(k))];
end
end
end
zplot=[zplot; zd(end)];

Function eigenrec

function [eigs, vecs, Amat, Bmat]=eigenrec(A, B, C, D)
% [eigs, vecs, Amat, Bmat]=eigenrec(A, B, C, D)
% Solve a rectangular eigenvalue problem of the form: X*A+B*X=lambda*(X*C+D*X)
% n=size(B,1); m=size(A,2); s=[n,m]; N=n*m;
for i=1:n
    IK=sub2ind(s, i*ones(1,m), km);
    Bikn=B(i, kn); Dikn=D(i, kn);
    for j=1:m
        I=sub2ind(s, i, j);
        Amat(I, IK)=A(km, j)'; Bmat(I, IK)=C(km, j)';
        KJ=sub2ind(s, kn, j*ones(1,n));
        Amat(I, KJ)=Amat(I, KJ)+ Bikn;
        Bmat(I, KJ)=Bmat(I, KJ)+ Dikn;
    end
end
[vecs, eigs]=eig(Bmat\Amat);
[eigs, k]=sort(diag(eigs));
vecs=reshape(vecs(:, k), n, m, N);

Function eigsym

function [evcs, eigvals]=eigsym(k, m, c)
% [evcs, eigvals]=eigsym(k, m, c)
% This function solves the eigenvalue of the constrained eigenvalue problem
% k*x=(lambda)*m*x, with c*x=0.
% Matrix k must be real symmetric and matrix
% m must be symmetric and positive definite;
% otherwise, computed results will be wrong.
% k - a real symmetric matrix
% m - a real symmetric positive definite matrix
% c - a matrix defining the constraint condition c*x=0. This matrix is omitted if no constraint exists.
% evecs - matrix of eigenvectors orthogonal with respect to k and m. The following relations apply:
% evecs'*m*evecs=identity_matrix
% evecs'*k*evecs=diag(eigvals).
% eigvals - a vector of the eigenvalues sorted in increasing order
% User m functions called: trifacsm

if nargin==3
q=null(c); m=q'*m*q; k=q'*k*q;
end
u=trifacsm(m); k=u'
/u; k=(k+k')/2;
[evecs,eigvals]=eig(k);
[eigvals,j]=sort(diag(eigvals));
evecs=evecs(:,j); evecs=u'
evecs;
if nargin==3, evecs=q*evecs; end

function [t,y,lambda]=fhrmck(m,c,k,f1,f2,w,tlim,nt,y0,v0)
% [t,y,lambda]=fhrmck(m,c,k,f1,f2,w,tlim,nt,y0,v0)
% This function uses eigenfunction analysis to solve the matrix differential equation
% m*y''(t)+c*y'(t)+k*y(t)=f1*cos(w*t)+f2*sin(w*t)
% with initial conditions of y(0)=y0, y'(0)=v0
% The solution is general unless 1) a zero or repeated eigenvalue occurs or 2) the system is undamped and the forcing function matches a natural frequency.
% If either error condition occurs, program execution terminates with t and y set to nan.
% m, c, k - mass, damping, and stiffness matrices
% f1, f2 - amplitude vectors for the sine and cosine forcing function components
% w - frequency of the forcing function
% tlim - a vector containing the minimum and maximum time limits for evaluation of the solution
% nt - the number of times at which the solution is evaluated within the chosen limits
% y0, v0 - initial position and velocity vectors
% t - vector of time values for the solution
% y - matrix of solution values where y(i, j) is the value of component j at time t(i)
% lam - the complex natural frequencies arranged in order of increasing absolute value

if nargin==0 % Generate default data using 2 masses
    m=eye(2,2); k=[2, -1; -1, 1]; c=.3*k;
    f1=[0; 1]; f2=[0; 0]; w=0.6; tlim=[0, 100]; nt=400;
end
n=size(m,1); t=linspace(tlim(1), tlim(2), nt);
if nargin<10, y0=zeros(n,1); v0=y0; end

% Determine eigenvalues and eigenvectors for the homogeneous solution
A=[zeros(n,n), eye(n,n); -m, k];
[U, lam]=eig(A); [lam, j]=sort(diag(lam)); U=U(:, j);

% Check for zero or repeated eigenvalues and for undamped resonance
wmin=abs(lam(1)); tol=wmin/1e6;
[dif, J]=min(abs(lam-i*w)); lj=num2str(lam(J));
if wmin==0, disp(''), disp('The homogeneous equation has a zero eigenvalue which is not allowed. '), disp(''), t=nan; y=nan; return
elseif any(abs(diff(lam))<tol)
    disp('A repeated eigenvalue occurred. '), disp(''),
    t=nan; y=nan; return
elseif dif<tol & sum(abs(c(:)))==0
    disp(''), disp(''),
    t=nan; y=nan; return
© 2003 by CRC Press LLC
disp('The system is undamped and the forcing')
disp(['function resonates with ',...  
    'eigenvalue ',lj])
disp('Execution is terminated.')
disp(' '), t=nan; y=nan; return
else
% Determine the particular solution
a=(-w^2*m+k+i*w*c)(f1-i*f2);
yp=real(a*exp(i*w*t));
yp0=real(a); vp0=real(i*w*a);
end

% Scale the homogeneous solution to satisfy the initial conditions
U=U*diag(U\(y0-yp0; v0-vp0));
yh=real(U(1:n,:)*exp(lam*t));

% Combine results to obtain the total solution

% Show data graphically only for default case
if nargin==0
    waterfall(t,(1:n),y'), xlabel('time axis')
ylabel('mass index'), zlabel('Displacements')
title(['DISPLACEMENT HISTORY FOR A ',...  
    int2str(n),'-MASS SYSTEM'])
colormap([1,0,0]), shg
end

Function findifco

function [c,e,m,crat]=findifco(k,a)
%  
% [c,e,m,crat]=findifco(k,a)
%  -------------------------------
% This function approximates the k’th derivative of a function using function values at n interpolation points. Let f(x) be a general function having its k’th derivative denoted by F(x,k). The finite difference approximation for the k’th derivative employing a stepsize h is given by:
% F(x,k)=Sum(c(j)*f(x+a(j)*h), j=1:n)/h^k +
% TruncationError
% with m=n-k being the order of truncation
% error which decreases like h^m and
% TruncationError=(h^m)*(e(1)*F(x,n)+...
% e(2)*F(x,n+1)*h+e(3)*F(x,n+2)*h^2+O(h^3))
% a - a vector of length n defining the
% interpolation points x+a(j)*h where
% x is an arbitrary parameter point
% k - order of derivative evaluated at x
% c - the weighting coefficients in the
% difference formula above. c(j) is
% the multiplier for value f(x+a(j)*h)
% e - error component vector in the above
% difference formula
% m - order of truncation order in the
% formula. The relation m=n-k applies.
% crat - a matrix of integers such that c is
% approximated by crat(1,:)/crat(2,:)

a=a(:); n=length(a); m=n-k; mat=ones(n,n+4);
for j=2:n+4; mat(:,j)=a/(j-1).*mat(:,j-1); end
A=pinv(mat(:,1:n)); ec=-A*mat(:,n+1:n+4);
c=A(k+1,:); e=-ec(k+1,:);
[ctop,cbot]=rat(c,1e-8); crat=[ctop(:)';cbot(:)'

Function gcquad

function [val,bp,wf]=gcquad(func,xlow,...
    xhigh,nquad,mparts,varargin)
% [val,bp,wf]=gcquad(func,xlow,...
%     xhigh,nquad,mparts,varargin)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function integrates a general function using
% a composite Gauss formula of arbitrary order. The
% integral value is returned along with base points
% and weight factors obtained by an eigenvalue based
% method. The integration interval is divided into
% mparts subintervals of equal length and integration
% over each part is performed with a Gauss formula

© 2003 by CRC Press LLC
% making nquad function evaluations. Results are
% exact for polynomials of degree up to 2*nquad-1.
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% func - name of a function to be integrated
% having an argument list of the form
% func(x,p1,p2,...) where any auxiliary
% parameters p1,p2,.. are passed through
% variable varargin. Use [ ] for the
% function name if only the base points
% and weight factors are needed.
% xlow,xhigh - integration limits
% nquad - order of Gauss formula chosen
% mparts - number of subintervals selected in
% the composite integration
% varargin - variable length parameter used to
% pass additional arguments needed in
% the integrand func
% val - numerical value of the integral
% bp,wf - vectors containing base points and
% weight factors in the composite
% integral formula

% A typical calculation such as:
% Fun=inline('(sin(w*t).^2).*exp(c*t)','t','w','c');
% A=0; B=12; nquad=21; mparts=10; w=10; c=8;
% [value,pcterr]=integrate(Fun,A,B,nquad,mparts,w,c);
% gives value = 1.935685556078172e+040 which is
% accurate within an error of 1.9e-13 percent.
% User m functions called: the function name passed
% in the argument list
%-----------------------------------------------

if isempty(nquad), nquad=10; end
if isempty(mparts), mparts=1; end

% Compute base points and weight factors
% for the single interval [-1,1]. (Ref:
% 'Methods of Numerical Integration' by
% P. Davis and P. Rabinowitz, page 93)
u=(1:nquad-1)./sqrt((2*(1:nquad-1)).^2-1);
[vc,bp]=eig(diag(u,-1)+diag(u,1));
[bp,k]=sort(diag(bp)); wf=2*vc(1,k).^2;
61: % Modify the base points and weight factors
62: % to apply for a composite interval
63: d=(xhigh-xlow)/mparts; d1=d/2;
64: dbp=d1*bp(:); dwf=d1*wf(:); dr=d*(1:mparts);
65: cbp=dbp(:,ones(1,mparts))+ ... 
66: dr(ones(nquad,1),:)+(xlow-d1);
67: cwf=dwf(:,ones(1,mparts)); wf=cwf(:,); bp=cbp(:,); 
68: 
69: % Compute the integral
70: if isempty(func) 
71: val=[]; 
72: else 
73: f=feval(func,bp,varargin{:}); val=wf’*f(:); 
74: end 

Function gridview

1 function gridview(x,y,xlabl,ylabl,titl)
2 %
3 % gridview(x,y,xlabl,ylabl,titl)
4 % -----------------------------
5 % %
6 % This function views a surface from the top
7 % to show the coordinate lines of the surface.
8 % It is useful for illustrating how coordinate
9 % lines distort in a conformal transformation.
10 % Calling gridview with no arguments depicts the
11 % mapping of a polar coordinate grid map under
12 % a transformation of the form
13 % z=R*(zeta+m/zeta).
14 %
15 % x,y - real matrices defining a
16 % curvilinear coordinate system
17 % xlabl,ylabl - labels for x and y axes
18 % titl - title for the graph
19 %
20 % User m functions called: cubrange
21 %-----------------------------
22 
23 % close
24 if nargin<5
25 xlabl='real axis'; ylabl='imaginary axis';

© 2003 by CRC Press LLC
% Default example using z=R*(zeta+m/zeta)
if nargin==0
    zeta=linspace(1,3,10)'* ... 
    exp(i*linspace(0,2*pi,81)); 
    a=2; b=1; R=(a+b)/2; m=(a-b)/(a+b); 
    z=R*(zeta+m./zeta); x=real(z); y=imag(z); 
    titl=[{'Circular Annulus Mapped onto an ', ... 
        'Elliptical Annulus'}];
end
range=cubrange([x(:),y(:)],1.1);

% The data define a curve
if size(x,1)==1 | size(x,2)==1
    plot(x,y,'-k'); xlabel(xlabl); ylabel(ylabl); 
    title(titl); axis('equal'); axis(range); 
    grid on; figure(gcf);
    if nargin==0
        print -deps gridviewl;
    end
% The data define a surface
else
    plot(x,y,'k-',x',y','k-') 
    xlabel(xlabl); ylabel(ylabl); title(titl); 
    axis('equal'); axis(range); grid on; 
    figure(gcf);
    if nargin==0
        print -deps gridview;
    end
end

%========================================================================

function range=cubrange(xyz,ovrsiz)

% range=cubrange(xyz,ovrsiz)
% -----------------------------
% This function determines limits for a square 
% or cube shaped region for plotting data values 
% in the columns of array xyz to an undistorted 
% scale

© 2003 by CRC Press LLC
% xyz - a matrix of the form [x,y] or [x,y,z] where x,y,z are vectors of coordinate points
% ovrsiz - a scale factor for increasing the window size. This parameter is set to one if only one input is given.
% range - a vector used by function axis to set undistorted. This vector has the form [xmin,xmax,ymin,ymax] when xyz has only two columns or the form [xmin,xmax,ymin,ymax,zmin,zmax] when xyz has three columns.

% User m functions called: none

Function inputv

function varargout=inputv(prompt)
% % [a1,a2,...,a_nargout]=inputv(prompt)
%---------------------------------------------------
% % This function reads several values on one line. The items should be separated by commas or blanks.
% % prompt - A string preceding the data entry. It is set to ’ ’ if no value of prompt is given.
% % a1,a2,...,a_nargout - The output variables
that are created. If not enough data values are given following the prompt, the remaining undefined values are set equal to NaN.

A typical function call is:

\[ [A,B,C,D]=inputv('Enter values of A,B,C,D: ') \]

--------------------------------------------

```matlab
if nargin==0, prompt=' ? '; end
u=input(prompt,'s'); v=eval(['[',u,']']);
ni=length(v); no=nargout;
varargout=cell(1,no); k=min(ni,no);
for j=1:k, varargout{j}=v(j); end
if no>ni
for j=ni+1:no, varargout{j}=nan; end
end
```

Function `lintrp`

```matlab
function y=lintrp(xd,yd,x)
% y=lintrp(xd,yd,x)
% ------------------
% This function performs piecewise linear interpolation through data values stored in xd, yd, where xd values are arranged in nondecreasing order. The function can handle discontinuous functions specified when some successive values in xd are equal. Then the repeated xd values are shifted by a small amount to remove the discontinuities.
% Interpolation for any points outside the range of xd is also performed by continuing the line segments through the outermost data pairs.
% xd,yd - vectors of interpolation data values
% x - matrix of values where interpolated values are required
```

© 2003 by CRC Press LLC
% y - matrix of interpolated values
k=find(diff(xd)==0);
if length(k)~=0
    xd(k+1)=xd(k+1)+(xd(end)-xd(1))*1e3*eps;
end
y=interp1(xd,yd,x,'linear','extrap');

Function manyrts

function roots=manyrts(func,a,b,nsteps,...
    maxrts,tol,varargin)
  % roots=manyrts(func,a,b,nsteps,maxrts,tol,...
  % varargin)
  % ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  % This function attempts to find multiple roots
  % of a function by searching an interval in steps
  % of equal length and finding a root in each
  % interval where a sign change occurs
  % func - name of a function of the form
  %       func(x,p1,p2,...) where additional
  %       parameters after the first are
  %       passed through varargin
  % a,b - upper and lower limits of the
  %       search interval
  % nsteps - number of intervals from a to b
  % which are checked to detect a
  % sign change
  % maxrts - maximum number of roots sought
  % within the search limits. The
  % search terminates when the number
  % of roots found equals maxrts.
  % tol - the root tolerance passed to
  % function fzero. A default value of
  % 1e-10 is used if no value is given
  % varargin - the cell variable provided to pass
  % multiple arguments to function func

if nargin<6, tol=1e-10; end;
if nargin<5, maxrts=100; end
if isstruct(tol), options=tol;
else
options=optimset('tolfun',tol,'tolx',tol);
end
x=linspace(a,b,nsteps); roots=[];
rtlast=-realmax;
for j=1:nsteps-1
xj=x(j); xj1=x(j+1);
fj=feval(func,xj,varargin{:});
fj1=feval(func,xj1,varargin{:});
if fj.*fj1<=0
    rt=fzero(func,[xj,xj1],...
        options,varargin{:});
    if (rt-rtlast)>tol
        roots=[roots,rt]; rtlast=rt;
    end
end
if length(roots)==maxrts, break, end
end

Function membanim

function membanim(u,x,y,t)
%
% function membanim(u,x,y,t)
% --------------------------
% This function animates the motion of a
% vibrating membrane
%
% u array in which component u(i,j,k) is the
% displacement for y(i),x(j),t(k)
% x,y arrays of x and y coordinates
% t vector of time values
%
% Compute the plot range
if nargin==0;
[u,x,y,t]=memrecwv(2,1,15.5,1.5,.5,5);
end
xmin=min(x(:)); xmax=max(x(:));
ymin=min(y(:)); ymax=max(y(:));
xmid=(xmin+xmax)/2; ymid=(ymin+ymax)/2;
d=max(xmax-xmin,ymax-ymin)/2; Nt=length(t);
range=[xmid-d,xmid+d,ymid-d,ymid+d,...
    3*min(u(:)),3*max(u(:))];
while 1  % Show the animation repeatedly
    disp(' '), disp('Press return for animation')
    dumy=input('or enter 0 to stop > ? ','s');
    if ~isempty(dumy)
        disp(' '), disp('All done'), break
    end

    % Plot positions for successive times
    for j=1:Nt
        surf(x,y,u(:,j)), axis(range)
        xlabel('x axis'), ylabel('y axis')
        zlabel('u axis'), titl=sprintf(...
            'MEMBRANE POSITION AT T=%5.2f',t(j));
        title(titl), colormap([1 1 1])
        colormap([127/255 1 212/255])
        axis off
        drawnow, shg, pause(.1)
    end
end

Function plotmotn

function plotmotn(x,y,titl,isave)

  % plotmotn(x,y,titl,isave)
  % ------------------------
  % This function plots the cable time
  % history described by coordinate values
  % stored in the rows of matrices x and y.
  %
  % x,y - matrices having successive rows
  % which describe position
  % configurations for the cable
  % titl - a title shown on the plots
  % isave - parameter controlling the form
  % of output. When isave is not input,
  % only one position at a time is shown
  % in rapid succession to animate the
  % motion. If isave is given a value,
  % then successive are all shown at
  % once to illustrate a kinematic
  % trace of the motion history.

© 2003 by CRC Press LLC
% User m functions called:  none

%----------------------------------------------

% Set a square window to contain all
% possible positions
[nt,n]=size(x);
if nargin==4, save =1; else, save=0; end
xmin=min(x(:)); xmax=max(x(:));
ymin=min(y(:)); ymax=max(y(:));
w=max(xmax-xmin,ymax-ymin)/2;
xmd=(xmin+xmax)/2; ymd=(ymin+ymax)/2;
hold off; clf; axis('normal'); axis('equal');
title(titl)
xlabel('x axis'); ylabel('y axis')
if save==0
for j=1:nt
  xj=x(j,:); yj=y(j,:);
  plot(xj,yj,'-k',xj,yj,'ok');
  axis(range), axis off
  title(titl)
  figure(gcf), drawnow, pause(.1)
end
pause(2)
else
hold off; close
for j=1:nt
  xj=x(j,:); yj=y(j,:);
  plot(xj,yj,'-k',xj,yj,'ok');
  axis(range), axis off, hold on
end
title(titl)
figure(gcf), drawnow, hold off, pause(2)
end

% Save plot history for subsequent printing
% print -deps plotmotn

Function polhedrn

function [v,rc,vrr,irr]=polhedrn(x,y,z,idface)
% %
% [v,rc,vrr,irr]=polhedrn(x,y,z,idface)
% This function determines the volume, 
% centroidal coordinates and inertial moments 
% for an arbitrary polyhedron.

% x,y,z - vectors containing the corner 
% indices of the polyhedron 
% idface - a matrix in which row j defines the 
% corner indices of the j’th face. 
% Each face is traversed in a 
% counterclockwise sense relative to 
% the outward normal. The column 
% dimension equals the largest number 
% of indices needed to define a face. 
% Rows requiring fewer than the 
% maximum number of corner indices are 
% padded with zeros on the right. 

% v - the volume of the polyhedron 
% rc - the centroidal radius 
% vrr - the integral of R*R’*d(vol) 
% irr - the inertia tensor for a rigid body 
% of unit mass obtained from vrr as 
% eye(3,3)*sum(diag(vrr))-vrr 

% User m functions called: pyramid

r=[x(:),y(:),z(:)]; nf=size(idface,1);
v=0; vr=0; vrr=0;
for k=1:nf
   i=idface(k,:); i=i(find(i>0));
   [u,ur,urr]=pyramid(r(i,:));
   v=v+u; vr=vr+ur; vrr=vrr+urr;
end
rc=vr/v; irr=eye(3,3)*sum(diag(vrr))-vrr;

Function polyxy

function [area,xbar,ybar,axx,axy,ayy]=polyxy(x,y)
% This function computes the area, centroidal
% coordinates, and inertial moments of an
% arbitrary polygon.
%
% x,y - vectors containing the corner
% coordinates. The boundary is
% traversed in a counterclockwise
% direction
%
% area - the polygon area
% xbar,ybar - the centroidal coordinates
% axx - integral of x^2*dx*dy
% axy - integral of xy*dx*dy
% ayy - integral of y^2*dx*dy
%
% User m functions called: none
%----------------------------------------------

n=1:length(x); n1=n+1;
x=[x(:);x(1)]; y=[y(:);y(1)];
a=(x(n).*y(n1)-y(n).*x(n1))';
area=sum(a)/2; a6=6*area;
xbar=a*(x(n)+x(n1))/a6; ybar=a*(y(n)+y(n1))/a6;
ayy=a*(y(n).^2+y(n).*y(n1)+y(n1).^2)/12;
axy=a*(x(n).*(2*y(n)+y(n1))+x(n1).* ... 
   (2*y(n1)+y(n)))./24;
axx=a*(x(n).^2+x(n).*x(n1)+x(n1).^2)/12;

Function quadlsqrt

function v=quadlsqrt(fname,type,a,b,tol,trace,varargin)
% v=quadlsqrt(fname,type,a,b,tol,trace,varargin)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function uses the MATLAB integrator quadl
% to evaluate integrals having square root type
% singularities at one or both ends of the
% integration interval a < x < b.
% The integrand has the form:
% func(x)/sqrt(x-a) if type=1.
% func(x)/sqrt(b-x) if type==2.
% func(x)/sqrt((x-a)*(b-x)) if type==3.
% func - the handle for a function continuous
% from x=a to x=b
% type - 1 if the integrand is singular at x=a
% 2 if the integrand is singular at x=b
% 3 if the integrand is singular at both x=a and x=b.
% a,b - integration limits with b > a

if nargin<6 | isempty(trace), trace=0; end
if nargin<5 | isempty(tol), tol=1e-8; end
if nargin<7
varargin{1}=type; varargin{2}=[a,b];
varargin{3}=fname;
else
n=length(varargin); c=[a,b]; varargin{n+1}=type;
varargin{n+2}=c; varargin{n+3}=fname;
end

if type==1 | type==2
v=2*quadl(@fshift,0,sqrt(b-a),...
tol,trace,varargin{:});
else
v=quadl(@fshift,0,pi,tol,trace,varargin{:});
end

function u=fshift(x,varargin)
% u=fshift(x,varargin)
% This function shifts arguments to produce
% a nonsingular integrand called by quadl
N=length(varargin); fname=varargin{N};
c=varargin{N-1}; type=varargin{N-2};
a=c(1); b=c(2); c1=(b+a)/2; c2=(b-a)/2;

switch type
    case 1, t=a+x.^2; case 2, t=b-x.^2;
    case 3, t=c1+c2*cos(x);
end

if N>3, u=feval(fname,t,varargin{1:N-3});
else, u=feval(fname,t); end
Function ratcof

1: function [a,b]=ratcof(xdata,ydata,ntop,nbot)

2: %

3: % [a,b]=ratcof(xdata,ydata,ntop,nbot)

4: -------------------------------

5: % Determine a and b to approximate ydata as
6: % a rational function of the variable xdata.
7: % The function has the form:
8: %
9: % y(x) = sum(1=>ntop) ( a(j)*x^(j-1) ) /
10: %    ( 1 + sum(1=>nbot) ( b(j)*x^(j) ) )
11: %
12: % xdata,ydata - input data vectors (real or
13: % complex)
14: % ntop,nbot - number of series terms used in
15: % the numerator and the
16: % denominator.
17: %----------------------------------------------
18: 21: ydata=ydata(:); xdata=xdata(:);
19: m=length(ydata);
20: if nargin==3, nbot=ntop; end;
21: x=ones(m,ntop+nbot); x(:,ntop+1)=-ydata.*xdata;
22: for i=2:ntop, x(:,i)=xdata.*x(:,i-1); end
23: for i=2:nbot
24:     x(:,i+ntop)=xdata.*x(:,i+ntop-1);
25: end
26: ab=pinv(x)*ydata; %ab=x\ydata;
27: a=ab(1:ntop); b=ab(ntop+1:ntop+nbot);

Function raterp

1: function y=raterp(a,b,x)

2: %
3: % y=raterp(a,b,x)
4: -------------------------
5: % This function interpolates using coefficients
6: % from function ratcof.
7: %
% a,b - polynomial coefficients from function
% ratcof
% x - argument at which function is evaluated
% y - computed rational function values
%----------------------------------------------
a=flipud(a(:)); b=flipud(b(:));
y=polyval(a,x)./(1+x.*polyval(b,x));

Function smdsolve

function [x,v]=smdsolve(m,c,k,f1,f2,w,x0,v0,t)
% [x,v]=smdsolve(m,c,k,f1,f2,w,x0,v0,t)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function solves the differential equation
% m*x''(t)+c*x'(t)+k*x(t)=f1*cos(w*t)+f2*sin(w*t)
% with x(0)=x0 and x'(0)=v0
% m,c,k - mass, damping and stiffness coefficients
% f1,f2 - magnitudes of cosine and sine terms in
% the forcing function
% w - frequency of the forcing function
% t - vector of times to evaluate the solution
% x,v - computed position and velocity vectors
ccrit=2*sqrt(m*k); wn=sqrt(k/m);
% If the system is undamped and resonance will
% occur, add a little damping
if c==0 & w==wn; c=ccrit/1e6; end;
% If damping is critical, modify the damping
% very slightly to avoid repeated roots
if c==ccrit; c=c*(1+1e-6); end
% Forced response solution
a=(f1-i*f2)/(k-m*w^2+i*c*w);
X0=real(a); V0=real(i*w*a);
X=real(a*exp(i*w*t)); V=real(i*w*a*exp(i*w*t));
% Homogeneous solution

© 2003 by CRC Press LLC
32: \[ r = \sqrt{c^2 - 4*m*k}; \]
33: \[ s_1 = \frac{-c + r}{2*m}; \quad s_2 = \frac{-c - r}{2*m}; \]
34: \[ p = [1,1;s_1,s_2][x_0-X_0;v_0-V_0]; \]
35: % Total solution satisfying the initial conditions
36: \[ x = X + \text{real}(p(1)*\exp(s_1*t) + p(2)*\exp(s_2*t)); \]
37: \[ v = V + \text{real}(p(1)*s_1*\exp(s_1*t) + p(2)*s_2*\exp(s_2*t)); \]

Function splined

1: function val=splined(xd,yd,x,if2)
2: %
3: % val=splined(xd,yd,x,if2)
4: % -------------------------------------
5: % This function evaluates the first or second
6: % derivative of the piecewise cubic
7: % interpolation curve defined by the intrinsic
8: % function spline provided in MATLAB. If fewer
9: % than four data points are input, then simple
10: % polynomial interpolation is employed
11: %
12: % xd,yd - data vectors determining the spline
13: % curve produced by function spline
14: % x     - vector of values where the first or
15: %         the second derivative are desired
16: % if2   - a parameter which is input only if
17: %         y''(x) is required. Otherwise, y'(x)
18: %         is returned.
19: %
20: % val   - the first or second derivative values
21: % for the spline
22: %
23: % User m functions called: none
24: %
25: m=length(xd); [b,c]=unmkpp(spline(xd,yd));
26: if n>3 % Use a cubic spline
27:   if nargin==3, c=[3*c(:,1),2*c(:,2),c(:,3)];
28:   else, c=[6*c(:,1),2*c(:,2)]; end
29: end
30: val=ppval(mkpp(b,c),x);
31: else % Use a simple polynomial
32:   c=polyder(polyfit(xd(:),yd(:),n-1));
33:   if nargin==4, c=polyder(c); end

© 2003 by CRC Press LLC
val=polyval(c,x);
end

Function splineg

function [val,b,c]=splineg(xd,yd,x,deriv,endc,b,c)
% [val,b,c]=splineg(xd,yd,x,deriv,endc,b,c)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
% For a cubic spline curve through data points
% xd,yd, this function evaluates y(x), y'(x),
% y''(x), or integral(y(x)*dx, xd(1) to x(j) )
% for j=1:length(x).The coefficients needed to
% evaluate the spline are also computed.
% xd,yd - data vectors defining the cubic
% spline curve
% x - vector of points where curve
% properties are computed.
% deriv - denoting the spline curve as y(x),
% deriv=0 gives a vector for y(x)
% deriv=1 gives a vector for y'(x)
% deriv=2 gives a vector for y''(x)
% deriv=3 gives a vector of values
% for integral(y(z)*dz) from xd(1)
% to x(j) for j=1:length(x)
% endc - endc=1 makes y'''(x) continuous at
% xd(2) and xd(end-1).
% endc=[2,left_slope,right_slope]
% imposes slope values at both ends.
% endc=[3,left_slope] imposes the left
% end slope and makes the discontinuity
% of y''' at xd(end-1) small.
% endc=[4,right_slope] imposes the right
% end slope and makes the discontinuity
% of y''' at xd(2) small.
% b,c coefficients needed to perform the
% spline interpolation. If these are not
% given, function unmkpp is called to
% generate them.
% val values y(x),y'(x),y''(x) or
% integral(y(z)dz, z=xd(1)..x) for
% deriv=0,1,2, or 3, respectively.

% User m files called: splincof
% -------------------------------------------
if nargin<5 | isempty(endc), endc=1; end
if nargin<7, [b,c]=splincof(xd,yd,endc); end
n=length(xd); [N,M]=size(c);

switch deriv

case 0 % Function value
    val=ppval(mkpp(b,c),x);

case 1 % First derivative
    C=[3*c(:,1),2*c(:,2),c(:,3)];
    val=ppval(mkpp(b,C),x);

case 2 % Second derivative
    C=[6*c(:,1),2*c(:,2)];
    val=ppval(mkpp(b,C),x);

case 3 % Integral values from xd(1) to x
    k=M:-1:1;
    C=[c./k(ones(N,1),:),zeros(N,1)];
    dx=xd(2:n)-xd(1:n-1); s=zeros(n-2,1);
    for j=1:n-2, s(j)=polyval(C(j,:),dx(j)); end
    C(:,5)=[0;cumsum(s)]; val=ppval(mkpp(b,C),x);
end

%=============================================

function [b,c]=splincof(xd,yd,endc)

% [b,c]=splincof(xd,yd,endc)
% ~~~~~~~~~~~~~~~~~~~~~~~~~~
% This function determines coefficients for
% cubic spline interpolation allowing four
% different types of end conditions.
% xd,yd - data vectors for the interpolation
% endc - endc=1 makes y'''(x) continuous at
% xd(2) and xd(end-1).
% endc=[2,left_slope,right_slope]
% imposes slope values at both ends.
% endc=[3,left_slope] imposes the left
% end slope and makes the discontinuity
% of y''' at xd(end-1) small.
endc=[4,right_slope] imposes the right
% end slope and makes the discontinuity
% of y''' at xd(2) small.
if nargin<3, endc=1; end;
type=endc(1); xd=xd(:); yd=yd(:);
switch type
    case 1
        % y'''(x) continuous at the xd(2) and xd(end-1)
        [b,c]=unmkpp(spline(xd,yd));
    case 2
        % Slope given at both ends
        [b,c]=unmkpp(spline(xd,[endc(2);yd;endc(3)]));
    case 3
        % Slope at left end given. Compute right end
        % slope.
        [b,c]=unmkpp(spline(xd,yd));
        c=[3*c(:,1),2*c(:,2),c(:,3)];
        sright=ppval(mkpp(b,c),xd(end));
        [b,c]=unmkpp(spline(xd,[endc(2);yd;sright]));
    case 4
        % Slope at right end known. Compute left end
        % slope.
        [b,c]=unmkpp(spline(xd,yd));
        c=[3*c(:,1),2*c(:,2),c(:,3)];
        sleft=ppval(mkpp(b,c),xd(1));
        [b,c]=unmkpp(spline(xd,[sleft;yd;endc(2)]));
end

Function spterp

function [v,c]=spterp(xd,yd,id,x,endv,c)
    % [v,c]=spterp(xd,yd,id,x,endv,c)
    % This function performs cubic spline interpo-
% lation. Values of y(x),y'(x),y''(x) or the
% integral(y(t)*dt, xd(1)..x) are obtained.
% xd, yd - data vectors with xd arranged in
% ascending order.
% id   - id equals 0,1,2,3 to compute y(x),
%       y'(x), integral(y(t)*dt,t=xd(1)..x),
%       respectively.
% v    - values of the function, first deriva-
%       tive, second derivative, or integral
%       from xd(1) to x
% c    - the coefficients defining the spline
%       curve.
% endv - vector giving the end conditions in
%       one of the following five forms:
%       endv=1 or endv omitted makes
%       c(2) and c(n-1) zero
%       endv=[2,left_end_slope,...
%         right_end_slope] to impose slope
%       values at each end
%       endv=[3,left_end_slope] imposes the
%       left end slope value and makes
%       c(n-1) zero
%       endv=[4,right_end_slope] imposes the
%       right end slope value and makes
%       c(2) zero
%       endv=5 defines a periodic spline by
%       making y,y',y" match at both ends

if nargin<5 | isempty(endv), endv=1; end
n=length(xd); sx=size(x); x=x(:); X=x-xd(1);
if nargin<6, c=spcof(xd,yd,endv); end
C=c(1:n); s1=c(n+1); m1=c(n+2); X=x-xd(1);
if id==0  % y(x)
v=yd(1)+s1*X+m1/2*X.*X+...
powermat(x,xd,3)*C/6;
elseif id==1  % y'(x)
v=s1+m1*X+powermat(x,xd,2)*C/2;
elseif id==2  % y''(x)
v=m1+powermat(x,xd,1)*C;
else        % integral(y(t)*dt, t=xd(1)..x)
v=yd(1)*X+s1/2*X.*X+m1/6*X.^3+...
powermat(x,xd,4)*C/24;

© 2003 by CRC Press LLC
v=reshape(v,sx);

function c=spcof(x,y,endv)

% This function determines spline interpolation
% coefficients consisting of the support
% reactions concatenated with y' and y'' at
% the left end.
% x,y - data vectors of interplation points.
% Denote n as the length of x.
% endv - vector of data for end conditions
% described in function speterp.
% c - a vector [c(1);...;c(n+2)] where the
% first n components are support
% reactions and the last two are
% values of y'(x(1)) and y''(x(1)).

if nargin<3, endv=1; end
x=x(:); y=y(:); n=length(x); u=x(2:n)-x(1);
a=zeros(n+2,n+2); a(1,1:n)=1;
a(2:n,:)= [powermat(x(2:n),x,3)/6,u,u.*u/2];
b=zeros(n+2,1); b(2:n)=y(2:n)-y(1);
if endv(1)==1 % Force, force condition
    a(n+1,2)=1; a(n+2,n-1)=1;
b(n+1)=endv(2); a(n+1,n+1)=1;
b(n+2)=endv(3); a(n+2,1:n)= [((x(n)-x').^2)/2,1,x(n)-x(1)];
elseif endv(1)==2 % Slope, slope condition
    a(n+1,1:n)=x(n)-x'; b(n)=0;
a(n+2,1:n)=1/2*(x(n)-x').^2;
a(n+2,n+2)=x(n)-x(1);
else
    error('Invalid value of endv in function spcof')
end
if endv(1)==1 & n<4, c=pinv(a)*b;
else, c=a/b; end

%==============================================
% This function evaluates various powers of a
% matrix used in cubic spline interpolation.
% x,X - arbitrary vectors of length n and N
% a - an n by M matrix of elements such that
% a(i,j)=(x(i)>X(j))*abs(x(i)-X(j))^p
x=x(:); n=length(x); X=X(:)'; N=length(X);
a=x(:,ones(1,N))-X(ones(n,1),:); a=a.*(a>0);
switch p, case 0, a=sign(a); case 1, return;
case 2, a=a.*a; case 3; a=a.*a.*a;
case 4, a=a.*a; a=a.*a; otherwise, a=a.^p; end

---

Function srfv

function [v,rc,vrr]=srfv(x,y,z)
% [v,rc,vrr]=srfv(x,y,z)
% This function computes the volume, centroidal
% coordinates, and inertial tensor for a volume
% covered by surface coordinates contained in
% arrays x,y,z
% x,y,z - matrices containing the coordinates
% of a grid of points covering the
% surface of the solid
% v - volume of the solid
% rc - centroidal coordinate vector of the
% solid
% vrr - inertial tensor for the solid with the
% mass density taken as unity
% User functions called: scatripl proptet
%-----------------------------------------------
% p=inline(...
% 'v*(eye(3)*(r(:)''*r(:))-r(:)*r(:)''),'v','r');
%d=mean([x(:,y(:,z(:))]);
%x=x-d(1); y=y-d(2); z=z-d(3);

[n,m]=size(x); i=1:n-1; I=i+1; j=1:m-1; J=j+1;
xij=x(i,j); yij=y(i,j); zij=z(i,j);
xIj=x(I,j); yIj=y(I,j); zIj=z(I,j);
xIJ=x(I,J); yIJ=y(I,J); zIJ=z(I,J);
xJ=x(i,J); yJ=y(i,J); zJ=z(i,J);

% Tetrahedron volumes
v1=scatripl(xij,yij,zij,xIj,yIj,zIj,xIJ,yIJ,zIJ);
v2=scatripl(xij,yij,zij,xIJ,yIJ,zIJ,xiJ,yiJ,ziJ);
v=sum(sum(v1+v2));

% First moments of volume
X1=xij+xIj+xIJ; X2=xij+xIJ+xiJ;
Y1=yij+yIj+yIJ; Y2=yij+yIJ+yiJ;
Z1=zij+zIj+zIJ; Z2=zij+zIJ+ziJ;
vx=sum(sum(v1.*X1+v2.*X2));
vy=sum(sum(v1.*Y1+v2.*Y2));
vz=sum(sum(v1.*Z1+v2.*Z2));

% Second moments of volume
vrr=proptet(v1,xij,yij,zij,xIj,yIj,zIj,...
  xIJ,yIJ,zIJ,x1,Y1,Z1)+...
  proptet(v2,xij,yij,zij,xIJ,yIJ,zIJ,...
  xiJ,yiJ,zij,X2,Y2,Z2);
rc=[vx,vy,vz]/v4; vs=sign(v);
v=abs(v)/6; vrr=vs*vrr/120;
vrr=[vrr([1 4 5]), vrr([4 2 6]), vrr([5 6 3])];
vec=eye(3,3)*sum(diag(vrr))'-vrr;

%vrr=vrr-p(v,rc)+p(v,rc+d); rc=rc+d;

Function strdynrk

function [t,x,v]=strdynrk(t,x0,v0,m,c,k,functim)
% [t,x,v]=strdynrk(t,x0,v0,m,c,k,functim)
% This function uses ode45 to solve the matrix
% differential equation: M*X''+C*X'+K*X=F(t)
© 2003 by CRC Press LLC
% t - vector of solution times
% x0,v0 - initial position and velocity vectors
% m,c,k - mass, damping and stiffness matrices
% functim - character name for the driving force
% x,v - arrays containing solution values for position and velocity

% A typical call to strdynrk function is:
% m=eye(3,3); k=[2,-1,0;-1,2,-1;0,-1,2];
% c=.05*k; x0=zeros(3,1); v0=zeros(3,1);
% t=linspace(0,10,101);
% [t,x,v]=strdynrk(t,x0,v0,m,c,k,'func');

global Mi C K F n n1 n2
Mi=inv(m); C=c; K=k; F=functim;
n=size(m,1); n1=1:n; n2=n+1:2*n;
[t,z]=ode45(@sde,t,[x0(:);v0(:)]);
x=z(:,n1); v=z(:,n2);

function zp=sde(t,z)
% zp=sde(t,z)
% This function determines the closest points on two surfaces and the distance between these points. It is similar to function srf2srf except that large

%================================
function f=func(t)
% f=func(t)
% This is an example forcing function for function strdynrk in the case of three degrees of freedom.
f=[-1;0;2]*sin(1.413*t);

Function surf2surf

function [d,r,R]=surf2surf(x,y,z,X,Y,Z,n)
% [d,r,R]=surf2surf(x,y,z,X,Y,Z,n)
% This function determines the closest points on two surfaces and the distance between these points. It is similar to function srf2srf except that large
% arrays can be processed.
% x, y, z - arrays of points on the first surface
% X, Y, Z - arrays of points on the second surface
% d - the minimum distance between the surfaces
% r, R - vectors containing the coordinates of the nearest points on the first and the second surface
% n - length of subvectors used to process the data arrays. Sending vectors of length n to srf2srf and taking the best of the subresults allows processing of large arrays of data points

% User m functions used: srf2srf

if nargin<7, n=500; end
N=prod(size(x)); M=prod(size(X)); d=realmax;
kN=max(1,floor(N/n)); kM=max(1,floor(M/n));
for i=1:kN
    i1=1+(i-1)*n; i2=min(i1+n,N); i12=i1:i2;
    xi=x(i12); yi=y(i12); zi=z(i12);
    for j=1:kM
        j1=1+(j-1)*n; j2=min(j1+n,M); j12=j1:j2;
        [dij,rij,Rij]=srf2srf(xi,yi,zi,X(j12),Y(j12),Z(j12));
        if dij<d, d=dij; r=rij; R=Rij; end
    end
end

%================================================================================================= %

function [d,r,R]=srf2srf(x,y,z,X,Y,Z)
% [d,r,R]=srf2srf(x,y,z,X,Y,Z)
% This function determines the closest points on two surfaces and the distance between these points.
% x, y, z - arrays of points on the first surface
% X, Y, Z - arrays of points on the second surface
% d - the minimum distance between the surfaces
% r, R - vectors containing the coordinates of the nearest points on the first and the second surface

x=x(:); y=y(:); z=z(:); n=length(x); v=ones(n,1);
X=X(:)'; Y=Y(:)'; Z=Z(:)'; N=length(X); h=ones(1,N);

d2=(x(:,h)-X(v,:)).^2; d2=d2+(y(:,h)-Y(v,:)).^2;

d2=d2+(z(:,h)-Z(v,:)).^2;

[u,i]=min(d2); [d,j]=min(u); i=i(j); d=sqrt(d);

r=[x(i);y(i);z(i)]; R=[X(j);Y(j);Z(j)];

**Function surfmany**

```matlab
function surfmany(varargin)
%function surfmany(x1,y1,z1,x2,y2,z2,...
%    x3,y3,z3,...,xn,yn,zn)
% This function plots any number of surfaces
% on the same set of axes without shape
% distortion. When no input is given, then a
% six-legged solid composed of spheres and
cylinders is shown.
%
% User m functions called: none
%----------------------------------------------

if nargin==0

% Default data for a six-legged solid
n=10; rs=.25; d=7; rs=2; rc=.75;
[xs,ys,zs]=sphere; [xc,yc,zc]=cylinder;
xs=rs*xs; ys=rs*ys; zs=rs*zs;
xc=rc*xc; yc=rc*yc; zc=2*d*zc-d;
x1=xs; y1=ys; z1=zs;
x2=zs+d; y2=ys; z2=xs;
x3=zs-d; y3=ys; z3=xs;
x4=xs; y4=zs-d; z4=ys;
x5=xs; y5=zs+d; z5=ys;
x6=xs; y6=ys; z6=zs+d;
x7=xs; y7=ys; z7=zs-d;
x8=xc; y8=yc; z8=zc;
x9=zc; y9=xc; z9=yc;
x10=yc; y10=zc; z10=xc;
varargin={x1,y1,z1,x2,y2,z2,x3,y3,z3,...
x4,y4,z4,x5,y5,z5,x6,y6,z6,x7,y7,z7,...
x8,y8,z8,x9,y9,z9,x10,y10,z10};

end

% Find the data range
n=length(varargin);
```

© 2003 by CRC Press LLC
36  \texttt{r=realmax*[1,-1,1,-1,1,-1];}
37  \texttt{s=inline('min([a;b]),'a','b');}
38  \texttt{b=inline('max([a;b]),'a','b');}
39  \texttt{for \texttt{k}=1:3:n}
40  \texttt{x=varargin{\texttt{k}}; y=varargin{\texttt{k}+1};}
41  \texttt{z=varargin{\texttt{k}+2};}
42  \texttt{x=x(:); y=y(:); z=z(:);}
43  \texttt{r(1)=s(r(1),x); r(2)=b(r(2),x);}
44  \texttt{r(3)=s(r(3),y); r(4)=b(r(4),y);}
45  \texttt{r(5)=s(r(5),z); r(6)=b(r(6),z);}
46  \texttt{end}
47
48  \% Plot each surface
49  \texttt{hold off, newplot}
50  \texttt{for \texttt{k}=1:3:n}
51  \texttt{x=varargin{\texttt{k}}; y=varargin{\texttt{k}+1};}
52  \texttt{z=varargin{\texttt{k}+2};}
53  \texttt{surf(x,y,z); axis(r), hold on}
54  \texttt{end}
55
56  \% Set axes and display the combined plot
57  \texttt{axis equal, axis(r), grid on}
58  \texttt{xlabel('x axis'), ylabel('y axis')}
59  \texttt{zlabel('z axis')}
60  \texttt{title('SEVERAL SURFACES COMBINED')}
61  \% colormap([127/255 1 212/255]); \% aquamarine
62  \texttt{colormap([1 1 1]); figure(gcf), hold off}

\textbf{Function volrevol}

\begin{verbatim}
function \texttt{[v,rg,Irr,X,Y,Z,aprop,xd,zd,kn]=...}
  \texttt{volrev(xd,zd,kn,th,nth,noplot)}
\end{verbatim}

% This function computes geometrical properties
% for a volume of revolution resulting when a
% closed curve in the \texttt{(x,z) plane is rotated},
% \% through given angular limits, about the \texttt{z} axis.
% The cross section of the volume is defined by
% a spline curve passed through data points

© 2003 by CRC Press LLC
% (xd,zd) in the same manner as was done in
% function areaprop for plane areas.

% xd,zd - data vectors defining the spline
% interpolated boundary, which is
% traversed in a counterclockwise
% direction
% kn - indices of any points where slope
% discontinuity is allowed to turn
% sharp corners
% p - vector of volume properties containing
% [v, xcg, ycg, zcg, vxx, vyy, vzz,...
% vxy, vyz, vzx] where v is the volume,
% (xcg,ycg,zcg) are coordinates of the
% centroid, and the remaining properties
% are volume integrals of the following
% integrand:
% [x.^, y.^2, z.^2, xy, yz, zx]*dxdydz
% X,Y,Z - data arrays containing points on the
% surface of revolution. Plotting these
% points shows the solid volume with
% the ends left open. Function fill3
% is used to plot the surface with ends
% closed
% apro - a vector containing properties of the
% area in the (x,z) plane that was used
% to generate the volume. apro=[area,...
% xcentroidal, ycentroidal, axx, axz, azz].

% User m functions called: rotasurf, gcquad,
% curve2d, anglefun, splined
%--------------------------------------------------------
if nargin==0
    t1=-pi:pi/6:0; t2=0:pi/6:pi;
    Zd=[0,exp(i*t1),1/2+i+exp(i*t2)/2,0,-1];
    xd=real(Zd)+4; zd=imag(Zd);
    kn=[1,2,8,9,15,16];
    th=[-pi/2,pi]; nth=31;
end

% Plot a surface of revolution based on the
% input data points
if nargin==6
    [X,Y,Z]=rotasurf(xd,zd,th,nth,1);
else

© 2003 by CRC Press LLC
60: end

62: % Obtain base points and weight factors for the
63: % composite Gauss formula of order seven used in
64: % the numerical integration
65: nd=length(xd); nseg=nd-1;
66: [dum,bp,wf]=gcquad([],1,nd,7,nseg);

68: % Evaluate complex points and derivative values
69: % on the spline curve which is rotated to form
70: % the volume of revolution
71: [u,uplot,up]=curve2d(xd,zd,kn,bp);
72: % plot(real(uplot),imag(uplot)), axis equal,shg
73: u=u(:); up=up(:); n=length(bp);
74: x=real(u); dx=real(up); z=imag(u);
75: dz=imag(up); da=x.*dz-z.*dx;

77: % Evaluate line integrals for area properties
78: p=[ones(n,1), x, z, x.^2, x.*z, z.^2, x.^3,...
79:     (x.^2).*z, x.*(z.^2)].*repmat(da,1,9);
80: p=(wf(:)'*p)./[2 3 3 4 4 5 5 5 1];

82: % Scale area properties by multipliers involving
83: % the rotation angle for the volume
84: f=anglefun(th(2))-anglefun(th(1));
85: v=f(1)*p(2); rg=f([2 3 1]).*p([4 4 5])/v;
86: vrrr=[f([4 5 2]); f([5 6 3]); f([2 3 1])].*...
87:     [p([7 7 8]); p([7 7 8]); p([8 8 9])];
88: Irr=eye(3)*sum(diag(vrr))-vrr;
89: aprop=[p(1),p(2:3)/p(1),p(4:6)];
References


© 2003 by CRC Press LLC


© 2003 by CRC Press LLC


© 2003 by CRC Press LLC


© 2003 by CRC Press LLC


